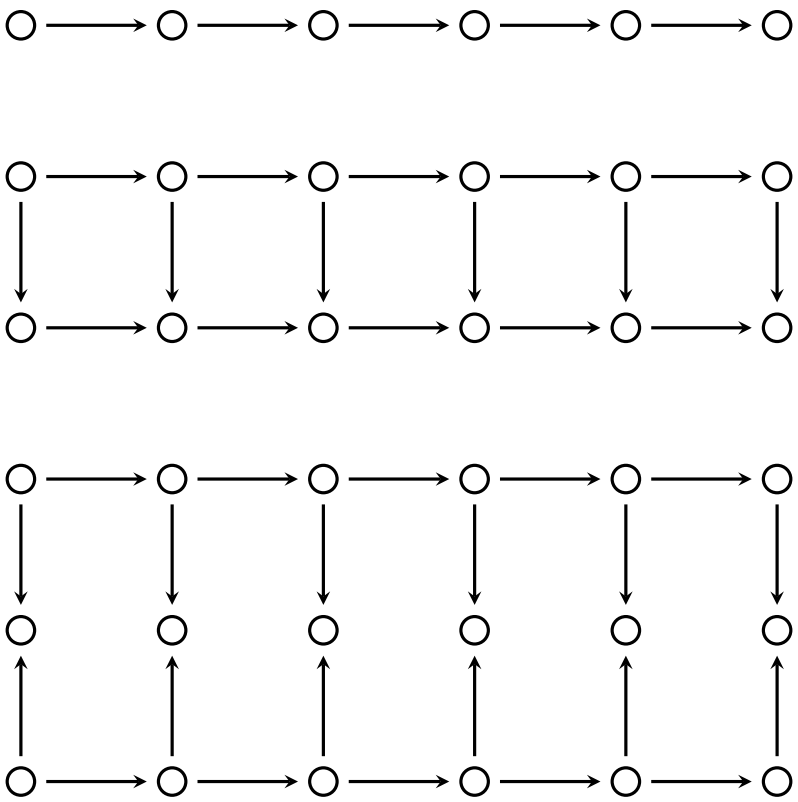


Compiling Swift Generics

Slava Pestov

November 10, 2025

Contents

Preface 11

I. Syntax 15

1. Introduction 17
1.1. Functions . . . 18
1.2. Nominal Types . . . 22
1.3. Protocols . . . 24
1.4. Associated Types . . . 28
1.5. Associated Requirements . . . 32
1.6. Related Work . . . 35

2. Compilation Model 37
2.1. Name Lookup . . . 41
2.2. Delayed Parsing . . . 45
2.3. Request Evaluator . . . 47
2.4. Incremental Builds . . . 51
2.5. Module System . . . 55
2.6. Source Code Reference . . . 57

3. Types 65
3.1. Fundamental Types . . . 70
3.2. More Types . . . 73
3.3. Special Types . . . 79
3.4. Source Code Reference . . . 83

4. Declarations 93
4.1. Generic Parameters . . . 97
4.2. Requirements . . . 99
4.3. Protocols . . . 103
4.4. Functions . . . 107
4.5. Storage . . . 114
4.6. Source Code Reference . . . 116

II. Semantics 127

5. Generic Signatures 129
5.1. Requirement Signatures . . . 132
5.2. Derived Requirements . . . 134
5.3. Valid Type Parameters . . . 140
5.4. Bound Type Parameters . . . 149
5.5. Reduced Type Parameters . . . 153
5.6. Generic Signature Queries . . . 159
5.7. Source Code Reference . . . 166

6. Substitution Maps 173
6.1. Nominal Types . . . 179
6.2. Nested Nominal Types . . . 182
6.3. Composition . . . 186
6.4. Subclassing . . . 190
6.5. SIL Type Lowering . . . 195
6.6. Source Code Reference . . . 207

7. Conformances 213
7.1. Conformance Lookup . . . 215
7.2. Conformance Substitution . . . 218
7.3. Type Witnesses . . . 222
7.4. Abstract Conformances . . . 227
7.5. Associated Conformances . . . 231
7.6. Associated Type Inference . . . 236
7.7. Source Code Reference . . . 248

8. Archetypes 255
8.1. Local Requirements . . . 259
8.2. Primary Archetypes . . . 261
8.3. The Type Parameter Graph . . . 263
8.4. The Archetype Builder . . . 269
8.5. Source Code Reference . . . 274

9. Type Resolution 277
9.1. Identifier Type Representations . . . 279
9.2. Member Type Representations . . . 285
9.3. Generic Arguments . . . 295
9.4. Unbound Generic Types . . . 304
9.5. Source Code Reference . . . 306

III. Subtleties 317

10. Extensions 319
10.1. Extension Binding . . . 322
10.2. Direct Lookup . . . 326
10.3. Constrained Extensions . . . 330
10.4. Conditional Conformances . . . 333
10.5. Source Code Reference . . . 343

11. Building Generic Signatures 347
11.1. Requirement Inference . . . 354
11.2. Decomposition and Desugaring . . . 359
11.3. Well-Formed Requirements . . . 365
11.4. Requirement Minimization . . . 377
11.5. Source Code Reference . . . 392

12. Conformance Paths 399
12.1. Validity and Existence . . . 404
12.2. The Conformance Path Graph . . . 408
12.3. Recursive Conformances . . . 414
12.4. The Halting Problem . . . 423
12.5. Source Code Reference . . . 435

13. Opaque Result Types 437
13.1. Opaque Archetypes . . . 442
13.2. Opaque Type Witnesses . . . 454
13.3. Runtime Representation . . . 459
13.4. Source Code Reference . . . 465

IV. The Requirement Machine 471

15. Basic Operation 473
15.1. Protocol Components . . . 480
15.2. Debugging Flags . . . 488
15.3. Source Code Reference . . . 489

16. Monoids 495
16.1. Finitely-Presented Monoids . . . 498
16.2. Equivalence of Terms . . . 503
16.3. A Swift Connection . . . 512
16.4. The Word Problem . . . 519

16.5. The Normal Form Algorithm 523

17. Symbols, Terms, and Rules 533
17.1. Correctness 536
17.2. Symbols 549
17.3. Terms 555
17.4. Rules 560
17.5. Protocol Type Aliases 566
17.6. The Normal Form Algorithm 571
17.7. Source Code Reference 576

18. Completion 585
18.1. Rule Simplification 597
18.2. Associated Types 601
18.3. More Critical Pairs 610
18.4. Tietze Transformations 618
18.5. Recursive Conformances 620
18.6. Source Code Reference 637

A. Mathematical Conventions 643

B. Derived Requirements 645

C. Substitution Algebra 647

Bibliography 649

Index 663

Command Line Flags 683

List of Algorithms

4A. Compute closure captures . . . 111
4B. Compute lowered closure captures . . . 112

5A. Generic parameter order . . . 155
5B. Protocol order . . . 155
5C. Associated type order . . . 156
5D. Type parameter order . . . 157
5E. Compute reduced type . . . 162

6A. Substitute type . . . 177
6B. Get superclass type . . . 192
6C. Get superclass type for declaration . . . 194
6D. Decide if type is address-only . . . 198
6E. Lower type with abstraction pattern . . . 205

7A. Global conformance lookup . . . 215
7B. Substitute dependent member type . . . 229
7C. Associated type inference . . . 240

8A. Expand conformance requirement . . . 270
8B. Merge potential archetypes . . . 271
8C. Resolve potential archetype . . . 271
8D. Historical `ArchetypeBuilder` algorithm . . . 272

9A. Check requirement . . . 301
9B. Check substitution map . . . 303

10A. Bind extensions . . . 323

11A. Decompose conformance requirement . . . 360
11B. Desugar same-type requirement . . . 364
11C. Desugar requirement . . . 364
11D. Requirement order . . . 392

12A. Local conformance lookup . . . 403
12B. Simplify conformance path . . . 404

12C. Conformance path order 411
12D. Find conformance path 412
12E. Evaluate tag system 426

13A. Apply substitution map to opaque archetype 446
13B. Map type parameter into opaque generic environment 450
13C. Weighted type parameter order 453
13D. Resolve opaque archetype 459
13E. Specialize opaque archetypes within a type 463
13F. Specialize opaque archetypes within a conformance 463
13G. Specialize opaque archetypes within a substitution map 464

14A. Apply substitution map to existential archetype 469

15A. Tarjan's algorithm 486
15B. Import rules from protocol components 487

16A. Normal form algorithm 524
16B. Shortlex order 527

17A. Protocol reduction order 550
17B. Associated type reduction order 551
17C. Build concrete type symbol 553
17D. Concrete type reduction order 554
17E. Concrete conformance reduction order 554
17F. Symbol reduction order 555
17G. Build term for explicit requirement 556
17H. Build term for associated requirement 557
17I. Weighted shortlex order 559
17J. Build rule from explicit requirement 560
17K. Build rule from associated requirement 561
17L. Build rule for protocol type alias 566
17M. Build protocol rules 570
17N. Lookup in rule trie 573
17O. Normal form algorithm using rule trie 573
17P. Insert rule in rule trie 574
17Q. Record rule 574

18A. Construct critical pair 591
18B. Resolve critical pair 591
18C. Overlap lookup in rule trie 594
18D. Find overlapping rules 595

18E. Knuth-Bendix completion procedure 596
18F. Left-simplify rules 598
18G. Right-simplify rules 599

19A. Simplify substitution terms 639

Preface

This is a book about the implementation of generic programming—also known as parametric polymorphism—in the Swift compiler. You won't learn how to *write* generic code in Swift here; the best reference for that is, of course, the official language guide [1]. This book is intended mainly for Swift compiler developers who interact with the generics implementation, other language designers who want to understand how Swift evolved, Swift programmers curious to peek under the hood, and finally, mathematicians interested in a practical application of string rewriting and the Knuth-Bendix completion procedure.

From the compiler developer's point of view, the *user* is the developer writing the code being compiled. The declarations, types, statements, and expressions that comprise the user's program become *data structures* the compiler must then analyze and manipulate. I assume some basic familiarity with these concepts, and compiler construction in general. For background reading, see [2, 3, 4, 5].

This book is divided into five parts. Part I gives a high-level overview of the Swift compiler architecture, and describes how types and declarations, and specifically, generic types and declarations, are modeled by the compiler.

- Chapter 1 summarizes every key concept in the generics implementation with a series of worked examples, and surveys capabilities for generic programming found in other programming languages.
- Chapter 2 covers Swift's compilation model and module system as well as the *request evaluator*, which adds an element of lazy evaluation to the typical "compilation pipeline" of parsing, type checking and code generation.
- Chapter 3 describes how the compiler models the *types* of values declared by the source program. Types form a miniature language of their own, and we often find ourselves taking them apart and re-assembling them in new ways. Generic parameter types, dependent member types, and generic nominal types are the three fundamental kinds; others are also summarized.
- Chapter 4 is about *declarations*, the building blocks of Swift code. Functions, structs, and protocols are examples of declarations. Various kinds of declarations can be *generic*. There is a common syntax for declaring generic parameters and stating *requirements*. Protocols can declare associated types and impose *associated requirements* on their associated types in a similar manner.

Part II focuses on the core *semantic* objects in the generics implementation. To grasp the mathematical asides, it helps to have had some practice working with definitions and proofs, at the level of an introductory course in calculus, linear algebra, or combinatorics. A summary of basic math appears in Appendix A.

- Chapter 5 defines the *generic signature*, which collects the generic parameters and explicit requirements of a generic declaration. The explicit requirements of a generate signature generate a set of *derived requirements* and *valid type parameters*, which explains how we type check code *inside* a generic declaration. This formalism is realized in the implementation via *generic signature queries*.
- Chapter 6 defines the *substitution map*, a mapping from generic parameter types to replacement types. The *type substitution algebra* will explain the operations of type substitution and substitution map composition. This explains how we type check a *reference* to (sometimes called a *specialization* of) a generic declaration.
- Chapter 7 defines the *conformance*, a description of how a concrete type fulfills the requirements of a protocol, in particular its associated types. In the type substitution algebra, conformances are to protocols what substitution maps are to generic signatures.
- Chapter 8 defines *archetypes* and *generic environments*, two abstractions used throughout the compiler. Also describes the *type parameter graph* that gives us an intuitive visualization of a generic signature.
- Chapter 9 describes *type resolution*, which uses name lookup and substitution to resolve syntactic representations to semantic types. Checking if a substitution map satisfies the requirements of its generic signature links our two formalisms.

Part III covers some additional language features and compiler internals, while further developing the derived requirements formalism and type substitution:

- Chapter 10 discusses extension declarations, which add members and conformances to existing types. Extensions can also declare *conditional conformances*, which have some interesting behaviors.
- Chapter 11 explains how we build a generic signature from syntax written in source, and gives a formal description of *requirement minimization*. This chapter also shows how invalid requirements are diagnosed, and defines a *well-formed generic signature* as one that passes these checks.
- Chapter 12 shows that *conformance paths* give us a way to evaluate expressions in the type substitution algebra, which completes the formalism. The concept of a *recursive conformance* is explored, and finally, the type substitution algebra is shown to be Turing-complete.

- Chapter 13 describes *opaque result types*, another kind of generic abstraction that allows a function to hide its concrete return type from its caller, by specifying the generic requirements this return type satisfies instead. This capability is built in terms of generic signatures, substitution maps, and archetypes.

- Chapter 14 is unfinished. It will describe existential types.

Part IV describes the Requirement Machine, a *decision procedure* for the derived requirements of Chapter 5. The original contribution here is that generic signature queries and requirement minimization are problems in the theory of *string rewriting*:

- Chapter 15 gives a high level overview of how both generic signature queries and requirement minimization recursively build a *requirement machine* for a generic signature from the requirement machines of its *protocol components*.

- Chapter 16 introduces *finitely-presented monoids* and the *word problem*, and then presents the theoretical result that a finitely-presented monoid can be encoded as a generic signature, such that word problems become generic signature queries. Therefore, generic signature queries are *at least* as hard as the word problem; that is, undecidable in the general case.

- Chapter 17 goes in the other direction and shows that a generic signature can be encoded in the form of a finitely-presented monoid, such that generic signature queries become word problems. Thus, derived requirements are *at most* as hard as the word problem—and the word problem can be solved in many cases using known techniques. This is the heart of our decision procedure.

- Chapter 18 describes the Knuth-Bendix algorithm, which attempts to solve the word problem by constructing a *convergent rewriting system*. Fundamental generic signature queries can then be answered via the *normal form algorithm*. This is the brain of our decision procedure.

- Chapter 19 is unfinished. It will describe the construction of a *property map* from a convergent rewriting system; the property map answers trickier generic signature queries.

- Chapter 20 is unfinished. It will present the algorithm for rewrite rule minimization, which is the final step in building a new generic signature.

The Swift compiler is written in C++. To avoid incidental complexity, concepts are described without direct reference to the source code. Instead, some chapters end with a **Source Code Reference** section, structured like an API reference. You can skip this material if you're only interested in the theory. No knowledge of C++ is assumed outside of these sections.

Occasional historical asides talk about how things came to be. Starting with Swift 2.2, the design of the Swift language has been guided by the Swift evolution process, where language changes are pitched, debated, and formalized in the open [6]. I will cite Swift evolution proposals when describing various language features. You will find a lot of other interesting material in the bibliography as well, not just evolution proposals.

This book does not say much about the runtime side of the separate compilation of generics, except for a brief overview of the model in relation to the type checker in Chapter 1. To learn more, I recommend watching a pair of LLVM Developer's Conference talks: [7] which gives a summary of the whole design, and [8] which describes some recent optimizations.

Also, while most of the material should be current as of Swift 6.2, some recent language extensions are not covered. These features are mostly additive and can be understood by reading the evolution proposals:

1. Parameter packs (or variadic generics), introduced in Swift 5.9 ([9, 10, 11]).
2. Noncopyable types, introduced in Swift 6 ([12, 13]).
3. Nonescapable types, introduced in Swift 6.2 ([14]).
4. Integer generic parameters, introduced in Swift 6.2 ([15]).

Source Code

The TEX code for this book lives in the Swift source repository:

`https://github.com/swiftlang/swift/tree/main/docs/Generics`

A periodically-updated typeset PDF is available from the Swift website:

`https://download.swift.org/docs/assets/generics.pdf`

Acknowledgments

I'd like to thank everyone who read earlier versions of the text, pointed out typos, and asked clarifying questions. Also, the Swift generics system itself is the result of over a decade of collaborative effort by countless people. This includes compiler developers, Swift evolution proposal authors, members of the evolution community, and all the users who reported bugs. This book attempts to give an overview of the sum total of all these contributions.

Part I.

Syntax

1. Introduction

Swift's generics implementation is best understood by first considering various design constraints faced by the compiler:

1. Generic functions should be independently type checked, without knowledge of all possible generic arguments that they are invoked with.
2. Shared libraries that export generic types and functions should be able to evolve resiliently without requiring recompilation of clients.
3. Generic struct or enum types should store their fields inline without boxing, so their layout needs to be defined abstractly in terms of their generic argument types.
4. This flexibility should only impose runtime overhead when absolutely necessary, such as when calling across module boundaries, or when complete type information is not available at compile time.

The high-level design can be summarized as follows:

1. The interface between a generic declaration and its caller is given by a list of **generic parameter types** and **requirements**. Inside the generic declaration itself, the requirements impose behaviors on its generic parameters. In turn, the caller supplies a list of generic arguments, which must satisfy the requirements.
2. The calling convention of a generic function requires the caller to pass **runtime type metadata** for each generic argument type. A type metadata record describes how to abstractly manipulate values of the represented type without compile-time knowledge of its concrete layout.
3. The runtime type metadata for a generic struct or enum encodes the layout of type. This layout information is computed recursively from the type metadata of its generic arguments. Runtime type metadata is constructed lazily when first requested, and then cached.
4. If a generic function's definition is visible from its call site, the optimizer can generate a **specialization** of the generic function for the given generic argument types. When this is possible, specialization eliminates the overhead of abstract value manipulation and runtime type metadata.

We're going to think of the compiler as *a library for modeling the concepts of the target language*. The Swift generics implementation defines four fundamental kinds of semantic objects: *generic signatures*, *substitution maps*, *requirement signatures*, and *conformances*. As we will see, they are understood as much by their inherent structure, as their relationships with each other. Subsequent chapters will dive into all the details, but first, we're going to present a series of worked examples.

1.1. Functions

Consider these two rather contrived function declarations:

```
func identity(_ x: Int) -> Int { return x }
func identity(_ x: String) -> String { return x }
```

Both have the same exact definition apart from the parameter and return type, and indeed you can write the same function for any concrete type. Our aesthetic sense might lead us to replace both with a single generic function:

```
func identity<T>(_ x: T) -> T { return x }
```

While this function declaration is rather simple, it illustrates some important concepts and allows us to introduce terminology. We'll see a full description of the compilation pipeline in the next chapter, but for now, let's consider a simplified view where we begin with parsing, then type checking, and finally code generation.

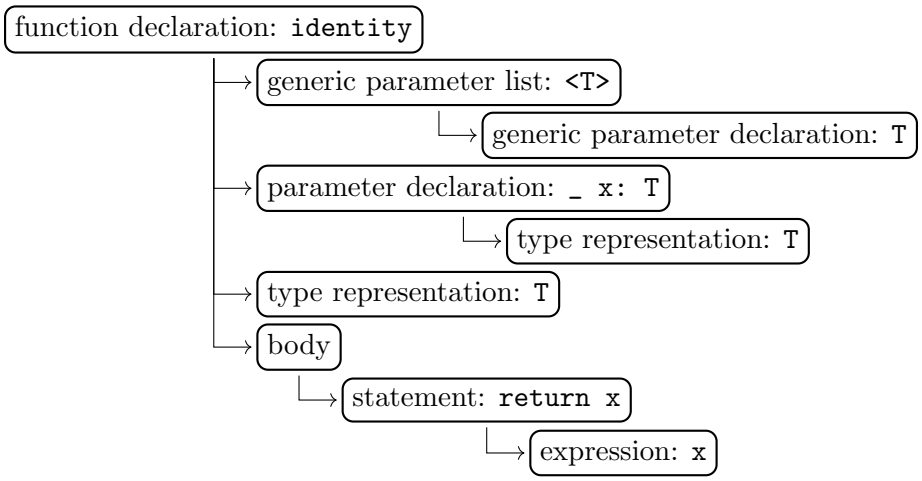
Parsing. Figure 1.1 shows the abstract syntax tree produced by the parser before type checking. The key elements:

1. The *generic parameter list* `<T>` introduces a single *generic parameter declaration* named `T`. This declaration declares the generic parameter type `T`, which is scoped to the entire source range of this function.

2. The parameter declaration "`_ x: T`" and the return type of `identity` both contain the *type representation* `T`. A type representation is a syntactic form that denotes a reference to an existing type. The parser does not perform name lookup, so the type representation `T` is just an identifier; it is not immediately associated with the generic parameter declaration `T`.

3. The function body consists of a `return` statement with the expression `x`. Again, the parser does not perform name lookup, so this expression is just the identifier `x`, not associated with the parameter declaration "`_ x: T`".

Figure 1.1.: The abstract syntax tree for `identity(_:)`



Type checking. The type checker translates these syntactic forms into higher-level semantic objects:

1. The generic parameter types declared by the generic parameter list are collected in the function's *generic signature*. In our case, the generic signature has the printed representation `<T>`. This is the simplest generic signature, apart from the empty generic signature of a non-generic declaration. We'll see more interesting generic signatures soon.

2. The *type resolution* procedure resolves the parameter type representation `T` to the generic parameter type `T`, by performing a name lookup.

3. The function's return type representation also resolves to the generic parameter type `T`.

The generic signature, together with the resolved parameter and return type, are packaged up into a *generic function type*, which becomes the *interface type* of the function declaration. We denote this generic function type as follows:

```
<T> (T) -> T
```

The name "`T`" is of no semantic consequence beyond name lookup. We will learn that every generic parameter type is uniquely identified within its lexical scope by its *depth* and *index*. The *canonical type* of the above generic function type is denoted as follows, where we replace `T` with the *canonical generic parameter type* `τ_0_0`:

```
<τ_0_0> (τ_0_0) -> τ_0_0
```

The interface type completely describes the type checking behavior of a reference to a declaration. Having computed the interface type of the function, the type checker moves on to the function's body. The type of the return statement's expression must match the return type of the function declaration. When we type check the return expression `x`, we use name lookup to find the parameter declaration "`_ x: T`". The interface type of this parameter declaration is the generic parameter type `T`, which is understood relative to the function's generic signature. The type assigned to the expression, however, is the *archetype* corresponding to `T`, denoted $[\![\texttt{T}]\!]$. We will learn later that an archetype is a self-describing form of a type parameter which behaves like a concrete type.

Code generation. We've now successfully type checked our function declaration. The next step in to actually lower the function to executable code. Recall the two concrete implementations that we folded together to get our generic function:

```
func identity(_ x: Int) -> Int { return x }
func identity(_ x: String) -> String { return x }
```

The calling conventions of these functions differ significantly:

1. The first function receives and returns the `Int` value in a machine register. The `Int` type is *trivial*, meaning that its values can be copied and moved without doing anything special. (C++ also calls this a "POD" type.)

2. The second function is trickier. A `String` is stored as a 16-byte value in memory, and contains a pointer to a reference-counted buffer. When manipulating values of a non-trivial type like `String`, memory ownership comes into play.

The default ownership rules for a Swift function are that the caller retains ownership over the parameter values passed into the callee, while the callee transfers ownership of the return value to the caller. Thus the generic implementation of `identity(_:)` must create a logical copy of `x` and then move this copy back to the caller, and do this in a manner that abstracts over all possible concrete types.

The calling convention for a generic function passes *runtime type metadata* for each generic parameter in the function's generic signature. Runtime type metadata describes the size and alignment of a concrete type, and provides implementations of the *move*, *copy* and *destroy* operations.

The move and copy operations for trivial type copy bytes, while destroy does nothing. With a reference type, the value is a reference-counted pointer, so copy and destroy operations update the reference count while a move leaves the reference count unchanged. The value operations for structs and enums are defined recursively from their members. Finally, weak references and existential types also have special value operations.

For copyable types, a move is semantically equivalent to a copy followed by a destroy, only more efficient. Traditionally, all types in the language were copyable. Swift 5.9 introduced *noncopyable types* [12], and Swift 6 extended generics to work with noncopyable types [13]. We will not discuss noncopyable types in this book.

More details

- Types: Chapter 3
- Generic parameter lists: Section 4.1
- Function declarations: Section 4.4
- Archetypes: Chapter 8
- Type resolution: Chapter 9

Substitution maps. Let us now turn our attention to the callers of generic functions. A *call expression* brings together a *callee* and a list of argument expressions. A callee is some expression with a function type; this can be a reference to a named function declaration, a reference to a type (which is sugar for invoking a constructor), a reference to a parameter or local variable of function type, or most generally, the result of another call expression. In our example, we can call the `identity(_:)` function by name:

```
let x = identity(3)
let y = identity("Hello, Swift")
```

In Swift, a call to a generic function does not specify the generic argument types in the syntax; instead, the type checker infers generic arguments by matching the function type of the callee against the types of the call argument expressions, as well as the expression's expected result type, if there is one. The inferred generic arguments are collected in a *substitution map*, which is a data structure that assigns a *replacement type* to each generic parameter type of the callee's generic signature.

The generic signature of `identity(_:)` has a single generic parameter type. Thus each of its substitution maps holds a single concrete type. We now introduce some notation. Here are two possible substitution maps, corresponding to the two calls shown above:

$$\Sigma_1 := \{\texttt{T} \mapsto \texttt{Int}\} \qquad \Sigma_2 := \{\texttt{T} \mapsto \texttt{String}\}$$

We can form the return type of a call by taking the function declaration's return type, and applying the corresponding substitution map. We denote this application using the "$\otimes$" operator. For now, this simply extracts the concrete type stored therein:

$$\texttt{T} \otimes \Sigma_1 = \texttt{Int} \qquad \texttt{T} \otimes \Sigma_2 = \texttt{String}$$

Substitution maps play a role in code generation. When lowering a call to a generic function, the compiler generates code to construct the runtime type metadata for each replacement type in the call expression's substitution map. In our example, the types `Int` and `String` are *nominal types* defined in the standard library. They are non-generic and have a fixed layout, so their runtime type metadata is accessed by calling a function, exported by the standard library, that returns the address of a constant symbol.

More details

- Substitution maps: Chapter 6

Specialization. Reification of runtime type metadata and the subsequent indirect manipulation of values incurs a performance penalty. As an alternative, if the definition of a generic function is visible at the call site, the optimizer can generate a *specialization* of the generic function by cloning the definition and applying the substitution map to all types appearing in the function's body. Definitions of generic functions are always visible to the specializer within their defining module. Shared library developers can also opt-in to exporting the body of a function across module boundaries with the `@inlinable` attribute.

More details

- `@inlinable` attribute: Section 2.5

1.2. Nominal Types

For our next example we consider a simple generic struct declaration:

```
struct Pair<T> {
  let first: T
  let second: T

  init(first: T, second: T) {
    self.first = first
    self.second = second
  }
}
```

A struct declaration is an example of a *nominal type declaration*. A *generic nominal type*, like `Pair<Int>` or `Pair<String>` is a reference to a generic nominal type declaration together with a list of generic argument types.

The in-memory layout of a struct value is determined by the interface types of its stored properties. Our `Pair` struct declares two stored properties, `first` and `second`, both with interface type `T`. Thus the layout of a `Pair` depends on the layout of the generic parameter type `T`.

The generic nominal type `Pair<T>`, formed by taking the generic parameter type `T` as the argument, is called the *declared interface type* of `Pair`. The type `Pair<Int>` is a *specialized type* of the declared interface type, `Pair<T>`. We can obtain `Pair<Int>` from `Pair<T>` by applying a substitution map:

$$\texttt{Pair<T>} \otimes \{\texttt{T} \mapsto \texttt{Int}\} = \texttt{Pair<Int>}$$

The above substitution map is known as the *context substitution map* of `Pair<Int>`. Every specialized type has a context substitution map, and if we apply the context substitution map to its declared interface type, we will get back the specialized type. Now, suppose we declare a local variable of type `Pair<Int>`:

```
let twoIntegers: Pair<Int> = ...
```

The compiler must allocate storage on the stack for this value. We take the context substitution map, and apply it to the interface type of each stored property. Since `Pair` has two stored properties of type `T`, we get:

$$\texttt{T} \otimes \{\texttt{T} \mapsto \texttt{Int}\} = \texttt{Int}$$

So a `Pair<Int>` consists of two consecutive `Int`s, which gives `Pair<Int>` a total size of 16 bytes and alignment of 8 bytes. Since `Pair<Int>` is trivial, the stack allocation does not require any special cleanup once we exit its scope.

Now, we complete our local variable declaration by writing down an initial value expression which calls the constructor:

```
let twoIntegers: Pair<Int> = Pair(first: 1, second: 2)
```

While the value has the type `Pair<Int>` at the call site, inside the constructor the value being initialized is of type `Pair<T>`. We construct the runtime type metadata for `Pair<Int>` by calling the *metadata access function* for `Pair` with the runtime type metadata for `Int` as an argument. The metadata for `Pair<Int>` has two parts:

1. A common prefix present in all runtime type metadata, which includes the total size and alignment of a value, and implementations of the move, copy and destroy operations.
2. A private area specific to the declaration of `Pair` itself, which stores the runtime type metadata for `T`, followed by the *field offset vector*, storing the offset of each stored property of this specialization of `Pair<T>`.

The metadata access function for a generic type takes the metadata for each generic argument, and calculates the offset of each stored property, also obtaining the size and alignment of the entire value. The move, copy and destroy operations of the aggregate type delegate to the corresponding operations in the generic argument metadata. The constructor of `Pair` then uses the runtime type metadata for both `Pair<T>` and `T` to correctly initialize the aggregate value from its two constituent parts.

Structural types, such as function types, tuple types, and metatypes, are similar to generic nominal types in that we call a metadata access function to obtain runtime type metadata for them, but this time, the metadata access function is part of the Swift runtime. For example, to construct metadata for the tuple type `(Int, Pair<String>)`, we first call the metadata access function for `Pair` to get `Pair<String>`, then call an entry point in the runtime to obtain `(Int, Pair<String>)`.

More details

- Declarations: Chapter 4
- Context substitution map: Section 6.1
- Structural types: Section 3.2

1.3. Protocols

Our `identity(_:)` and `Pair` declarations abstract over arbitrary concrete types, but this in turn limits their generic parameter `T` to the common capabilities shared by all types—the move, copy and destroy operations. By stating *generic requirements*, a generic declaration can impose various restrictions on the concrete types used as generic arguments, which in turn endows its generic parameter types with new capabilities provided by those concrete types.

A *protocol* specifies some additional capabilities that a concrete type may have. A generic declaration can state a *conformance requirement* on a generic parameter type, which the caller must satisfy with a concrete type that *conforms* to this protocol:

```
protocol Shape {
  func draw()
}

func drawShapes<S: Shape>(_ shapes: Array<S>) {
  for shape in shapes {
    shape.draw()
  }
}
```

The `drawShapes(_:)` function takes an array of values, all of the same type, which must conform to `Shape`. So far we only encountered the generic signature `<T>`. More generally, a generic signature lists one or more generic parameter types, together with their requirements. The generic signature of `drawShapes(_:)` has the single requirement [`S: Shape`]. We will use the following notation for generic signatures with requirements:

```
<S where S: Shape>
```

The interface type of `drawShapes(_:)` incorporates this generic signature into a generic function type:

```
<S where S: Shape> (Array<S>) -> ()
```

We can change `drawShapes(_:)` function to state the conformance requirement with a trailing `where` clause, or we can avoid naming the generic parameter `S` by using an *opaque parameter type* instead:

```
func drawShapes<S>(_ shapes: Array<S>) where S: Shape
func drawShapes(_ shapes: Array<some Shape>)
```

All three forms of `drawShapes(_:)` are ultimately equivalent, because they define the same generic signature, up to the choice of generic parameter name. In general, when there is more than one way to spell the same underlying language construct due to syntax sugar, the semantic objects "desugar" these differences into the same uniform representation.

More details

- Protocols: Section 4.3
- Requirements: Section 4.2
- Generic signatures: Chapter 5

Qualified lookup. Once we have a generic signature, we can type check the body of `drawShapes(_:)`. The `for` loop introduces a local variable "`shape`" of type ⟦`S`⟧ (we re-iterate that the generic parameter type `S` is represented as the archetype ⟦`S`⟧ inside a function body, but the distinction doesn't matter right now). This variable is referenced inside the `for` loop by the *member reference expression* "`shape.draw`":

```
for shape in shapes {
  shape.draw()
}
```

Our generic signature has the conformance requirement [`S: Shape`], so the caller must provide a replacement type for `S` conforming to `Shape`. We'll return to the caller's side of the equation shortly, but inside the callee, the requirement also tells us that the archetype ⟦`S`⟧ conforms to `Shape`. To resolve the member reference "`shape.draw`", the type checker performs a *qualified lookup* of the identifier `draw` on the base type of ⟦`S`⟧, which is the type of "`shape`". A qualified lookup into an archetype visits each protocol the archetype conforms to, so we find and return the `draw()` method of the `Shape` protocol.

How do we lower the call expression "`shape.draw()`" to executable code? As well as the runtime type metadata for `S`, the calling convention for `drawShapes(_:)` has another parameter, corresponding to the conformance requirement [`S: Shape`]. This parameter is used to pass the *witness table* for the conformance. The layout of a witness table is determined by the protocol; a witness table for a conformance to `Shape` has a single entry, the implementation of the `draw()` method. So to call "`shape.draw()`", we load this function pointer from the witness table and call it, passing in the `shape`.

More details

- Name lookup: Section 2.1

Conformances. This `Circle` struct states a *conformance* to the `Shape` protocol:

```
struct Circle: Shape {
  let radius: Double
  func draw() {...}
}
```

The *conformance checker* ensures that the declaration of `Circle` contains a *witness* for the `draw()` method of `Shape`, and records this fact in a *normal conformance*. We denote this normal conformance by [`Circle: Shape`]. When we visit `Circle` in code generation, we emit its runtime type metadata, together with the witness table for the normal conformance [`Circle: Shape`]. This witness table contains a pointer to the implementation of `Circle.draw()`.

Now, let's call `drawShapes(_:)` with an array of circles and look at the substitution map for the call:

```
drawShapes([Circle(radius: 1), Circle(radius: 2)])
```

When the callee's generic signature has conformance requirements, the substitution map must store a conformance for each conformance requirement. This is the "proof" that the concrete replacement type actually conforms to the protocol, as required. We denote a substitution map with conformances as follows:

$$\{\texttt{S} \mapsto \texttt{Circle};\ [\texttt{S: Shape}] \mapsto [\texttt{Circle: Shape}]\}$$

To find the conformance, the type checker performs a *global conformance lookup* with the concrete type and protocol. We use this notation for global conformance lookup:

$$\langle \texttt{Shape} \rangle \otimes \texttt{Circle} = [\texttt{Circle: Shape}]$$

When generating code for the call to `drawShapes(_:)`, we visit each entry in the substitution map, emitting a reference to runtime type metadata for each replacement type, and a reference to the witness table for each conformance. In our case, we pass the runtime type metadata for `Circle` and witness table for [`Circle: Shape`].

More details

- Conformances: Chapter 7
- Conformance lookup: Section 7.1

Existential types. Note that `drawShapes(_:)` operates on a *homogeneous* array of shapes. While the array contains an arbitrary number of elements, `drawShapes(_:)` only receives a single runtime type metadata for `S`, and one witness table for the conformance requirement [`S: Shape`], which together describe the behavior of each element in the array. If we instead want a function taking a *heterogeneous* array of shapes, we can use an *existential type* as the element type of our array:

```
func drawShapes(_ shapes: Array<any Shape>) {
  for shape in shapes {
    shape.draw()
  }
}
```

This function uses the `Shape` protocol in a new way. The existential type `any Shape` is a container for a value of some concrete type, together with its runtime type metadata and a witness table describing the conformance. This container stores small values inline, otherwise it points to a heap-allocated box. Observe the difference between the type `Array<S>` from the previous variant of `drawShapes(_:)`, and `Array<any Shape>` here. Every element of the latter has its own runtime type metadata and witness table, so we can mix multiple types of shapes in one array. In the implementation, existential types are built on top of the core primitives of the generics system.

More details

- Existential types: Chapter 14

1.4. Associated Types

The standard library's `IteratorProtocol` declares an associated type. This allows us to abstract over iterators whose element type depends on the conformance:

```
protocol IteratorProtocol {
  associatedtype Element
  mutating func next() -> Element?
}
```

A conforming type must declare a member type named `Element`, and a `next()` method returning an optional value of this type. This member type, which can be a type alias or nominal type, is the *type witness* for the associated type `Element`.

Let's declare a `Nat` type conforming to `IteratorProtocol` with an `Element` type of `Int`, for generating an infinite stream of consecutive natural numbers:

```
struct Nat: IteratorProtocol {
  typealias Element = Int  // (can also be omitted in this case)
  var x = 0

  mutating func next() -> Int? {
    defer { x += 1 }
    return x
  }
}
```

We say that `Int` is the *type witness* for `[IteratorProtocol]Element` in the conformance `[Nat: IteratorProtocol]`. We can express this in our type substitution algebra, using the type witness *projection* operation on normal conformances:

$$\langle \texttt{IteratorProtocol} | \texttt{Element} \otimes [\texttt{Nat: IteratorProtocol}] = \texttt{Int}$$

(More accurately, the type witness here is the `Element` *type alias type*, whose canonical type is `Int`.) Finally, we can actually omit the declaration of the `Element` type alias above, in which case *associated type inference* will be able to deduce it for us.

More details

- Type witnesses: Section 7.3
- Associated type inference: Section 7.5

Dependent member types. This function reads a pair of elements from an iterator:

```
func readTwo<I: IteratorProtocol>(_ iter: inout I) -> Pair<I.Element> {
  return Pair(first: iter.next()!, second: iter.next()!)
}
```

The return type is the generic nominal type `Pair<I.Element>`, constructed from the declaration of `Pair` and the generic argument type `I.Element`. This generic argument type is a *dependent member type*, made from the base type `I` and a reference to the associated type declaration `[IteratorProtocol]Element`. This dependent member type represents the type witness in the conformance [`I: IteratorProtocol`].

Suppose we call `readTwo(_:)` with a value of type `Nat`:

```
var iter = Nat()
print(readTwo(&iter))
```

The substitution map for the call stores the replacement type `Nat` and the conformance of `Nat` to `IteratorProtocol`. Let's denote this substitution map by Σ:

$$\begin{aligned}\Sigma := \{&\texttt{I} \mapsto \texttt{Nat};\\ &[\texttt{I: IteratorProtocol}] \mapsto [\texttt{Nat: IteratorProtocol}]\}\end{aligned}$$

To get the return type of our call, we evaluate $\texttt{Pair<I.Element>} \otimes \Sigma$. Applying a substitution map to a generic nominal type recursively applies the substitution map to each generic argument, so it remains to define $\texttt{I.Element} \otimes \Sigma$. Since this dependent member type abstracts over the type witness in the conformance, it must be the case that $\texttt{I.Element} \otimes \Sigma = \texttt{Int}$. We will eventually derive the below equation that relates dependent member type substitution with type witness projection:

$$\begin{aligned}&\texttt{I.Element} \otimes \Sigma\\ &\quad= \langle \texttt{IteratorProtocol} | \texttt{Element} \otimes [\texttt{I: IteratorProtocol}] \otimes \Sigma\\ &\quad= \langle \texttt{IteratorProtocol} | \texttt{Element} \otimes [\texttt{Nat: IteratorProtocol}]\\ &\quad= \texttt{Int}\end{aligned}$$

Once we have the above, we can conclude that the return type of our call to `readTwo(_:)` is $\texttt{Pair<I.Element>} \otimes \Sigma = \texttt{Pair<Int>}$.

More details

- Dependent member type substitution: Section 7.4, Chapter 12

Type parameters. Generic parameter types and dependent member types are the two kinds of *type parameters*. The generic signature of `readTwo(_:)` defines two type parameters, `I` and `I.Element`.

As with generic parameter types, dependent member types map to archetypes in the body of a generic function. We can reveal a little more about the structure of archetypes now, and say that an archetype packages a type parameter together with a generic signature. While a type parameter is like a "name" which can only be understood in relation to some generic signature, an archetype inherently "knows" what requirements it is subject to.

More details

- Type parameters: Section 3.1
- Primary archetypes: Section 8.2

Bound and unbound. There are actually two kinds of dependent member types. A *bound* dependent member type references an associated type declaration. We will denote this by `I.[IteratorProtocol]Element`, despite this not being valid language syntax. An *unbound* dependent member type references an identifier, which we denote by the source language style `I.Element`. The two forms have distinct representations, but we will see they are equivalent in a strong sense. We will often prefer the unbound form as a notational convenience, but bound dependent member types will later become important in our study of type substitution.

More details

- Member type representations: Section 9.1
- Bound type parameters: Section 5.4

Code generation. Inside the body of `readTwo(_:)`, the call expression `iter.next()` has the type ⟦`I.Element`⟧`?`, which we force-unwrap using the "`!`" operator to yield a value of type ⟦`I.Element`⟧. To manipulate a value of this type abstractly, we need its runtime type metadata.

To recover the runtime type metadata for a dependent member type at run time, we consult the witness table for the conformance. This mirrors how at compile time, we substitute a dependent member type by projecting a type witness from the conformance.

A witness table for a conformance to `IteratorProtocol` consists of a pair of function pointers, the first to recover the runtime type metadata for `Element`, and the second for the implementation of `next()`. The witness table for our [`Nat: IteratorProtocol`] thus references the runtime type metadata for `Int` from the standard library.

Same-type requirements. To introduce another fundamental requirement kind, we compose `Pair` and `IteratorProtocol` in a new way, writing a function that takes two iterators and reads an element from each one:

```
func readTwoParallel<I, J>(_ i: I, _ j: J) -> Pair<I.Element>
    where I: IteratorProtocol, J: IteratorProtocol,
          I.Element == J.Element {
  return Pair(first: i.next()!, second: j.next()!)
}
```

The generic signature of our `readTwoParallel(_:)` states the *same-type requirement* [`I.Element == J.Element`]:

```
<I, J where I: IteratorProtocol, J: IteratorProtocol,
            I.Element == J.Element>
```

This generic signature defines four type parameters, `I`, `J`, `I.Element`, and `J.Element`, where the final two abstract over the same concrete type, forming an *equivalence class*. It is often convenient to refer to an entire equivalence class by a representative type parameter. To make this choice in a deterministic fashion, we *order* type parameters and say the smallest type parameter inside an equivalence class is a *reduced type*. In our example, `I.Element` is the reduced type of `J.Element`.

In the presence of same-type requirements, an archetype represents a reduced type parameter (and thus an entire equivalence class of type parameters). In the body of `readTwoParallel(_:)`, the expressions `i.next()!` and `j.next()!` both return the type ⟦`I.Element`⟧, and the call to the `Pair` constructor is made with this substitution map:

$$\{\texttt{T} \mapsto ⟦\texttt{I.Element}⟧\}$$

More details

- Reduced type parameters: Section 5.5
- The type parameter graph: Section 8.3

Checking generic arguments. When type checking a call to `readTwoParallel(_:)`, we must ensure the same-type requirement is satisfied. Suppose we define two new iterator types, `BabyNames` and `CatNames`, both witnessing the `Element` associated type with `String`, and we call `readTwoParallel(_:)` with these types:

```
var i = BabyNames()
var j = CatNames()
print(readTwoParallel(&i, &j))
```

We're making the call with this substitution map:

$$\begin{aligned}\Sigma := \{&\texttt{I} \mapsto \texttt{BabyNames},\\ &\texttt{J} \mapsto \texttt{CatNames};\\ &[\texttt{I: IteratorProtocol}] \mapsto [\texttt{BabyNames: IteratorProtocol}]\\ &[\texttt{J: IteratorProtocol}] \mapsto [\texttt{CatNames: IteratorProtocol}]\}\end{aligned}$$

The type checker applies Σ to both sides of the same-type requirement, which gives us $\texttt{I.Element} \otimes \Sigma = \texttt{String}$ and $\texttt{J.Element} \otimes \Sigma = \texttt{String}$. The result is `String` in both cases, so we get the following *substituted requirement*:

$$[\texttt{I.Element == J.Element}] \otimes \Sigma = [\texttt{String == String}]$$

The substituted requirement is satisfied so the code is well-typed, and we see that the call returns `Pair<String>`. Suppose instead we had substituted `I` with `Nat`:

```
var i = Nat()
var j = CatNames()
print(readTwoParallel(&i, &j))  // error
```

In this case, we get the substituted requirement [`Int == String`] which is unsatisfied, so the call is malformed and the type checker must diagnose an error.

More details

- Checking generic arguments: Section 9.3

1.5. Associated Requirements

Protocols can impose *associated requirements* on their associated types. A conforming type must then satisfy these requirements. This capability is what gives Swift generics much of their distinctive flavor. Perhaps the simplest example is the `Sequence` protocol in the standard library, which abstracts over types that can produce a fresh iterator on demand:

```
protocol Sequence {
  associatedtype Element
  associatedtype Iterator: IteratorProtocol
    where Element == Iterator.Element

  func makeIterator() -> Iterator
}
```

The associated requirements of a protocol are recorded in the protocol's *requirement signature*. The `Sequence` protocol states two associated requirements:

- The conformance requirement [`Self.Iterator: IteratorProtocol`], stated using the sugared form, as a constraint type in the inheritance clause of the `Iterator` associated type.
- The same-type requirement [`Self.Element == Self.Iterator.Element`], which we state in a trailing `where` clause attached to the associated type.

Associated requirements are like the requirements in a generic signature, except they are rooted in the protocol `Self` type. Once again, there are multiple equivalent syntactic forms for stating them. For example, we could write out the conformance requirement explicitly, and the above `where` clause could be attached to the protocol itself for the same semantic effect.

Now consider this generic signature, and call it G:

```
<T, U where T: Sequence, U: Sequence,
            T.Element == U.Element>
```

We can informally describe G by classifying its type parameters into equivalence classes:

- `T`, which conforms to `Sequence`.
- `U`, which conforms to `Sequence`.
- `T.Element`, `U.Element`, `T.Iterator.Element`, and `U.Iterator.Element`, which are all in one equivalence class.
- `T.Iterator`, which conforms to `IteratorProtocol`.
- `U.Iterator`, which conforms to `IteratorProtocol`.

To make this kind of analysis precise, we will develop a theory of *derived requirements* to reason about requirements that are not explicitly stated, but are logical consequences of other requirements. The following are some interesting derived requirements of G:

[`T.Iterator: IteratorProtocol`]

[`U.Iterator: IteratorProtocol`]

[`T.Element == T.Iterator.Element`]

[`U.Element == U.Iterator.Element`]

[`T.Iterator.Element == U.Iterator.Element`]

At this point, it's worth clarifying that type parameters have a recursive structure; the base type of `U.Iterator.Element` is another dependent member type, `U.Iterator`. Note that not every such combination is meaningful. The theory of derived requirements will also describe the subset of *valid type parameters* of a generic signature.

Conformances. A normal conformance also stores an *associated conformance* for each associated conformance requirement of its protocol. The *associated conformance projection* operation recovers this conformance. At run time, a witness table for a conformance has a corresponding entry for each associated conformance. For example, a witness table for a conformance to `Sequence` has four entries:

1. A metadata access function for `Element`.
2. A metadata access function for `Iterator`.
3. A *witness table access function* for [`Self.Iterator: IteratorProtocol`].
4. A function pointer for the implementation of `makeIterator()`.

A *protocol inheritance* relation is represented by an associated conformance requirement with a subject type of `Self`. For example, the standard library `Collection` protocol inherits from `Sequence`, so the associated conformance requirement [`Self: Sequence`] appears in the requirement signature of `Collection`:

```
protocol Collection: Sequence {...}
```

Starting from a conformance [`Array<Int>: Collection`], we can get at the conformance to `Sequence` via associated conformance projection:

$$\langle \texttt{Self: Sequence} \rangle \otimes [\texttt{Array<Int>: Collection}] = [\texttt{Array<Int>: Sequence}]$$

This also means that at run time, we can recover a witness table for a conformance to `Sequence`, from a witness table for a conformance to `Collection`. We will take a closer look at the other associated requirements of the `Collection` protocol in due time. This will lead us into the topic of *recursive* associated conformance requirements, which as we show, enable the type substitution algebra to encode any computable function.

More details

- Requirement signatures: Section 5.1
- Derived requirements: Section 5.2
- Valid type parameters: Section 5.3
- Associated conformances: Section 7.5
- Recursive conformances: Section 12.3

1.6. Related Work

A calling convention based around a runtime representation of types was explored in a 1996 paper [16]. Swift protocols are similar in spirit to Haskell type classes, described in [17], and subsequently [18] and [19]. Swift witness tables follow the "dictionary passing" implementation strategy for type classes.

Associated types were introduced in [20]. In Swift parlance, the original formulation loosely corresponds to a world where each associated type is witnessed by a distinct nested nominal type. There were no associated requirements in the model. The original motivating example in their paper translates to Swift as follows:

```
protocol ArrayElem {
  associatedtype Array
  func index(_: Array, _: Int) -> Self
}
```

A subsequent work introduced *associated type synonyms* [21]. In Swift, this translates to an associated type witnessed by a type alias, again without associated requirements. The motivating example looks like this in Swift:

```
protocol Collects {
  associatedtype Elem
  static var empty: Self { get }
  func insert(_: Elem) -> Self
  func toList() -> [Elem]
}
```

Other relevant papers from the Haskell world include [22] and [23].

C++. While C++ templates are synonymous with "generic programming" to many programmers, C++ is somewhat unusual compared to most languages with parametric polymorphism, because templates are fundamentally syntactic in nature. Compiling a template declaration only does some minimal amount of semantic analysis, with most type checking deferred until *after* template expansion. There is no formal notion of requirements on template parameters, so whether template expansion succeeds or fails at a given expansion point entirely depends on how the template's body uses the specified template arguments.

This unusual flexibility enables some advanced metaprogramming techniques [24]. On the other hand, because a template declaration's body must be visible at each expansion point, heavy template use is fundamentally at odds with separate compilation. Library authors can find themselves implementing a large amount of logic in header files, and error messages on template expansion failure can be difficult to understand.

Swift's "value semantics" are an evolution of a certain philosophy that originated in the C++ community [25]. Swift generics also draw inspiration from "C++0x concepts," a proposal to add checked requirements to C++ templates, based around type classes and associated types ([26, 27]). Concepts could even state associated requirements, but the full ramifications of this were perhaps not yet apparent to the authors:

> *"Concepts often include requirements on associated types. For example, a container's associated iterator* `A` *would be required to model the* `Iterator` *concept. This form of concept composition is slightly different from refinement but close enough that we do not wish to clutter our presentation [. . .]"*

Rust. Rust generics are separately type checked, but Rust does not define a calling convention for unspecialized generic code, so there is no separate compilation. Instead, the implementation of a generic function is *specialized*, or *monomorphized*, for every unique set of generic arguments.

Rust's *traits* are similar to Swift's protocols; traits can declare associated types and associated conformance requirements. Rust generics also allow some kinds of abstraction not supported by Swift, such as lifetime variables, generic associated types [28], and const generics [29]. On the flip side, Rust does not allow same-type requirements to be stated in full generality [30]. Instead, trait bounds can constrain associated types with a syntactic form resembling Swift's parameterized protocol types (Section 4.3), but we will show in Example 18.10 that Swift's same-type requirements are more general.

Rust's "`where` clause elaboration" is more limited than Swift's derived requirements formalism, and associated requirements sometimes need to be re-stated in the `where` clause of a generic declaration [31]. An early attempt at formalizing Rust traits appears in a PhD dissertation from 2015 [32]. A more recent effort is "Chalk," an implementation of a Prolog-like solver based on Horn clauses [33].

Java. Traditional Java generics are implemented by erasing generic argument types at compile time, so that values of generic parameter type are always represented by object pointers, and primitive value types must be boxed first. This avoids the complexity of dependent layout, at the cost of more runtime checks and heap allocations. Java generics also support *variance*, a form of subtyping between different instantiations of the same generic type, defined in terms of the subtype relation on their corresponding generic arguments [34]. An effort to extend the Java virtual machine with support for user-defined value types and reified generics is currently underway [35].

Hylo. Hylo is a research language with a focus on mutable value semantics [36]. Hylo's generic programming capabilities are similar to Swift and Rust. The Hylo compiler implementation incorporates some ideas from this book, using the theory of string rewriting to reason about generic requirements [37].

2. Compilation Model

Most developers interact with the Swift compiler through the Xcode build system or the Swift package manager, but for simplicity's sake we're just going to consider direct invocation of `swiftc` from the command line. The `swiftc` command runs the *Swift driver*, which invokes the *Swift frontend* program to actually compile each source file; then, depending on the usage mode, the driver runs additional tools, such as the linker, to produce the final build artifact. Most of this book concerns the frontend, but we will briefly review the operation of the driver now.

In the Swift module system, all source files in a module must be built together. The Swift driver takes a list of source files on the command line, which become the *main module* being built. By default, the Swift driver generates an executable from the main module:

```
$ swiftc main.swift other.swift stuff.swift
```

Executables must define a *main function*, which is the entry point invoked when the executable is run. There are three mechanisms for doing so:

1. If the user invokes the driver with a single source file, this file becomes the *main source file* of the module. If there are multiple source files and one of them is named `main.swift`, then this file becomes the main source file. The main source file is special, in that it can contain statements at the top level, outside of a function body. Top-level statements are collected into *top-level code declarations*, and the frontend generates a main function which executes each top-level code declaration in source order. Source files other than the main source file cannot contain statements at the top level.

2. In the absence of a main source file, the user can annotate a struct, enum or class declaration with the `@main` attribute, in which case the declaration must contain a static method named `main()`. This method becomes the main entry point. This attribute was introduced in Swift 5.3 [38].

3. The older `@NSApplicationMain` and `@UIApplicationMain` attributes, deprecated since Swift 5.10 [39], provide a similar mechanism specific to Apple platforms. Attaching one of these attributes to a class conforming to `NSApplicationMain` or `UIApplicationMain`, respectively, will generate a main entry point which calls the `NSApplicationMain()` or `UIApplicationMain()` system framework function.

The `-emit-library` and `-emit-module` flags instruct the driver to generate a shared library together with the *serialized module* file that is consumed by the compiler when importing the library (Section 2.5):

```
$ swiftc algorithm.swift utils.swift -module-name SudokuSolver
      -emit-library -emit-module
```

Frontend jobs. The Swift frontend itself is single-threaded, but the driver can benefit from multi-core concurrency by running multiple frontend jobs in parallel. Each frontend job compiles one or more source files; these are the *primary source files* of the frontend job. All non-primary source files are the *secondary source files* of the frontend job. The assignment of primary source files to each frontend job is determined by the *compilation mode*:

- The `-wmo` driver flag selects *whole module mode*, typically used for release builds. In this mode, the driver schedules a single frontend job. The primary files of this job are all the source files in the main module, and there are no secondary files. In whole module mode, the frontend is able to perform more aggressive optimization across source file boundaries, hence its usage for release builds.

- The `-disable-batch-mode` driver flag selects *single file mode*, with one frontend job per source file. In this mode, each frontend job has a single primary file, with all other files being secondary files. Single file mode was the default for debug builds until Swift 4.1, however these days it is only used for testing the compiler.

  Single file mode incurs overhead in the form of duplicated work between frontend jobs; if two source files reference some declaration from a third source file, all three frontend jobs will need to parse and type check the third declaration; there is no caching or shared state between frontend jobs. (The next two sections detail how the frontend deals with secondary files, with delayed parsing and the request evaluator respectively.)

- The `-enable-batch-mode` driver flag selects *batch mode*, which is a happy medium between whole module and single file mode. In batch mode, the list of source files is partitioned into fixed-size batches, up to the maximum batch size. The source files in each batch become the primary files of each frontend job.

  By compiling multiple primary files in a single frontend job, batch mode amortizes the cost of parsing and type checking work performed on secondary files. At the same time, it still schedules multiple frontend jobs for parallelism on multi-core systems. Batch mode was first introduced in Swift 4.2, and is now the default for debug builds.

Each source file is a primary source file of exactly one frontend job, and within a single frontend job, the primary files and secondary files form a partition of the full list of source files in the module. A single source file is therefore the minimum unit of parallelism. By default, the number of concurrent frontend jobs is determined by the number of CPU cores; this can be overridden with the `-j` driver flag. If there are more frontend jobs than can be run simultaneously, the driver queues them and kicks them off as other frontend jobs complete. In batch mode and single file mode, the driver can also perform an *incremental build* by re-using the result of previous compilations, providing an additional compile-time speedup. Incremental builds are described in Section 2.3.

The `-###` driver flag performs a "dry run" which prints all commands to run without actually doing anything. In the below example, the driver schedules three frontend jobs, with each job having a single primary source file and two secondary files. The final command is the linker invocation, which combines the output of each frontend job into our binary executable.

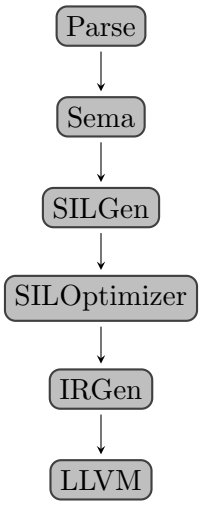
```
$ swiftc m.swift v.swift c.swift -###
swift-frontend -frontend -c -primary-file m.swift v.swift c.swift ...
swift-frontend -frontend -c m.swift -primary-file v.swift c.swift ...
swift-frontend -frontend -c m.swift v.swift -primary-file c.swift ...
ld m.o v.o c.o -o main
```

Compilation pipeline. Figure 2.1 shows a high-level view of the Swift frontend; this resembles the classic multi-pass compiler design, described in [2] or [3] for example:

1. **Parse:** Source files are parsed, building the abstract syntax tree.
2. **Sema:** Semantic analysis is performed, producing a type-checked syntax tree. (We'll see shortly the first two stages are not completely sequential.)
3. **SILGen:** The syntax tree is lowered to "raw SIL." SIL is the Swift Intermediate Language, an SSA (static single assignment) program representation [40, 41].
4. **SILOptimizer:** The raw SIL is transformed into "canonical SIL" by a series of *mandatory passes*, which analyze the control flow graph and emit diagnostics; for example, *definite initialization* ensures that all storage locations are initialized.

   When the `-O` command line flag is specified, the canonical SIL is optimized by a series of *performance passes* to improve run-time performance and code size.
5. **IRGen:** The optimized SIL is then transformed into LLVM IR.
6. **LLVM:** Finally, the LLVM IR is handed off to LLVM, which performs various lower level optimizations before generating machine code. (LLVM is, of course, the project formerly known as the "Low Level Virtual Machine [42].")

Figure 2.1.: The compilation pipeline



Debugging flags. Various command-line flags are provided to run the pipeline until a certain phase, and dump the output of that phase to the terminal (or some other file, in conjunction with the `-o` flag). These are useful for debugging the compiler:

- `-dump-parse` runs only the parser, and prints the syntax tree as an *s-expression*. (The term comes from Lisp. An s-expression represents a tree structure by nested parenthesized lists; e.g. `(a (b c) d)` is a node with three children `a`, `(b c)` and `d`, and `(b c)` has two children `b` and `c`.)
- `-dump-ast` runs only the parser and Sema, and prints the type-checked syntax tree as an s-expression.
- `-print-ast` prints the type-checked syntax tree in a form that approximates what was written in source code. This is useful for getting a sense of what declarations the compiler synthesized, for example for derived conformances to protocols like `Equatable`.
- `-emit-silgen` runs only Sema and SILGen, and prints the raw SIL output by SILGen.
- `-emit-sil` prints the canonical SIL output by the SIL optimizer. To see the output of the performance pipeline, also pass `-O`.
- `-emit-ir` prints the LLVM IR output by IRGen.
- `-S` prints the assembly output by LLVM.

Each pipeline phase can emit warnings and errors, collectively known as *diagnostics*. The parser attempts to recover from errors; the presence of parse errors does not prevent Sema from running. On the other hand, if Sema emits errors, compilation stops; SILGen does not attempt to lower an invalid abstract syntax tree to SIL (but SILGen can emit its own diagnostics, including those that result from lazy type checking of declarations in secondary files).

The compilation pipeline will vary slightly depending on what the driver and frontend were asked to produce. When the frontend is instructed to emit a serialized module file only, and not an object file, compilation stops after the SIL optimizer. When generating a textual interface file or TBD file, compilation stops after Sema. (Textual interfaces are discussed in Section 2.5. A TBD file is a list of symbols in a shared library, which can be consumed by the linker and is faster to generate than the shared library itself; we're not going to talk about them here.)

Frontend flags. The flags we listed above for dumping various stages of compiler output are understood by both the driver and the frontend; the driver passes them down to the frontend. Various other flags used for compiler development and debugging are only known to the frontend. If the driver is invoked with the `-frontend` flag as the first command line flag, then instead of scheduling frontend jobs, the driver spawns a single frontend job, passing it the rest of the command line without further processing:

```
$ swiftc -frontend -typecheck -primary-file a.swift b.swift
```

Another mechanism for passing flags to the frontend is provided by the `-Xfrontend` driver flag. When this flag appears in a command-line invocation, the driver proceeds to schedule jobs as usual, but the command line argument that immediately follows is passed directly to each frontend job:

```
$ swiftc a.swift b.swift -Xfrontend -dump-requirement-machine
```

2.1. Name Lookup

Name lookup is the process of resolving identifiers to declarations. The Swift compiler does not have a distinct "name binding" phase; instead, name lookup is queried from various points in the frontend process. Broadly speaking, there are two kinds of name lookup: *unqualified lookup* and *qualified lookup*. An unqualified lookup resolves a single identifier "`foo`", while qualified lookup resolves an identifier "`bar`" relative to a base, such as a member reference expression "`foo.bar`". There are also three important variants of these two fundamental kinds, for looking up top-level declarations in other modules, resolving operators, and performing dynamic lookups of Objective-C methods.

Unqualified lookup. An unqualified lookup is always performed relative to the source location where the identifier actually appears. The source location may either be in a primary or secondary file.

Unqualified lookup consults the source file's *scope tree*, which is constructed by walking the source file's abstract syntax tree. The root scope is the source file itself. Each scope has an associated source range, and zero or more child scopes; each child scope's source range must be a subrange of the source range of its parent, and the source ranges of sibling scopes are disjoint. Each scope introduces zero or more *variable bindings*.

Unqualified lookup first finds the innermost scope containing the source location, and proceeds to walk the scope tree up to the root, searching each parent node for bindings named by the given identifier. If the lookup reaches the root node, a *top-level lookup* is performed next. This will look for top-level declarations named by the given identifier, first in all source files of the main module, followed by all imported modules.

The `-dump-scope-maps` frontend flag dumps the scope map for each source file in the main module. For example, with this program:

```
func id<T>(_ t: T) -> T {
  return t
}
```

We get the scope map below:

```
ASTSourceFileScope 0x14c131908, [1:1 - 5:1] 'id.swift'
`-AbstractFunctionDeclScope 0x14c1392c0, [1:1 - 4:1] 'id(_:)'
  `-GenericParamScope 0x14c139118, [1:1 - 4:1] param 0 'T'
    |-ParameterListScope 0x14c139238, [1:11 - 1:18]
    `-FunctionBodyScope 0x14c1392c0, [1:25 - 4:1]
      `-BraceStmtScope 0x14c139510, [1:25 - 4:1]
        `-PatternEntryDeclScope 0x14c139450, [2:7 - 4:1] entry 0 'x'
          `-PatternEntryInitializerScope 0x14c139450, [2:11 - 2:11] entry 0 'x'
```

Unqualified lookup is an important part of type resolution (Section 9.1).

Qualified lookup. A qualified lookup searches in a list of type declarations for a member with the given name. Qualified lookup recursively visits the conformed protocols of each struct, enum and class declaration, and the superclass of each class declaration. If the found member was from a protocol or superclass, we apply a substitution map, which will be described in Section 9.2. The primitive operation, which searches inside a single type declaration and its extensions only, is called *direct lookup* (Section 10.2).

Module lookup. A qualified lookup where the base is a module declaration searches for a top-level declaration in the given module and any other modules that it re-exports via "`@_exported import`".

Dynamic lookup. A qualified lookup whose base is the `AnyObject` type implements the legacy Objective-C behavior of a message send to `id`, which can invoke any method defined in any Objective-C class or protocol. In Swift, the so-called *dynamic lookup* searches a global lookup table constructed to contain all `@objc` members of all classes and protocols:

- Any class can contain `@objc` members, and the attribute can either be explicitly stated, or inferred if the method overrides an `@objc` method from the superclass.
- Protocol members are `@objc` only if the protocol itself is `@objc`.

Operator lookup. Operator symbols are declared at the top level of a module by *operator declarations*. Operator declarations have a fixity (prefix, infix, or postfix), and infix operators also have a *precedence group*. Precedence groups are partially ordered with respect to other precedence groups. Standard operators like `+` and `*` and their precedence groups are thus defined in the standard library, rather than being built-in to the language itself.

The parser parses an arithmetic expression like `2 + 3 * 6` into a flat list of nodes and operator symbols, called a *sequence expression*. The parser does not know the precedence, fixity or associativity of the `+` and `*` operators. Indeed, it does not know that they exist at all. The *pre-check* pass of the expression type checker looks up operator symbols, in this case `+` and `*`, and transforms sequence expressions into the more familiar nested tree form, according to the operator's fixity, precedence and associativity.

Operator symbols do not themselves have an implementation; they are just names. An operator symbol can be used as the name of a function implementing the operator on a specific type (for prefix and postfix operators) or a specific pair of types (for infix operators). Operator functions can be declared either at the top level, or as a member of a type. As far as a name lookup is concerned, the interesting thing about operator functions is that they are visible globally, even when declared inside of a type. Operator functions are found by consulting the operator lookup table, which contains top-level operator functions as well as member operator functions of all declared types.

When the compiler type checks the expression `2 + 3 * 6`, it must pick two specific operator functions for `+` and `*` among all the possibilities, in order for this expression to type check. In this case, the overloads for `Int` are chosen, because `Int` is the default literal type for the literals `2`, `3` and `6`.

Listing 2.1 shows the definition of some custom operators and precedence groups. Note that the overload of `++` inside struct `Chicken` returns `Int`, and the overload of `++` inside struct `Sausage` returns `Bool`. The closure value stored in `fn` applies `++` to two anonymous closure parameters, `$0` and `$1`. While they do not have declared types, by simply coercing the *return type* to `Bool`, we are able to unambiguously pick the overload of `++` declared in `Sausage`. (Whether this is good style is left to the reader to judge.)

Listing 2.1.: Operator lookup in action

```
prefix operator <&>
infix operator ++: MyPrecedence
infix operator **: MyPrecedence

precedencegroup MyPrecedence {
  associativity: right
  higherThan: AdditionPrecedence
}

// Member operator examples
struct Chicken {
  static prefix func <&>(x: Chicken) {}
  static func ++(lhs: Chicken, rhs: Chicken) -> Int {}
}

struct Sausage {
  static func ++(lhs: Sausage, rhs: Sausage) -> Bool {}
}

// Top-level operator example
func **(lhs: Sausage, rhs: Sausage) -> Sausage {}

// Global operator lookup finds Sausage.++
// 'fn' has type (Sausage, Sausage) -> Bool
let fn = { ($0 ++ $1) as Bool }
```

Initially, infix operators defined their precedence as an integer value; Swift 3 introduced named precedence groups [43]. The global lookup for operator functions dates back to when all operator functions were declared at the top level. Swift 3 also introduced the ability to declare operator functions as members of types, but the global lookup behavior was retained [44].

2.2. Delayed Parsing

The "compilation pipeline" model as described is an over-simplification of the actual state of affairs. Ultimately, each frontend job only needs to generate machine code from the declarations in its primary files, so all stages from SILGen onward operate on the frontend job's primary files only. The situation while parsing and type checking is more subtle, because name lookup must find declarations in other source files, even secondary files. This requires having the abstract syntax tree for secondary files as well. However, it would be inefficient if every frontend job was required to fully parse all secondary files, because the time spent in the parser would be proportional to the number of frontend jobs multiplied by the number of source files, negating the benefits of parallelism.

The *delayed parsing* optimization solves this dilemma. When parsing a secondary file for the first time, the parser does not construct syntax tree nodes for the bodies of top-level types, extensions, and functions. Instead, it operates in a high-speed mode where comments are skipped and pairs of braces are matched, but very little other work is performed. This outputs a "skeleton" representation of each secondary file. (In whole module mode, there is no delayed parsing. There are no secondary files, and delayed parsing of declarations in primary files is pointless, since they are always needed for type checking and code generation anyway.) If we later need the body of a type or extension declaration from a secondary file—for example, if a name lookup into this declaration is performed while type checking an expression in a primary file—then we will parse the source range of the declaration again, this time building the full syntax tree.

While it is possible to construct a pathological program where every source file triggers delayed parsing of all declarations in every other file, this is unlikely in practice.

Operator lookup. For delayed parsing to work, the skipped members of types and extensions must have no observable effect on compilation. This is always true with two exceptions: operator lookup, and dynamic lookup. As explained in the previous section, operator functions are visible globally, even when declared as a method of a type. To deal with this, the parser looks for the keyword "`func`" followed by an operator symbol when skipping a type or extension body in a secondary file. The first time an operator lookup is performed, the bodies of all types and extensions that contain operator functions are parsed again. Most types and extensions do not define operator functions, so this occurs rarely in practice.

Dynamic lookup. The situation with dynamic lookup is similar, since a method call on a value of type `AnyObject` must consult a global lookup table constructed from `@objc` members of classes, and the (implicitly `@objc`) members of `@objc` protocols. Unlike operator functions, classes and `@objc` protocols are quite common in Swift programs, however `AnyObject` lookup itself is rarely used. The first time a frontend job encounters a dynamic `AnyObject` method call, all class bodies flagged as potentially containing `@objc` methods are eagerly parsed.

There's actually one more complication here. Classes can be nested inside of other types, whose bodies are skipped if they appear in a secondary file. To find such classes when building the `AnyObject` lookup table, we rely on a similar trick as operator lookup. When skipping the body of a type, the parser looks for occurrences of the "`class`" keyword. If the body contains this keyword, we record this fact, so that this type can be completely parsed later if needed.

Most Swift programs, even those making heavy use of Objective-C interoperability, do not contain a dynamic `AnyObject` method call in every source file, so delayed parsing remains effective.

Example 2.1. Listing 2.2 shows an example of this behavior. This program consists of three files. Suppose that the driver kicks off three frontend jobs, with a single primary file for each frontend job.

Listing 2.2.: Delayed parsing with `AnyObject` lookup

```
// a.swift
func f(x: AnyObject) {
  x.foo()
}
```

```
// b.swift
func g() {
  f()
}
```

```
// c.swift
struct Outer {
  class Inner {
    @objc func foo() {}
  }
}
```

The frontend jobs each do the following:

- The frontend job with the primary file `a.swift` will parse `b.swift` and `c.swift` as secondary files. The body of `g()` in `b.swift` is skipped. The parser also skips the body of `Outer`, but records that it contains the `class` keyword. The function `f()` in `a.swift` contains a dynamic `AnyObject` call, so this frontend job will construct the global lookup table, triggering parsing of `Outer` and `Inner` in `c.swift`.
- The frontend job with the primary file `b.swift` will parse `a.swift` and `c.swift` as secondary files. This primary file does not reference anything from `c.swift` at all, so `Outer` remains unparsed in this frontend job. Type checking the call to `f()` from `g()` also does not require parsing the *body* of `f()`.
- The frontend job with the primary file `c.swift` will parse `a.swift` and `b.swift` as secondary files, skipping the body of `f()` and `g()`.

2.3. Request Evaluator

The *request evaluator* generalizes the idea behind delayed parsing to all of type checking. As with parsing, the classic compiler design, where a single semantic analysis pass walks declarations in source order, is not well-suited for Swift:

- Declarations may be written in any order within a Swift source file, without being forward declared (unlike Pascal or C). Expressions and type annotations can also reference declarations in other source files without restriction. Finally, certain kinds of circular references are permitted.

  In particular, this means that within a single frontend job, an entity in a primary file may reference a declaration that has not yet been type checked, or is in the process of being type checked.

- Ordering issues aside, there is also the potential overhead of duplicated work across frontend jobs. Every time a frontend job type checks a declaration in a secondary file, some of the benefit of parallelism is lost, since this secondary file is necessarily the primary file of some other frontend job, and the same declaration must therefore be type checked again in the other job.

  For this reason, we want to minimize time spent type checking declarations in secondary files.

Thus, the work of type checking is split up into small, fine-grained *requests* which are evaluated on demand, instead of sequentially. There is still a semantic analysis pass that visits the declarations of each primary file in source order, but it merely kicks off requests and emits diagnostics.

Concretely, a *request* packages a list of input parameters together with an *evaluation function*. With the exception of emitting diagnostics, the request function's result should only depend on those inputs, and the results of other requests. The request evaluator directly invokes the evaluation function, and caches the result. Clients only evaluate the request via the request evaluator framework, which returns the cached value if present, detects request cycles automatically, and tracks dependency information for incremental builds.

The Swift frontend defines hundreds of request kinds; for our purposes, the most important ones are:

- The **type-check primary file request** visits each declaration in a primary source file. It is responsible for kicking off enough requests to ensure that SILGen can proceed if all requests succeeded without emitting diagnostics.
- The **AST lowering request** is the entry point into SILGen, generating SIL from the abstract syntax tree for a source file.
- The **unqualified lookup request** and **qualified lookup request** perform the two kinds of name lookup described in the previous section.
- The **interface type request** is explained in Chapter 4.
- The **generic signature request** is explained in Chapter 11.

Example 2.2. Consider what happens when we type check this program:

```
let food = cook()
func cook() -> Food {}
struct Food {}
```

Notice how the initial value expression of `food` references the `cook()` function, and the return type of `cook()` is the `Food` struct declared immediately after, `Food` is the inferred type of `food`. This plays out with the request evaluator:

1. The **type-check primary file request** begins by visiting the declaration of `food` and performing various semantic checks.
2. One of these checks evaluates the **interface type request** with the declaration of `food`. This is a variable declaration, so the evaluation function will type check the initial value expression and return the type of the result.
   a) In order to type check the expression `cook()`, the **interface type request** is evaluated again, this time with the declaration of `cook` as its input parameter.
   b) The interface type of `cook()` has not been computed yet, so the request evaluator invokes the request evaluation function.

3. After computing the interface type of `food` and performing other semantic checks, the **type-check primary file request** moves on to the declaration of `cook`:
   a) The **interface type request** is evaluated once again, with the input parameter being the declaration of `cook`.
   b) The result was already cached, so the request evaluator immediately returns the cached result without computing it again.
4. Finally, we type check the declaration of `Food`, evaluating any remaining requests.

The **type-check primary file request** is special, because it does not return a value; it is evaluated for the side effect of emitting diagnostics, whereas most other requests return a value. The implementation of the **type-check primary file request** guarantees that if no diagnostics were emitted, then SILGen can generate valid SIL for all declarations in a primary file. However, the next example shows that SILGen can encounter invalid declarations, and diagnose errors in secondary files.

Example 2.3. Suppose we run a frontend job with the below primary file:

```
// a.swift
func open(_: Box) {}
```

We're going to look at what happens when `Box` is defined in a secondary file with a semantic error:

```
// b.swift
struct Box {
  let contents: DoesNotExist
}
```

Our frontend job does not emit any diagnostics in the semantic analysis pass, because the `contents` stored property of `Box` is not actually referenced while type checking the primary file `a.swift`. However when SILGen runs, it needs to determine whether the parameter of type `Box` to the `open()` function needs to be passed directly in registers, or via an address by computing the *type lowering* for the `Box` type. The type lowering procedure recursively computes the type lowering of each stored property of `Box`; this evaluates the **interface type request** for the `contents` property of `Box`, which emits a diagnostic because the identifier "`DoesNotExist`" does not resolve to a valid type. The interface type of the stored property then becomes the error type. We will discuss type lowering in Section 6.5.

The request evaluator framework was first introduced in Swift 4.2 [45]. In subsequent releases, various ad-hoc mechanisms were gradually converted into request evaluator requests, with resulting gains to compiler performance, stability, and implementation maintainability.

Cycles. In a language that permits forward references, one can write a syntactically well-formed program where all identifiers reference valid declarations, but the program is nonetheless invalid, because of circularity. The classic example of this is a pair of classes where each class inherits from the other:

```
class A: B {}
class B: A {}
```

Implementing bespoke logic to detect circularity is error-prone and tedious, and a missing circularity check can result in a crash or infinite loop when the compiler encounters an invalid input program. The request evaluator centralizes cycle detection by maintaining a stack of *active requests*. Before evaluating a request, the request evaluator first checks if the active request stack already contains an equal request. In this case, calling the evaluation function would result in infinite recursion, so instead the request evaluator diagnoses an error and returns a request-specific sentinel value.

```
$ swiftc cycle.swift
cycle.swift:1:7: error: ‘A’ inherits from itself
class A: B {}
      ^
cycle.swift:2:7: note: class ‘B’ declared here
class B: A {}
      ^
```

The circularity diagnostic can be customized for each request kind; the default just says “circular reference.” If the compiler is invoked with the `-debug-cycles` frontend flag, the active request stack is also printed:

```
$ swiftc cycle.swift -Xfrontend -debug-cycles
===CYCLE DETECTED===
 ‘--TypeCheckPrimaryFileRequest(source_file "cycle.swift")
     ‘--SuperclassDeclRequest(cycle.(file).A@cycle.swift:1:7)
         ‘--SuperclassDeclRequest(cycle.(file).B@cycle.swift:2:7)
             ‘--SuperclassDeclRequest(cycle.(file).A@cycle.swift:1:7)
```

Performance analysis. A handful of command-line flags are provided to help with understanding compile-time performance. The `-stats-output-dir` flag is followed by the name of a directory, which must already exist. Each frontend job writes a new JSON file to this directory, with various counters and timers. When used in conjunction with the `-fine-grained-timers` flag, the compiler will count the number of request evaluations, and the total time spent in request evaluation, broken down into each kind

of request. The output can be sliced and diced in various ways; one can actually make pretty effective use of "`awk`" [46] for example, despite the JSON format:

```
$ mkdir /tmp/stats
$ swiftc -stats-output-dir -fine-grained-timers /tmp/stats ...
$ awk -f '/InterfaceTypeRequest.wall/ { x += $2 } END { print x }' \
    /tmp/stats/*.json
```

Another command-line flag is `-trace-stats-events`. It must be passed in conjunction with `-stats-output-dir`, and enables output of a trace file to the statistics directory. The trace file is a sequence of time-stamped events that mark the start and end of each request evaluation function, in CSV format. More details about these flags in [47].

2.4. Incremental Builds

The request evaluator also records dependencies for incremental compilation, enabled by the `-incremental` driver flag. The goal of incremental compilation is to prove which files do not need to be rebuilt, in the least conservative way possible. The quality of an incremental compilation implementation can be judged as follows (the author would like to acknowledge David Ungar for this explanation):

1. Perform a clean build of all source files in the program, and collect the object files.
2. Make a change to one or more source files in the input program.
3. Do an incremental build, which rebuilds some subset of source files in the input program. If a source file was rebuilt but the resulting object file is identical to the one saved in Step 1, the incremental build performed *wasted work*.
4. Finally, do another clean build, which yet again rebuilds all source files in the input program. If a source file was rebuilt and the resulting object file is different to the one saved in Step 1, the incremental build was *incorrect*.

This highlights the difficulty of the incremental compilation problem. Rebuilding *too many* files is an annoyance; rebuilding *too few* files is a correctness issue. A correct but ineffective implementation would rebuild all source files every time. On the other hand, the opposite approach of only rebuilding the subset of source files that have changed since the last compiler invocation is too aggressive. To see why it is incorrect, consider the program shown in Listing 2.3. Let's say the programmer builds the program, adds the overload `f: (Int) -> ()`, then builds it again. The new overload is more specific, so the call `f(123)` in `b.swift` now refers to the new overload; therefore, `b.swift` must also be rebuilt.

Listing 2.3.: Rebuilding a file after adding a new overload

```
// a.swift
func f<T>(_: T) {}

// new overload added in second version of file
func f(_: Int) {}
```

```
// b.swift
func g() {
  // new overload is selected after a.swift is updated
  f(123)
}
```

The approach taken by the Swift compiler is to construct a *dependency graph*. The frontend outputs a *dependency file* for each source file, recording all names the source file *provides*, and all names the type checker *requires* while compiling the source file. Dependency files use a binary format with the "`.swiftdeps`" file name extension. The list of provided names in the dependency file is generated by walking the abstract syntax tree and collecting all visible declarations in each source file. The list of required names is generated by the request evaluator, using the stack of active requests. Every cached request has a list of required names, and a request can optionally be either a dependency sink, or dependency source:

- A *dependency sink* is a name lookup request which records a required name. When a dependency sink request is evaluated, the request evaluator walks the stack of active requests, adding the identifier to each active request's list of required names. Thus, for every request, we track the name lookups that took place from the evaluation function.

  An important caveat is that when a request with a cached value is evaluated again, the request's cached list of required names must again be "replayed," adding them to each active request that depends on the cached value.

- A *dependency source* is a request which appears at the top of the request stack, such as the **type-check primary file request** or the **AST lowering request**. A dependency source scopes some amount of work to a source file.

  After the evaluation of a dependency source request completes, all required names attributed to the request are added to the source file's list of required names.

The driver makes use of the dependency files generated by the frontend to actually perform an incremental build. This happens in two phases:

1. The first phase rebuilds all source files which have changed since the last compilation. This is the minimum set that must be rebuilt.

2. The second phase reads the dependency files, and collects all names provided by the source files rebuilt in the first phase. The source files which depend on those names are then rebuilt.

Example 2.4. To understand how request caching interacts with dependency recording, consider the program shown in Listing 2.4. Suppose the driver decides to compile *both* `a.swift` and `b.swift` in the same frontend job (in fact, the issue at hand can only appear in batch mode, when a frontend job has more than one primary file). First, the **type-check primary file request** runs with the source file `a.swift`.

1. While type checking the body of `breakfast()`, the type checker evaluates the **unqualified lookup request** to resolve the identifier "`soup`."

2. This records the identifier "`soup`" in the require names list of each active request. There is one active request, the **type-check primary file request** for `a.swift`.

3. The lookup finds the declaration of `soup()` in `c.swift`.

Listing 2.4.: Recording incremental dependencies

```
// a.swift
func breakfast() {
  soup(nil)
}
```

```
// b.swift
func lunch() {
  soup(nil)
}
```

```
// c.swift
func soup(_: Pumpkin?) {}
struct Pumpkin {}
```

4. The type checker evaluates the **interface type request** with the declaration of `soup()`.
   a) The **interface type request** evaluates the **unqualified lookup request** with the identifier "`Pumpkin`."
   b) This records the identifier "`Pumpkin`" in the required names list of each active request, of which there are now two: the **interface type request** for `soup()`, and the **type-check primary file request** for `a.swift`.
5. The **type-check primary file request** for `a.swift` completes. The required names list for this request contains two identifiers, "`soup`" and "`Pumpkin`"; both are added to the required names list of the source file `a.swift`.

Next, the **type-check primary file request** runs with the source file `b.swift`.

1. While type checking the body of `lunch()`, the type checker evaluates the **unqualified lookup request** with the identifier "`soup`."
2. This records the identifier "`soup`" in the required names list of each active request. There is one active request, the **type-check primary file request** for `b.swift`.
3. The lookup finds the declaration of `soup()` in `c.swift`.
4. The type checker evaluates the **interface type request** with the declaration of `soup()`.
5. This request has already been evaluated, and the cached result is returned. The required names list for this request is the single identifier "`Pumpkin`." This required names list is replayed, as if the request was being evaluated for the first time. This adds the identifier "`Pumpkin`" to the required names list of each active request, of which there is just one: the **type-check primary file request** for `b.swift`.
6. The **type-check primary file request** for `b.swift` completes. The required names list for this request contains two identifiers, "`soup`" and "`Pumpkin`"; both are added to the required names list of the source file `b.swift`.

The frontend job writes out the dependency files for `a.swift` and `b.swift` upon completion. Both source files require the names "`soup`" and "`Pumpkin`." The dependency of `b.swift` on "`Pumpkin`" is correctly recorded because evaluating a request with a cached value replays the request's required names list in Step 2 above.

There's a bit more to the incremental build story than this; in particular, we haven't talked about the "interface hash" mechanism, meant to avoid rebuilding of dependent source files for changes were limited to comments, whitespace or function bodies. We're already far afield from the goal of describing Swift generics though, so the curious reader can refer to [45] and [48] for details.

2.5. Module System

The frontend represents a module by a *module declaration* containing one or more *file units*. The list of source files in a compiler invocation form the *main module*. The main module is special, because its abstract syntax tree is constructed directly by parsing source code; the file units are *source files*. There are three other kinds of modules:

1. **Serialized modules** that consist of one or more *serialized AST file units*. When the main module imports another module written in Swift, the frontend reads a serialized module that was previously built.

2. **Imported modules** that consist of one or more *Clang file units*. These are the modules implemented in C, Objective-C, or C++.

3. **The builtin module**, which has exactly one file unit that contains types and intrinsics implemented by the compiler itself.

The main module depends on other modules via the `import` keyword, which parses as an *import declaration*. After parsing, one of the first stages in semantic analysis loads all modules imported the main module. The standard library is defined in the `Swift` module, which is imported automatically unless the frontend was invoked with the `-parse-stdlib` flag, used when building the standard library itself. As for the builtin module, it is ordinarily not visible, but the `-parse-stdlib` flag also causes it to be implicitly imported (Section 3.3).

Serialized modules. The `-emit-module` flag instructs the compiler to generate a serialized module. Serialized module files use the "`.swiftmodule`" file name extension. Serialized modules are stored in a binary format, closely tied to the specific version of the Swift compiler. (When building a shared library for distribution, it is better to publish a textual interface instead, as described at the end of this section.)

Name lookup into a serialized module lazily constructs declarations by deserializing records from this binary format as needed. Deserialized declarations generally look like parsed and fully type-checked declarations, but they sometimes contain less information. For example, in Section 4.2, we will encounter various syntactic representations of requirements, such as `where` clauses. Since this information is only used when type checking the declaration, it is not serialized. Instead, deserialized declarations only need to store a generic signature, described in Chapter 5.

A parsed function declaration has a body, consisting of statements and expressions. This body is not preserved by serialization, so a deserialized function declaration does not have a body. To implement the `@inlinable` attribute, which makes a function definition available for inlining and specialization across module boundaries, we serialize the SIL representation of the function.

Imported modules. An imported module is implemented in C, Objective-C, or C++. The Swift compiler embeds a copy of Clang and uses it to parse module maps, header files, and binary precompiled headers. Name lookup into an imported module lazily constructs Swift declarations from their corresponding Clang declarations. The Swift compiler component responsible for this is known as the "Clang importer."

Imported function declarations generally do not have bodies if the entry point was previously emitted by Clang and is available externally. Occasionally the Clang importer synthesizes accessor methods and other such trivia, which do have bodies represented as Swift statements and expressions. C functions not available externally, such as `static inline` functions declared in header files, are emitted by having Swift IRGen call into Clang.

Invoking the compiler with the `-import-objc-header` flag followed by a header file name specifies a *bridging header*. This is a shortcut for making C declarations in the bridging header visible to all other source files in the main module, without having to define a separate Clang module first. This is implemented by adding a Clang file unit corresponding to the bridging header to the main module. For this reason, compiler code should not assume that all file units in the main module are Swift source files.

Textual interfaces. The binary module format depends on compiler internals and no attempt is made to preserve compatibility across compiler releases. When building a shared library for distribution, it is better to generate a *textual interface*:

```
$ swiftc Horse.swift -enable-library-evolution -emit-module-interface
```

Unlike the serialized module format, textual interfaces only describe the public declarations of a module. The `-enable-library-evolution` flag enables *resilience*, which is a prerequisite for emitting a textual interface. Resilience instructs clients to use more abstract access patterns which are guaranteed to only depend on the public declarations of a module. For example, it allows new stored properties to be added to a public struct. Resilience is documented in [49, 50].

Textual interface files use the "`.swiftinterface`" file name extension. They are generated by the AST printer, which prints declarations in a format that looks very much like Swift source code, with a few exceptions:

1. Non-`@inlinable` function bodies are skipped. Bodies of `@inlinable` functions are printed verbatim, including comments, except that `#if` conditions are evaluated.

2. Various synthesized declarations, such as type alias declarations from associated type inference, witnesses for derived conformances such as `Equatable`, and so on, are written out explicitly.

3. Opaque result types also require special handling (Section 13.2).

Note that (1) above means the textual interface format is target-specific; a separate textual interface needs to be generated for each target platform, alongside the shared library itself.

When a module defined by a textual interface is imported for the first time, a frontend job parses and type checks the textual interface, and generates a serialized module file which is then consumed by the original frontend job. Serialized module files generated in this manner are cached, and can be reused between invocations of the same compiler version.

The `@inlinable` attribute was introduced in Swift 4.2 [51]. The Swift ABI was formally stabilized in Swift 5, when the standard library became part of the operating system on Apple platforms. Library evolution support and textual interfaces became user-visible features in Swift 5.1 [52]. A recent paper describes a formal model for reasoning about the Swift ABI [53].

2.6. Source Code Reference

The Swift driver is now implemented in Swift, and lives in a separate repository from the rest of the compiler:

```
https://github.com/swiftlang/swift-driver
```

The Swift frontend, standard library, and runtime are found in the main repository:

```
https://github.com/swiftlang/swift
```

The major components of the Swift frontend live in their own subdirectories of the main repository. The entities modeling the abstract syntax tree are defined in `lib/AST/` and `include/swift/AST/`; among these, types and declarations are important for the purposes of this book, and will be covered in Chapter 3 and Chapter 4. The core of the SIL intermediate language is implemented in `lib/SIL/` and `include/swift/SIL/`.

Each stage of the compilation pipeline has its own subdirectory:

- `lib/Parse/`
- `lib/Sema/`
- `lib/SILGen/`
- `lib/SILOptimizer/`
- `lib/IRGen/`

The AST Context

Key source files:

- include/swift/AST/ASTContext.h
- lib/AST/ASTContext.cpp

`ASTContext`	*class*

This is a singleton class representing the frontend instance. An AST context provides a memory allocation arena, unique allocation for various immutable data types used throughout the compiler, and storage for various other global singletons.

Request Evaluator

Key source files:

- include/swift/AST/Evaluator.h
- lib/AST/Evaluator.cpp

`SimpleRequest`	*template class*

Each request kind is a subclass of `SimpleRequest`. Subclasses implement the evaluation function by overriding the `evaluate()` method of `SimpleRequest`.

`RequestFlags`	*enum class*

One of the template parameters to `SimpleRequest` is a value of this type. A handful of flags specify the caching policy. Exactly one of these must be specified:

- `RequestFlags::Uncached` indicates that no caching is to be performed.
- `RequestFlags::Cached` indicates that results should be automatically cached.
- `RequestFlags::SeparatelyCached` indicates that the request's results should be cached by the request implementation itself.
- `RequestFlags::SplitCached` is a hybrid strategy that combines both automatic and separate caching, as described below.

Another pair of flags define how this request interacts with the dependency tracking mechanism for incremental builds (Section 2.4). At most one of the below should be specified:

- `RequestFlags::DependencySource` marks this request as a dependency source. This is set if the request directly performs a name lookup.
- `RequestFlags::DependencySink` marks this request as a dependency sink. This is set for top-level request associated with an entire source file.

Due to expressivity limitations in C++, a bit of boilerplate is involved in the definition of a new request kind. For example, consider the `InterfaceTypeRequest`, which takes a `ValueDecl` as input and returns a `Type` as output:

- The request type ID is declared in `include/swift/AST/TypeCheckerTypeIDZone.def`.
- The `InterfaceTypeRequest` class is declared in `include/swift/AST/TypeCheckRequests.h`.
- The `InterfaceTypeRequest::evaluate()` method is defined in `lib/Sema/TypeCheckDecl.cpp`.
- The request is separately cached, so the `InterfaceTypeRequest` class also overrides the `isCached()`, `getCachedResult()`, and `cacheResult()` methods, described below.

These methods are implemented in `lib/AST/TypeCheckRequestFunctions.cpp`.

`Evaluator` *class*

The `Evaluator` class is a singleton, stored in the `evaluator` instance variable of the global `ASTContext` singleton.

Requests are evaluated by calling the `evaluateOrDefault()` top-level function. This function receives the request evaluator singleton, the request to evaluate, and a sentinel value to return in case of circularity. The request evaluator will either return a cached value, or invoke the evaluation function and cache the result.

For example, the implementation of the `ValueDecl::getInterfaceType()` method evaluates the `InterfaceTypeRequest` as follows:

```
Type ValueDecl::getInterfaceType() const {
  auto &ctx = getASTContext();
  return evaluateOrDefault(
    ctx.evaluator,
    InterfaceTypeRequest{const_cast<ValueDecl *>(this)},
    ErrorType::get(ctx)));
}
```

Request Caching

We will now discuss the various forms of caching available by specifying `RequestFlags`.

An `Uncached` request is appropriate when the evaluation function wraps some other bit of code that performs its own caching, or when external conditions guarantee that this request will only be evaluated once for each possible input.

A `Cached` request will store its results in a per-request kind `DenseMap` whose keys are the inputs given to the evaluation function. This requires no additional work on the side of the request, but it has the disadvantage that the overhead of the `DenseMap` becomes significant when the number of distinct keys is too great. The request implementation only needs to declare one additional method in this case, named `isCached()`. This allows opting out of caching for certain inputs only; always returning `true` is the most common implementation.

A `SeparatelyCached` request must declare `isCached()` as above, together with two additional methods, `getCachedResult()` and `cacheResult()`. Separate caching takes additional work to implement, but it avoids the cost of the `DenseMap` when the cached value can be stored directly inside the input value.

For example, in a whole-module build, the `InterfaceTypeRequest` will be evaluated against almost every single `ValueDecl`. For this reason, the `InterfaceTypeRequest` uses separate caching, storing the interface type directly in an instance variable of the `ValueDecl` itself.

A `SplitCached` request is like `SeparatelyCached` in that the request must declare `isCached()`, `getCachedResult()`, and `cacheResult()` methods, which are called by the evaluator. The evaluator also allocates a `DenseMap`, just like the `Cached` case, but the evaluator does not store anything there itself. The request's `getCachedResult()` and `cacheResult()` methods either store the result somewhere of their own choosing, or they pass it down to the evaluator's cache, using a pair of methods on the `Evaluator` singleton:

- `getCachedNonEmptyOutput()` takes a request, and returns a `std::optional` with the cached value, or `std::nullopt` if there isn't one.
- `cacheNonEmptyOutput()` takes a request and the result, and updates the cache.

Split caching works best when the request is evaluated against a large number of inputs, but when the result is almost always some empty placeholder value. The idea is that the empty value is stored directly, perhaps with a single bit, whereas all other results are cached by the request evaluator.

For example, we use split caching for the `OpaqueResultTypeRequest` of Chapter 13. This request is evaluated against almost every `ValueDecl`, but most value declarations don't have an opaque result type, so the result is almost always `nullptr`. We don't want to build a `DenseMap` whose keys range over all value declarations and where most values

are `nullptr`. We also don't want to use separate caching, since that would require adding a new instance variable to `ValueDecl` whose value is almost always `nullptr`. Split caching allows us to reserve a single bit in the `ValueDecl` to indicate the presence of an opaque result type, while the actual declaration of the opaque result type is then stored in the request evaluator's cache, but only for those value declarations that actually have one.

If the `-analyze-request-evaluator` frontend flag is specified, the frontend job prints statistics about the request evaluator cache upon completion. This information is useful when choosing the appropriate caching strategy for a new request.

Name Lookup

Key source files:

- `include/swift/AST/NameLookup.h`
- `include/swift/AST/NameLookupRequests.h`
- `lib/AST/NameLookup.cpp`
- `lib/AST/UnqualifiedLookup.cpp`

The "ASTScope" subsystem implements unqualified lookup for local bindings. This code is internal to the name lookup implementation; the rest of the compiler does not generally interact with ASTScope directly:

- `include/swift/AST/ASTScope.h`
- `lib/AST/ASTScope.cpp`
- `lib/AST/ASTScopeCreation.cpp`
- `lib/AST/ASTScopeLookup.cpp`
- `lib/AST/ASTScopePrinting.cpp`
- `lib/AST/ASTScopeSourceRange.cpp`

`UnqualifiedLookupRequest` *class*

Unqualified lookups are performed by evaluating an instance of this request kind. The request takes an `UnqualifiedLookupDescriptor` as input.

`UnqualifiedLookupDescriptor` *class*

Encapsulates the input parameters for an unqualified lookup:

- The name to look up.
- The declaration context where the lookup starts.
- The source location where the name was written in source. If not specified, this becomes a top-level lookup.
- Various flags, described below.

`UnqualifiedLookupFlags` *enum class*

Flags passed as part of an `UnqualifiedLookupDescriptor`.

- `UnqualifiedLookupFlags::TypeLookup`: if set, lookup ignores declarations other than type declarations. This is used in type resolution.
- `UnqualifiedLookupFlags::AllowProtocolMembers`: if set, lookup finds members of protocols and protocol extensions. Generally should always be set, except to avoid request cycles in cases where it is known the result of the lookup cannot appear in a protocol or protocol extensions.
- `UnqualifiedLookupFlags::IgnoreAccessControl` if set, lookup ignores access control. Generally should never be set, except when recovering from errors in diagnostics.
- `UnqualifiedLookupFlags::IncludeOuterResults` if set, lookup stops after finding results in an innermost scope, or to always proceed to a top-level lookup.

`DeclContext` *class*

Declaration contexts will be introduced in Chapter 4, and the `DeclContext` class in Section 4.6.

- The various overloads of `lookupQualified()` perform a qualified name lookup into the given base type. The "`this`" parameter—the `DeclContext *` which the method is called on determines the visibility of declarations found via lookup through imports and access control; `this` is *not* the base type of the lookup.

`NLOptions` *enum*

Similar to `UnqualifiedLookupFlags`, but for `DeclContext::lookupQualified()`.

- `NL_OnlyTypes`: if set, lookup ignores declarations other than type declarations. This is used in type resolution.

- `NL_ProtocolMembers`: if set, lookup finds members of protocols and protocol extensions. Generally should always be set, except to avoid request cycles in cases where it is known the result of the lookup cannot appear in a protocol or protocol extension.
- `NL_IgnoreAccessControl`: if set, lookup ignores access control. Generally should never be set, except when recovering from errors in diagnostics.

`NominalTypeDecl` *class*

Nominal type declarations will be introduced in Chapter 4, and the `NominalTypeDecl` class in Section 4.6. The implementation of direct lookup and lazy member loading is discussed in Section 10.5.

- `lookupDirect()` performs a direct lookup, which only searches the nominal type declaration itself and its extensions, ignoring access control.

`lookupInModule()` *function*

Searches for top-level declarations within a module. Operates in one of two modes, depending on the arguments given:

- Qualified lookup into a specific module. Looks inside the given module and all of its `@_exported` imports.
- Unqualified lookup from the top-level of a source file. Looks inside all modules imported from this source file, as well as any `@_exported` imports from other source files in the main module.

Primary File Type Checking

Key source files:

- `lib/Sema/TypeCheckDeclPrimary.cpp`

The `TypeCheckPrimaryFileRequest` calls the `typeCheckDecl()` global function, which uses the visitor pattern to switch on the declaration kind. For each declaration kind, it performs various semantic checks and kicks off requests which may emit diagnostics.

Module System

`ModuleDecl` *class*

A module.

- `getName()` returns the module's name.
- `getFiles()` returns an array of `FileUnit`.
- `isMainModule()` answers if this is the main module.

`FileUnit` *class*

Abstract base class representing a file unit.

`SourceFile` *class*

Represents a parsed source file from disk. Inherits from `FileUnit`.

- `getTopLevelItems()` returns an array of all top-level items in this source file.
- `isPrimary()` returns `true` if this is a primary file, `false` if this is a secondary file.
- `isScriptMode()` answers if this is the main file of a module.
- `getScope()` returns the root of the scope tree for unqualified lookup.

Imported and serialized modules

Support for imported and serialized modules can be found in a pair sub-directories:

- `lib/ClangImporter/`
- `lib/Serialization/`

AST printer

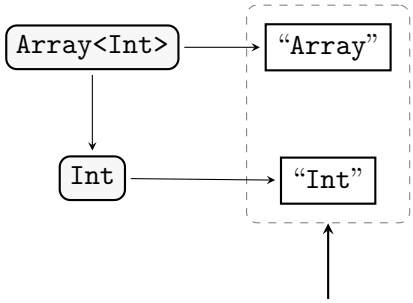
The AST printer generates textual interface files:

- `include/swift/AST/ASTPrinter.h`
- `lib/AST/ASTPrinter.cpp`

3. Types

Reasoning about types is a central concern in the implementation of a statically typed language. In Swift, various syntactic forms such as `Int`, `Array<String>`, and `(Bool) -> ()` denote references to types. A *type representation* is the syntactic form of a type annotation written in source, as constructed by the parser. A *type* is a higher-level semantic object. Types are constructed from type representations by *type resolution*. They can also be built and taken apart directly.

A type representation:



Suppose the text "`Array<Int>`" appears in a valid position of the language grammar. First, the lexer splits the input text into the tokens "`Array`", "`<`", "`Int`", and "`>`". The parser inspects each token and builds a type representation.

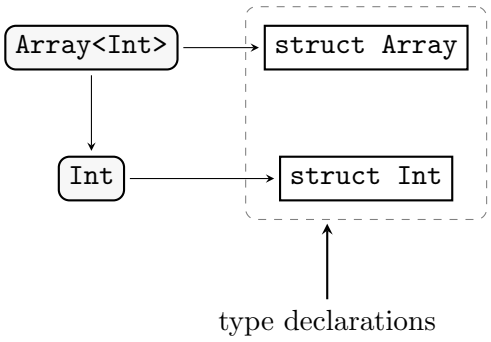
The parsed type representation `Array<Int>` has a tree structure, combining the identifier "`Array`" with the type representation `Int`. The latter is a leaf node just storing an identifier. As syntactic objects, type representations only store identifiers, which bear no relation to actual type declarations. The "shape" of a tree node is determined by the type representation's *kind*, corresponding to syntactic productions.

To resolve this type representation, we first use name lookup to find the type declarations for `Array` and `Int`. From these, we form the generic nominal type `Array<Int>`.

The generic nominal type `Array<Int>` points at the `Array` type declaration, and contains a child node for its generic argument, which is the type `Int`. The latter type also points at the declaration of `Int`, and not just an identifier.

Types are categorized by *kind*, just like type representations. Each kind has a constructor operation to form a new type from *structural components*, and corresponding operations to take an existing type apart.

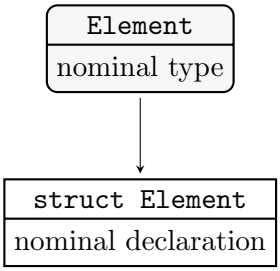
A type:



Categorization by kind. Type representation kinds are given by various productions in the grammar, such as identifiers, function type representations "`(Int) -> ()`", tuple type representations "`(Bool, String)`", and so on. The latter two kinds resolve to function types and tuple types, but type kinds are finer grained than type representation kinds in general, as we can see if we consider identifier type representations. An identifier type representation "`Element`" can resolve to one of several kinds of types; we consider a few now.

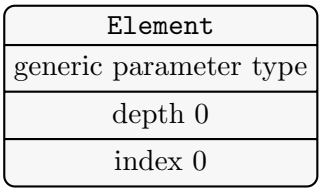
One possibility is that the identifier names a nominal type declaration. In this case, the resolved type is a **nominal type** pointing at this declaration.

```
struct Element {...}
var x: Element
```



The type checker can then perform further qualified lookups to find members of "`x`", query the stored properties of the struct to determine its layout, and so on.

The identifier might instead name a generic parameter from an outer scope, in which case the resolved type is a **generic parameter type**.

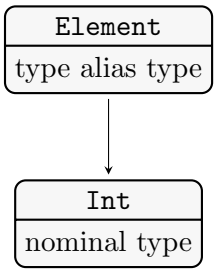



```
struct Container<Element> {
  var x: Element
}
```

Generic parameters are declared in a generic parameter list attached to a declaration. They are scoped to the body of this declaration, and can be uniquely identified by a pair of integers, the *depth* and *index*. Generic parameter types can also store a name, but the name only has significance when printed in diagnostics.

Yet another possibility is that the identifier names a type alias declaration. In this case, a special **type alias type** is returned, which wraps the *underlying type* `Int`. It behaves like `Int` in all ways, except that it can be printed back as "`Element`".

```
typealias Element = Int
var x: Element
```



Outside of type resolution, type representations do not play a big role in the compiler, so we punt on the topic of type representations until Chapter 9 and just focus on types for now. For our current purposes, it suffices to say that type resolution is really just one possible mechanism by which types are constructed. The expression checker builds types by solving a constraint system, and the generics system builds types via substitution, to give two examples.

Structural components. A type is constructed from structural components, which may either be other types, or non-type information. Common examples include: nominal types, which consist of a pointer to a declaration, together with a list of generic argument types; tuple types, which have element types and labels; and function types, which contain parameter types, return types, and various additional bits like `@escaping` and `inout`. We will give a full accounting of all type kinds and their structural components in the second half of this chapter.

Once created, types are immutable. To say that a type *contains* another type means that the latter appears as a structural component of the former, perhaps nested several levels deep. We will often talk about *replacing* a type contained by another type. This is understood as constructing a new type with the same kind as the original type, preserving all structural components except for the one being replaced. The original type is never mutated directly.

More generally, types can be transformed by taking the type apart by kind, recursively transforming each structural component, and forming a new type of the same kind from the new components. To preview Chapter 6, if `Element` is a generic parameter type, the type `Array<Int>` can be formed from `Array<Element>` by replacing `Element` with `Int`; this is called *type substitution*. The compiler provides various utilities to simplify the task of implementing recursive walks and transformations over kinds of types; type substitution is one example of such a transformation.

Canonical types. It is possible for two types to differ in their spelling, and yet be equivalent semantically:

- The Swift language defines some shorthands for common types, such as `T?` for `Optional<T>`, `[T]` for `Array<T>`, and `[K: V]` for `Dictionary<K, V>`.
- Type alias declarations introduce a new name for some existing underlying type, equivalent to writing out the underlying type in place of the type alias. The standard library, for example, declares a type alias `Void` with underlying type `()`.
- Another form of fiction along these lines is the preservation of generic parameter names. Generic parameter types written in source have a name, like "`Element`," and should be printed back as such in diagnostics, but internally they are uniquely identified in their generic signature by a pair of integers, the depth and the index. This is detailed in Section 4.1.

These constructions are the so-called *sugared types*. A sugared type has a desugaring into a more primitive form in terms of its structural components. The compiler constructs type sugar in type resolution, and attempts to preserve it as much as possible when transforming types. Preserving sugar in diagnostics can be especially helpful with more complex type aliases and such.

A *canonical type* is a type that does not recursively contain any sugared types. Type sugar has no semantic effect, for the most part. For example, it would not make sense to define two overloads of the same function that only differ by sugared types. For this reason, many operations on types compute the canonical type first to avoid having to consider sugared types in case analysis. After type checking, compiler passes such as SILGen and IRGen only deal with canonical types. The Swift runtime reifies canonical types as runtime type metadata.

The compiler can transform an arbitrary type into a canonical type by the process of *canonicalization*, which recursively replaces sugared types with their desugared form; in this way, `[(Int?, Void)]` becomes `Array<(Optional<Int>, ())>`. This operation is very cheap; each type caches a pointer to its canonical type, which is computed as needed (so types are not completely immutable, as we said previously; but the mutability cannot be observed from outside).

One notable case where the type checker's behavior *does* depend on type sugar is with default initialization of variables: if a variable's type is declared as the sugared optional type `T?` for some `T`, the variable's initial value expression is assumed to be `nil` if none was provided. Spelling the type as `Optional<T>` avoids the default initialization behavior:

```
var x: Int?
print(x)  // prints 'nil'

var y: Optional<Int>
print(y)  // error: use of uninitialized variable 'y'
```

Another exception is requirement inference with a generic type alias (Section 11.1).

Type equality. Types are uniquely allocated, which is made possible by them being immutable. A type `(Int) -> ()` has a unique pointer identity within a compilation; inside the tuple type `((Int) -> (), (Int) -> ())`, the two element types have the same pointer value in memory. From this, three levels of equality are defined on types:

1. **Type pointer equality** checks if two types are exactly equal as trees.
2. **Canonical type equality** checks if two types are equal after sugar is removed.
3. **Reduced type equality** checks if two types have the same reduced type with respect to a generic signature, in a sense we will make precise.

Each level of equality implies the next, so (1) ⇒ (2) and (2) ⇒ (3). If both types are canonical, then (1) and (2) coincide; if both are reduced, (1), (2) and (3) all coincide. Type pointer equality is only infrequently used, precisely because it is too strict; we usually do not want to consider two types to be distinct if they only differ by sugar.

Figure 3.1.: Some examples of type equality

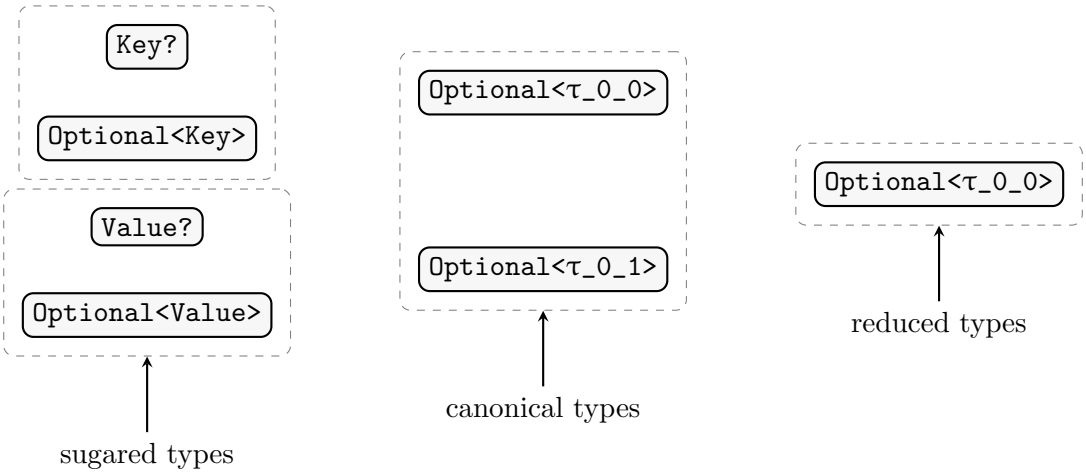


Figure 3.1 demonstrates type equality in the following extension declaration, where the `Key` (`τ_0_0`) and `Value` (`τ_0_1`) generic parameters of `Dictionary` are declared equivalent with a same-type requirement:

```
extension Dictionary where Key == Value {
  func foo(a: Key?, b: Optional<Key>, c: Value?, d: Optional<Value>) {}
}
```

Consider the four types `Key?`, `Optional<Key>`, `Value?`, and `Optional<Value>`:

- All four are distinct under type pointer equality.
- The first two have the canonical type `Optional<τ_0_0>`. Thus, the first two are equal under canonical type equality.
- The last two have the canonical type `Optional<τ_0_1>`. Again, they are equal to each other under canonical type equality (but distinct from the first two).
- When we consider the generic signature of the extension, we're left with just one reduced type, because both canonical types reduce to `Optional<τ_0_0>` via the same-type requirement. Thus, all four of the original types are equal under reduced type equality.

Reduced type equality means "equivalent as a consequence of one or more same-type requirements." We will define this equivalence of type parameters in Section 5.3 using the derived requirements formalism, and then generalize to all interface types in Section 5.6. Presenting a decision procedure for reduced type equality is one of our main results in this book; key developments take place in Section 16.5 and Chapter 17.

3.1. Fundamental Types

We've looked at some behaviors of types in general, and informally introduced a few kinds. We now give a full accounting of all kinds of types, starting from those most important from the point of view of the generics implementation.

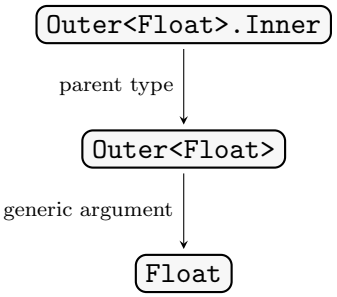
Nominal types. A *nominal type* is declared by a non-generic struct, enum or class declaration, such as `Int`. A *generic nominal type* is declared by a generic struct, enum or class declaration. They have these structural components:

- A pointer to the nominal type declaration.
- A parent type, if the nominal type declaration is nested inside of another nominal type declaration.
- A list of generic arguments, if the nominal type declaration is generic.

The parent type records the generic arguments of outer nominal type declarations. For example, with the below declarations, we can form the nominal type `Outer<Float>.Inner`:

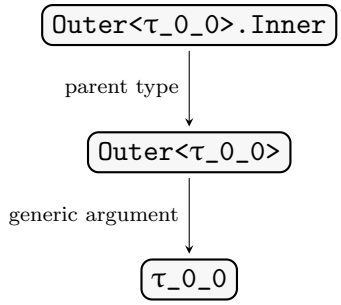
```
struct Outer<T> {
  struct Inner {}
}
```

Nominal type nesting



A nominal type declaration defines a family of nominal types, all sharing the same declaration and parent type structure, but having different generic arguments. Each of these types is a *specialized type* of the nominal type declaration.

Declared interface type



The specialized type where each generic argument is set to the corresponding generic parameter type is the *declared interface type* of the nominal type declaration. It is "universal": any specialized type is obtainable from the declared interface type by assigning a replacement type to each generic parameter type. This assignment is known as a *substitution map*, and the substitution map defined by the generic arguments of a specialized type is its *context substitution map* (Section 6.1). Finally, in the case where neither the nominal type declaration nor any of its parents are generic, the declaration defines just one *fully-concrete* specialized type, with an empty context substitution map.

Generic parameter types. A generic parameter type abstracts over a generic argument provided by the caller. Generic parameter types are declared by generic parameter declarations, described in Section 4.1. The sugared form references the declaration, and prints as the declaration's name; the canonical form only stores a depth and an index. Care must be taken not to print canonical generic parameter types in diagnostics, to avoid surfacing the "`τ_1_2`" notation to the user. (We will show how to transform a canonical generic parameter type into its sugared form at the end of Section 5.7.)

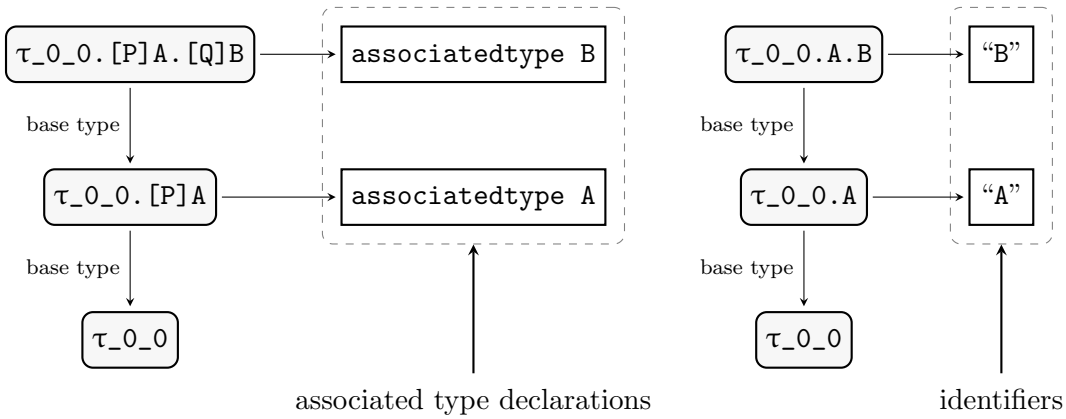
Dependent member types. A dependent member type abstracts over a concrete type that fulfills an associated type requirement. It has two structural components:

- A base type, which is a generic parameter type or another dependent member type.
- An identifier (in which case this is an *unbound* dependent member type), or an associated type declaration (in which case it is *bound*).

In Chapter 9, we describe the two stages of type resolution. Unbound dependent member types appear in the structural resolution stage, when we resolve the requirements in the `where` clause to feed into the generic signature construction procedure. Once we have a generic signature, we move on to interface resolution stage, and dependent member types written elsewhere are fully resolved into their bound form.

If `T` is the base type, `P` is a protocol and `A` is an associated type declared inside this protocol, we denote the bound dependent member type by `T.[P]A`, and the unbound dependent member type by `T.A`. The base type `T` can be another dependent member type, so we get a recursive structure like `τ_0_0.[P]A.[Q]B`. Figure 3.2 illustrates this with two dependent member types, bound and unbound.

Figure 3.2.: Bound and unbound dependent member types



A *type parameter* is a generic parameter type or dependent member type (whose base is then another type parameter). An *interface type* is a type that might contain type parameters, but is not necessarily a type parameter itself. A significant portion of this book is devoted to understanding type parameters. Key topics include:

- Semantic validity of type parameters (Section 5.2, Section 5.3).
- Generic signature queries (Section 5.6).
- Dependent member type substitution (Section 7.4, Chapter 12).
- Type resolution with bound and unbound type parameters (Chapter 9).

Finally, a note about terminology. This usage of "dependent" comes from C++, and this is not a type dependent on a value in the "lambda cube" sense.

Archetype types. Type parameters derive their meaning from the requirements of a generic signature; they are only "names" of external entities, in a sense. Archetypes are an alternate "self-describing" representation. Archetypes are instantiated from a *generic environment*, which stores a generic signature together with other information (Chapter 8). A *contextual type* is a type that might contain archetypes but is not necessarily an archetype itself.

Archetypes occur inside expressions and SIL instructions. Archetypes also represent references to opaque result types (Section 13.1) and the type of the payload inside of an existential (Section 14.1). In diagnostics, an archetype is printed as the type parameter it represents. We will denote by $[\![\texttt{T}]\!]$ the archetype for the type parameter `T` in some generic environment understood from context.

The fundamental type kinds we surveyed above—nominal types, type parameters, and archetypes—are *the Swift types that can conform to protocols*. In other words, they can satisfy the left-hand side of a conformance requirement, with the details for each type given later in Section 7.1. Also important are *constraint types*; these are *the types appearing on the right hand side of conformance requirements*. Constraint types themselves are never the types of value-producing expressions. (A value can have an existential type though, which wraps a constraint type, as we will see next.)

Protocol types. A protocol type is the most fundamental kind of constraint type; a conformance requirement involving any other kind of constraint type can always be split up into simpler conformance requirements. A protocol type is a kind of nominal type, so it will have a parent type if the protocol declaration is nested inside of another nominal type declaration. Unlike other nominal types, protocols cannot be nested in generic contexts (Section 6.2), so neither the protocol type itself nor any of its parents can have generic arguments. Thus, there is exactly one protocol type corresponding to each protocol declaration.

Protocol composition types. A protocol composition type may contain any number of protocol types, and at most one class type or `AnyObject` layout constraint:

```
P & Q
P & AnyObject
SomeClass<Int> & P
```

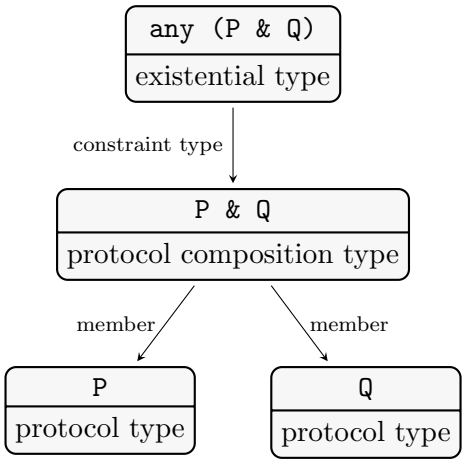
The empty protocol composition is spelled `Any`. If the right-hand side of a conformance requirement is a protocol composition type, we generate a series of simpler requirements, one for each member of the composition (Section 11.2). Swift 2.2 and prior used the syntax "`protocol<P, Q>`"; Swift 3 introduced the modern spelling [54]. So now `Any` is a special case, but of course it used to be "`typealias Any = protocol<>`".

Parameterized protocol types. A parameterized protocol type stores a protocol type together with a list of generic arguments. As a constraint type, it expands into a conformance requirement together with one or more same-type requirements, for each of the protocol's *primary associated types* (Section 4.3). The written representation looks just like a generic nominal type, except the named declaration is a protocol, for example, `Sequence<Int>`. Parameterized protocol types were introduced in Swift 5.7 [55] (the evolution proposal calls them "constrained protocol types").

3.2. More Types

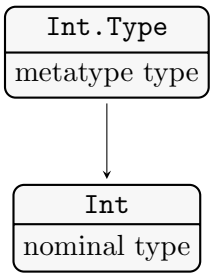
Various *structural types* can also be formed from other types. (Not to be confused with the types produced by the *structural resolution stage*, discussed in Chapter 9.)

Existential types. An existential type has one structural component, the *constraint type*. An existential value is a container holding a value of some unknown dynamic type that is known to satisfy the constraint; to the right we show the existential type `any (P & Q)`, which stores a value conforming to both `P` and `Q`.



The `any` keyword was added in Swift 5.6 [56]; in Swift releases prior, existential types and constraint types were the same concept in the language and implementation. (For the sake of source compatibility, a constraint type without the `any` keyword still resolves to an existential type except when it appears on the right-hand side of a conformance requirement.)

Existential types are covered in Chapter 14.

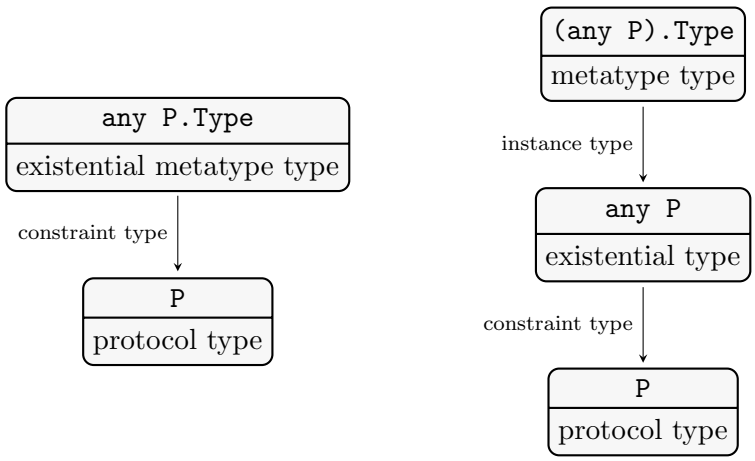


Metatype types. A type `T` can be the callee in a call expression, `T(...)`; this is shorthand for a constructor call `T.init(...)`. It can serve as the base of a static method call, `T.foo(...)`, where the type is passed as the `self` parameter. Finally, it can be directly referenced by the expression `T.self`. In all cases, the type becomes a *value*, and this value must itself be assigned a type; this type is called a *metatype*. The metatype of a type `T` is written as `T.Type`. The type `T` is the *instance type* of the metatype. For example, the type of the expression "`Int.self`" is the metatype `Int.Type`, whose instance type is `Int`. Metatypes are sometimes referred to as *concrete metatypes*, to distinguish them from existential metatypes, which we introduce below. Most concrete metatypes are singleton types with one value, the instance type itself. One exception is that the class metatype for a non-final class also has all subclasses of the class as values.

Existential metatype types. An existential metatype is a container for an unknown concrete metatype, whose instance type is known to satisfy the constraint type.

Existential metatypes are not the same as metatypes whose instance type is existential, as we can demonstrate by considering language semantics. If `P` is a protocol and `S` is a type conforming to `P`, then the existential metatype `any P.Type` can store the value `S.self`. As the existential type `any P` does not conform to `P`, the value `(any P).self` cannot be stored inside of the existential metatype `any P.Type`. In fact, the type of this value is the *concrete* metatype `(any P).Type`. Figure 3.3 compares the recursive structure of `any P.Type` against `(any P).Type`.

Figure 3.3.: Existential metatype and metatype of existential

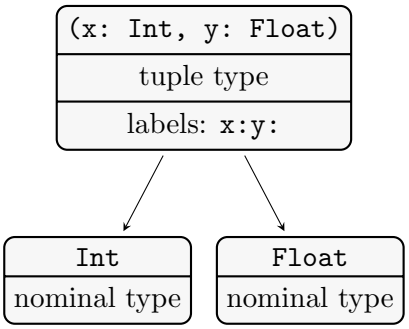


Prior to the introduction of the `any` keyword, existential metatypes were written as `P.Type`, and the concrete metatype of an existential as `P.Protocol`. This was a source of confusion, because when `T` is a non-protocol type, `T.Type` is otherwise always a concrete metatype. The below table compares the old and new syntax:

Old syntax	New syntax	Type kind
`(T).Type`	`(T).Type`	Concrete metatype (unchanged)
`P.Protocol`	`(any P).Type`	Concrete metatype of existential
`P.Type`	`any P.Type`	Existential metatype

Tuple types. A tuple type is a list of element types together with optional labels. If the list of element types is empty, we get the unique empty tuple type `()`.

An unlabeled one-element tuple type cannot be formed at all; `(T)` resolves to the same type as `T`. Labeled one-element tuple types `(foo: T)` are valid in the grammar, but are rejected by type resolution. SILGen creates them internally when it materializes the payload of an enum case (for instance, "`case person(name: String)`"), but they do not appear as the types of expressions.



Function types. A function type is the type of the callee in a call expression. It contains a parameter list, a return type, and non-type attributes. The attributes include the function's effect, lifetime, and calling convention. The effects are `throws` and `async` (the latter is from the Swift 5.5 concurrency model [57]). Values of function type with non-escaping lifetime are second-class; they can only be passed to another function, captured by a non-escaping closure, or called. Escaping functions can also be returned, or stored inside other values, as usual. The four calling conventions are:

- The default "thick" convention, where the function is passed as a function pointer together with a reference-counted closure context.
- `@convention(thin)`: The value is a single function pointer using the Swift calling convention, without a closure context. Thin functions cannot have captures.
- `@convention(c)`: The value is a single function pointer, and the parameter and return types must be representable in C. Once again, captures are not permitted.
- `@convention(block)`: The value is an Objective-C block. Captures are permitted, but the parameter and return types must be representable in Objective-C.

Each entry in the parameter list contains a parameter type and some non-type bits:

- The **ownership specifier**, whose possible values correspond to default ownership, `inout`, `borrowing`, or `consuming`.

  The `inout` kind is key to Swift's mutable value type model; the interested reader can consult [58] for details. The other two were introduced in Swift 5.9 [59].

- The **variadic** flag, in which case the parameter type must be an array type.

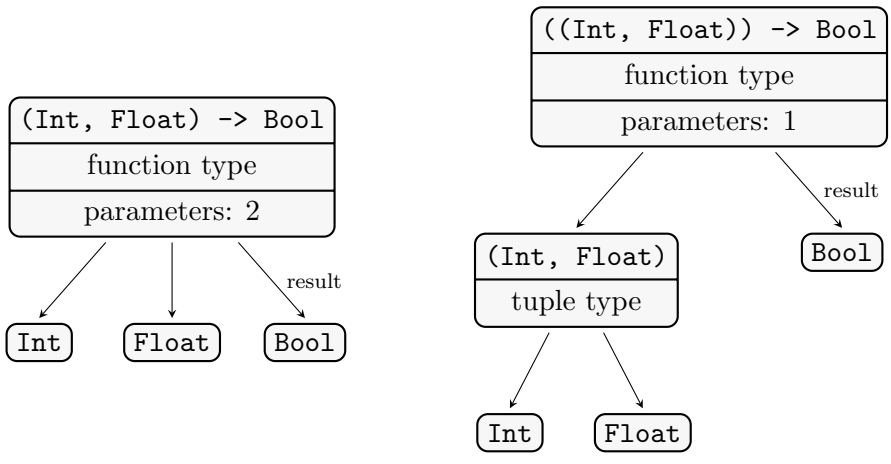
  When type checking a call to function value with a variadic parameter, the type checker collects multiple expressions from the call argument list into an implicit array expression. Otherwise, variadic parameters behave exactly like arrays once we get to SILGen and below.

- The `@autoclosure` attribute, in which case the parameter type must be another function type of the form `() -> T` for some type `T`.

  This instructs the type checker to treat the corresponding argument in the caller as if it was a value of type `T`, rather than a function type `() -> T`. The argument is then wrapped inside an implicit closure expression. In the body of the callee, an `@autoclosure` parameter behaves exactly like an ordinary function value, and can be called to evaluate the expression provided by the caller.

In functional languages such as ML and Haskell, all function types conceptually take a single parameter type. This is not the case in Swift, however. Figure 3.4 illustrates the distinction between `(Int, Float) -> Bool` and `((Int, Float)) -> Bool`; the first has two parameters, and the second has a single parameter that is of tuple type.

Figure 3.4.: Function type with two parameters, or a single tuple parameter



For convenience, the type checker defines an implicit conversion, called a "tuple splat," between function types taking a single tuple and function types of multiple arguments. This implicit conversion is only available when passing a function value as an argument to a call, and only when the function type's parameter list can be represented as a tuple (hence, it has no parameter attributes). An example:

```
func apply<T, U>(fn: (T) -> U, arg: T) -> U {
  return fn(arg)
}

// Tuple splat conversion:
// - the type of (+) is (Int, Int) -> (),
// - but apply() expects ((Int, Int)) -> ().
print(apply(fn: (+), arg: (1, 2)))
```

Another subtle point with function types is that argument labels are part of a function declaration's *name*, not a function declaration's *type*. A closure, being an anonymous function, is always called without argument labels. This includes a closure formed from an unapplied reference to a function declaration—even if the function declaration *does* have argument labels:

```
func subtract(minuend x: Int, subtrahend y: Int) -> Int {
  return x - y
}

print(subtract(minuend: 3, subtrahend: 1))  // prints 2

let fn1 = subtract  // declaration name can omit argument labels
print(fn1(3, 1))  // prints 2

let fn2 = subtract(minuend:subtrahend:)  // full declaration name
print(fn2(3, 1))  // prints 2
```

The history of Swift function types is an interesting case study in language evolution. Originally, Swift followed the classical functional language model, where a function type always had a *single* parameter type, which could be a tuple type to simulate a function of multiple parameters. Tuple types used to be able to contain `inout` and variadic elements, and furthermore, the argument labels of a function declaration were part of the function declaration's type. The existence of such "non-materializable" tuple types introduced complications throughout the type system, and argument labels had inconsistent behavior in different contexts.

The syntax for referencing a declaration name with argument labels was adopted in Swift 2.2 [60]. Subsequently, argument labels were dropped from function types in Swift 3 [61]. The distinction between a function taking multiple arguments and a function taking a single tuple argument was first hinted at in Swift 3 with [62] and [63], and became explicit in Swift 4 [64]. At the same time, Swift 4 also introduced the "tuple splat" function conversion to simulate the Swift 3 model for the cases where the old behavior was convenient. For example, the element type of `Dictionary` is a key/value pair, but often it is more convenient to call `Collection.map()` with a closure taking two arguments, instead of a single tuple argument.

Once the above proposals were implemented, the compiler continued to model function types as having a single input type for quite some time, despite this being completely hidden from the user. After Swift 5, the function type representation fully converged with the semantic model of the language.

Note that while metatypes, tuple types, and function types all an play an important role in the Swift language as a whole, they are not essential to the formal analysis of generics; one can construct a toy implementation of Swift generics with generic nominal types and type parameters alone. Indeed, from the viewpoint of the generics model, structural types have no intrinsic behaviors, other than possibly containing type parameters which can be substituted.

We finish this section by turning to sugared types. Sugared generic parameter types were already described in Section 3.1. Of the remaining kinds of sugared types, type alias types are defined by the user, and the other three are built-in to the language.

Type alias types. A type alias type represents a reference to a type alias declaration. It consists of an optional parent type, a list of generic arguments, and the substituted underlying type. The canonical type of a type alias type is the substituted underlying type. The parent type is only present if the type alias declaration is a member of another type declaration, and the generic arguments are only present if the type alias is generic (Section 9.1). The parent type and generic arguments are preserved by type resolution so that the type alias type can be printed faithfully, and also for requirement inference (Section 11.1).

Optional types. The optional type is written as `T?` for some payload type `T`; its canonical type is `Optional<T>`.

Array types. The array type is written as `[E]` for some element type `E`; its canonical type is `Array<E>`.

Dictionary types. The dictionary type is written as `[K: V]` for some key type `K` and value type `V`; its canonical type is `Dictionary<K, V>`.

3.3. Special Types

We now discuss a few of the remaining types, which are all weird in their own unique ways. They tend to only be valid in specific contexts, and some do not represent actual types of values at all. Their unexpected appearance can be a source of counterexamples and failed assertions. They all play important roles in the expression type checker, but again, do not really give us anything new if we consider the generics model from a purely formal viewpoint.

Generic function types. A generic function type has the same structural components as a function type, except it also stores a generic signature:

```
<S where S: Sequence> (S) -> S.Element
```

A generic function type represents the interface type of a generic function or subscript declaration, before substitutions are applied. An expression that references a generic function or subscript declaration will always apply substitutions first, so a *value* in the Swift language cannot have a generic function type. In particular, a generic function type cannot be a parameter or result of another function type; the Swift type system does not support *higher-rank polymorphism*. Type inference with higher-rank types is known to be undecidable; see [65] and [66].

Generic function types have a special behavior when their canonical type is computed. Since generic function types carry a generic signature, the parameter types and return type of a *canonical* generic function type are actually *reduced* types with respect to this generic signature (Section 5.5).

Reference storage types. A reference storage type is the type of a variable declaration adorned with the `weak`, `unowned`, or `unowned(unsafe)` attribute. These modifiers change the reference counting behavior of the class type, class-constrained archetype, or class-constrained existential type they are applied to. Reference storage types arise as the interface types of variable declarations, and as the types of values in SIL. The types of expressions never contain reference storage types.

Placeholder types. A placeholder type represents a generic argument to be inferred by the type checker. The written representation is the underscore "`_`". They can only appear in a handful of restricted contexts, and do not appear in the types of expressions or the interface types of declarations after type checking. The constraint solver replaces placeholder types with type variables when solving the constraint system. For example, here, the interface type of the `myPets` local variable is inferred as `Array<String>`:

```
let myPets: Array<_> = ["Zelda", "Giblet"]
```

Placeholder types were introduced in Swift 5.6 [67].

Unbound generic types. Unbound generic types predate placeholder types, and can be seen as a special case. An unbound generic type is written as a named reference to a generic type declaration, without generic arguments applied. An unbound generic type behaves like a generic nominal type where all generic arguments are placeholder types. In previous example, we wrote the generic nominal type `Array<_>` with a placeholder type. The unbound generic type `Array` could have been used instead:

```
let myPets: Array = ["Zelda", "Giblet"]
```

Unbound generic types are also occasionally useful in diagnostics, to print the name of a type declaration only (like `Outer.Inner`) without the generic parameters of its declared interface type (`Outer<T>.Inner<U>`, for example). We will discuss the behavior of type resolution with unbound generic types in Section 9.4.

Dynamic Self types. The dynamic `Self` type appears when a method in a class declares a return type of `Self`. This enforces a guarantee that the return value must have the same *dynamic* type as the `self` parameter passed in to the method call, which might be a subclass of the static type of `self`. The dynamic `Self` type has one structural component, a class type. The dynamic `Self` type behaves like an existential type in certain respects, but it's just a hard-coded behavior in the type checker and SILGen.

Note that the identifier "`Self`" only has this interpretation inside a class. In a protocol declaration, `Self` is the implicit generic parameter (Section 4.3). In a struct or enum declaration, `Self` is the declared interface type (Section 9.1).

Listing 3.1 demonstrates some of the behaviors of the dynamic `Self` type. Two invalid cases are shown; `invalid1()` is rejected because the type checker cannot prove that the return type is always an instance of the dynamic type of `self`, and `invalid2()` is rejected because `Self` appears in contravariant position. This feature comes from Objective-C, where it is called `instancetype`.

Type variable types. A type variable represents the future inferred type of an expression. The expression type checker builds a *constraint system* by walking an expression recursively, assigning new type variables to each sub-expression and then recording constraints to relate these type variables. The constraint solver then searches for a *solution* that assigns a concrete type to each type variable in a way that satisfies the constraints. Solving the constraint system can have three possible outcomes:

- **One solution**—the expression and all of its sub-expressions are well-typed.
- **No solutions**—the constraints cannot be satisfied, so the expression is invalid.
- **Multiple solutions**—the expression is ambiguous, because there is more than one possible assignment of concrete types that satisfies the constraints.

Listing 3.1.: Dynamic `Self` type example

```
class Base {
  required init() {}

  func dynamicSelf() -> Self {
    // the type of ‘self’ in a method returning ‘Self’ is
    // the dynamic Self type.
    return self
  }

  func clone() -> Self {
    return Self()
  }

  func invalid1() -> Self {
    return Base()  // error
  }

  func invalid2(_: Self) {}  // error
}

class Derived: Base {}

let y = Derived().dynamicSelf()  // y has type ‘Derived’
let z = Derived().clone()  // z has type ‘Derived’
```

If there is exactly one solution, we update each expression in the AST with its concrete type. In the case of multiple solutions, we attempt to rank the solutions using a heuristic; if one is clearly "better" than the others, we proceed as in the case of one solution. If there was no solution, or the are multiple ambiguous solutions and none is better than the rest, we diagnose an error.

Type variable types and the structures that contain them cannot outlive the the *constraint solver arena*, whose lifetime is scoped to the invocation of the expression type checker on a single expression. Structures that *contain* type variables, such as other types and substitution maps, are also allocated in the constraint solver arena. For this reason, type variable types must never "escape" from the constraint solver arena. Assertions are used to rule out unexpected appearance of type variable types.

The printed representation of a type variable is `$Tn`, where `n` is an incrementing integer local to the constraint system. One way to see type variables and constraint solving in action is to pass the `-Xfrontend -debug-constraints` compiler flag. To learn more about the expression type checker, see [68, 69].

L-value types. An l-value type represents the type of an expression appearing on the left hand side of an assignment operator (hence the "l" in l-value), or as an argument to an `inout` parameter in a function call. L-value types wrap an *object type* which is the type of the stored value; they print out as `@lvalue T` where `T` is the object type, but this is not valid syntax in the language.

L-value types appear in type-checked expressions. The reader familiar with C++ might think of an l-value type as somewhat analogous to a C++ mutable reference type "`T &`"—unlike C++ though, they are not directly visible in the source language. L-value types do not appear in the types of SIL instructions; SILGen lowers l-value accesses into accessor calls or direct manipulation of memory.

Error types. An error type represents an unknown type in an erroneous program. There are two forms that an error type may take: it either wraps some other type, or it is the so-called singleton error type. In some sense, the error type resembles the "Not a Number" value of floating point arithmetic.

Type substitution returns an error type when it encounters an invalid conformance (Chapter 7). In this case, the error type wraps the original type being substituted, so that it prints as the original type, to make types coming from malformed conformances more readable in diagnostics.

Type resolution returns the singleton error type when the type representation being resolved is invalid in some way. The singleton error type prints as `<<error type>>`. To avoid user confusion, diagnostics containing the singleton error type should not be emitted. Generally, any expression whose type contains an error type does not need to be diagnosed, because a diagnostic will have been emitted elsewhere.

Built-in types. What users think of as fundamental types, such as `Int` and `Bool`, are defined as structs in the standard library. These structs wrap the various *built-in types* which are understood directly by the compiler. Built-in types are not nominal types, so they cannot contain members, cannot have new members added via extensions, and cannot conform to protocols. Values of built-in types are manipulated using special *compiler intrinsics*. The standard library wraps built-in types in nominal types, and defines methods and operators on those nominal types which call the intrinsic functions, thereby presenting the actual interface expected by users.

For example, the `Int` struct defines a single stored property named `_value` with type `Builtin.Int`. The `+` operator on `Int` is implemented by extracting the `_value` stored property from a pair of `Int` values, calling the `Builtin.sadd_with_overflow_Int64` compiler intrinsic to add them together, and finally, wrapping the resulting `Builtin.Int` in a new instance of `Int`.

Built-in types and their intrinsics are defined in the special `Builtin` module, which is a module constructed by the compiler itself, and not built from source code. The `Builtin` module is only visible when the compiler is invoked with the `-parse-stdlib` frontend flag; the standard library is built with this flag, but user code never interacts with the `Builtin` module directly.

3.4. Source Code Reference

Key source files:

- include/swift/AST/Type.h
- include/swift/AST/Types.h
- lib/AST/Type.cpp

Other source files:

- include/swift/AST/TypeNodes.def
- include/swift/AST/TypeVisitor.h
- include/swift/AST/CanTypeVisitor.h

`Type`	*class*

Represents an immutable, uniqued type. Meant to be passed as a value, it stores a single instance variable, a `TypeBase *` pointer.

The `getPointer()` method returns this pointer. The pointer is not `const`, however neither `TypeBase` nor any of its subclasses define any mutating methods. The pointer

may be `nullptr`; the default constructor `Type()` constructs an instance of a null type. Most methods will crash if called on a null type; only the implicit `bool` conversion and `getPointer()` are safe.

The `getPointer()` method is only used occasionally, because types are usually passed as `Type` and not `TypeBase *`, and `Type` overloads `operator->` to forward method calls to the `TypeBase *` pointer. While most operations on types are actually methods on `TypeBase`, a few methods are also defined on `Type` itself (these are called with "`.`" instead of "`->`").

- **Various traversals:** `walk()` is a general pre-order traversal where the callback returns a tri-state value—continue, stop, or skip a sub-tree. Built on top of this are two simpler variants; `findIf()` takes a boolean predicate, and `visit()` takes a void-returning callback which offers no way to terminate the traversal.
- **Transformations:** `transformWithPosition()` and `transformRec()`. The first is the most general, and the other one changes the signature of the callback to ignore a `TypePosition` parameter. The callback is invoked on all types contained within a type, recursively. It can either elect to replace a type with a new type, or leave a type unchanged and instead try to transform any of its child types.
- **Substitution:** `subst()` implements type substitution, which is a particularly common kind of transform which replaces generic parameters or archetypes with concrete types (Section 6.6).
- **Printing:** `print()` outputs the string form of a type, with many customization options; `dump()` prints the tree structure of a type in an s-expression form. The latter is extremely useful for invoking from inside a debugger, or ad-hoc print debug statements.

The `Type` class explicitly deletes the overloads of `operator==` and `operator!=` to make the choice between pointer and canonical equality explicit. To check type pointer equality of possibly-sugared types, first unwrap both sides with a `getPointer()` call, and then compare the `TypeBase *` values for equality:

```
if (lhsType.getPointer() == rhsType.getPointer())
  ...;
```

Canonical type equality is more commonly used. To check for canonical type equality, call the `isEqual()` method on `TypeBase`. Unlike the pointer equality check, the below only works if both types are non-`nullptr`:

```
if (lhsType->isEqual(rhsType))
  ...;
```

`TypeBase` *class*

The root of the type kind hierarchy. Its instances are always uniqued and allocated by the AST context, either the permanent arena or constraint solver arena. Instances are usually wrapped in `Type`. The various subclasses correspond to the different kinds of types:

- `NominalType` and its four subclasses:
  - `StructType`,
  - `EnumType`,
  - `ClassType`,
  - `ProtocolType`.
- `BoundGenericNominalType` and its three subclasses:
  - `BoundGenericStructType`,
  - `BoundGenericEnumType`,
  - `BoundGenericClassType`.
- `GenericTypeParamType`, `DependentMemberType`, the two type parameter types.
- `ArchetypeType`, and its three subclasses:
  - `PrimaryArchetypeType`,
  - `ExistentialArchetypeType`,
  - `OpaqueArchetypeType`.
- Constraint types:
  - `ProtocolCompositionType`,
  - `ParameterizedProtocolType`,
- Existential types:
  - `ExistentialType`,
  - `ExistentialMetatypeType`,
- The structural types `TupleType`, `MetatypeType`.
- `AnyFunctionType` and its two subclasses:
  - `FunctionType`,
  - `GenericFunctionType`.

- `SugarType` and its four subclasses:
  - `TypeAliasType`,
  - `OptionalType`,
  - `ArrayType`,
  - `DictionaryType`.
- `BuiltinType` and its subclasses (there are a bunch of esoteric ones; only a few are shown below):
  - `BuiltinRawPointerType`,
  - `BuiltinVectorType`,
  - `BuiltinIntegerType`,
  - `BuiltinIntegerLiteralType`,
  - `BuiltinNativeObjectType`,
  - `BuiltinBridgeObjectType`.
- `ReferenceStorageType` and its two subclasses:
  - `WeakStorageType`,
  - `UnownedStorageType`.
- Everything else:
  - `DynamicSelfType`,
  - `UnboundGenericType`,
  - `PlaceholderType`,
  - `TypeVariableType`,
  - `LValueType`,
  - `ErrorType`.

Each concrete subclass defines some set of static factory methods, usually named `get()` or similar, which take the structural components and construct a new, uniqued type of this kind. There are also getter methods, prefixed with `get`, which project the structural components of each kind of type. It would be needlessly duplicative to list all of the getter methods for each subclass of `TypeBase`; they can all be found in `include/swift/AST/Types.h`.

Desugaring casts. Subclasses of `TypeBase *` are identifiable at runtime via the `is<>`, `castTo<>` and `getAs<>` template methods.

To check if a type has a specific kind, use `is<>`:

```
Type type = ...;

if (type->is<FunctionType>())
  ...;
```

To conditionally cast a type to a specific kind, use `getAs<>`, which returns `nullptr` if the cast fails:

```
if (FunctionType *funcTy = type->getAs<FunctionType>())
  ...;
```

Finally, to assert that a type has the required kind, use `castTo<>`:

```
FunctionType *funcTy = type->castTo<FunctionType>();
```

These template methods desugar the type if it is a sugared type, and they can never cast *to* a sugared type. This is usually what is intended, because type sugar is not meant to have any effect behavior. For example, if `type` is the `Swift.Void` type alias type, then `type->is<TupleType>()` returns true, because it is for all intents and purposes a tuple (an empty tuple), except when printed in diagnostics.

Direct casts. Alternatively, there are also the three top-level template functions `isa<>`, `dyn_cast<>`, and `cast<>`, that operate on `TypeBase *`. Using these with `Type` is an error; the pointer must be explicitly unwrapped with `getPointer()` first. These casts do not desugar, and thus they permit casting to a sugared type. This is the mechanism used when sugared types must be distinguished from canonical types for some reason:

```
Type type = ...;

if (isa<OptionalType>(type.getPointer()))
  ...;
```

Canonical types. The `getCanonicalType()` method outputs a `CanType` wrapping the canonical form of this `TypeBase *`. The canonical type is computed once and memoized, so this operation is cheap. The actual computation is done in `computeCanonicalType()`. Finally, the `isCanonical()` method checks if a type is already canonical.

Visitors. The simplest way to exhaustively handle each kind of type is to switch over the kind, which is an instance of the `TypeKind` enum, like this:

```
Type ty = ...;
switch (ty->getKind()) {
case TypeKind::Struct: {
  auto *structTy = ty->castTo<StructType>();
  ...
}
case TypeKind::Enum:
case TypeKind::Class:
  ...
}
```

However, in most cases it is more convenient to use the *visitor pattern* instead, by subclassing `TypeVisitor` and overriding various `visit`*`Kind`*`Type()` methods. Then, invoking the visitor's `visit()` method with a type performs the same switch and dynamic cast dance above:

```
class MyVisitor: public TypeVisitor<MyVisitor> {
public:
  void visitStructType(StructType *ty) {
    ...
  }
};

MyVisitor visitor;

Type ty = ...;
visitor.visit(ty);
```

The `TypeVisitor` also defines various methods corresponding to abstract base classes in the `TypeBase` hierarchy, so, for example, you can override `visitNominalType()` to handle all nominal types at once.

The `TypeVisitor` preserves information if it receives a sugared type; for example, visiting `Int?` will call `visitOptionalType()`, while visiting `Optional<Int>` will call `visitBoundGenericEnumType()`. In the common situation where the semantics of your operation do not depend on type sugar, you can use the `CanTypeVisitor` template class instead. Here, the `visit()` method takes a `CanType`, so `Int?` will need to be canonicalized to `Optional<Int>` before being passed in.

Nominal types. The below methods on `TypeBase` perform a desugaring cast to a nominal type (so they will also accept a type alias type or other sugared type), and then they return the nominal type declaration, or `nullptr` if the type isn't of a nominal kind:

- `getAnyNominal()` returns the nominal type declaration of `UnboundGenericType`, `NominalType` or `BoundGenericNominalType`, or `nullptr`.
- `getNominalOrBoundGenericNominal()` returns the nominal type declaration of a `NominalType` or `BoundGenericNominalType`, or `nullptr`.
- `getStructOrBoundGenericStruct()` returns the type declaration of a `StructType` or `BoundGenericStructType`, or `nullptr`.
- `getEnumOrBoundGenericEnum()` returns the enum declaration of an `EnumType` or `BoundGenericEnumType`, or `nullptr`.
- `getClassOrBoundGenericClass()` returns the class declaration of a `ClassType` or `BoundGenericClassType`, or `nullptr`.
- `getNominalParent()` returns the parent type stored by an `UnboundGenericType`, `NominalType` or `BoundGenericNominalType`, or `nullptr`.

Recursive properties. Various predicates determine if a type recursively contains other types with certain properties. These are computed when a type is constructed and are therefore cheap to check:

- `hasTypeVariable()` determines whether the type was allocated in the permanent arena or the constraint solver arena.
- `hasPrimaryArchetype()`, `hasOpaqueArchetype()`, `hasOpenedExistential()`.
- `hasTypeParameter()`.
- `hasUnboundGenericType()`, `hasDynamicSelf()`, `hasPlaceholder()`.
- `hasLValueType()` determines whether the type contains an l-value type.

Utility operations. These encapsulate frequently-useful patterns.

- `getOptionalObjectType()` returns the payload type `T` of a generic nominal type `Optional<T>` that was formed from the `Optional` enum declaration in the standard library. For any other kind of type, returns the null type.
- `getMetatypeInstanceType()` returns the type `T` if this type is `T.Type`, otherwise it returns `T` (and not the null type).
- `mayHaveMembers()` checks if this is a nominal type, archetype, existential type, or dynamic Self type.

Recovering the AST context. All non-canonical types point at their canonical type, and canonical types point at the AST context.

- `getASTContext()` returns the singleton AST context from a type.

`CanType` *class*

The `CanType` class wraps a `TypeBase *` pointer which is known to be canonical. The pointer can be recovered with the `getPointer()` method. It forwards various methods to either `Type` or `TypeBase *`. There is an implicit conversion from `CanType` to `Type`. In the other direction, the explicit one-argument constructor `CanType(Type)` asserts that the type is canonical; however, most of the time the `getCanonicalType()` method on `TypeBase` is used instead.

The `operator==` and `operator!=` operators are used to test `CanType` for type pointer equality. The `isEqual()` method described earlier implements canonical equality on sugared types by first canonicalizing both sides, and then checking the resulting canonical types for type pointer equality. Therefore, the following are equivalent:

```
if (lhsType->isEqual(rhsType)) ...;
if (lhsType->getCanonicalType() == rhsType->getCanonicalType()) ...;
```

The `CanType` class can be used with the `isa<>`, `cast<>` and `dyn_cast<>` templates. Instead of returning the actual `TypeBase` subclass, the latter two return a *canonical type wrapper* for that subclass. Every subclass of `TypeBase` has a corresponding canonical type wrapper; if the subclass is named `FooType`, the canonical wrapper is named `CanFooType`. Canonical type wrappers forward `operator->` to the specific `TypeBase` subclass, and define methods of their own (called with ".") which project the known-canonical components of the type.

For example, `FunctionType` has a `getResult()` method returning `Type`, so the canonical type wrapper `CanFunctionType` has a `getResult()` method returning a `CanType`. The wrapper methods are not exhaustive, and their use is not required because you can instead make explicit calls to `CanType(Type)` or `getCanonicalType()` after projecting a type that is known to be canonical.

```
CanType canTy = ...;
CanFunctionType canFuncTy = cast<FunctionType>(canTy);

// method on CanFunctionType: returns CanType(canFuncTy->getResult())
CanType canResultTy = canFuncTy.getResult();

// operator-> forwards to method on FunctionType: returns Type
CanType resultTy = CanType(canFuncTy->getResult());
```

`AnyFunctionType` *class*

This is the base class of `FunctionType` and `GenericFunctionType`.

- `getParams()` returns an array of `AnyFunctionType::Param`.
- `getResult()` returns the return type.
- `getExtInfo()` returns an instance of `AnyFunctionType::ExtInfo` storing the additional non-type attributes.

`AnyFunctionType::Param` *class*

This represents a parameter in a function type's parameter list.

- `getPlainType()` returns the type of the parameter. If the parameter is variadic (`T...`), this is the element type `T`.
- `getParameterType()` same as above, but if the parameter is variadic, returns the type `Array<T>`.
- `isVariadic()`, `isAutoClosure()` are the special behaviors.
- `getValueOwnership()` returns an ownership specifier encoded as an instance of the `ValueOwnership` enum.

`ValueOwnership` *enum class*

The return type of `AnyFunctionType::Param::getValueOwnership()`.

- `ValueOwnership::Default`
- `ValueOwnership::InOut`
- `ValueOwnership::Shared`
- `ValueOwnership::Owned`

`AnyFunctionType::ExtInfo` *class*

This represents the non-type attributes of a function type.

4. Declarations

Declarations are the building blocks of Swift programs. In Chapter 2, we saw that the user's entire program is a *module declaration* at the root of a hierarchy, with *file units* as immediate children. Now, we will see that a file unit holds a list of *top-level declarations*, which correspond to the main divisions in a source file, and these declarations can have further declarations nested within them. We will investigate the different kinds of declarations into a taxonomy, as we did with types in Chapter 3. Then, we focus on the syntactic representations for declaring generic parameters and stating requirements, which are common to all generic declarations. Finally, we end the chapter with a discussion of functions, closures, and captured values.

We begin with two major divisions in the declaration taxonomy:

1. A *value declaration* is a one that can be referenced by name from an expression; this includes variables, functions, and such. Every value declaration has an *interface type*, which is the type we assign to an expression naming this declaration.

2. A *type declaration* is one that can be referenced by name from within a type representation. This includes structs, type aliases, and so on. A type declaration declares a type, called the *declared interface type* of the type declaration.

Not all declarations are value declarations. An extension declaration adds members to an existing nominal type declaration, as we'll see in Chapter 10, but an extension does not itself have a name. A *top-level code declaration* holds the statements and expressions written at the top level of a source file, and again, it does not have a name, semantically.

Declaration contexts. Every declaration is contained in a *declaration context*, and a declaration context is anything that *contains* declarations. Consider this program:

```
func squares(_ nums: [Int]) -> [Int] {
  return nums.map { x in x * x }
}
```

The parameter declaration "`x`" is a child of the closure expression "`{ x in x * x }`", and not a direct child of the enclosing function declaration. So a closure expression is a declaration context, but not a declaration. On the other hand, a parameter declaration is a declaration, but not a declaration context. Finally, the `squares()` function itself is both a declaration, and a declaration context.

Type declarations. Since the Swift grammar allows type representations to appear inside expressions, every type declaration is *also* a value declaration. The interface type of a type declaration is the metatype formed from its declared interface type. This is a mouthful, but the basic idea should be familiar to every Swift programmer. Consider a global variable with a type annotation and an initial value:

```
struct Horse {}
let myHorse: Horse = Horse()
```

The struct declaration `Horse` is referenced twice here, first in the type annotation that follows "`:`", and then again in the initial value expression that follows "`=`". Inside the type annotation, `Horse` means the *declared interface type* of `Horse`, which is simply the nominal type `Horse`; we're declaring that the `myHorse` variable stores a value whose type has the stated name. The second reference to `Horse`, from the initial value expression, refers to *the type itself* as a value, so this uses its *interface type*, which is the metatype `Horse.Type`. (Recall the diagram from Section 3.2.) Furthermore, this metatype value is the callee of a call expression, which is a shorthand for calling a constructor member named `init`. We can write this out as follows to be even more explicit:

```
let myHorseType: Horse.Type = Horse.self
let myHorse: Horse = myHorseType.init()
```

Nominal type declarations. Declared by the `struct`, `enum`, and `class` keywords; Swift 5.5 also added `actor`, which to us, is just a class [70]. Nominal type declarations are declaration contexts, and the declarations they contain are called their *member declarations*. A function member declaration is commonly called a *method*, a member variable is a *property*, and a *member type declaration* is exactly that.

Structs and classes can contain a special kind of property declaration called a *stored property declaration*. Struct values directly store their stored properties, while a class value is a reference to a heap allocated box that contains its stored properties. A value of enum type stores exactly one element among several; instead of stored properties, enum declarations contain *enum element declarations*, introduced with the `case` keyword.

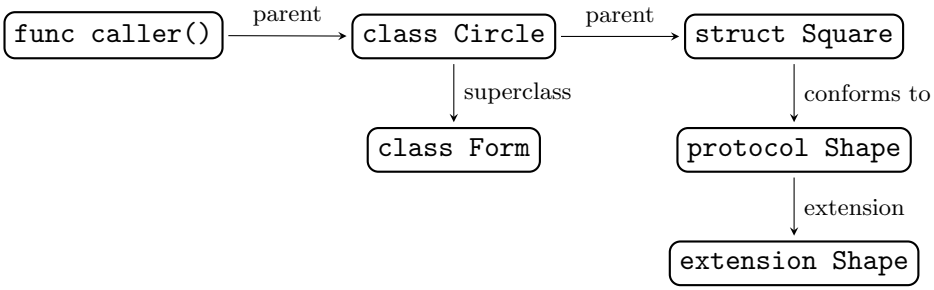
The members of a nominal type declaration are visible to name lookup (Section 2.1), both in the nominal type declaration's scope (unqualified lookup) and outside (qualified lookup). Listing 4.1 shows three features we will discuss in detail later:

- A class can inherit from a superclass type, and members of the superclass are also visible from the subclass (Section 6.4).
- Nominal type declarations can conform to protocols (Chapter 7).
- Extensions add members to existing nominal type declarations (Chapter 10).

Listing 4.1.: Some behaviors of name lookup

```
class Form { static func callee1() {} }
protocol Shape { static func callee2() }
extension Shape { static func callee3() {} }

struct Square: Shape {
  class Circle: Form {
    static func caller() {
      ...  // unqualified lookup from here
    }
  }
}
```

The body of `caller()` can reference `callee1()`, `callee2()` or `callee3()` with a single identifier. To resolve the identifier, unqualified lookup must search through these declaration contexts, starting in the top-left corner:



We will say more about name lookup in Chapter 9 and Section 10.2.

A nominal type declaration declares a new type with its own name and identity (hence "nominal"). The declared interface type of a nominal type declaration is called a nominal type, which we talked about in Section 3.1:

```
struct Universe {     // declared interface type: Universe
  struct Galaxy {}    // declared interface type: Universe.Galaxy

  func solarSystem() {
    struct Planet()   // declared interface type: Planet
  }
}
```

The declared interface type of the `Galaxy` struct is `Universe.Galaxy`, while the declared interface type of `Planet` is just `Planet`, with no parent type. This reflects the semantic difference; `Galaxy` is visible to qualified lookup as a member of `Universe`, while `Planet` is only visible to unqualified lookup within the scope of `solarSystem()`; we call it a *local type declaration*. Section 6.2 gives more detail about nominal type nesting.

Type alias declarations. These are introduced by the `typealias` keyword. The underlying type is written on the right-hand side of "`=`":

```
typealias Hands = Int  // one hand is four inches
func measure(horse: Horse) -> Hands {...}
let metatype = Hands.self
```

The declared interface type of a type alias declaration is a type alias type. The canonical type of this type alias type is just the underlying type. Therefore, if we print the return type of `measure()` in a diagnostic message, we will print it as "`Hands`", but otherwise it behaves as if it were an `Int`.

As with all type declarations, the interface type of a type alias declaration is the metatype of its declared interface type. In the above, the expression "`Hands.self`" has the metatype type `Hands.Type`. This is a sugared type, canonically equal to `Int.Type`.

While type aliases are declaration contexts, the only declarations a type alias can contain are generic parameter declarations, in the event the type alias is generic.

Other type declarations. We've now seen the first two kinds of type declarations. In the next two sections, we expand on this by looking at the declarations of generic parameters, protocols, and associated types. At this point, our foray into Swift generics can begin in earnest. Here are all the type declaration kinds and their declared interface types, with a representative specimen of each:

Type declaration	**Declared interface type**
Nominal type declaration: `struct Horse {...}`	Nominal type: `Horse`
Type alias declaration: `typealias Hands = Int`	Type alias type: `Hands` (canonically `Int`)
Generic parameter declaration: `<T: Sequence>`	Generic parameter type: `T` (canonically `τ_0_0`)
Protocol declaration: `protocol Sequence {...}`	Protocol type: `Sequence`
Associated type declaration: `associatedtype Element`	Dependent member type: `Self.Element`

4.1. Generic Parameters

Various kinds of declarations can have a generic parameter list. We call them *generic declarations*. We start with those where the generic parameter list is written in source: structs, enums, classes, type aliases, functions, constructors, and subscripts. In all cases, the generic parameter list is declared by following the generic declaration's name with the `<...>` syntax:

```
struct Outer<T> {...}
```

Each comma-separated element in this list is a *generic parameter declaration*; this is a type declaration that declares a generic parameter type. Generic parameter declarations are scoped to the entire source range of their parent declaration, the one with the generic parameter list. A declaration with a generic parameter list can be nested inside of another generic declaration; each inner generic declaration is effectively parameterized its own generic parameters, together with all generic parameters of its outer declarations. Thus, unqualified lookup will "see" all outer generic parameter declarations.

Any declaration kind that can have a generic parameter list is also a declaration context in our taxonomy, because it contains other declarations; namely, its generic parameter declarations. We say that a declaration context is a *generic context* if at least one parent context has a generic parameter list.

The name of a generic parameter declaration plays no role after unqualified lookup. Instead, to each generic parameter declaration, we assign a pair of integers (or more accurately, natural numbers; they're non-negative), the *depth* and the *index*:

- The depth selects a generic parameter list; the generic parameters declared by the outermost generic parameter list are at depth zero, and we increment the depth by one for each nested generic parameter list.
- The index selects a generic parameter within a generic parameter list; we number sibling generic parameter declarations consecutively starting from zero.

Let's write some nested generic declarations inside the `Outer` struct above. In the following, `two()` is generic over `T` and `U`, while `four()` is generic over `T`, `V`, `W` and `X`:

```
struct Outer<T> {
  func two<U>(u: U) -> T {...}

  struct Both<V, W> {
    func four<X>() -> X {...}
  }
}
```

When type resolution resolves the type representation "`T`" in the return type of `two()`, it outputs a generic parameter type that prints as "`T`", if it appears in a diagnostic for example. This is a sugared type. Every generic parameter type also has a canonical form which only records the depth and index; we denote a canonical generic parameter type by "`τ_d_i`", where `d` is the depth and `i` is the index. Two generic parameter types are canonically equal if they have the same depth and index. This is sound, because the depth and index unambiguously identify a generic parameter within its lexical scope.

Let's enumerate all generic parameters visible within `two()`,

Name:	`T`	`U`
Depth:	0	1
Index:	0	0
Type:	`τ_0_0`	`τ_1_0`

and `four()`,

Name:	`T`	`V`	`W`	`X`
Depth:	0	1	1	2
Index:	0	0	1	0
Type:	`τ_0_0`	`τ_1_0`	`τ_1_1`	`τ_2_0`

The generic parameter `U` of `two()` has the same declared interface type, `τ_1_0`, as the generic parameter `V` of `four()`. This is not a problem because the source ranges of their parent declarations, `two()` and `Both`, do not intersect.

The numbering by depth can be seen in the declared interface type of a nested generic nominal type declaration. For example, the declared interface type of `Outer.Both` is the generic nominal type `Outer<τ_0_0>.Both<τ_1_0, τ_1_1>`.

Implicit generic parameters. Sometimes the generic parameter list is not written in source. Every protocol declaration has a generic parameter list with a single generic parameter named `Self` (Section 4.3), and every extension declaration has a generic parameter list cloned from that of the extended type (Chapter 10). These implicit generic parameters can be referenced by name within their scope, just like the generic parameter declarations in a parsed generic parameter list (Section 9.1).

Function, constructor and subscript declarations can also declare *opaque parameters* with the `some` keyword, possibly in combination with a generic parameter list:

```
func pickElement<E>(_ elts: some Sequence<E>) -> E {...}
```

An opaque parameter simultaneously declares a parameter value, a generic parameter type that is the type of the value, and a requirement this type must satisfy. Here, we can

refer to "`elts`" from an expression inside the function body, but we cannot name the *type* of "`elts`" in a type representation. From expression context however, the type of an opaque parameter can be obtained via the `type(of:)` special form, which produces a metatype value. This allows for invoking static methods on these types.

The **generic parameter list request** appends the opaque parameters to the parsed generic parameter list, so they follow the parsed generic parameters in index order. In `pickElement()`, the generic parameter `E` has canonical type `τ_0_0`, while the opaque parameter associated with "`elts`" has canonical type `τ_0_1`. Opaque parameter declarations also state a constraint type, which imposes a requirement on this unnamed generic parameter. We will discuss this in the next section. Note that when `some` appears in the return type of a function, it declares an *opaque result type*, which is a related but different feature (Chapter 13).

In Chapter 5, we will discuss the generic signature, a data structure which collects all generic parameters that parameterize a declaration, independent of surface syntax.

4.2. Requirements

The requirements of a generic declaration constrain the generic argument types that can be provided by the caller. This endows the generic declaration's type parameters with new capabilities, so they abstract over the concrete types that satisfy those requirements. We use the following encoding for requirements in the theory and implementation.

Definition 4.1. A *requirement* is a triple consisting of a *requirement kind*, a subject type `T` (usually a type parameter), and one final piece of information that depends on the requirement kind:

- A **conformance requirement** [`T: P`] states that the replacement type for `T` must conform to `P`, which must be a protocol, protocol composition, or parameterized protocol type.
- A **superclass requirement** [`T: C`] states that the replacement type for `T` must be a subclass of some class type `C`.
- A **layout requirement** [`T: AnyObject`] states that the replacement type for `T` must be represented as a single reference-counted pointer at run time.
- A **same-type requirement** [`T == U`] states that the replacement types for `T` and `U` must be canonically equal.

When looking at concrete instances of requirements in a self-contained snippet of code, there is no ambiguity in using the same notation for the first three kinds, because the type referenced by the right-hand side determines the requirement kind. When talking about requirements in the abstract, we will explicitly state that `P` is some protocol, or `C` is some class, before talking about [`T: P`] or [`T: C`].

Constraint types. Before we introduce the trailing `where` clause syntax for stating requirements in a fully general way, let's look at the shorthand of stating a *constraint type* in the inheritance clause of a generic parameter declaration:

```
func allEqual<E: Equatable>(_ elements: [E]) {...}
```

The generic parameter declaration `E` declares the generic parameter type `τ_0_0`, and it also states the constraint type `Equatable`. This is a protocol declared in the standard library, so the stated requirement is the conformance requirement [`τ_0_0: Equatable`]. More generally, the constraint type is one of the following:

1. A protocol type, like `Equatable`.
2. A parameterized protocol type, like `Sequence<String>`.
3. A protocol composition type, like `Sequence & MyClass`.
4. A class type, like `NSObject`.
5. The `AnyObject` *layout constraint*, which restricts the possible concrete types to those represented as a single reference-counted pointer.

In the first three cases, the stated requirement becomes a conformance requirement. Otherwise, it is a superclass or layout requirement. In all cases, the subject type of the requirement is the declared interface type of the generic parameter.

Example 4.2. Notice how the generic parameter `B` of `open()` states the constraint type `Box<C>`, and this refers to the second generic parameter, `C`:

```
func open<B: Box<C>, C>(box: B) -> C {
  return box.contents!
}

class Box<Contents> {
  var contents: Contents? = nil
}
```

This illustrates a property of the scope tree: generic parameters are visible in the entire source range of a generic declaration, including in the generic parameter list itself. The declaration of `open()` thus states the superclass requirement [`τ_0_0: Box<τ_0_1>`]. Here is a possible usage of `open()`, which we leave unexplained:

```
struct Vegetable {}
class FarmBox: Box<Vegetable> {}
let vegetable: Vegetable = open(box: FarmBox())
```

Opaque parameters. An opaque parameter declaration is written with a constraint type after the `some` keyword (Section 4.1). This specifies a conformance, superclass or layout requirement on the generic parameter type introduced by the opaque parameter declaration. For example, the following two declarations are equivalent:

```
func pickElement<E>(_ elts: some Sequence<E>) -> E {...}
func pickElement<E, S: Sequence<E>>(_ elts: S) -> E {...}
```

We will see later that constraint types also appear in various other positions, and in all cases, they state a requirement with some distinguished subject type:

1. In the inheritance clause of a protocol or associated type (Section 4.3).
2. Following the `some` keyword in return position, where it declares an opaque result type (Chapter 13).
3. Following the `any` keyword for introducing an existential type (Chapter 14), with the exception that the constraint type cannot be a class by itself (for example, we allow "`any MyClass & Equatable`", but "`any MyClass`" is just "`MyClass`").

Trailing where clauses. Requirements can also be stated in a `where` clause attached to the generic declaration. This allows for more generality than can be expressed in the inheritance clause of a generic parameter alone.

A `where` clause entry defines a requirement whose subject type is written explicitly, so that dependent member types can be subject to requirements; here, we state two requirements, [`τ_0_0: Sequence`] and [`τ_0_0.Element: Comparable`]:

```
func isSorted<S>(_: S) where S: Sequence, S.Element: Comparable {...}
```

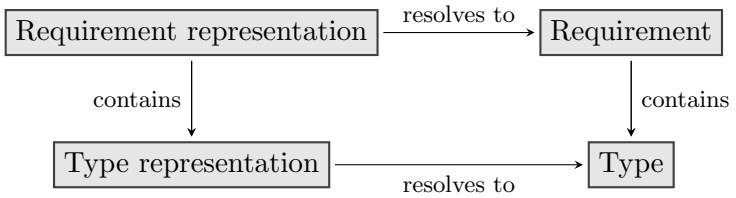
A `where` clause can also state a same-type requirement. In the next example, we state two conformance requirements using the inheritance clause syntax, another conformance requirement, and the same-type requirement [`τ_0_0.Element == τ_0_1.Element`]:

```
func merge<S1: Sequence, S2: Sequence>(_: S1, _: S2) -> [S1.Element]
    where S1.Element: Comparable, S1.Element == S2.Element {...}
```

Note that there is no way to refer to an opaque parameter type within the function's `where` clause, but every declaration using opaque parameters can always be rewritten into an equivalent one using named generic parameters, so no generality is lost.

We saw in Chapter 3 that when the parser reads a type annotation in the source, it constructs a type representation, a lower-level syntactic object which must be resolved to obtain a type. Similarly, requirements have a syntactic form, called a *requirement*

representation. The parser constructs requirement representations while reading a `where` clause. The relationship between the syntactic and semantic entities is shown in this diagram:



There are only two kinds of requirement representations, because the "`:`" form cannot distinguish conformance, superclass and layout requirements until we resolve the type representation on the right-hand side:

1. A **constraint requirement representation** "`T: C`", where `T` and `C` are type representations.
2. A **same-type requirement representation** "`T == U`", where `T` and `U` are type representations.

Recall that a conformance requirement has a protocol type, protocol composition type, or parameterized protocol type on the right-hand side (Section 3.1). In the case of a protocol composition, we decompose the requirement into simpler requirements, one for each member of the composition. For example, if `MyClass` is a class, the requirement [`τ_0_0: Sequence & MyClass`] splits up as [`τ_0_0: Sequence`] and [`τ_0_0: MyClass`], the latter being a superclass requirement. The empty protocol composition, written `Any`, is a trivial case; stating a conformance requirement to `Any` does nothing in a `where` clause, but it is allowed. Parameterized protocol types also decompose, as we'll see in Section 4.3.

In the next section, we will introduce a formal system for derived requirements, where we will assume that only conformance requirements to protocol types remain, and that the subject type of a requirement is always a type parameter, and not an arbitrary type. Section 11.2 will show how we eliminate these unnecessary forms of generality.

Contextually-generic declarations. A generic declaration nested inside of another generic declaration can state a `where` clause, without introducing new generic parameters of its own. This is called a *contextually-generic declaration*:

```
enum LinkedList<Element> {...}

extension LinkedList {
  func sum() -> Element where Element: AdditiveArithmetic {...}
}
```

There is no semantic distinction between attaching a `where` clause to a member of a type, or moving the member to a constrained extension (Section 10.3), so the above is equivalent to the following:

```
extension LinkedList where Element: AdditiveArithmetic {
  func sum() -> Element {...}
}
```

However, for historical reasons, these two declarations have distinct mangled symbol names, so the above is not an ABI-compatible transformation.

In Chapter 5, we will see that the generic signature of a declaration records all of its requirements, regardless of they were stated in source.

History. The syntax described in this section has evolved over time:

- The `where` clause used to be written within the "`<`" and "`>`", but was moved to the current "trailing" position in Swift 3 [71].
- Generic type aliases were introduced in Swift 3 [72].
- Protocol compositions involving class types were introduced in Swift 4 [73].
- Generic subscripts were introduced in Swift 4 [74].
- Implementation limitations prevented the `where` clause from stating requirements that constrain outer generic parameters until Swift 3, and contextually-generic declarations were not allowed until Swift 5.3 [75].
- Opaque parameter declarations were introduced in Swift 5.7 [76].

4.3. Protocols

The `protocol` keyword introduces a *protocol declaration*, which is a special kind of nominal type declaration. The members of a protocol, with the exception of type aliases, are requirements that must be witnessed by corresponding members in a conforming type. The protocol's properties, subscripts, and methods are called *value requirements*. They don't have bodies, but are otherwise like member declarations of a concrete nominal type. The declared interface type of a protocol declaration is a protocol type.

Every protocol has an implicit generic parameter list with a single generic parameter named `Self`, which abstracts over the conforming type. The declared interface type of `Self` is always `τ_0_0`; protocols cannot be nested in other generic contexts (Section 6.2), nor can they declare any other generic parameters.

The `associatedtype` keyword introduces an *associated type declaration*, which can only appear inside of a protocol. The declared interface type is a dependent member type (Section 3.1). Specifically, the declared interface type of an associated type `A` in a protocol `P` is the bound dependent member type denoted `Self.[P]A`, formed from the base type of `Self` together with `A`. A nominal type conforming to a protocol with associated types must declare a type witness for each associated type (Section 7.3).

Protocols can also impose *associated requirements* on `Self` and its dependent member types. A conforming type and its type witnesses must together satisfy these associated requirements. There are a number of ways to state associated requirements in the language, so we will review them now.

Protocol inheritance clauses. A protocol can have an inheritance clause with a list of one or more comma-separated constraint types. Each inheritance clause entry states an associated requirement with a subject type of `Self`. These are additional requirements the conforming type itself must satisfy in order to conform.

An associated conformance requirement with a subject type of `Self` establishes a *protocol inheritance* relationship. The protocol stating the requirement is the *derived protocol*, and the protocol on the right-hand side is the *base protocol*. The derived protocol is said to *inherit* from (or sometimes, *refine*) the base protocol. A qualified lookup into a protocol (or concrete nominal types) will traverse all of its base protocols (or all base protocols of each conformed protocol of the concrete nominal type).

For example, the standard library's `Collection` protocol inherits from `Sequence` by stating the associated requirement [`Self: Sequence`]:

```
protocol Collection: Sequence {...}
```

Protocols can restrict their conforming types to those with a reference-counted pointer representation by stating an `AnyObject` layout constraint in the inheritance clause:

```
protocol BoxProtocol: AnyObject {...}
```

Protocols can also limit their conforming types to subclasses of some superclass:

```
class Plant {}
class Animal {}
protocol Duck: Animal {}
class MockDuck: Plant, Duck {}  // error: not a subclass of Animal
```

A protocol is *class-constrained* if the [`Self: AnyObject`] associated layout requirement is either explicitly stated, or a consequence of some other associated requirement. We'll say more about the semantics of protocol inheritance clauses and name lookup in Section 5.1, Section 9.1, and Chapter 11.

Primary associated types. A protocol can declare a list of *primary associated types* with a syntax resembling that of a generic parameter list:

```
protocol IteratorProtocol<Element> {
  associatedtype Element
  mutating func next() -> Element?
}
```

While generic parameter lists introduce new generic parameter declarations, the entries in the primary associated type list reference *existing* associated type declarations, either in the protocol itself, or some base protocol.

A *parameterized protocol type* can be formed from a reference to a protocol with primary associated types, by taking a list of generic argument types, one for each primary associated type. On the right-hand side of a conformance requirement, a parameterized protocol type decomposes into a conformance requirement to the protocol, followed by a series of same-type requirements. The following are equivalent:

```
func sumOfSquares<I>(_: I) -> Int
    where I: IteratorProtocol<Int> {...}
func sumOfSquares<I>(_: I) -> Int
    where I: IteratorProtocol, I.Element == Int {...}
```

More details appear in Section 11.2. Parameterized protocol types and primary associated types were added to the language in Swift 5.7 [55].

Associated requirements. An associated type declaration can have an inheritance clause, consisting of one or more comma-separated constraint types. Each entry defines a requirement on the declared interface type of the associated type declaration, so we get [`Self.Data: Codable`] and [`Self.Data: Hashable`] below:

```
associatedtype Data: Codable, Hashable
```

An associated type declaration can also have a trailing `where` clause, which allows associated requirements to be stated in full generality. The standard library `Sequence` protocol showcases both primary associated types, and associated requirements:

```
protocol Sequence<Element> {
  associatedtype Iterator: IteratorProtocol
  associatedtype Element where Element == Iterator.Element

  func makeIterator() -> Iterator
}
```

The associated conformance requirement on `Self.Iterator` could have been stated using a `where` clause instead:

```
associatedtype Iterator where Iterator: IteratorProtocol
```

A `where` clause can also be attached to the protocol itself; there is no semantic difference between that and attaching it to an associated type declaration:

```
protocol Sequence where Iterator: IteratorProtocol,
                        Element == Iterator.Element {...}
```

Finally, we can explicitly qualify the member types with `Self`:

```
protocol Sequence where Self.Iterator: IteratorProtocol,
                        Self.Element == Self.Iterator.Element {...}
```

In all cases, we state the same two associated requirements. Our notation is to subscript the associated requirement with the name of the protocol stating this requirement:

$$[\texttt{Self.Iterator: IteratorProtocol}]_{\texttt{Sequence}}$$
$$[\texttt{Self.Element == Self.Iterator.Element}]_{\texttt{Sequence}}$$

Let's summarize all the ways of stating associated requirements in a protocol `P`:

- The protocol can state an inheritance clause. Each entry defines a conformance, superclass, or layout requirement with a subject type of `Self`.
- An associated type declaration `A` can state an inheritance clause. Each entry defines a conformance, superclass or layout requirement with a subject type of `Self.[P]A`.
- Arbitrary associated requirements can be stated in trailing `where` clauses, attached to the protocol or any of its associated types, in any combination.

A protocol's associated requirements are collected in its requirement signature, which we will see is dual to a generic signature in some sense (Section 5.1). How concrete types satisfy the requirement signature will be discussed in Chapter 7.

Self requirements. The `where` clause of a protocol method or subscript requirement cannot constrain `Self` or its associated types. For example, the following protocol is rejected, because there would be no way to implement the `minElement()` requirement in a concrete conforming type whose `Element` type is *not* `Comparable`:

```
protocol SetProtocol {
  associatedtype Element  // we want 'where Element: Comparable' here
  func minElement() -> Element where Element: Comparable  // error
}
```

History. Older releases of Swift permitted protocols and associated types to state constraint types in their inheritance clauses, but more general associated requirements did not exist. Associated requirements were introduced when the trailing `where` clause syntax was generalized to associated types and protocols in Swift 4 [77].

4.4. Functions

A function declaration can appear at the top level of a source file, as a member of a nominal type or extension (which we recall is called a method declaration), or as a *local function* nested inside of another function. In this section, we will describe the computation of the interface type of a function declaration, and then conclude with a discussion how closure expressions and local functions capture values.

The **interface type request** computes the interface type of a function declaration. This is a function type or generic function type, constructed from the interface types of the function's parameter declarations, together with its return type, and generic signature, if any. If no return type is given, it becomes the empty tuple type `()`:

```
func f(x: Int, y: String) -> Bool {...}
// Interface type: (Int, String) -> Bool

func g() {...}
// Interface type: () -> ()
```

Method declarations. In addition to the formal parameters declared in its parameter list, a method declaration also has an implicit `self` parameter, to receive the value on the left-hand side of the "`.`" in the member reference expression. The interface type of a method declaration is a function type which receives the `self` parameter, and returns another function which then takes the method's formal parameters. The "`->`" syntax for a function type associates to the right, so `A -> B -> C` means `A -> (B -> C)`:

```
struct Universe {
  func wormhole(x: Int, y: String) -> Bool {...}
  // Interface type: (Universe) -> (Int, String) -> Bool

  static func bigBang() {}
  // Interface type: (Universe.Type) -> () -> ()

  mutating func teleport() {}
  // Interface type: (inout Universe) -> () -> ()
}
```

We derive the interface type and ownership specifier for the `self` parameter as follows:

- We start with the *self interface type* of the method's parent declaration context. In a struct, enum, or class, this is the same as the declared interface type. In a protocol, this is the protocol `Self` type (Section 4.3). In an extension, the self interface type is that of the extended type.
- If the method is declared inside a class, and if it returns the dynamic `Self` type, we wrap the `self` type in the dynamic `Self` type (Section 3.3).
- If the method is `static`, we further wrap the `self` type in a metatype.
- If the method is `mutating`, we pass the `self` parameter `inout`.

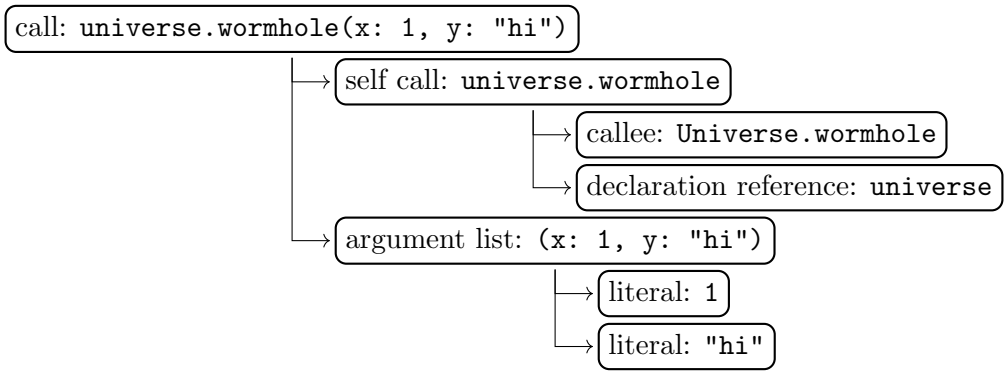
Let's compare the interface type of a method declaration with the structure of a method call expression in the abstract syntax tree, shown in Figure 4.1:

- The outer expression's callee, `universe.wormhole`, is itself a call expression, so we must evaluate this inner call expression first.

  The inner call expression applies the argument `universe` to the `self` parameter of `Universe.wormhole()`. This represents the method lookup. The return type of the inner call expression is `(Int, String) -> Bool`.
- The outer call expression applies the argument list `(x: 1, y: "hi")` to the result of the inner call expression. This represents the method call itself, with a return type of `Bool`.

The extra function call disappears in SILGen, where we lower a method to a SIL function that receives all formal parameters and the `self` parameter at once.

Figure 4.1.: The method call `universe.wormhole(x: 1, y: "hi")`



A partially-applied method reference, such as `universe.wormhole`, can also be used as a value of type `(Int, String) -> Bool`. This binds the `self` parameter but does not call the method. We handle this by wrapping the method reference in a closure that invokes the method; this closure has the same type as the method reference. In the lambda calculus formalism, this is what's known as *η-expansion*:

```
{ x, y in universe.wormhole(x: x, y: y) }
```

The unapplied form, `Universe.wormhole`, desugars into a closure returning a closure:

```
{ mySelf in { x y in mySelf.wormhole(x: x, y: y) } }
```

This desugaring simplifies SILGen, because we only need to implement a lowering for fully-applied method calls that pass all formal parameters and `self` at once.

```
struct Example {
  func instanceMethod() {}
  static func staticMethod() {}

  struct Lookup {
    func innerMethod() {}

    func test() {
      instanceMethod()  // bad
      staticMethod()    // ok
      innerMethod()     // ok
    }
  }

  func anotherMethod(x: Int) {
    struct Local {
      func test() {
        print(x)        // bad
      }
    }
  }
}
```

All function declarations must be followed by a body in the source language, except for protocol requirements. A function body can contain statements, expressions, and other declarations. (Unlike types and declarations, we will not exhaustively survey all statements and expressions in this book.) The example on the left shows some call expressions.

In a method body, an unqualified reference to a member of the innermost nominal type declaration is interpreted as having an implicit "`self.`" qualification. Thus, instance methods can refer to other instance methods this way, and static methods can refer to other static methods.

An unqualified reference to a member of an outer nominal type can only be made if the member is static, because there is no "outer `self` parameter" to invoke the method with; a *value* of the nested type does not contain a *value* of its parent type.

For the same reason, methods inside local types cannot refer to local variables declared outside of the local type. (Contrast this with Java inner classes for example, which can be declared as `static` or instance members of their outer class; a non-`static` inner class captures a "`this`" reference from the outer class. Inner classes nested in methods can also capture local variables in Java.)

Constructor declarations. Constructor declarations are introduced with the `init` keyword. The parent context of a constructor must be a nominal type or extension.

From the outside, the interface type of a constructor looks like a static method that returns a new instance of the type, but inside the constructor, `self` is the instance being initialized, so the interface type of `self` is the nominal type, and not its metatype. In a struct or enum, `self` is also `inout`. Constructors can delegate to other constructors in various ways. To model the delegation with a call expression, the *initializer interface type* describes the type of an in-place initialization at a location provided by the caller:

```
struct Universe {
  init(age: Int) {...}
  // Interface type: (Universe.Type) -> (Int) -> Universe
  // Initializer interface type: (inout Universe) -> (Int) -> Universe
}
```

Destructor declarations. Introduced with the `deinit` keyword, which is only valid inside a class. A destructor cannot have a generic parameter list or `where` clause. Its interface type is that of a method with no formal parameters and a return type of `()`.

Local contexts. A *local context* is any declaration context that is not a module, source file, type declaration or extension. Swift allows variable, function, and type declarations to appear in local context. The following are local contexts:

- Top-level code declarations.
- Function declarations.
- Closure expressions.
- If a variable is not itself in local context (for example, it's a member of a nominal type declaration), then its initial value expression defines a new local context.
- Subscript and enum element declarations are local contexts, because they can contain parameter declarations (and also a generic parameter list, in the case of a subscript declaration).

Local functions and closures can *capture* references to other local declarations from outer scopes. We use the standard technique of *closure conversion* to lower functions with captured values into ones without. We can understand this process as introducing an additional parameter for each captured value, followed by a walk of the function body to replace references to those captured values with references to the corresponding parameters. In Swift, this is part of SILGen's lowering process, and not a separate transformation on the abstract syntax tree.

The **capture info request** computes the list of values captured by the given function and all of its nested local functions and closure expressions.

```
func f() {
  let x = 0, y = 0

  func g() {
    var z = 0
    print(x)

    func h() {
      print(y, z)
    }
  }
}
```

Consider the three nested functions shown on the left. We proceed to compute their captures from the inside out.

The innermost function `h()` captures `y` and `z`. The middle function `g()` captures `x`. It also captures `y`, because `h()` captures `y`, but it does not capture `z`, because `z` is declared by `g()` itself. Finally, `f()` is declared at the top level, so it doesn't have any captures.

We can summarize this as follows (see Appendix A for a summary of set notation):

Function	Captures
`f()`	$\varnothing$
`g()`	{`x`, `y`}
`h()`	{`y`, `z`}

Algorithm 4A (Compute closure captures)**.** As input, takes the type-checked body of a closure expression or local function F. Outputs the set of captures of F.

1. Initialize the return value with an empty set, $C \leftarrow \varnothing$.
2. Recursively walk the type-checked body of F and handle each element:
3. (Declaration references) If F contains an expression that references some local variable or local function d by name, let `PARENT`(d) denote the declaration context containing d. This is either F itself, or an outer local context, because we found d by unqualified lookup from F.

   If `PARENT`(d) $\neq F$, set $C \leftarrow C \cup \{d\}$.
4. (Nested closures) If F contains a nested closure expression or local function F', then any captures of F' not declared by F are also captures of F.

   Recursively compute the captures of F'. For each d captured by F' such that `PARENT`(d) $\neq F$, set $C \leftarrow C \cup \{d\}$.
5. (Local types) If F contains a local type, do not walk into the children of the local type. Local types do not capture values; we enforce this in the next step.
6. (Diagnose) After the recursive walk, consider each element $d \in C$. If the path of parent declaration contexts from F to d contains a nominal type declaration, we have an unsupported capture inside a local type. Diagnose an error.
7. Return C.

```
func f() {
  let x = 0, y = 0, z = 0

  func g() { print(x); h() }
  func h() { print(y); g() }
  func i() { print(z); h() }
}
```

Local functions can also reference each other recursively. Consider the functions shown on the right and notice how `f()` and `g()` are mutually recursive. At runtime, we cannot represent this by forming two closure contexts where each one retains the other, because then neither context will ever be released.

We use a second algorithm to obtain the list of *lowered captures*, by replacing any captured local functions with their corresponding capture lists, repeating this until fixed point. The final list contains variable declarations only. With our example, the captures and lowered captures of each function are as follows:

Function	**Captures**	**Lowered**
`f()`	∅	∅
`g()`	{`x`, `h()`}	{`x`, `y`}
`h()`	{`y`, `g()`}	{`x`, `y`}
`i()`	{`z`, `h()`}	{`x`, `y`, `z`}

(As a special case, if a set of local functions reference each other but capture no other state from the outer declaration context, their lowered captures will be empty, so no runtime context allocation is necessary.)

Algorithm 4B (Compute lowered closure captures)**.** As input takes the type-checked body of a closure expression or local function F. Outputs the set of variable declarations transitively captured by F.

1. Initialize the set $C \leftarrow \varnothing$; this will be the return value. Initialize an empty worklist. Initialize an empty visited set. Add F to the worklist.
2. If the worklist is empty, return C. Otherwise, remove the next function F from the worklist.
3. If F is in the visited set, go back to Step 2. Otherwise, add F to the visited set.
4. Compute the captures of F using Algorithm 4A and consider each capture d. If d is a local variable declaration, set $C \leftarrow C \cup \{d\}$. If d is a local function declaration, add d to the worklist.
5. Go back to Step 2.

This completely explains captures of `let` variables, but mutable `var` variables and `inout` parameters merit further explanation.

A *non-escaping* closure can capture a `var` or `inout` by simply capturing the memory address of the storage location. This is safe, because a non-escaping closure cannot outlive the dynamic extent of the storage location.

An `@escaping` closure can also capture a `var`, which requires promoting the `var` to a heap-allocated box with a reference count, with all variable accesses indirecting through the box. The below example can be found in every Lisp textbook. Each invocation of `counter()` allocates a new counter value on the heap, and returns three closures that reference the box; the box itself is completely hidden by the abstraction:

```
func counter() -> (read: () -> Int, inc: () -> (), dec: () -> ()) {
  var count = 0  // promoted to a box
  return ({ count }, { count += 1 }, { count -= 1 })
}
```

Before Swift 3, `@escaping` closures were permitted to capture `inout` parameters as well. To make this safe, the contents of the `inout` parameter were first copied into a heap-allocated box, which was captured by the closure. The contents of this box were then copied back before the function returned to its caller. This was essentially equivalent to doing the following transform, where we introduce `_n`:

```
func changeValue(_ n: inout Int) {
  var _n = n  // copy the value

  let escapingFn = {
    _n += 1   // capture the box
  }

  n = _n      // write it back
}
```

In this scheme, if the closure outlived the dynamic extent of the `inout` parameter, any subsequent writes from within the closure were silently dropped. This was a source of user confusion, so Swift 3 banned `inout` captures from escaping closures instead [78].

In SILGen, captured values introduce new parameters at the end of the function's parameter list, and a closure value is formed from a function with captures by *partially applying* the captured values. This produces a new function value with the required type, so the captured values are "sliced off." In IRGen, we lower a partial application by allocating space for storing the captures (either on the stack for a non-escaping function type, otherwise it's on the heap for `@escaping`), and then we emit a thunk, which takes a pointer to the context as an argument, unpacks the captured values from the context, and passes them as individual arguments to the original function. This thunk together with the context forms a *thick function* value which represents the closure.

If nothing is captured (or if all captured values are zero bytes in size), we can pass a null pointer as the context, without performing a heap allocation. If there is exactly one captured value and this value can be represented as a reference-counted pointer, we can also elide the allocation by passing the captured value as the context pointer instead. For example, if a closure's single capture is an instance of a class type, nothing is allocated. If the single capture is the heap-allocated box that wraps a `var`, we must still allocate the box for the `var`, but we avoid a second context allocation.

4.5. Storage

Storage declarations represent locations that can be read and written.

Parameter declarations. Functions, enum elements, and subscripts can have parameter lists; each parameter is represented by a parameter declaration. Parameter declarations are a kind of variable declaration.

Variable declarations. Variables that are not parameters are introduced with `var` and `let`. A variable is either *stored* or *computed*; the behavior of a computed variable is determined by its *accessor declarations*. The *value interface type* of a variable is the type of its value. The interface type of a variable is obtained by taking the type of its value, and possibly wrapping it in a reference storage type if the variable is declared as `weak` or `unowned`.

Variable declarations are always created alongside a *pattern binding declaration* which represents the various ways in which variables can be bound to values in Swift. A pattern binding declaration consists of one or more *pattern binding entries*. Each pattern binding entry has a *pattern* and an optional *initial value expression*. A pattern declares zero or more variables.

Here is a pattern binding declaration with a single entry that does not declare any variables:

```
let _ = ignored()
```

Here is a pattern binding declaration with a single entry, where the pattern declares a single variable:

```
let x = 123
```

We can write a more complex pattern, for example binding the first element of a tuple while discarding the second element:

```
let (x, _) = (123, "hello")
```

Here is a pattern binding declaration with a single entry, whose pattern declares two variables `x` and `y`:

```
let (x, y) = (123, "hello")
```

Here is a pattern binding declaration with two entries, declaring `x` and `y` respectively:

```
let x = 123, y = "hello"
```

And finally, here we have two pattern binding declarations, where each pattern binding declaration has a single entry declaring a single variable:

```
let x = 123
let y = "hello"
```

When a pattern binding declaration appears outside of a local context, each entry must declare at least one variable, so we reject both of the following:

```
let _ = 123

struct S {
  let _ = "hello"
}
```

A funny quirk of the pattern grammar is that typed patterns and tuple patterns do not compose in the way one might think. If "`let x: Int`" is a typed pattern declaring a variable `x` type with annotation `Int`, and "`let (x, y)`" is a tuple pattern declaring two variables `x` and `y`, we might expect "`let (x: Int, y: String)`" to declare two variables `x` and `y` with type annotations `Int` and `String` respectively; what actually happens is we get a tuple pattern declaring two variables named `Int` and `String` that binds a two-element tuple with *labels* `x` and `y`:

```
let (x: Int, y: String) = (x: 123, y: "hello")
print(Int)  // huh? prints 123
print(String)  // weird! prints "hello"
```

Subscript declarations. Subscripts are introduced with the `subscript` keyword. They can only appear as members of nominal types and extensions. The interface type of a subscript is a function type taking the index parameters and returning the storage type. The value interface type of a subscript is just the storage type. For historical reasons, the interface type of a subscript does not include the `Self` clause, the way that method declarations do. Subscripts can either be instance or static members; static subscripts were introduced in Swift 5.1 [79].

Accessor declarations. Each storage declaration has a set of accessor declarations, which are a special kind of function declaration. The accessor declarations are siblings of the storage declaration in the declaration context hierarchy. The interface type of an accessor depends the accessor kind. For example, getters return the value, and setters take the new value as a parameter. Variable accessors do not take any other parameters; subscript accessors also take the subscript's index parameters. We will not need any more details about accessor and storage declarations in this book.

4.6. Source Code Reference

Key source files:

- include/swift/AST/Decl.h
- include/swift/AST/DeclContext.h
- lib/AST/Decl.cpp
- lib/AST/DeclContext.cpp

Other source files:

- include/swift/AST/DeclNodes.def
- include/swift/AST/ASTVisitor.h
- include/swift/AST/ASTWalker.h

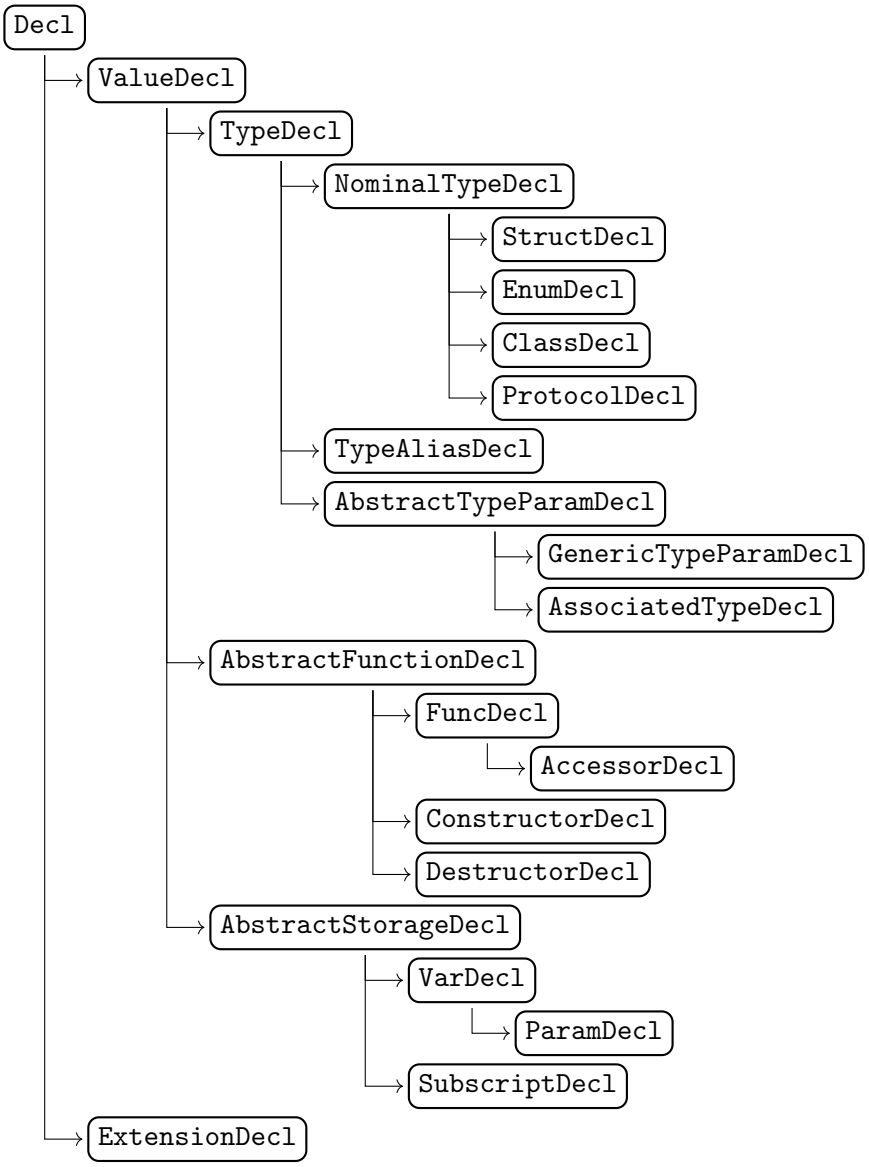
`Decl` *class*

Base class of declarations. Figure 4.2 shows various subclasses, which correspond to the different kinds of declarations described previously in this chapter.

Instances are always allocated in the permanent arena of the `ASTContext`, either when the declaration is parsed, or synthesized. The top-level `isa<>`, `cast<>` and `dyn_cast<>` template functions support dynamic casting from `Decl *` to any of its subclasses.

- `getDeclContext()` returns the parent `DeclContext` of this declaration.
- `getInnermostDeclContext()` if this declaration is also a declaration context, returns the declaration as a `DeclContext`, otherwise returns the parent `DeclContext`.
- `getASTContext()` returns the singleton AST context from a declaration.

Figure 4.2.: The `Decl` class hierarchy



Visitors. To exhaustively handle each kind of declaration, the simplest way is to switch over the kind, which is an instance of the `DeclKind` enum, like this:

```
Decl *decl = ...;
switch (decl->getKind()) {
case DeclKind::Struct: {
  auto *structDecl = decl->castTo<StructDecl>();
  ...
}
case DeclKind::Enum:
  ...
case DeclKind::Class:
  ...
}
```

However, just as with types, is can be more convenient to use the visitor pattern, by subclassing `ASTVisitor` and overriding various `visit`*Kind*`Decl()` methods. The visitor's `visit()` method performs the switch and dynamic cast dance above, and calls each method:

```
class MyVisitor: public ASTVisitor<MyVisitor> {
public:
  void visitStructDecl(StructType *decl) {
    ...
  }
};

MyVisitor visitor;

Decl *decl = ...;
visitor.visit(decl);
```

The `ASTVisitor` also defines various methods corresponding to abstract base classes in the `Decl` hierarchy, so for example an override of `visitNominalTypeDecl()` will handle all nominal type declarations at once. The `ASTVisitor` is more general than just visiting declarations; it also supports visiting statements, expressions, and type representations.

A more elaborate form is implemented by the `ASTWalker`. While the visitor visits a single declaration, the walker traverses nested declarations, statements and expressions in a pre-order walk.

`ValueDecl` *class*

Base class of named declarations.

- `getDeclName()` returns the declaration's name.
- `getInterfaceType()` returns the declaration's interface type.

Type Declarations

`TypeDecl` *class*

Base class of type declarations.

- `getDeclaredInterfaceType()` returns the type of an instance of this declaration.

`NominalTypeDecl` *class*

Base class of nominal type declarations. Also a `DeclContext`.

- `getSelfInterfaceType()` returns this nominal type declaration's self interface type (Section 4.4). This is the same as the declared interface type, except if this is a protocol declaration. A protocol's declared interface type is a nominal type, but its self interface type is the generic parameter `Self`.
- `getDeclaredType()` returns the type of an instance of this declaration, without generic arguments. If the declaration is generic, this is an unbound generic type. If this declaration is not generic, this is the same as the declared interface type. This is occasionally used in diagnostics instead of the declared interface type, when the generic parameter types are irrelevant.

`TypeAliasDecl` *class*

A type alias declaration. Also a `DeclContext`.

- `getDeclaredInterfaceType()` returns the underlying type of the type alias declaration, wrapped in type alias type sugar.
- `getUnderlyingType()` returns the underlying type of the type alias declaration, without wrapping it in type alias type sugar.

Declaration Contexts

`DeclContext` *class*

Base class for declaration contexts. See also Section 5.7.

The top-level `isa<>`, `cast<>` and `dyn_cast<>` template functions also support dynamic casting from a `DeclContext *` to any of its subclasses.

There are a handful of subclasses which are not also subclasses of `Decl *`:

- `ClosureExpr`.
- `FileUnit` and its various subclasses, such as `SourceFile`.
- A few other less interesting ones found in the source.

Methods for understanding the nesting of declaration contexts:

- `getAsDecl()` if declaration context is also a declaration, returns the declaration, otherwise returns `nullptr`.
- `getParent()` returns the parent declaration context.
- `isModuleScopeContext()` returns true if this is a `ModuleDecl` or `FileUnit`.
- `isTypeContext()` returns true if this is a nominal type declaration or an extension.
- `isLocalContext()` returns true if this is not a module scope context or type context.
- `getParentModule()` returns the module declaration at the root of the hierarchy.
- `getModuleScopeContext()` returns the innermost parent which is a `ModuleDecl` or `FileUnit`.
- `getParentSourceFile()` returns the innermost parent which is a source file, or `nullptr` if this declaration context was not parsed from source.
- `getInnermostDeclarationDeclContext()` returns the innermost parent which is also a declaration, or `nullptr`.
- `getInnermostDeclarationTypeContext()` returns the innermost parent which is also a nominal type or extension, or `nullptr`.

Operations on type contexts:

- `getSelfNominalDecl()` returns the nominal type declaration if this is a type context, or `nullptr`.
- `getSelfStructDecl()` as above but result is a `StructDecl *` or `nullptr`.
- `getSelfEnumDecl()` as above but result is a `EnumDecl *` or `nullptr`.
- `getSelfClassDecl()` as above but result is a `ClassDecl *` or `nullptr`.
- `getSelfProtocolDecl()` as above but result is a `ProtocolDecl *` or `nullptr`.

- `getDeclaredInterfaceType()` delegates to the method on `NominalTypeDecl` or `ExtensionDecl` as appropriate.
- `getSelfInterfaceType()` is similar.

Generic parameters and requirements:

- `isGenericContext()` answers true if either this generic context or one of its parents has a generic parameter list.
- `isInnermostContextGeneric()` answers if this declaration context itself has a generic parameter list. Compare with `isGenericContext()`.

Generic Contexts

Key source files:

- include/swift/AST/GenericParamList.h
- include/swift/AST/Requirement.h
- lib/AST/GenericParamList.cpp
- lib/AST/NameLookup.cpp
- lib/AST/Requirement.cpp

`GenericContext` *class*

Subclass of `DeclContext`. Base class for declaration kinds which can have a generic parameter list. See also Section 5.7.

- `getParsedGenericParams()` returns the declaration's parsed generic parameter list, or `nullptr`.
- `getGenericParams()` returns the declaration's full generic parameter list, which includes any implicit generic parameters. Evaluates a `GenericParamListRequest`.
- `isGeneric()` answers if this declaration has a generic parameter list. This is equivalent to calling `DeclContext::isInnermostContextGeneric()`. Compare with `DeclContext::isGenericContext()`.
- `getGenericContextDepth()` returns the depth of the declaration's generic parameter list, or `(unsigned)-1` if neither this declaration nor any outer declaration is generic.

- `getTrailingWhereClause()` returns the declaration's trailing `where` clause, or `nullptr`.

Trailing `where` clauses are not preserved in serialized generic contexts. Most code should look at `GenericContext::getGenericSignature()` instead (Section 5.7), except when actually building the generic signature.

`GenericParamList` *class*

A generic parameter list.

- `getParams()` returns an array of generic parameter declarations.
- `getOuterParameters()` returns the outer generic parameter list, linking multiple generic parameter lists for the same generic context. Only used for extensions of nested generic types.

`GenericParamListRequest` *class*

This request creates the full generic parameter list for a declaration. Kicked off from `GenericContext::getGenericParams()`.

- For protocols, this creates the implicit `Self` parameter.
- For functions and subscripts, calls `createOpaqueParameterGenericParams()` to walk the formal parameter list and look for `OpaqueTypeRepr`s.
- For extensions, calls `createExtensionGenericParams()` which clones the generic parameter lists of the extended nominal itself and all of its outer generic contexts, and links them together via `GenericParamList::getOuterParameters()`.

`GenericTypeParamDecl` *class*

A generic parameter declaration.

- `getDepth()` returns the depth of the generic parameter declaration.
- `getIndex()` returns the index of the generic parameter declaration.
- `getName()` returns the name of the generic parameter declaration.
- `getDeclaredInterfaceType()` returns the sugared generic parameter type for this declaration, which prints as the generic parameter's name.
- `isOpaque()` answers if this generic parameter is associated with an opaque parameter.

- `getOpaqueTypeRepr()` returns the associated `OpaqueReturnTypeRepr` if this is an opaque parameter, otherwise `nullptr`.
- `getInherited()` returns the generic parameter declaration's inheritance clause.

Inheritance clauses are not preserved in serialized generic parameter declarations. Requirements stated on generic parameter declarations are part of the corresponding generic context's generic signature, so except when actually building the generic signature, most code uses `GenericContext::getGenericSignature()` instead (Section 5.7).

`GenericTypeParamType` *class*

A generic parameter type.

- `getDepth()` returns the depth of the generic parameter declaration.
- `getIndex()` returns the index of the generic parameter declaration.
- `getName()` returns the name of the generic parameter declaration if this is the sugared form, otherwise returns a string of the form "`τ_d_i`".

`TrailingWhereClause` *class*

The syntactic representation of a trailing `where` clause.

- `getRequirements()` returns an array of `RequirementRepr`.

`RequirementRepr` *class*

The syntactic representation of a requirement in a trailing `where` clause.

- `getKind()` returns a `RequirementReprKind`.
- `getFirstTypeRepr()` returns the first `TypeRepr` of a same-type requirement.
- `getSecondTypeRepr()` returns the second `TypeRepr` of a same-type requirement.
- `getSubjectTypeRepr()` returns the first `TypeRepr` of a constraint or layout requirement.
- `getConstraintTypeRepr()` returns the second `TypeRepr` of a constraint requirement.
- `getLayoutConstraint()` returns the layout constraint of a layout requirement.

`RequirementReprKind` *enum class*

Return type of `RequirementRepr::getKind()`.

- `RequirementRepr::TypeConstraint`
- `RequirementRepr::SameType`
- `RequirementRepr::LayoutConstraint`

`WhereClauseOwner` *class*

Represents a reference to some set of requirement representations which can be resolved to requirements, for example a trailing `where` clause. This is used by various requests, such as the `RequirementRequest` below, and the `InferredGenericSignatureRequest` in Section 11.5.

- `getRequirements()` returns an array of `RequirementRepr`.
- `visitRequirements()` resolves each requirement representation and invokes a callback with the `RequirementRepr` and resolved `Requirement`.

`RequirementRequest` *class*

Request which can be evaluated to resolve a single requirement representation in a `WhereClauseOwner`. Used by `WhereClauseOwner::visitRequirements()`.

`ProtocolDecl` *class*

A protocol declaration.

- `getTrailingWhereClause()` returns this protocol `where` clause, or `nullptr`.
- `getAssociatedTypes()` returns an array of all associated type declarations in this protocol.
- `getPrimaryAssociatedTypes()` returns an array of all primary associated type declarations in this protocol.
- `getInherited()` returns this protocol's inheritance clause.

Trailing `where` clauses and inheritance clauses are not preserved in serialized protocol declarations. Except when actually building the requirement signature, most code uses `ProtocolDecl::getRequirementSignature()` instead (Section 5.7).

The last three utility methods operate on the requirement signature, so are safe to use on deserialized protocols:

- `getInheritedProtocols()` returns an array of all protocols directly inherited by this protocol, computed from the inheritance clause.

- `inheritsFrom()` determines if this protocol inherits from the given protocol, possibly transitively.
- `getSuperclassDecl()` returns the protocol's superclass declaration.

`AssociatedTypeDecl` *class*

An associated type declaration.

- `getTrailingWhereClause()` returns this associated type's trailing `where` clause, or `nullptr`.
- `getInherited()` returns this associated type's inheritance clause.

Trailing `where` clauses and inheritance clauses are not preserved in serialized associated type declarations. Requirements on associated types are part of a protocol's requirement signature, so except when actually building the requirement signature, most code uses `ProtocolDecl::getRequirementSignature()` instead (Section 5.7).

Function Declarations

`AbstractFunctionDecl` *class*

Base class of function-like declarations. Also a `DeclContext`.

- `getImplicitSelfDecl()` returns the implicit `self` parameter if this is a method, `nullptr` otherwise.
- `getParameters()` returns the function's parameter list.
- `getMethodInterfaceType()` returns the type of a method without the `Self` clause.
- `getResultInterfaceType()` returns the return type of this function or method.

`ParameterList` *class*

The parameter list of `AbstractFunctionDecl`, `EnumElementDecl`, or `SubscriptDecl`.

- `size()` returns the number of parameters.
- `get()` returns the `ParamDecl` at the given index.

`ConstructorDecl` *class*

A constructor declaration.

- `getInitializerInterfaceType()` returns the initializer interface type, used when type checking `super.init()` delegation.

Closure Conversion

Key source files:

- include/swift/AST/CaptureInfo.h
- include/swift/SIL/TypeLowering.h
- lib/AST/CaptureInfo.cpp
- lib/Sema/TypeCheckCaptures.cpp
- lib/SIL/IR/TypeLowering.cpp

`CaptureInfo`	*class*

An immutable list of captured values.

`CaptureInfoRequest::evaluate`	*method*

Computes the `CaptureInfo`. This is Algorithm 4A.

`TypeConverter::getLoweredLocalCaptures()`	*method*

Computes the lowered `CaptureInfo`. This is Algorithm 4B. See Section 6.5 and Section 6.6 for a discussion of SIL type lowering and the `TypeConverter`.

Storage Declarations

`AbstractStorageDecl`	*class*

Base class for storage declarations.

- `getValueInterfaceType()` returns the type of the stored value, without `weak` or `unowned` storage qualifiers.

`VarDecl`	*class*

Subclass of `AbstractStorageDecl`.

`SubscriptDecl`	*class*

Subclass of `AbstractStorageDecl` and `DeclContext`.

`AccessorDecl`	*class*

Subclass of `AbstractFunctionDecl`.

Part II.

Semantics

5. Generic Signatures

Generic signatures are semantic objects that describe the type checking behavior of generic declarations. Declarations can nest, and outer generic parameters are visible inside inner declarations, so a generic signature is a "flat" representation that collects all *generic parameter types* and *requirements* that apply in the declaration's scope. This abstracts away the concrete syntax described in the previous chapter:

- The list of generic parameter types begins with all outer generic parameters, which are followed by generic parameters from the explicit generic parameter list `<...>`, together with any generic parameter types implicitly introduced by opaque parameter declarations, like the "`some P`" in "`func f(_: some P)`".
- The list of requirements in a generic signature includes the requirements stated by outer generic declarations, as well as any requirements from generic parameter inheritance clauses, the trailing `where` clause, and opaque parameters. A fourth mechanism, called requirement inference, will be described in Section 11.1.

The following notation for generic signatures is used throughout this book, as well as debugging output from the compiler:

$$\underbrace{\texttt{<A, B, C, ...}}_{\text{generic parameters}}\ \texttt{where}\ \underbrace{\texttt{A: Sequence, B: Equatable, A.Element == Int, ...>}}_{\text{requirements}}$$

The requirements in a generic signature use the same semantic representation as the requirements in a trailing `where` clause, given by Definition 4.1, with a few additional properties. A generic signature always omits any redundant requirements from the list, and the remaining ones are rewritten to be "as simple as possible." We will fully explain this in Chapter 11, when we see how the generic signature of a declaration is built from the syntactic forms described in the previous chapter, but for now, we're just going to assume we're working with an existing generic signature that was given to us.

After some preliminaries, we will go on to introduce a formal system for reasoning about requirements and type parameters in Section 5.2. This makes precise the earlier concept of the *interface type* of a declaration—it contains type parameters whose validity can be derived from the declaration's generic signature. Section 5.6 describes *generic signature queries*, which are fundamental primitives in the implementation, used by the rest of the compiler to answer questions about generic signatures. These questions will be precisely stated using our formal system.

Debugging. The `-debug-generic-signatures` frontend flag gives us a glimpse into the generics implementation by printing the generic signature of each declaration being type checked. Here is a simple program with three nested generic declarations:

```
struct Outer<T: Sequence> {
  struct Inner<U> {
    func transform() where T.Element == U {
      ...
    }
  }
}
```

If we run the compiler with this flag, we see that the generic signature at each level of nesting incorporates all information from the outer declaration's generic signature:

```
debug.(file).Outer@debug.swift:1:8
Generic signature: <T where T : Sequence>

debug.(file).Outer.Inner@debug.swift:2:10
Generic signature: <T, U where T : Sequence>

debug.(file).Outer.Inner.transform()@debug.swift:3:10
Generic signature: <T, U where T : Sequence, U == T.[Sequence]Element>
```

This flag also allows us to observe *requirement minimization*. Here are three functions, each one generic over a pair of types conforming to `Sequence`:

```
func sameElt<S1: Sequence, S2: Sequence>(_ s1: S1, _ s2: S2)
    where S1.Element == S2.Element {...}

func sameIter<S1: Sequence, S2: Sequence>(_ s1: S1, _ s2: S2)
    where S1.Iterator == S2.Iterator {...}

func sameEltAndIter<S1: Sequence, S2: Sequence>(_ s1: S1, _ s2: S2)
    where S1.Element == S2.Element,
          S1.Iterator == S2.Iterator {...}
```

The first function expects two sequences with the same `Element` associated type, so "`sameElt(Array<Int>(), Set<Int>())`" for example. The second function requires both have the same `Iterator` type. The third function requires both conditions, but a consequence of how `Sequence` is declared in the standard library is that the second

condition is stronger than the first; that is, if two sequences have the same `Iterator` type, they *also* have the same `Element` type. In other words, the same-type requirement [`S1.Element == S2.Element`] is *redundant* in `sameEltAndIter()`. (We will be able to *prove* this fact when we revisit this generic signature in Example 5.11.) When we compile this program with `-debug-generic-signatures`, we observe that the generic signature of `sameElt()` is distinct from the other two, while `sameIter()` and `sameEltAndIter()` have a generic signature that looks like this:

```
<S1, S2 where S1: Sequence, S2: Sequence,
              S1.[Sequence]Iterator == S2.[Sequence]Iterator>
```

Requirement minimization is described in Section 11.4.

Canonical signatures. Generic signatures are immutable and uniqued, so two generic signatures with the same structure and pointer-equal types are always pointer-equal generic signatures. While requirements in a generic signature are minimal and sorted in a certain order, two otherwise-identical generic signatures can still differ by type sugar. A generic signature is *canonical* if all listed generic parameter types are canonical, and any types appearing in requirements are canonical. To compute a canonical signature from an arbitrary generic signature, we replace any sugared types appearing in the signature with canonical types. This gives us the notion of canonical equality of generic signatures; two generic signatures are canonically equal if their canonical signatures are pointer-equal.

The generic signatures of `sameIter()` and `sameEltAndIter()` above are canonically equal, but not pointer-equal; while their generic parameters have the same *names*, the sugared generic parameter types refer to actual *declarations*, which are distinct. Their canonical signature looks like this:

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
           τ_0_0.[Sequence]Iterator == τ_0_1.[Sequence]Iterator>
```

Empty generic signature. A generic declaration without a generic parameter list or trailing `where` clause inherits the generic signature from the parent context. If no outer parent context is generic, we get the *empty generic signature* with no generic parameters or requirements. The interface types described by the empty generic signature are the fully concrete types, that is, types that do not contain any type parameters.

Protocol generic signature. In Section 4.3 we saw that every protocol declaration has a generic parameter named `Self` that conforms to the protocol itself. The protocol `Self` type is always canonically equal to `τ_0_0`. We will denote the generic signature of a protocol `P` by $G_{\texttt{P}}$:

$G_{\texttt{P}}$:= `<Self where Self: P>`

5.1. Requirement Signatures

Every protocol has a *requirement signature*, which collects the protocol's associated type declarations and the associated requirements imposed upon them. This abstracts away the concrete syntax from Section 4.3. There is a duality between generic signatures and requirement signatures, illustrated with the following diagram that shows each kind of entity with a concrete specimen to remind us of the notation:

Generic signature	**Requirement signature**
Generic parameters: `τ_0_1`	Associated types: `associatedtype Iterator`
Generic requirements: [`τ_0_1: Sequence`]	Associated requirements: [`Self.Iterator: IteratorProtocol`]$_{\texttt{Sequence}}$

(Requirement signatures also describe the type alias members of the protocol; these are called *protocol type aliases*. They're not part of the formalism described in the next section, but we will discuss them in Section 17.5.)

If a generic signature G states a conformance requirement [`T: P`], the requirement signature of `P` generates additional structure, which we describe informally at first:

- For every associated type `A` of `P`, we can talk about the dependent member type `T.[P]A`, which abstracts over the type witness in the conformance.
- Because the associated requirements are satisfied by every concrete conforming type, they also hold "in relation" to the abstract type parameter `T`.

Therefore, to interpret a generic signature, we may need to consult the requirement signatures of some set of protocols that the generic signature "depends on." The question of *which* protocols exactly is settled in Section 15.1, but for now, we can take our generic signatures to exist in the universe of all protocols visible to name lookup, declared in the source program and all serialized modules. This set of protocols is always finite.

Remember how the generic signature of a protocol `P` just has a single conformance requirement [`Self: P`]; thus, the protocol generic signature of `P` is the simplest generic signature that allows us to "look inside" the requirement signature of `P`.

The `-debug-generic-signatures` frontend flag also prints the requirement signature of each protocol as its type checked. While we print a requirement signature as a generic signature with the single generic parameter `Self`, requirement signatures and generic signatures are distinct in theory and implementation.

Example 5.1. We recall that the `Sequence` protocol states two associated requirements:

[`Self.Iterator: IteratorProtocol`]$_{\texttt{Sequence}}$
[`Self.Element == Self.Iterator.Element`]$_{\texttt{Sequence}}$

To motivate the formal system we will introduce in the next section, we consider how the associated requirements of a protocol can be observed from inside a generic declaration that states a conformance requirement to this protocol. Here is a generic function that states a conformance requirement to `Sequence`—two, in fact:

```
func firstTwoEqual<S1: Sequence, S2: Sequence>(_ s1: S1, _ s2: S2)
    where S1.Element == S2.Element, S1.Element: Equatable {
  var iter1 = s1.makeIterator()
  var iter2 = s2.makeIterator()
  return iter1.next()! == iter2.next()!
}
```

Instead of the sugared generic parameter types "`S1`" and "`S2`", we will use canonical types to emphasize that type sugar has no semantic effect. Here is the generic signature of `firstTwoEqual()`:

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
                    τ_0_0.Element: Equatable,
                    τ_0_0.Element == τ_0_1.Element>
```

We expect roughly the following to happen while type checking the function body:

1. "`s1`" has type `τ_0_0`, which conforms to `Sequence`, so it has a `makeIterator()` method that we can call.
2. This method returns `Self.Iterator`. We substitute `Self` with `τ_0_0`, and conclude that "`iter1`" has type `τ_0_0.Iterator`.
3. Similarly, "`iter2`" has type `τ_0_1.Iterator`.

Then, consider the sub-expression "`iter1.next()!`":

1. The requirement signature of `Sequence` tells us that `τ_0_0.Iterator` conforms to `IteratorProtocol`, so we know it has a `next()` method.
2. This method returns `Optional<Self.Element>`, and we substitute `Self` with `τ_0_0.Iterator` to get `Optional<τ_0_0.Iterator.Element>`.
3. The forced unwrap expression has type `τ_0_0.Iterator.Element`.

A similar analysis shows that "`iter2.next()!`" has type `τ_0_1.Iterator.Element`. We now look at the interface type of the `==` operator in the `Equatable` protocol:

```
<Self where Self: Equatable> (Self, Self) -> Bool
```

To type check the expression "`iter1.next()! == iter2.next()!`", the compiler must prove that the two requirements below are consequences of the explicit requirements of our generic signature (together with the requirement signature of `Sequence`):

[`τ_0_0.Iterator.Element == τ_0_1.Iterator.Element`]

[`τ_0_0.Iterator.Element: Equatable`]

We will formalize what this means first, by defining the set of *derived requirements* of a generic signature. In Part IV, we will then describe a *decision procedure* that will tell us if a requirement can be derived or not. It will answer affirmatively for the above two requirements, but give a negative answer for the below requirement for example, which is not a consequence of our signature:

[`τ_0_0.Iterator: Equatable`]

Our decision procedure will also tell us that `τ_0_0.Iterator.Element` is a valid type parameter, but something like `τ_0_0.Element.Iterator` is not.

5.2. Derived Requirements

The next three sections define the *derived requirements* and *valid type parameters* of a generic signature. This will allow us to precisely state those questions the compiler must be able to answer about generic signatures while type checking generic declarations.

We do this by working in a system of deductive reasoning, or a *formal system*, of the sort studied in mathematical logic [80]. We define our formal system in relation to a fixed generic signature G, so each generic signature has its own corresponding formal system. A formal system has three constituent parts, which we describe somewhat informally:

1. A description of all possible *statements* in the formal system, generated by finite combination from a fixed set of symbols.

   Our statements are requirements, such as [`τ_0_0.Element: Equatable`], and type parameters, such as `τ_0_1.Iterator`; their syntactic structure is quite simple, as we've already seen.

2. A finite set of *elementary statements* which are assumed to be true.

   Our elementary statements are the generic parameters and explicit requirements of our generic signature G; all generic parameters are valid type parameters, and all explicit requirements are trivially "derived."

3. A specification of the *inference rules* for deriving new statements from previous statements, in a manner that depends only on their formal syntactic structure.

   We will describe inference rules for requirements and type parameters shortly.

The *theory* of a formal system is the set of all statements we can prove within the system. This always includes the elementary statements, but it should not include *all* possible statements. (A theory where *everything* is true explains nothing.)

The theory of a generic signature is the set union of its derived requirements and valid type parameters. To demonstrate membership in this set, we write down a *derivation*—a finite sequence of steps that give a constructive proof of our desired statement as a consequence of the formal system:

1. A derivation necessarily begins by deriving one or more elementary statements.
2. This is followed by zero or more steps that derive new statements from previous statements via inference rules.
3. The derivation ends with the final step proving the conclusion. (Instead, we can think of a derivation as proving multiple conclusions, if we consider *all* conclusions from each intermediate step.)

There can be many ways to prove the same thing, and we allow "useless" steps whose conclusions are unused, so derivations are not unique in general. The point of writing down a derivation is that we can *check* the reasoning at each step, and convince ourselves the conclusion is a true statement in the theory.

A remark for practitioners. We use our formal system to *specify* various behaviors of the implementation, but the compiler itself does not directly encode derivations as data structures. (We learn in Chapter 17 that we reason about derived requirements and valid type parameters by translating into a *string rewriting* problem.)

Notation. A derivation step will always be written on a single line, with the conclusion first; then on the right-hand side we write the "kind" of derivation step in small caps, followed by the list of assumptions. The conclusion and assumptions are statements:

conclusion (KIND *assumption*)

An *elementary derivation step* has no assumptions, so it proves an elementary statement. Any other kind of derivation step applies an inference rule to one or more assumptions, which are themselves conclusions of previous steps. When discussing a specific derivation step, we write the assumptions in place. When listing a derivation, we prefer to be concise so we *number* each step and refer to the conclusion of a prior step by number:

1. *an elementary statement of "Pooh" sort* (POOH)
2. *a consequence of the above by the "Piglet" principle* (PIGLET 1)

We use the "turnstile operator" $\vdash$ as a predicate. If G is a generic signature and D is a requirement or type parameter, $G \vdash D$ means D is an element of the theory of G, as in "if $G \vdash D$, then ...".

Our formal system has 6 kinds of elementary statements and 17 inference rules; we will discuss them all over the course of this section and the next two:

1. We start with conformance requirements, and same-type requirements between type parameters. The theory completely describes the behavior of this subset of the language. We will also sketch out the remaining requirement kinds.
2. Section 5.3 defines more inference rules, so that the derived same-type requirements of a generic signature generate an equivalence relation on its type parameters. We will study the equivalence classes generated by this relation.
3. We only consider type parameters formed from unbound dependent member types at first. In Section 5.4, we extend our theory to explain bound dependent member types, but this won't reveal anything fundamentally new.

Elementary statements. Let G be a generic signature. We can derive each generic parameter `τ_d_i` of G:

`τ_d_i` (Generic)

Let [`T: P`] be an explicit conformance requirement of G, so `T` is some type parameter and `P` is a protocol. We can derive this requirement:

[`T: P`] (Conf)

Let [`T == U`] be an explicit same-type requirement of G, with `T` and `U` type parameters. We can derive this requirement:

[`T == U`] (Same)

Requirement signatures. We now assume we have derived a specific conformance requirement [`T: P`] for some type parameter `T` and protocol `P`. Each element of the requirement signature of `P` defines an inference rule that generates a new statement:

[`T: P`] + requirement signature of `P` = more statements

The three kinds of inference rules are AssocName, AssocConf, and AssocSame. They correspond to requirement signature elements in the same way that the Generic, Conf, and Same elementary statements correspond to generic signature elements.

From each associated type declaration `A` of `P`, the AssocName inference rule derives the unbound dependent member type `T.A`, with base type `T` and identifier `A`. Notice how this step is a consequence of its assumption, the assumption being the original conformance requirement:

`T.A` (AssocName [`T: P`])

From each associated conformance requirement of `P`, the AssocConf inference rule derives the conformance requirement obtained by substituting `T` in place of `Self`. An arbitrary associated conformance requirement takes the form $[\texttt{Self.U: Q}]_{\texttt{P}}$ for some type parameter `Self.U` and protocol `Q`. The substituted type parameter we described above is denoted by "`T.U`":

$$[\texttt{T.U: Q}] \qquad (\textsc{AssocConf}\ [\texttt{Self.U: Q}]_{\texttt{P}}\ [\texttt{T: P}])$$

From each associated same-type requirement of `P`, the AssocSame inference rule derives a new same-type requirement by substituting `T` in place of `Self`. The most general form of an associated same-type requirement is $[\texttt{Self.U == Self.V}]_{\texttt{P}}$ for type parameters `Self.U` and `Self.V`, so we're forming a same-type requirement from the two substituted type parameters "`T.U`" and "`T.V`":

$$[\texttt{T.U == T.V}] \qquad (\textsc{AssocSame}\ [\texttt{Self.U == Self.V}]_{\texttt{P}}\ [\texttt{T: P}])$$

We will omit the associated requirement from the list of assumptions when listing a derivation in a fixed generic signature, because there is no ambiguity in doing so.

A few words about the meta-syntactic variables used above, such as "`T`" and "`U`" and so on. In the conclusion of an elementary derivation step, they specify the most general form of an explicit requirement of G; we're saying that we have a fixed set of elementary derivation steps, and each one can be obtained from the schema by suitable substitution of concrete entities in place of "`T`", "`U`" and "`P`". When meta-syntactic variables appear in the assumptions of a derivation step as above, they mean something else—we're defining the inference rule by pattern matching, and we can make use of the rule as long as a suitable substitution of the meta-syntactic variables matches each assumption to the conclusion of a previous step.

The general form of a type parameter inside an associated requirement is denoted by "`Self.U`" because it's a type parameter for the protocol generic signature $G_{\texttt{P}}$, so it must be `Self` recursively wrapped in dependent member types, *zero* or more times. This means that `Self` itself is a valid substitution for "`Self.U`". This comes up with protocol inheritance. If `Derived` inherits from `Base`, we have the associated requirement $[\texttt{Self: Base}]_{\texttt{Derived}}$. Given a derivation of `[T: Derived]`, we get a derivation of `[T: Base]` by adding an AssocConf derivation step for $[\texttt{Self: Base}]_{\texttt{Derived}}$.

Example 5.2. The inference rules presented so far can already prove some interesting statements, but we they're not sufficient to explain everything in the informal overview from Example 5.1. Let's revisit the generic signature from that example:

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
                    τ_0_0.Element: Equatable,
                    τ_0_0.Element == τ_0_1.Element>
```

The `Sequence` protocol declares two associated types `Iterator` and `Element`, and since `τ_0_0` conforms to `Sequence`, we can derive `τ_0_0.Element`, for example:

1. [`τ_0_0: Sequence`] (Conf)
2. `τ_0_0.Element` (AssocName 1)

We can derive `τ_0_0.Iterator` similarly. Now consider `τ_0_0.Iterator.Element`. We recall that `Sequence` declares two associated requirements:

[`Self.Iterator: IteratorProtocol`]$_{\texttt{Sequence}}$
[`Self.Element == Self.Iterator.Element`]$_{\texttt{Sequence}}$

The first associated requirement allows us to derive that `τ_0_0.Iterator` conforms to `IteratorProtocol`, and from there we can derive `τ_0_0.Iterator.Element`:

1. [`τ_0_0: Sequence`] (Conf)
2. [`τ_0_0.Iterator: IteratorProtocol`] (AssocConf 1)
3. `τ_0_0.Iterator.Element` (AssocName 2)

We can also derive `τ_0_1.Iterator.Element` if we start from [`τ_0_1: Sequence`]:

1. [`τ_0_1: Sequence`] (Conf)
2. [`τ_0_1.Iterator: IteratorProtocol`] (AssocConf 1)
3. `τ_0_1.Iterator.Element` (AssocName 2)

Remember our original goal in Example 5.1 was to establish a same-type requirement relating `τ_0_0.Iterator.Element` with `τ_0_1.Iterator.Element`, and not just to show that these two type parameters exist. We're going to attempt to write down a derivation. The associated same-type requirement of `Sequence` gives us a same-type requirement between `τ_0_0.Element` and `τ_0_0.Iterator.Element`:

1. [`τ_0_0: Sequence`] (Conf)
2. [`τ_0_0.Element == τ_0_0.Iterator.Element`] (AssocSame 1)

We can also derive a similar statement about `τ_0_1`:

3. [`τ_0_1: Sequence`] (Conf)
4. [`τ_0_1.Element == τ_0_1.Iterator.Element`] (AssocSame 1)

Now remember we have an explicit same-type requirement also:

5. [`τ_0_0.Element == τ_0_1.Element`] (Same)

At this point, we have three same-type requirements (2), (4) and (5) above, but so far, we have no way to conclude anything else:

$$[\tau_0_0.\texttt{Element} == \tau_0_0.\texttt{Iterator.Element}]$$
$$[\tau_0_1.\texttt{Element} == \tau_0_1.\texttt{Iterator.Element}]$$
$$[\tau_0_0.\texttt{Element} == \tau_0_1.\texttt{Element}]$$

We put this example aside until the next section, when we introduce more inference rules for working with same-type requirements between type parameters.

Other requirements. We now extend our formal system with concrete same-type requirements, superclass requirements, and layout requirements, with the caveat that this extended theory is incomplete. All inferences made by these rules are correct, but they do not describe all implemented behaviors of these other requirement kinds. We will see examples of this incompleteness in Section 5.6 and Section 11.4.

We add some more elementary statements. Let G be a generic signature. If $[\texttt{T} == \texttt{X}]$ is an explicit same-type requirement of G, where $\texttt{T}$ is a type parameter and $\texttt{X}$ is a concrete type (so $\texttt{X}$ is not a type parameter but it may contain type parameters):

$$[\texttt{T} == \texttt{X}] \qquad \text{(Concrete)}$$

If $[\texttt{T: C}]$ is an explicit superclass requirement of G, so $\texttt{C}$ is a class type:

$$[\texttt{T: C}] \qquad \text{(Super)}$$

If $[\texttt{T: AnyObject}]$ is an explicit layout requirement of G:

$$[\texttt{T: AnyObject}] \qquad \text{(Layout)}$$

We also have inference rules for the associated requirement forms of the above, so in what follows, we assume we can derive $[\texttt{T: P}]$ for some $\texttt{T}$ and $\texttt{P}$.

Suppose that $\texttt{P}$ declares an associated same-type requirement $[\texttt{Self.U} == \texttt{X}]_{\texttt{P}}$, for some $\texttt{Self.U}$ and concrete type $\texttt{X}$. We substitute $\texttt{T}$ for $\texttt{Self}$ inside $\texttt{Self.U}$ to get a type parameter $\texttt{T.U}$, and inside $\texttt{X}$ to get a concrete type $\texttt{X}'$. We can then derive:

$$[\texttt{T.U} == \texttt{X}'] \qquad \text{(AssocConcrete } [\texttt{Self.U} == \texttt{X}]_{\texttt{P}}\ [\texttt{T: P}]\text{)}$$

For an associated superclass requirement $[\texttt{Self.U: C}]_{\texttt{P}}$, we substitute $\texttt{T}$ for $\texttt{Self}$ in $\texttt{C}$ to get $\texttt{C}'$:

$$[\texttt{T.U: C}'] \qquad \text{(AssocSuper } [\texttt{Self.U: C}]_{\texttt{P}}\ [\texttt{T: P}]\text{)}$$

Finally, an associated layout requirement does not have anything to substitute on the right-hand side, so we derive the same statement about $\texttt{T.U}$:

$$[\texttt{T.U: AnyObject}] \qquad \text{(AssocLayout } [\texttt{Self.U: AnyObject}]_{\texttt{P}}\ [\texttt{T: P}]\text{)}$$

At this point, we've seen all of the kinds of elementary statements that exist, but a few more inference rules remain to be defined. Here is a quick summary of all the elementary statements and inference rules we've seen so far:

Basic model:		
Generic	Conf	Same
AssocName	AssocConf	AssocSame
Other requirements:		
Concrete	Super	Layout
AssocConcrete	AssocSuper	AssocLayout

5.3. Valid Type Parameters

In this section we're going to take a closer look at the valid type parameters that can be derived from a generic signature. Chapter 3 introduced two notions of equality of type parameters: canonical type equality tells us if they have the same spelling, and *reduced type equality*, relative to a generic signature, also takes same-type requirements into account. We will now define reduced type equality on type parameters, and generalize it to all interface types in Section 5.6.

Reduced type equality is an *equivalence relation*, so let's review the concept now. Given a fixed set that we call the *domain*, a relation models some property that a pair of elements might have. In programming languages, relations are typically functions taking a pair of values to a true or false result. In mathematics, we instead imagine a relation is the set of those ordered pairs (x, y) such that the property is true of x and y.

Definition 5.3. Let S be a set. A *relation* with domain S is a subset of the Cartesian product $S \times S$.

Definition 5.4. An *equivalence relation* $R \subseteq S \times S$ is reflexive, symmetric and transitive:

- R is *reflexive* if $(x, x) \in R$ for all $x \in S$.
- R is *symmetric* if $(x, y) \in R$ implies that $(y, x) \in R$ for all x, $y \in S$.
- R is *transitive* if (x, y), $(y, z) \in R$ implies that $(x, z) \in R$ for all x, y, $z \in S$.

Imagine a fraction a/b as a purely formal object, a pair of integers a, b with $b \neq 0$. While $1/2$, $2/4$, and $(-3)/(-6)$ are distinct fractions, they measure the same proportion, as we see by cross-multiplying denominators. We have two equivalence relations:

- a/b and c/d are "identical" if $a = b$ and $c = d$.
- a/b and c/d are "equivalent" if $ad = bc$.

Definition 5.5. Let R be an equivalence relation with domain S, and let $x \in S$. The *equivalence class* of x, denoted $[\![x]\!]$, is the set of all $y \in S$ such that $(x, y) \in R$.

Under our "equivalence of proportion" relation, the equivalence class of $1/2$ is the set of all fractions of the form $n/2n$, as n ranges over all non-zero integers. Every fraction belongs to exactly one equivalence class, so our equivalence relation partitions the set of fractions into disjoint equivalence classes. We now increase the level of abstraction, and consider the set whose *elements* are equivalence classes of fractions. This is $\mathbb{Q}$, the set of *rational numbers*, where two equivalent fractions represent the same rational number. We can repeat this construction with any equivalence relation.

Proposition 5.6. Let R be an equivalence relation with domain S. The equivalence classes of R form a disjoint partition of S.

Proof. We need to establish these two facts:

1. Every $x \in S$ belongs to *at least* one equivalence class.
2. Every $x \in S$ belongs to *at most* one equivalence class, so if $x \in [\![y]\!]$ and $x \in [\![z]\!]$ for some y, $z \in S$, then in fact $[\![y]\!] = [\![z]\!]$ (this is equality of sets, meaning they have the same elements).

For the first part, the assumption that R is reflexive says that $(x, x) \in R$ for all $x \in S$. By definition of $[\![x]\!]$, this means that $x \in [\![x]\!]$, so every x is, at the very least, an element of its *own* equivalence class $[\![x]\!]$.

For the second part, assume that for some y, $z \in S$, there exists an $x \in [\![y]\!] \cap [\![z]\!]$. To show that $[\![y]\!] = [\![z]\!]$, we prove that $[\![y]\!] \subseteq [\![z]\!]$ and $[\![z]\!] \subseteq [\![y]\!]$. To see that $[\![y]\!] \subseteq [\![z]\!]$, we take an arbitrary element $t \in [\![y]\!]$, and chase a series of equivalences to derive that $t \in [\![z]\!]$:

1. $t \in [\![y]\!]$ and $x \in [\![y]\!]$, so (t, y), $(x, y) \in R$.
2. $(x, y) \in R$, but R is symmetric, so $(y, x) \in R$.
3. (t, y), $(y, x) \in R$, but R is transitive, so $(t, x) \in R$.
4. (t, x), $(x, z) \in R$, but R is transitive, so $(t, z) \in R$; that is, $t \in [\![z]\!]$.

This gives us $[\![y]\!] \subseteq [\![z]\!]$. To prove $[\![z]\!] \subseteq [\![y]\!]$, we pretend to copy and paste the above, and swap y and z everywhere that they appear. (We tend to avoid writing out the other direction of the proof in a situation like this where it is completely mechanical.)

We've shown that if two equivalence classes of S have at least one element in common, they must in fact coincide. The only other possibility is that the two equivalence classes are disjoint, that is, their intersection is the empty set. Note that we made use of all three defining properties of an equivalence relation; the result no longer holds if we drop the assumption of reflexivity, symmetry, or transitivity. □

Reduced type equality. We now define an equivalence relation on type parameters. We want to say that two type parameters are equivalent if we can derive a same-type requirement between them:

Definition 5.7. Let G be a generic signature. The *reduced type equality* relation on the valid type parameters of G is the set of all pairs `T` and `U` such that $G \vdash$ `[T == U]`.

For this to work, we must extend our formal system with three new inference rules, one for each defining axiom of an equivalence relation. The first rule says that if we can derive a valid type parameter `T`, we can derive the trivial same-type requirement `[T == T]`. Note that we're deriving a *requirement* from a *type parameter*; this is the only time we can do that:

`[T == T]` (Reflex `T`)

Next, if we can derive a same-type requirement `[T == U]` for type parameters `T` and `U`, we can derive the opposite same-type requirement:

`[U == T]` (Sym `[T == U]`)

Finally, if we can derive two same-type requirements `[T == U]` and `[U == V]` where the right-hand side of the first is identical to the left-hand side of the second, we can derive a same-type requirement that gets us from one end to the other in one jump:

`[T == V]` (Trans `[T == U]` `[U == V]`)

Given these new rules, we immediately see:

Proposition 5.8. Reduced type equality is an equivalence relation.

Proof. Each axiom follows from the corresponding inference rule:

- (Reflexivity) Suppose that `T` is a valid type parameter of G. Given a derivation $G \vdash$ `T`, we derive $G \vdash$ `[T == T]` via Reflex. Thus, `T` is equivalent to itself.
- (Symmetry) Suppose that `T` is equivalent to `U`. Given a derivation $G \vdash$ `[T == U]`, we derive $G \vdash$ `[U == T]` via Sym. Thus, `U` is equivalent to `T`.
- (Transitivity) Suppose that `T` is equivalent to `U`, and `U` is equivalent to `V`. Given the two derivations $G \vdash$ `[T == U]` and $G \vdash$ `[U == V]`, we concatenate them together, and derive $G \vdash$ `[T == V]` via Trans. Thus, `T` is equivalent to `V`.

We are careful to limit our relation's domain to the valid type parameters of G, instead of considering all type parameters that can be formed syntactically. If `T` is some invalid type parameter that cannot be derived from G, we cannot use Reflex to derive `[T == T]`, so we would no longer have an equivalence relation. □

So far, nothing prevents us from deriving a requirement [`T == U`] where `T` or `U` is not itself derivable. If this happens, we exclude the ordered pair from our relation. We will not worry about this for now, because Section 11.3 will show that such situations can be ruled out by diagnosing invalid requirements written by the user.

Example 5.9. We can use these inference rules to derive more requirements. Returning to Example 5.2, we were able to derive (2), (4) and (5), but then we got stuck:

1. [`τ_0_0: Sequence`] (Conf)
2. [`τ_0_0.Element == τ_0_0.Iterator.Element`] (AssocSame 1)
3. [`τ_0_1: Sequence`] (Conf)
4. [`τ_0_1.Element == τ_0_1.Iterator.Element`] (AssocSame 3)
5. [`τ_0_0.Element == τ_0_1.Element`] (Same)

We proceed as follows. We apply Sym to (2), and then observe that (6), (5) and (4) now form a chain where the right-hand side of each one is identical to the left-hand side of the next. Two applications of Trans give us what we're looking for:

6. [`τ_0_0.Iterator.Element == τ_0_0.Element`] (Sym 2)
7. [`τ_0_0.Iterator.Element == τ_0_1.Element`] (Trans 6 5)
8. [`τ_0_0.Iterator.Element == τ_0_1.Iterator.Element`] (Trans 7 4)

We've see that `τ_0_0.Iterator.Element` is equivalent to `τ_0_1.Iterator.Element`. Along the way, we've discovered that our generic signature has these four single-element equivalence classes,

{`τ_0_0`}, {`τ_0_1`}, {`τ_0_0.Iterator`}, {`τ_0_1.Iterator`},

and one final equivalence class formed from the other four type parameters,

{`τ_0_0.Element`, `τ_0_1.Element`,
`τ_0_0.Iterator.Element`, `τ_0_1.Iterator.Element`}.

To prove that `τ_0_0.Iterator.Element` conforms to `Equatable`, we need another rule.

Compatibility. Our equivalence relation on fractions had an interesting property: if we pick any two fractions from a pair of equivalence classes and add them together (or multiply, etc), the equivalence class of the result does not depend on our choice of representatives. In Swift generics, we have an analogous goal: any observable behavior of a type parameter, be it conforming to a protocol, having a superclass bound, being fixed to a concrete type, or having a dependent member type with a given name, should only depend on its equivalence class, and not on the "spelling."

Once again, we wave our magic wand, and decree that reduced type equality is to have this property, by adding new inference rules. These rules relate same-type requirements with the other requirement kinds by replacing the other requirement's subject type. That is, if we can derive a same-type requirement [`T == U`] and a conformance requirement [`U: P`], we can derive [`T: P`]:

[`T: P`] (SameConf [`U: P`] [`T == U`])

A same-type requirement [`T == U`] also composes with the other requirement kinds:

[`T == X`] (SameConcrete [`U == X`] [`T == U`])
[`T: C`] (SameSuper [`U: C`] [`T == U`])
[`T: AnyObject`] (SameLayout [`U: AnyObject`] [`T == U`])

There is a second compatibility condition. We want equivalent type parameters to have equivalent member types, as follows. Suppose we can derive a conformance requirement [`U: P`] and a same-type requirement [`T == U`]. For every associated type `A` of `P`, we can derive a same-type requirement [`T.A == U.A`]:

[`T.A == U.A`] (SameName [`U: P`] [`T == U`])

We have yet to describe the inference rules for bound dependent member types, so that's coming up next. Apart from that, our formal system is complete, so let's look at a few more examples to justify these rules we just added.

Example 5.10. To continue Example 5.9, we can use the SameConf inference rule to derive [`τ_0_0.Iterator.Element: Equatable`] from [`τ_0_0.Element: Equatable`], because as we saw, `τ_0_0.Element` and `τ_0_0.Iterator.Element` are equivalent:

1. [`τ_0_0.Element: Equatable`] (Conf)
2. [`τ_0_0.Element == τ_0_0.Iterator.Element`] (AssocSame 1)
3. [`τ_0_0.Iterator.Element == τ_0_0.Element`] (Sym 2)
3. [`τ_0_0.Iterator.Element: Equatable`] (SameConf 1 3)

In our generic signature, this *entire* equivalence class conforms to `Equatable`:

{`τ_0_0.Element`, `τ_0_1.Element`,
`τ_0_0.Iterator.Element`, `τ_0_1.Iterator.Element`}.

Example 5.11. To see the SameName inference rule in action, we recall the generic signature of `sameIter()` from the beginning of the chapter:

```
<S1, S2 where S1: Sequence, S2: Sequence,
              S1.Iterator == S2.Iterator>
```

We also saw another declaration, `sameIterAndElt()`, which states both requirements:

[τ_0_0.Iterator == τ_0_1.Iterator]

[τ_0_0.Element == τ_0_1.Element]

We claimed that requirement minimization drops the second requirement, because it is redundant, giving `sameIterAndElt()` the same generic signature as `sameIter()`. This means that we should be able to derive the second requirement without invoking *itself* as an elementary statement. To do this, we take [τ_0_0.Iterator == τ_0_1.Iterator], and apply SameName to derive the same-type requirement (4):

1. [τ_0_0.Iterator == τ_0_1.Iterator] (Same)
2. [τ_0_1: Sequence] (Conf)
3. [τ_0_1.Iterator: Sequence] (AssocConf 1)
4. [τ_0_0.Iterator.Element == τ_0_1.Iterator.Element] (SameName 3 1)

We then rewrite both sides of (4) into their short form by making use of the associated same-type requirement of `Sequence`:

5. [τ_0_0: Sequence] (Conf)
6. [τ_0_0.Element == τ_0_0.Iterator.Element] (AssocSame 5)
7. [τ_0_1.Element == τ_0_1.Iterator.Element] (AssocSame 2)
8. [τ_0_1.Iterator.Element == τ_0_1.Element] (Sym 7)
9. [τ_0_0.Element == τ_0_1.Iterator.Element] (Trans 6 4)
10. [τ_0_0.Element == τ_0_1.Element] (Trans 9 8)

We could not have derived this requirement without SameName.

Example 5.12. We can conjure up a generic signature with infinitely many equivalence classes. We start with this protocol:

```
protocol N {
  associatedtype A: N
}
```

Now, consider the protocol generic signature $G_{\texttt{N}}$. We can derive an infinite sequence of conformance requirements from [τ_0_0: N], by repeated application of the AssocConf inference rule with the associated conformance requirement $[\texttt{Self.A: N}]_{\texttt{N}}$:

1. [τ_0_0: N] (Conf)
2. [τ_0_0.A: N] (AssocConf 1)
3. [τ_0_0.A.A: N] (AssocConf 2)
4. ...

We can also derive infinitely many valid type parameters:

5. τ_0_0 (Generic)
6. $\tau_0_0.A$ (AssocName 2)
7. $\tau_0_0.A.A$ (AssocName 3)
8. ...

Each one of these type parameters is in its own equivalence class, because we cannot derive any non-trivial same-type requirements. We've shown that $G_{\texttt{N}}$ defines an infinite set of equivalence classes.

Example 5.13. Here is a simplified form of the standard library `Collection` protocol:

```
protocol Collection: Sequence {
  associatedtype SubSequence: Collection
      where Element == SubSequence.Element
            SubSequence == SubSequence.SubSequence
}
```

Our protocol inherits the `Element` and `Iterator` associated types from `Sequence`, then declares a new associated type, and states these four associated requirements:

$$[\texttt{Self: Sequence}]_{\texttt{Collection}}$$
$$[\texttt{Self.SubSequence: Collection}]_{\texttt{Collection}}$$
$$[\texttt{Self.Element == Self.SubSequence.Element}]_{\texttt{Collection}}$$
$$[\texttt{Self.SubSequence == Self.SubSequence.SubSequence}]_{\texttt{Collection}}$$

We can read the associated requirements as follows:

1. The first conformance requirement states that all collections are sequences.
2. The second conformance requirement states that a subsequence of a collection is another collection, of a possibly different concrete type.
3. The first same-type requirement states that a subsequence of an arbitrary collection always has the same element type as the original.
4. The second same-type requirement states that a subsequence of a subsequence must have the same type as a subsequence of the original collection.

For example, the `SubSequence` of `Array<Int>` is `ArraySlice<Int>`, and `SubSequence` of `ArraySlice<Int>` is again `ArraySlice<Int>`. (Type witnesses of concrete conformances are discussed in Section 7.3).

We're going to look at the protocol generic signature $G_{\texttt{Collection}}$ and attempt to understand its equivalence classes. We can use the protocol inheritance relationship to derive `τ_0_0.Iterator`:

1. [`τ_0_0: Collection`] (Conf)
2. [`τ_0_0: Sequence`] (AssocConf 1)
3. `τ_0_0.Iterator` (AssocName 2)

The first thing we notice is that `τ_0_0` and `τ_0_0.Iterator` are not equivalent to each other or to any other type parameter. Each one is in an equivalence class by itself, so we have our first two equivalence classes.

To understand how the remaining equivalence classes are formed, we notice that $[\texttt{Self.SubSequence: Collection}]_{\texttt{Collection}}$ generates an infinite family of conformance requirements, like $[\texttt{Self.A: N}]_{\texttt{N}}$ did in the previous example:

1. [`τ_0_0: Collection`] (Conf)
2. [`τ_0_0.SubSequence: Collection`] (AssocConf 1)
3. [`τ_0_0.SubSequence.SubSequence: Collection`] (AssocConf 2)
4. ...

To talk about this phenomenon, we introduce the following notation:

$$\texttt{τ_0_0.SubSequence}^n := \begin{cases} \texttt{τ_0_0} & n = 0 \\ \texttt{τ_0_0.SubSequence} & n = 1 \\ \texttt{τ_0_0.SubSequence}^{n-1}\texttt{.SubSequence} & n > 1 \end{cases}$$

So the theory of $G_{\texttt{Collection}}$ contains $[\texttt{τ_0_0.SubSequence}^n\texttt{: Collection}]$ for all $n \geq 0$. Furthermore, for each $n \geq 0$, we can derive the following from (1) below:

1. $[\texttt{τ_0_0.SubSequence}^n\texttt{: Collection}]$ (...)
2. $[\texttt{τ_0_0.SubSequence}^n\texttt{: Sequence}]$ (AssocConf 1)
3. $[\texttt{τ_0_0.SubSequence}^n\texttt{.Element == τ_0_0.SubSequence}^{n+1}\texttt{.Element}]$ (AssocSame 1)
4. $[\texttt{τ_0_0.SubSequence}^n\texttt{.Element == τ_0_0.SubSequence}^n\texttt{.Iterator.Element}]$ (AssocSame 2)

The infinite families (3) and (4) define a single equivalence class. This equivalence class contains $\texttt{τ_0_0.SubSequence}^n\texttt{.Element}$ and $\texttt{τ_0_0.SubSequence}^n\texttt{.Iterator.Element}$, for all $n \geq 0$. (In particular, it contains `τ_0_0.Element`.)

We can also generate another infinite family of same-type requirements:

5. $[\texttt{τ_0_0.SubSequence}^{n+1}\texttt{ == τ_0_0.SubSequence}^{n+2}]$ (AssocSame 1)

We get an equivalence class containing $\texttt{τ_0_0.SubSequence}^n$ for $n \geq 1$. Finally, applying SameName gives us one more infinite family:

6. $[\texttt{τ_0_0.SubSequence}^{n+1}\texttt{.Iterator == τ_0_0.SubSequence}^{n+2}\texttt{.Iterator}]$ (SameName 5 2)

These form our final equivalence class; the `SubSequence` may have a distinct `Iterator` from the original sequence.

We see that $G_{\texttt{Collection}}$ defines five equivalence classes, and three of those contain infinitely many representative type parameters each:

$$\{\texttt{τ_0_0}\}, \tag{1}$$

$$\{\texttt{τ_0_0.Element}, \texttt{τ_0_0.SubSequence.Element}, \ldots\} \tag{2}$$
$$\cup \{\texttt{τ_0_0.Iterator.Element}, \texttt{τ_0_0.SubSequence.Iterator.Element}, \ldots\},$$

$$\{\texttt{τ_0_0.Iterator}\}, \tag{3}$$

$$\{\texttt{τ_0_0.SubSequence}, \texttt{τ_0_0.SubSequence.SubSequence}, \ldots\}, \tag{4}$$

$$\{\texttt{τ_0_0.SubSequence.Iterator}, \tag{5}$$
$$\texttt{τ_0_0.SubSequence.SubSequence.Iterator},$$
$$\ldots\}$$

To describe the conformance requirements that apply to each equivalence class, it is sufficient to pick a single representative conformance requirement for each combination of type parameter and protocol, because the SameConf inference rule allows us to derive the rest. This table summarizes our "pen and paper" investigation:

Representative:	**Conforms to:**
`τ_0_0`	`Collection` and `Sequence`
`τ_0_0.Element`	none
`τ_0_0.Iterator`	`IteratorProtocol`
`τ_0_0.SubSequence`	`Collection` and `Sequence`
`τ_0_0.SubSequence.Iterator`	`IteratorProtocol`

That's pretty much all there is to say about the generic signature $G_{\texttt{Collection}}$, however our above description is somewhat lacking because it doesn't make the member type relationships apparent. We will revisit all of the generic signatures we saw here in Section 8.3, when we introduce the *type parameter graph*. We will define an edge relation on equivalence classes, which become the vertices of a graph. This will give us a visual aid for understanding member type relationships.

Notice how our formal system can generate an infinite theory! We saw that $G_{\texttt{N}}$ has an infinite set of finite equivalence classes, while in $G_{\texttt{Collection}}$, the equivalence classes themselves were infinite. Of course both can happen simultaneously. A generic signature with an infinite set of infinite equivalence classes will appear in Section 16.3.

5.4. Bound Type Parameters

We saw in Section 3.1 that dependent member types come in two varieties:

- Unbound dependent member types refer to an identifier. Denoted `T.A` for some base type `T` and identifier `A`.
- Bound dependent member types refer to an associated type declaration. Denoted `T.[P]A` for some base type `T` and associated type declaration `A` of some protocol `P`.

Definition 5.14. We define the following for convenience:

- An *unbound type parameter* is a generic parameter type, or an unbound dependent member type whose base type is another unbound type parameter.
- A *bound type parameter* is a generic parameter type, or a bound dependent member type whose base type is another bound type parameter.

The bound and unbound type parameters do not exhaustively partition the set of all type parameters. A generic parameter type is *both* bound and unbound per the above, while a type parameter like `τ_0_0.[Sequence]Iterator.Element` is neither bound nor unbound, because it contains a mix of bound and unbound dependent member types.

The fundamental tension here is the following:

- Unbound type parameters types appear when type resolution resolves the requirements in a `where` clause; they directly represent what was written by the user.
- Type substitution only operates on bound dependent member types, because as we will see in Section 7.4, we need all three pieces of information that describe one as a valid type parameter: the base type, the protocol, and the associated type declaration.

We often apply substitution maps to requirements of generic signatures and requirement signatures, and the interface types of declarations. Indeed, all of these semantic objects must only contain bound type parameters. We resolve this as follows:

- Requirement minimization (Section 11.4) converts unbound type parameters in the `where` clause into bound type parameters when building a generic signature.
- Generic signature queries (Section 5.6) accept unbound type parameters. One can obtain a bound type parameter from an unbound type parameter by asking the generic signature for the unbound type parameter's *reduced type*.
- Type resolution uses generic signature queries to form bound type parameters when resolving the interface type of a declaration (Chapter 9).

We now extend our formal system to describe these behaviors. As always, let G be a generic signature. We assume that $G \vdash$ [`T: P`] for some `T` and `P`.

We first add an inference rule analogous to AssocName, except that it derives a bound dependent member type. From each associated type declaration `A` of `P`, the AssocDecl inference rule derives the bound dependent member type `T.[P]A`, with base type `T`, referencing the associated type declaration `A`:

`T.[P]A` (AssocDecl [`T: P`])

To encode the equivalence of bound and unbound dependent member types, we also add an inference rule that derives a same-type requirement between the two. That is, from each associated type declaration `A` of `P`, the AssocBind inference rule derives a same-type requirement between the two dependent member types that AssocDecl and AssocName would derive under the same assumptions:

[`T.[P]A == T.A`] (AssocBind [`T: P`])

The SameName inference rule, which gave us [`T.A == U.A`] from [`U: P`] and [`T == U`], also has a bound type parameter equivalent. Under the same assumptions, SameDecl derives [`T.[P]A == U.[P]A`]:

[`T.[P]A == U.[P]A`] (SameDecl [`U: P`] [`T == U`])

The SameDecl inference rule is fundamentally different from all of the others, because its introduction does not contribute any novel statements to our theory:

Proposition 5.15. Any derivation containing a SameDecl step can be transformed into one without.

Proof. Given $G \vdash$ [`U: P`] and $G \vdash$ [`T == U`], we can derive [`T.[P]A == U.[P]A`] without SameDecl by adding the below steps, for any suitable choice of `T`, `U`, `P` and `A`:

1. [`U: P`] (...)
2. [`T == U`] (...)
3. [`T.A == U.A`] (SameName 1 2)
4. [`U.[P]A == U.A`] (AssocBind 1)
5. [`U.A == U.[P]A`] (Sym 4)
6. [`T: P`] (SameConf 1 2)
7. [`T.[P]A == T.A`] (AssocBind 6)
8. [`T.[P]A == U.A`] (Trans 7 3)
9. [`T.[P]A == U.[P]A`] (Trans 8 5)

However, it is certainly more convenient to write 1 derivation step instead of 7 every time we need this equivalence. Thus, SameDecl is syntax sugar for our formal system. □

Example 5.16. Consider the protocol generic signature $G_{\texttt{Sequence}}$, and recall that `Sequence` states two associated requirements:

$$[\texttt{Self.Iterator: IteratorProtocol}]_{\texttt{Sequence}}$$
$$[\texttt{Self.Element == Self.Iterator.Element}]_{\texttt{Sequence}}$$

Let's continue to pretend that those associated requirements are written with unbound type parameters at first, but allow use of the new inference rules. We can derive the bound type parameter `τ_0_0.[Sequence]Iterator.[IteratorProtocol]Element`:

1. [`τ_0_0: Sequence`] (Conf)
2. [`τ_0_0.Iterator: IteratorProtocol`] (AssocConf 1)
3. [`τ_0_0.[Sequence]Iterator == τ_0_0.Iterator`] (AssocBind 1)
4. [`τ_0_0.[Sequence]Iterator: IteratorProtocol`] (SameConf 2 3)
5. `τ_0_0.[Sequence]Iterator.[IteratorProtocol]Element` (AssocDecl 4)

We used the new inference rules in (3) and (5). Notice how (4) looks a lot like our associated conformance requirement, but with a bound type parameter.

Now, we can also define our formal system so that the explicit requirements of our generic signature and the associated requirements of protocols are written in terms of bound type parameters, as they actually would be in the implementation:

$$[\texttt{Self.[Sequence]Iterator: IteratorProtocol}]_{\texttt{Sequence}}$$
$$[\texttt{Self.[Sequence]Element == Self.[Sequence]Iterator.[IteratorProtocol]Element}]_{\texttt{Sequence}}$$

We see that `τ_0_0.[Sequence]Iterator.[IteratorProtocol]Element` can be derived, now with fewer steps:

1. [`τ_0_0: Sequence`] (Conf)
2. [`τ_0_0.[Sequence]Iterator: IteratorProtocol`] (AssocConf 1)
3. `τ_0_0.[Sequence]Iterator.[IteratorProtocol]Element` (AssocDecl 2)

And there's a symmetry here—we can just as easily derive the unbound type parameter `τ_0_0.Iterator.Element` from our associated requirements that contain bound type parameters:

1. [`τ_0_0: Sequence`] (Conf)
2. [`τ_0_0.[Sequence]Iterator: IteratorProtocol`] (AssocConf 1)
3. [`τ_0_0.[Sequence]Iterator == τ_0_0.Iterator`] (AssocBind 1)
4. [`τ_0_0.Iterator == τ_0_0.[Sequence]Iterator`] (Sym 3)
5. [`τ_0_0.Iterator: IteratorProtocol`] (SameConf 2 4)
6. `τ_0_0.Iterator.Element` (AssocName 5)

In fact, we will prove the following in Chapter 11:

- Theorem 11.9 will show that every equivalence class of valid type parameters always contains at least one bound and at least one unbound type parameter.
- Proposition 11.10 will tell us that we can start with a list of explicit requirements written with bound or unbound type parameters without changing the theory.

Example 5.17. Suppose a generic parameter `τ_0_0` conforms to two protocols `P1` and `P2`, and both declare an associated type named `A`. The same unbound dependent type `τ_0_0.A` can be derived from either conformance requirement, and this effectively merges the associated requirements imposed on `A` in the language semantics. This remains true once we add bound dependent member types, because we can derive:

1. [`τ_0_0: P1`] (Conf)
2. [`τ_0_0.[P1]A == τ_0_0.A`] (AssocBind 1)
3. [`τ_0_0: P2`] (Conf)
4. [`τ_0_0.[P2]A == τ_0_0.A`] (AssocBind 3)
5. [`τ_0_0.A == τ_0_0.[P2]A`] (Sym 4)
6. [`τ_0_0.[P1]A == τ_0_0.[P2]A`] (Trans 2 5)

A special case is when the first protocol inherits from the other. That is, re-stating an associated type declaration from an inherited protocol does not change the equivalence class structure of a generic signature.

Bound type parameters don't seem to yield anything new. Why bother then? Type substitution must issue generic signature queries in the general case anyway, and we could just consult the generic signature every time we decompose a dependent member type. Instead, our representation basically pre-computes certain information and encodes it in the structure of the dependent member type itself. This avoids generic signature queries in simple cases, and our formal system proves this is equivalent.

Summary. A complete summary of all elementary statements and inference rules in the derived requirements formalism appears in Appendix B. Bound type parameters are trivial from a theoretical standpoint, and our theory of concrete type requirements is incomplete, so we will mostly work with this subset of elementary statements and inference rules:

Elementary statements:	Generic	Conf	Same
Inference rules:	AssocName Reflex	AssocConf Sym SameConf	AssocSame Trans SameName

So far, we've only used our formal system to derive concrete statements *in* a fixed generic signature. In later chapters, we prove results *about* generic signatures:

- In Chapter 11, we describe how we diagnose invalid requirements, and show that if no diagnostics are emitted, our formal system has a particularly nice theory.
- In Chapter 12, we take a closer look at derived conformance requirements, to describe substitution of dependent member types; in particular, we prove that a certain algorithm must terminate.
- In Chapter 16, we use derived requirements to show that a Swift protocol can encode an arbitrary finitely-presented monoid, which demonstrates that a generic signature can have an undecidable theory.
- In Chapter 17, we will translate the explicit requirements of a generic signature into rewrite rules, and then show that derived requirements correspond to *rewrite paths* under this mapping. This provides a correctness proof for the implementation.

5.5. Reduced Type Parameters

An equivalence class of type parameters might be infinite, or finite but large. For this reason, we cannot model an equivalence class as a set in the implementation. Instead, we define a way to consistently select a unique representative from each equivalence class, called the *reduced type parameter* of the equivalence class. The reduced type parameter can "stand in" for its entire equivalence class, and the set of all reduced type parameters gives another description of the equivalence class structure of a generic signature.

To continue the analogy with fractions from the previous section, we don't usually think of a single rational number as an infinite set of fractions. Instead, after performing a series of arithmetic operations, we *reduce* the result to lowest terms by eliminating common factors from the numerator and denominator. For example, $2/4$ and $(-3)/(-6)$ reduce to $1/2$, while $1/2$ is already reduced. This gives us a new way to check if two fractions are equivalent: we reduce both, and then check if the reduced fractions are identical. A key fact is that we can find a reduced fraction by *ordering* the elements of its equivalence class: the reduced fraction has the smallest positive denominator.

Definition 5.18. A *partial order* $R \subseteq S \times S$ is anti-reflexive and transitive:

- R is *anti-reflexive* if $(x, x) \notin R$ for all $x \in S$.
- R is *transitive* if (x, y), $(y, z) \in R$ implies that $(x, z) \in R$.

If R is clear from context, we write $x < y$ instead of $(x, y) \in R$ and $x \not< y$ instead of $(x, y) \notin R$. We also define $>$, $\leq$ and $\geq$ in the usual way in terms of $<$.

Note that $x < y$ and $y < x$ cannot both be true at once, because then we'd have $x < x$ by transitivity, which contradicts the requirement that $x \not< x$ for all $x \in S$. Therefore in a partial order, *at most* one of the below is true for a pair of elements x, $y \in S$:

1. $x = y$,
2. $x < y$,
3. $x > y$.

If none of these are true, we say x and y are *incomparable*. We will consider general partial orders later, but for now we're going to restrict our attention to those orders where *exactly* one of the three conditions above is true, for any pair of elements:

Definition 5.19. A *linear order* is a partial order without incomparable elements. In a linear order, $x \not< y$ is another way of saying $x \geq y$.

We will define a linear order on type parameters, denoted by $<$. This will express the idea that if $G \vdash$ `[T == U]` and `T` $<$ `U`, then `T` is "more reduced" than `U`, within this equivalence class that contains both. The minimum out of all representatives is then the reduced type parameter itself. We will start by measuring the "complexity" of a type parameter by counting the number of dependent member types involved in its construction:

Definition 5.20. The *length* of a type parameter `T`, denoted by $|$`T`$|$, is a natural number. The length of a generic parameter type is 1, while the length of a dependent member type `U.A` or `U.[P]A` is recursively defined as $|$`U`$| + 1$.

We want the reduced type parameter to be one of minimum possible length, and we also want to be able to apply substitution maps to it, so it ought to be a bound type parameter. This gives us two conditions our type parameter order must satisfy:

1. If $|$`T`$| < |$`U`$|$, then `T` $<$ `U`.
2. Bound dependent member types precede unbound dependent member types.

Example 5.21. In Example 5.13, we studied a simplified `Collection` protocol. We saw that the equivalence class of `τ_0_0.SubSequence` in $G_{\texttt{Collection}}$ contains an infinite sequence of type parameters of increasing length,

$$\{\texttt{τ_0_0.SubSequence}^n\}_{n \geq 1}.$$

After adding inference rules for bound dependent member types, our equivalence class will also contain some of those:

$$\{\texttt{τ_0_0.[Collection]SubSequence}^n\}_{n \geq 1}.$$

(It also contains "mixed" bound and unbound type parameters.) The reduced type parameter of this equivalence class must be `τ_0_0.[Collection]SubSequence`.

Each equivalence class of $G_{\texttt{Collection}}$ has a unique representative of minimum length, which isn't true of all generic signatures in general. This means that if two equivalent type parameters have the same length, they only differ in being bound or unbound, so our two conditions completely determine the reduced type parameter of each equivalence class:

```
τ_0_0
τ_0_0.[Sequence]Element
τ_0_0.[Sequence]Iterator
τ_0_0.[Collection]SubSequence
τ_0_0.[Collection]SubSequence.[Sequence]Iterator
```

(The real `Collection` protocol defines these reduced types together with a few more equivalence classes related to the `Index` and `Indices` associated types.)

In general, we must also order type parameters of equal length, so we show how we do this now. All of the below algorithms take a pair of values x and y and then compute whether $x < y$, $x > y$, or $x = y$ simultaneously. We start with the generic parameters, which are the type parameters of length 1.

Algorithm 5A (Generic parameter order)**.** Takes two generic parameter types `τ_d_i` and `τ_D_I` as input. Returns one of "<", ">", or "=" as output.

1. If `d` < `D`, return "<".
2. If `d` > `D`, return ">".
3. If `d` = `D` and `i` < `I`, return "<".
4. If `d` = `D` and `i` > `I`, return ">".
5. If `d` = `D` and `i` = `I`, return "=".

To order dependent member types, we must order associated type declarations, and to do that we must order protocol declarations.

Algorithm 5B (Protocol order)**.** Takes protocols `P` and `Q` as input, and returns one of "<", ">", or "=" as output.

1. Compare the parent module names of `P` and `Q` with the usual lexicographic order on identifiers. Return the result if it is "<" or ">". Otherwise, `P` and `Q` are declared in the same module, so keep going.
2. Compare the names of `P` and `Q` and return the result if it is "<" or ">". If `P` and `Q` are actually the same protocol, return "=". Otherwise, the program is invalid because it declares two protocols with the same name. Any tie-breaker can be used, such as source location.

Suppose the `Barn` module declares a `Horse` protocol, and the `Swift` module declares `Collection`. Then `Horse` precedes `Collection`, because they are declared in different modules, and `Barn` < `Swift`. If the `Barn` module also declares a `Saddle` protocol, then `Horse` < `Saddle`, because both are from the same module, so we compare their protocol names.

In the previous section, we showed that re-stating an inherited associated type has no effect in our formal system. We also want this to have no effect on reduced types, so our type parameter order must prefer certain associated type declarations over others:

Definition 5.22. A *root associated type* is an associated type whose parent protocol does not inherit from any protocol having an associated type with the same name.

In the following, `Derived.Foo` is not a root, because `Derived` inherits `Base` and `Base` also declares an associated type named `Foo`. However, `Derived.Bar` is a root:

```
protocol Base {
  associatedtype Foo  // root
}

protocol Derived: Base {
  associatedtype Foo  // not a root
  associatedtype Bar  // root
}
```

We define the associated type order in terms of the protocol order.

Algorithm 5C (Associated type order)**.** Takes associated type declarations $\texttt{A}_1$ and $\texttt{A}_2$ as input, and returns one of "<", ">", or "=" as output.

1. First, compare their names lexicographically. Return the result if it is "<" or ">". Otherwise, both associated types have the same name, so keep going.
2. If $\texttt{A}_1$ is a root associated type and $\texttt{A}_2$ is not, return "<".
3. If $\texttt{A}_2$ is a root associated type and $\texttt{A}_1$ is not, return ">".
4. Otherwise, $\texttt{A}_1$ and $\texttt{A}_2$ have the same "root-ness." Compare the parent protocols of $\texttt{A}_1$ and $\texttt{A}_2$ using Algorithm 5B and return the result if it is "<" or ">".
5. Otherwise, we have two associated types with the same name in the same protocol. If $\texttt{A}_1$ and $\texttt{A}_2$ are the same associated type declaration, return "=".
6. Otherwise, the program is invalid. As with the protocol order, any tie-breaker can be used, such as source location.

We now have enough to order all type parameters. The type parameter order does not distinguish type sugar, so it outputs "=" if and only if `T` and `U` are canonically equal.

Algorithm 5D (Type parameter order)**.** Takes type parameters `T` and `U` as input, and returns one of "<", ">", or "=" as output.

1. If `T` is a generic parameter type and `U` is a dependent member type, then $|\texttt{T}| < |\texttt{U}|$, so return "<".
2. If `T` is a dependent member type and `U` is a generic parameter type, then $|\texttt{T}| > |\texttt{U}|$, so return ">".
3. If `T` and `U` are both generic parameter types, compare them using Algorithm 5A and return the result.
4. Otherwise, both are dependent member types. Recursively compare the base type of `T` with the base type of `U`, and return the result it is "<" or ">".
5. Otherwise, `T` and `U` have canonically equal base types.
6. If `T` is bound and `U` is unbound, return "<".
7. If `T` is unbound and `U` is bound, return ">".
8. If `T` and `U` are both unbound dependent member types, compare their names lexicographically and return the result.
9. If `T` and `U` are both bound dependent member types, compare their associated type declarations using Algorithm 5C and return the result.

Definition 5.23. Let G be a generic signature. A valid type parameter `T` of G is a *reduced type parameter* if $G \vdash [\texttt{T == U}]$ implies that $\texttt{T} \leq \texttt{U}$ for all `U`.

The type parameter order is a linear order, so if we can identify a reduced type parameter in some equivalence class, we know that it must be unique. However, we haven't seen how to calculate the reduced type parameter yet. Indeed, we haven't even established that an equivalence class of type parameters *has* a minimum element. For example, the set of integers $\mathbb{Z}$ certainly does not, under the usual linear order:

$$\cdots < -2 < -1 < 0 < 1 < 2 < \cdots$$

We wish to rule out this possibility:

Definition 5.24. A linear order with domain S is *well-founded* if every non-empty subset of S contains a minimum element.

In the more general case of a partial order, this is not the right definition, because a set consisting of two incomparable elements does not have a minimum element. For a partial order, the correct condition is that the domain S must not contain an *infinite descending chain*, which is an infinite sequence of elements $x_i \in S$ such that:

$$x_1 > x_2 > x_3 > \ldots$$

For a linear order the two conditions are equivalent.

Any partial order on a finite set is always well-founded. The usual linear order on the natural numbers $\mathbb{N}$ is also well-founded (essentially by *definition* of $\mathbb{N}$).

Proposition 5.25. The type parameter order of Algorithm 5D is well-founded.

Proof. Let S be any non-empty set of type parameters, possibly infinite. We must prove that S contains a minimum element.

First, define the set $L(S) \subseteq \mathbb{N}$ by taking the length of each type parameter in S. While $L(S)$ might be an infinite set, it must have a minimum element under the linear order on $\mathbb{N}$, say $n \in L(S)$. Let $S_n \subseteq S$ be the subset of S containing only type parameters of length n. If S_n has a minimum element, it will also be the minimum element of S.

On the other hand, a set of type parameters of fixed length must be finite. If our entire program and all imported modules declare a total of g distinct generic parameters and a associated types, then the number of type parameters of length n, generated by all generic signatures, cannot exceed $g(2a)^n$. (We count each associated type twice, to account for bound and unbound type parameters, but this exact number is irrelevant; the important fact is that it's finite.)

Thus, the finite set S_n contains a minimum element, and so does S, and since S was chosen arbitrarily, we conclude that the type parameter order is well-founded. □

The type parameter order plays an important role in the Swift ABI. For example, the mangled symbol name of a generic function incorporates the function's parameter types and return type, and the mangler takes reduced types to ensure that cosmetic changes to the declaration have no effect on binary compatibility.

To define reduced type parameters, we compared elements in a *single* equivalence class. We also use the type parameter order to establish an order relation *between* equivalence classes, by comparing their reduced types. We will extend this to a partial order on *requirements* in Algorithm 11D. This requirement order surfaces in the Swift ABI:

1. The calling convention for a generic function passes a witness table for each explicit conformance requirement in the function's generic signature, sorted in this order.

2. The in-memory layout of a witness table stores an associated witness table for each associated conformance requirement in the protocol's requirement signature, sorted in this order.

In Chapter 8, we introduce *archetypes*, a self-describing representation of a reduced type parameter packaged up with a generic signature, which behaves more like a concrete type inside the compiler. An archetype thus represents an entire equivalence class of type parameters, and for this reason we will also use the equivalence class notation $[\![\texttt{T}]\!]$ to denote the archetype corresponding to the type parameter `T`.

A more complete treatment of equivalence relations and partial orders can be found in a discrete mathematics textbook, such as [81]. Something to keep in mind is that some authors say that a partial order is reflexive instead of anti-reflexive, so $\leq$ is the basic operation. This necessitates the additional condition that if both $x \leq y$ and $y \geq x$ are true, then $x = y$; without this assumption, we have a *preorder*. Sometimes, a linear order is also called a *total order*, and a set with a well-founded order is said to be *well-ordered*. The type parameter order is actually a special case of a *shortlex order*. Under reasonable assumptions, a shortlex order is always well-founded. We will see another instance of a shortlex order in Section 12.2, and then generalize the concept when we introduce the theory of string rewriting in Section 16.5.

5.6. Generic Signature Queries

In the implementation, we answer questions about the derived requirements and valid type parameters of a generic signature by invoking *generic signature queries*. These are methods on the `GenericSignature` class, used throughout the compiler. We can formalize the behavior of these queries using the notation we developed earlier in this chapter. Each query defines a mathematical function that takes a generic signature and at least one other piece of data, and evaluates to some property of this signature.

Basic queries. We saw that the equivalence class structure of a generic signature is generated by conformance requirements, and same-type requirements between type parameters. The first four queries allow us to answer questions about this structure, and their behavior is completely defined by the derived requirements formalism.

- **Query:** `requiresProtocol`$(G, \texttt{T}, \texttt{P})$

  Input: type parameter `T`, protocol `P`.

  Output: true or false: $G \vdash [\texttt{T: P}]$?

  Decide if a type parameter conforms to a given protocol.

- **Query:** `areReducedTypeParametersEqual`$(G, \texttt{T}, \texttt{U})$

  Input: type parameter `T`, type parameter `U`.

  Output: true or false: $G \vdash [\texttt{T == U}]$?

  Decide if two type parameters are in the same equivalence class.

- **Query:** `isValidTypeParameter`(G, `T`)

  Input: type parameter `T`.

  Output: true or false: $G \vdash$ `T`?

  Decide if a type parameter is valid.

- **Query:** `getRequiredProtocols`(G, `T`)

  Input: type parameter `T`.

  Output: All protocols `P` such that $G \vdash$ [`T: P`].

  Produce a list of all protocols this type parameter is known to conform to.

From a theoretical point of view, the two fundamental queries are `requiresProtocol`() and `areReducedTypeParametersEqual`(). The other two, `isValidTypeParameter`() and `getRequiredProtocols`(), while primitive in the implementation, can be formalized in terms of the others. Here is `isValidTypeParameter`(G, `T`):

- A generic parameter type is valid if it appears in G.
- An unbound dependent member type `U.A` is valid if there exists a protocol `P` in `getRequiredProtocols`(G, `U`) such that `P` declares an associated type named `A`.
- A bound dependent member type `U.[P]A` is valid if `requiresProtocol`(G, `U`, `P`).

As for `getRequiredProtocols`(), because the "for all" quantifier is selecting from a finite universe of protocols, a correct but inefficient implementation would repeatedly check if `requiresProtocol`(G, `T`, `P`) for each `P` in turn. The real implementation instead builds a data structure that can perform this lookup more efficiently (Chapter 19).

The `getRequiredProtocols`() query is used when type checking a member reference expression like "`foo.bar`" where the type of "`foo`" is a type parameter, because we resolve "`bar`" by a qualified name lookup into this list of protocols. This list is minimal in the sense that no protocol inherits from any other, because qualified lookup recursively visits all inherited protocols anyway. The protocols are also sorted using Algorithm 5B.

Example 5.26. Let G be the generic signature from Example 5.1:

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
                    τ_0_0.Element: Equatable,
                    τ_0_0.Element == τ_0_1.Element>
```

1. `requiresProtocol`(G, `τ_0_0.Element`, `Equatable`) is true.
2. `requiresProtocol`(G, `τ_0_0.Iterator.Element`, `Equatable`) is true.
3. `requiresProtocol`(G, `τ_0_0.Iterator`, `Equatable`) is false.

4. areReducedTypeParametersEqual(G, τ_0_0.Element, τ_0_1.Element) is true.
5. areReducedTypeParametersEqual(G, τ_0_0.Iterator, τ_0_1.Iterator) is false.
6. isValidTypeParameter(G, τ_0_0.Element) is true.
7. isValidTypeParameter(G, τ_0_0.Iterator.Element) is true.
8. isValidTypeParameter(G, τ_0_0.Element.Iterator) is false.
9. getRequiredProtocols(G, τ_0_0.Iterator) is {IteratorProtocol}.

Concrete types. The next two queries concern concrete same-type requirements. We can ask if a type parameter is fixed to a concrete type, and then we can ask for this concrete type. (We use "⊢" below, but recall from Section 5.2 that we don't have a complete set of inference rules to describe the implemented behavior of concrete same-type requirements.)

- **Query:** isConcreteType(G, T)

  Input: type parameter T.

  Output: true or false: exists some concrete type X such that $G \vdash$ [T == X]?

  Decide if a type parameter is fixed to a concrete type.

- **Query:** getConcreteType(G, T)

  Input: type parameter T.

  Output: some concrete type X such that $G \vdash$ [T == X].

  Output the fixed concrete type of a type parameter.

Example 5.27. Let G be the generic signature,

```
<τ_0_0 where τ_0_0: Foo, τ_0_0.[Foo]B == Int>
```

with protocol Foo as below:

```
protocol Foo {
  associatedtype A where A == Array<B>
  associatedtype B
}
```

1. isConcreteType(G, τ_0_0.[Foo]A) is true.
2. getConcreteType(G, τ_0_0.[Foo]A) is Array<τ_0_0.[Foo]B>.
3. getConcreteType(G, τ_0_0.[Foo]B) is Int.

Reduced types. We can now generalize reduced type equality of type parameters to get an equivalence relation on interface types, that is, those types that *contain* type parameters. This equivalence relation, known as *reduced type equality of interface types*, can be characterized by its two defining properties.

The first property is that if a generic signature fixes a type parameter to a concrete type, an interface type containing this type parameter is then equivalent to one where this type parameter has been replaced by its concrete type. In the previous example, `τ_0_0.[Foo]A` was fixed to `Array<τ_0_0.[Foo]B>`, and `τ_0_0.[Foo]B` was fixed to `Int`, so the following three interface types are equivalent in that signature:

```
τ_0_0.[Foo]A
Array<τ_0_0.[Foo]B>
Array<Int>
```

The second property is that replacing a type parameter with an equivalent type parameter, anywhere that it appears in an interface type, will always produce an equivalent interface type. Consider this signature, with the same protocol `Foo` as before, but instead of fixing `τ_0_0.[Foo]B` to `Int`, we say that `τ_0_0.[Foo]B` is equivalent to `τ_0_1`:

```
<τ_0_0, τ_0_1 where τ_0_0: Foo, τ_0_1 == τ_0_0.[Foo]B>
```

In this generic signature, the following three interface types are equivalent:

```
τ_0_0.[Foo]A
Array<τ_0_0.[Foo]B>
Array<τ_0_1>
```

Suppose we take an interface type, and apply a pair of transformations until fixed point: first, we replace every type parameter that is fixed to concrete type, and second, we replace every other type parameter with the reduced type parameter of its equivalence class. A key fact is that the resulting interface type is equivalent to the original.

Algorithm 5E (Compute reduced type)**.** Takes a generic signature G, and an interface type `X`, as input. Outputs the reduced type of `X`.

1. If `X` is actually a type parameter `T`:
   a) If `isConcreteType`(G, `T`): recursively reduce `getConcreteType`(G, `T`).
   b) Otherwise, return the reduced type parameter in the equivalence class of `T`.
2. Otherwise, `X` is a concrete type. If `X` does not have any children, return `X`.
3. Otherwise, recursively reduce each child type of `X`, and construct a new type from these reduced child types together with any non-type attributes of `X`.

This algorithm outputs a special kind of interface type:

Definition 5.28. Let G be a generic signature. An interface type is a *reduced type* of G if the following two conditions hold for each type parameter it contains:

1. Every such type parameter is a reduced type parameter of G.
2. No such type parameter is fixed to a concrete type by G.

In particular, a fully-concrete type, one without type parameters, is reduced.

To guarantee termination in Algorithm 5E, we must disallow generic signatures with self-referential same-type requirements. For example, if $G \vdash$ [`τ_0_0 == Array<τ_0_0>`], Step 1a would get stuck reducing `τ_0_0`, `Array<τ_0_0>`, `Array<Array<τ_0_0>>`, and so on. We will describe how in Section 19.1. Given that this can't happen, it follows that every equivalence class of interface types contains a unique reduced type. Therefore:

Proposition 5.29. Let G be a generic signature. Two interface types are equivalent under the reduced type equality relation if and only if they have the same reduced type. (More precisely, if their reduced types are canonically equal.)

Thus, the following pair of generic signature queries decide reduced type equality:

- **Query:** `isReducedType`(G, `T`)

  Input: interface type `T`.

  Output: true or false: is `T` canonically equal to its reduced type?

  Decide if an interface type is already a reduced type.

- **Query:** `getReducedType`(G, `T`)

  Input: interface type `T`.

  Output: the reduced type of `T`.

  Compute the reduced type using Algorithm 5E. This always outputs a canonical type, so any type sugar in the original type is lost.

Example 5.30. Reduced type equality on interface types is a *coarser* relation than reduced type equality on type parameters; two equivalent type parameters are also equivalent as interface types, but the converse does *not* hold. Let G be this generic signature, and note that [`τ_0_0 == τ_0_1`] *cannot* be derived from G:

```
<τ_0_0, τ_0_1 where τ_0_0 == Int, τ_0_1 == Int>
```

1. `areReducedTypeParametersEqual`(G, `τ_0_0`, `τ_0_1`) is false.
2. `getReducedType`(G, `τ_0_0`) is `Int`.
3. `getReducedType`(G, `τ_0_1`) is `Int`.

Thus, `τ_0_0` and `τ_0_1` are equivalent as interface types, but not as type parameters.

Other requirements. The final set of queries concern superclass and layout requirements. Once again, we note that "$\vdash$" is aspirational, because not all implemented behaviors of these requirement kinds are described by our formal system.

- **Query:** `getSuperclassBound`$(G, \texttt{T})$

  Input: type parameter `T`.

  Output: some concrete class type `C` such that $G \vdash$ [`T: C`].

  Output the superclass bound, if the type parameter has one.

- **Query:** `requiresClass`$(G, \texttt{T})$

  Input: type parameter `T`.

  Output: true or false: $G \vdash$ [`T: AnyObject`]?

  Decide if `T` has a single reference-counted pointer representation.

- **Query:** `getLayoutConstraint`$(G, \texttt{T})$

  Input: type parameter `T`.

  Output: some layout constraint `L` such that `T` $\vdash$ [`T: L`].

  Output the layout constraint, if the type parameter has one.

  The `AnyObject` layout constraint is the only one that can be explicitly written in source. A second kind of layout constraint, `_NativeClass`, is implied by a superclass bound being a native Swift class, meaning a class not inheriting from `NSObject`. The `_NativeClass` layout constraint in turn implies the `AnyObject` layout constraint.

  The two differ in how reference counting operations are lowered in IRGen. Classes of unknown ancestry use the Objective-C runtime entry points, whereas native class instances use different entry points in the Swift runtime.

Example 5.31. Let G be the generic signature,

```
<τ_0_0, τ_0_1, τ_0_2 where τ_0_0: Form, τ_0_1: Shape, τ_0_2: Entity>
```

together with the three declarations:

```
class Shape {}
protocol Form: AnyObject {}
protocol Entity: Shape, Form {}
```

1. `getSuperclassBound`(G, `τ_0_0`) is null.
2. `getSuperclassBound`(G, `τ_0_1`) is `Shape`.

3. `getSuperclassBound`(G, `τ_0_2`) is `Shape`.
4. `requiresClass`(G, `τ_0_0`) is true.
5. `requiresClass`(G, `τ_0_1`) is true.
6. `requiresClass`(G, `τ_0_2`) is true.

We can write down a derivation of `requiresClass`(G, `τ_0_0`):

1. [`τ_0_0: Form`] (Conf)
2. [`τ_0_0: AnyObject`] (AssocLayout 1)

However, we cannot write down a derivation of `requiresClass`(G, `τ_0_1`), even though we *should* be able to. The implementation says this derived requirement holds, because any concrete replacement type for `τ_0_1` that satisfies the superclass requirement must also satisfy the layout requirement. We are unable to make this inference within our formal system, so it is *incomplete*. In this case, the missing rule is one that allows us to derive [`T: AnyObject`] from a superclass requirement [`T: C`]. A formal description of the missing rules remains a work in progress.

Sugared types. To avoid printing canonical generic parameter types like `τ_d_i` in diagnostics, a special query can be used to transform a canonical type into a sugared type, using the generic parameter names of a given generic signature. This is not really a semantic query at all, it's just a syntactic transformation of the type.

- **Query:** `getSugaredType`(G, `T`)

  Input: interface type `T`.

  Output: a canonically equal interface type, written using the sugared types of G.

Combined queries. To simplify archetype construction inside a generic environment (Section 8.1), we use a special entry point to perform multiple lookups at once. Its behavior is otherwise completely determined by the other queries.

- **Query:** `getLocalRequirements`(G, `T`)

  Input: type parameter `T`.

  Output: a single structure with the result of several queries.

  Output all known data points about the equivalence class of `T`:

  `getRequiredProtocols`(G, `T`)
  `getSuperclassBound`(G, `T`)
  `getLayoutConstraint`(G, `T`)

5.7. Source Code Reference

Key source files:

- include/swift/AST/GenericSignature.h
- include/swift/AST/Requirement.h
- include/swift/AST/RequirementSignature.h
- lib/AST/GenericSignature.cpp

Other source files:

- include/swift/AST/Decl.h
- include/swift/AST/DeclContext.h
- lib/AST/Decl.cpp
- lib/AST/DeclContext.cpp

`DeclContext` *class*

See also Section 4.6.

- `getGenericSignatureOfContext()` returns the generic signature of the innermost generic context, or the empty generic signature if there isn't one.

`GenericContext` *class*

See also Section 4.6.

- `getGenericSignature()` returns the declaration's generic signature, computing it first if necessary. If the declaration does not have a generic parameter list or trailing `where` clause, returns the generic signature of the parent context.

`GenericSignature` *class*

Represents an immutable, uniqued generic signature. Meant to be passed as a value, it stores a single instance variable, a `GenericSignatureImpl *` pointer.

The `getPointer()` method returns this pointer. The pointer is not `const`, however `GenericSignatureImpl` does not define any mutating methods.

The pointer may be `nullptr`, representing an empty generic signature; the default constructor `GenericSignature()` constructs this value. There is an implicit `bool` conversion which tests for the empty generic signature.

The `getPointer()` method is only used occasionally, because the `GenericSignature` class overloads `operator->` to forward method calls to the `GenericSignatureImpl *` pointer. Some operations on generic signatures are methods on `GenericSignature` (called with "`.`") and some on `GenericSignatureImpl` (called with "`->`").

Methods of `GenericSignature` are safe to call with an empty generic signature, which is presented as having no generic parameters or requirements. Methods forwarded to `GenericSignatureImpl` can only be invoked if the signature is non-empty.

The `GenericSignature` class explicitly deletes `operator==` and `operator!=` to make the choice between pointer and canonical equality explicit. To check pointer equality of generic signatures, first unwrap both sides with a `getPointer()` call:

```
if (lhsSig.getPointer() == rhsSig.getPointer())
  ...;
```

The more common canonical signature equality check is implemented by the `isEqual()` method on `GenericSignatureImpl`:

```
if (lhsSig->isEqual(rhsSig))
  ...;
```

Various accessor methods:

- `getGenericParams()` returns an array of `GenericTypeParamType`. If the generic signature is empty, this is the empty array, otherwise it contains at least one generic parameter.
- `getInnermostGenericParams()` returns an array of `GenericTypeParamType` with the innermost generic parameters only, that is, those with the highest depth. If the generic signature is empty, this is the empty array, otherwise it contains at least one generic parameter.
- `getRequirements()` returns an array of `Requirement`. If the generic signature is empty, this is the empty array.
- `getCanonicalSignature()` returns the canonical signature. If the generic signature is empty, returns the canonical empty generic signature.
- `getPointer()` returns the underlying `GenericSignatureImpl *`.

Computing reduced types:

- `getReducedType()` returns the reduced type of an interface type for this generic signature. If the generic signature is empty, the type must be fully concrete, and is returned unchanged.

Other:

- `print()` prints the generic signature, with various options to control the output.
- `dump()` prints the generic signature, meant for use from the debugger or ad-hoc print debug statements.

Also see Section 11.5.

`GenericSignatureImpl` *class*

The backing storage of a generic signature. Instances of this class are allocated in the AST context, and are always passed by pointer.

- `isEqual()` checks if two generic signatures are canonically equal.
- `getSugaredType()` given a type containing canonical type parameters that is understood to be written with respect to this generic signature, replaces the generic parameter types with their "sugared" forms, so that the name is preserved when the type is printed out to a string.
- `forEachParam()` invokes a callback on each generic parameter of the signature; the callback also receives a boolean indicating if the generic parameter type is reduced or not—a generic parameter on the left hand side of a same-type requirement is not reduced.
- `areAllParamsConcrete()` checks if all generic parameters are fixed to concrete types via same-type requirements, which makes the generic signature somewhat like an empty generic signature. Fully-concrete generic signatures are lowered away at the SIL level.

The generic signature queries from Section 5.6 are methods on `GenericSignatureImpl`:

- Predicate queries:
  - `isValidTypeParameter()`
  - `requiresProtocol()`
  - `requiresClass()`
  - `isConcreteType()`
- Property queries:
  - `getRequiredProtocols()`
  - `getSuperclassBound()`
  - `getConcreteType()`

  - `getLayoutConstraint()`

- Reduced type queries:
  - `areReducedTypeParametersEqual()`
  - `isReducedType()`
  - `getReducedType()`

`CanGenericSignature` *class*

The `CanGenericSignature` class wraps a `GenericSignatureImpl *` pointer which is known to be canonical. The pointer can be recovered with the `getPointer()` method. There is an implicit conversion from `CanGenericSignature` to `GenericSignature`. The `operator->` forwards method calls to the underlying `GenericSignatureImpl`.

The `operator==` and `operator!=` operators are used to test `CanGenericSignature` for pointer equality. The `isEqual()` method of `GenericSignatureImpl` implements canonical equality on arbitrary generic signatures by first canonicalizing both sides, then checking the resulting canonical signatures for pointer equality. Therefore, the following are equivalent:

```
if (lhsSig->isEqual(rhsSig))
  ...;

if (lhsSig.getCanonicalSignature() == rhsSig.getCanonicalSignature())
  ...;
```

The `CanGenericSignature` class inherits from `GenericSignature`, and so inherits all of the same methods. Additionally, it overrides `getGenericParams()` to return an array of `CanGenericTypeParamType`.

`Requirement` *class*

A generic requirement. Consists of a kind, a left-hand side which is always a `Type`, and a right-hand side, which depends on the kind. See also Section 9.5 and Section 11.5.

- `Requirement(RequirementKind, Type, Type)` constructs any requirement kind except for `RequirementKind::Layout`.
- `Requirement(RequirementKind, Type, LayoutConstraint)` constructs a `RequirementKind::Layout`.
- `getKind()` returns the `RequirementKind`.
- `getFirstType()` returns the left-hand side type.

- `getSecondType()` returns the right-hand side type if this requirement has any kind except for `RequirementKind::Layout`; otherwise asserts.
- `getProtocolDecl()` returns the right-hand side protocol declaration if this is a `RequirementKind::Conformance`; otherwise asserts.
- `getLayoutConstraint()` returns the right-hand side layout constraint if this is a `RequirementKind::Layout`; otherwise asserts.

`RequirementKind` *enum class*

Return type of `Requirement::getKind()`.

- `RequirementKind::Conformance`
- `RequirementKind::Superclass`
- `RequirementKind::Layout`
- `RequirementKind::SameType`

`ProtocolDecl` *class*

See also Section 4.6 and Section 11.5.

- `getRequirementSignature()` returns the protocol's requirement signature, first computing it, if necessary.
- `requiresClass()` checks if the protocol is a class-constrained protocol.

`RequirementSignature` *class*

A protocol requirement signature.

- `getRequirements()` returns an array of `Requirement`.
- `getTypeAliases()` returns an array of `ProtocolTypeAlias`.

Also see Section 11.5.

`ProtocolTypeAlias` *class*

A protocol type alias descriptor.

- `getName()` returns the name of the alias.
- `getUnderlyingType()` returns the underlying type of the type alias. This is a type written in terms of the type parameters of the requirement signature.

`TypeBase` *class*

See also Section 3.4.

- `isTypeParameter()` checks if this type is a type parameter; that is, a generic parameter type, or a `DependentMemberType` whose base is another type parameter.
- `hasTypeParameter()` checks if this type is itself a type parameter, or if it contains a type parameter in structural position. For example, `Array<τ_0_0>` will answer `false` to `isTypeParameter()`, but `true` to `hasTypeParameter()`.

`DependentMemberType` *class*

A type abstracting over a type witness in a conformance.

- `getBase()` returns the base type; for example, given `τ_0_0.Foo.Bar`, will answer `τ_0_0.Foo`.
- `getName()` returns the identifier naming the associated type.
- `getAssocType()` if this is a bound `DependentMemberType`, returns the associated type declaration, otherwise if it is unbound, returns `nullptr`.

`TypeDecl` *class*

See also Section 4.6.

- `compare()` compares two protocols by the protocol order (Definition 5B), returning one of the following:
  - -1 if this protocol precedes the given protocol,
  - 0 if both protocol declarations are equal,
  - 1 if this protocol follows the given protocol.

`compareDependentTypes()` *function*

Implements the type parameter order (Algorithm 5D), returning one of the following:

- -1 if the left hand side precedes the right hand side,
- 0 if the two type parameters are equal as canonical types,
- 1 if the left hand side follows the right hand side.

6. Substitution Maps

Substitution maps are the semantic objects that describe *references* to generic declarations. If we think of a generic signature as a *contract* between a declaration and its client, then the previous chapter's derived requirements formalism gave us one side of this contract: within the *body* of a generic declaration, we start from the assumption that the concrete replacement types, whatever they might be, must satisfy the generic signature's explicit requirements, and from this, we derive various consequences which then hold. In this chapter, we turn our attention to the client's side of this contract, which leads to the study of *type substitution*.

Input generic signature. The "shape" of a substitution map is determined by its *input generic signature*. In this chapter, we will primarily focus on the simplest case, where this generic signature has no requirements, but we will summarize the general case shortly. Subsequent chapters will explain how requirements interact with substitution.

Example 6.1. Let's look at this generic function, and think about its callers:

```
func combine<T, U>(_ t: T, _ u: U) -> (T, Array<U>) {
  return (t, [u])
}
```

Our generic signature has two generic parameters and no requirements. Let's call it G:

$G :=$ `<T, U>`, or `<τ_0_0, τ_0_1>`.

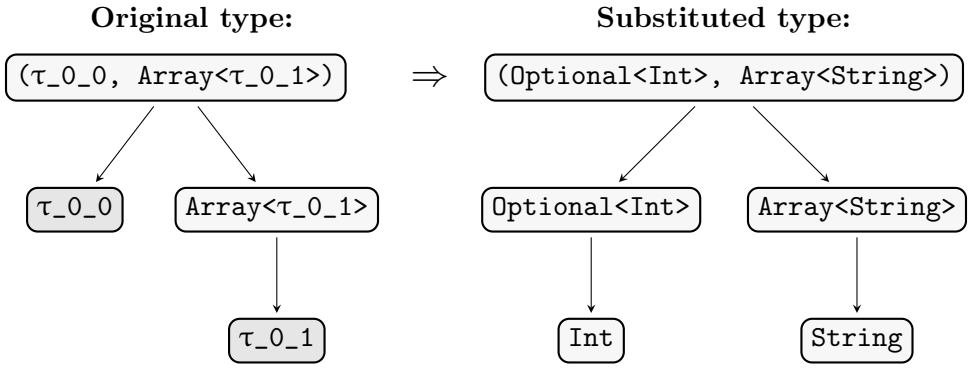
Suppose we call our function with these two values; what is the type of the result?

```
let t: Optional<Int> = 3
let u: String = "Hello world"

let result = combine(t, u)  // what is the return type of this call?
```

By matching the function's declared parameter types against the types of the argument expressions, the expression type checker can deduce that `T` must be `Optional<Int>`, and also that `U` must be `String`. This gives us our substitution map; let's call it Σ:

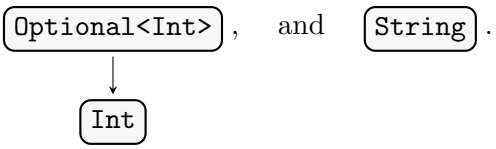
$\Sigma :=$ {`τ_0_0` $\mapsto$ `Optional<Int>`,
`τ_0_1` $\mapsto$ `String`}

Figure 6.1.: Type substitution



Our notation for specifying a substitution map lists the replacement type for each generic parameter of its input generic signature, in order. Type substitution disregards type sugar, and to remind ourselves of this fact, we write the canonical generic parameter type on the left-hand side of the "$\mapsto$" symbol. We often name our substitution maps Σ or some variation, like Σ_1, Σ_2, Σ', and so on.

Recall from Chapter 3 that types have a tree structure, so the two replacement types in our substitution map Σ have an in-memory form that looks something like this:



Now, to figure out the inferred type of "`result`," we note that our function's declared return type is the tuple type `(τ_0_0, Array<τ_0_1>)`. Let's call this the *original type*. This type contains the generic parameters of G. To get the *substituted type*, we replace each occurrence of `τ_0_0` and `τ_0_1` in the original type with its replacement type. We conclude that the substituted type is `(Optional<Int>, Array<String>)`. Figure 6.1 illustrates the transformation.

This operation is called *type substitution*. We will introduce some notation to talk about it first, before we look at the formal algorithm. With Σ as above, we denote this type substitution operation as follows, with the original type on the left of the $\otimes$ operator, and the substitution map on the right:

$$\texttt{(τ_0_0, Array<τ_0_1>)} \otimes \Sigma = \texttt{(Optional<Int>, Array<String>)}$$

The above example showed one way to "observe" type substitution: we write down a generic function declaration, call it with some list of generic argument types, and then look at the return type of the call. Here is another similar trick.

Example 6.2. Consider type resolution with a generic type alias:

```
typealias Foo<T, U> = (T, Array<U>)

let x: Foo<Optional<Int>, String> = ...
```

Here, we're referencing `Foo` with the same generic arguments as before, so we get the same substitution map Σ. To resolve the type annotation on "`x`", we apply Σ to the underlying type of `Foo`:

$$\texttt{(τ_0_0, Array<τ_0_1>)} \otimes \Sigma = \texttt{(Optional<Int>, Array<String>)}$$

Since the underlying type of the alias and the generic arguments of a reference are both arbitrary, we can use generic type aliases to encode any type substitution operation. We will discuss type resolution of generic type aliases in Section 9.1 and Section 9.2.

Output generic signature. In our examples so far, the replacement types in the substitution map were fully concrete. To observe a substitution map whose replacement types contain type parameters, we can reference a generic declaration from the body of some other generic declaration. We say that a generic signature H is the *output generic signature* of a substitution map Σ if all type parameters that appear in the replacement types of Σ are valid type parameters of H.

Example 6.3. In the following, the generic `callee()` function is referenced from the body of `caller()`:

```
func callee<T>(_ value: T) -> Array<T> {
  return [value]
}

func caller<X, Y>(_ x: X, y: Y) -> Array<(X, Y)> {
  return callee((x, y))  // call here
}
```

The call expression passes in a tuple value as the argument, so the replacement type for `τ_0_0` is the tuple type `(τ_0_0, τ_0_1)`:

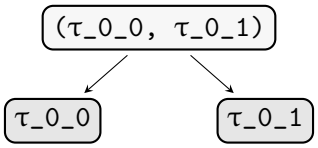
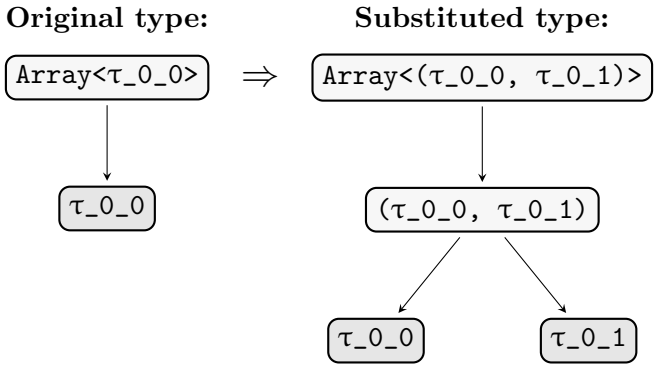


Figure 6.2.: Type substitution with output generic signature



Our input generic signature is the generic signature of `callee()`:

$G :=$ `<τ_0_0>`

Our output generic signature is the generic signature of `caller()`:

$H :=$ `<τ_0_0, τ_0_1>`

Here is the substitution map:

$\Sigma :=$ {`τ_0_0` $\mapsto$ `(τ_0_0, τ_0_1)`}

Now, we apply Σ to the return type of `callee()` to get the substituted type:

`Array<τ_0_0>` $\otimes \Sigma =$ `Array<(τ_0_0, τ_0_1)>`

Figure 6.2 illustrates the transformation.

Next, we introduce a bit of notation to formalize a concept we've already seen:

Definition 6.4. Let G be a generic signature. We write $\textsc{Type}(G)$ to denote the set of interface types of G. In particular, this set contains the following:

- All valid type parameters of G.
- All nominal types, formed from the non-generic nominal type declarations.
- All generic nominal types, formed from the generic nominal type declarations and all combinations of generic arguments from $\textsc{Type}(G)$.
- All structural types—such as function types, tuple types, and metatypes—formed from the elements of $\textsc{Type}(G)$.

We can also talk about the set of all substitution maps that map the interface types of one fixed generic signature into another:

Definition 6.5. Let G and H be generic signatures. We write $\textsc{Sub}(G, H)$ to denote the set of all substitution maps that have input generic signature G and output generic signature H.

These are infinite sets, so we do not realize them in the implementation. Instead, we use them as a notational aid to understand type substitution. An important and subtle point is that the output generic signature is not stored in the substitution map itself; it is implicit from usage. Thus, we must take care not to get our output generic signatures "mixed up" by keeping the relationships between these sets in mind.

Suppose we are given a $\Sigma \in \textsc{Sub}(G, H)$. Note that the replacement types of Σ are elements of $\textsc{Type}(H)$, and furthermore, if we take some $\texttt{X} \in \textsc{Type}(G)$ and replace each type parameter that occurs in $\texttt{X}$ with an element of $\textsc{Type}(H)$, the transformed type will then also be an element of $\textsc{Type}(H)$. In particular, $\texttt{X} \otimes \Sigma \in \textsc{Type}(H)$, and we can think of type substitution as the following mapping between sets:

$$\textsc{Type}(G) \otimes \textsc{Sub}(G, H) \longrightarrow \textsc{Type}(H)$$

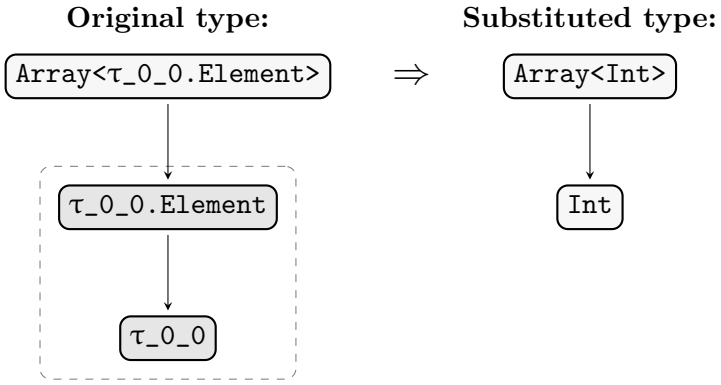
We model type substitution a binary operation on types and substitution maps, instead of using an "applicative" notation like "$\Sigma(\texttt{T})$" where the substitution map is represented as a unary function. The reason for this will soon become clear, when we encounter additional forms of the $\otimes$ operator that allow us to write down more complex expressions. The different forms of $\otimes$ will be related by various identities, which will define the *type substitution algebra*.

The below algorithm for type substitution formalizes the structural transformation we saw earlier. It also calls out to a handful of subroutines to handle the dependent member type and archetype cases, which we will fill out later.

Algorithm 6A (Substitute type)**.** Takes a type $\texttt{X}$, and a substitution map Σ with input generic signature G. Outputs $\texttt{X} \otimes \Sigma$.

1. (Type parameter) If $\texttt{X}$ is a type parameter $\texttt{T}$:
   a) If `isValidTypeParameter`(G, $\texttt{T}$) is false, return the error type.
   b) If $\texttt{T}$ is a generic parameter type, return the replacement type for $\texttt{T}$ in Σ.
   c) If $\texttt{T}$ is a dependent member type, apply Algorithm 7B, and return the result.
2. (Archetype) If $\texttt{X}$ is a primary archetype $[\![\texttt{T}]\!]_G$, return $\texttt{T} \otimes \Sigma$ (Section 8.2). If $\texttt{X}$ is an opaque archetype, call Algorithm 13A. If $\texttt{X}$ is existential, call Algorithm 14A.
3. (Base case) If $\texttt{X}$ does not contain type parameters or archetypes: return $\texttt{X}$.
4. (Recurse) Apply Σ to each child of $\texttt{X}$ recursively, and form a new type from these substituted child types, preserving any non-type structural components of $\texttt{X}$.

Figure 6.3.: Type substitution with dependent member type



Dependent member types. If a generic signature states conformance requirements to protocols with associated types, the generic signature's valid type parameters also include all dependent member types derived from these conformance requirements, in addition to the generic parameters stated in the generic signature.

```
func extract<S: Sequence>(_ s: S) -> Array<S.Element> {...}
let mySet: Set<Int> = [1, 2, 3]
let x = extract(mySet)
```

A substitution map records a *root conformance* for each conformance requirement of its input generic signature. In the call to `extract()`, the substitution map is:

$$\Sigma := \{\texttt{τ_0_0} \mapsto \texttt{Set<Int>},$$
$$\quad [\texttt{τ_0_0: Sequence}] \mapsto [\texttt{Set<Int>: Sequence}]\}$$

Type substitution treats a dependent member type as an indivisible atomic element, and replaces it in one shot with a substituted type. (We do not recurse into the base type.) In Section 7.4, we will see that the substituted type is computed from one of the conformances stored in the substitution map. We leave this unexplained for now:

$$\texttt{Array<τ_0_0.Element>} \otimes \Sigma = \texttt{Array<Int>}$$

Figure 6.3 illustrates the transformation.

Archetypes. In Chapter 8, we will meet *archetypes*, an alternate representation which combines a type parameter with a generic signature. There are three kinds of archetype. A *primary* archetype acts like the type parameter it represents, as far as type substitution is concerned. Substitution of *opaque* and *existential* archetypes is different, and will be described in Section 13.1 and Section 14.1.

Substitution failure. Type substitution outputs an error type if the original type contains a type parameter that is not valid in the substitution map's input generic signature. Type substitution might also produce an error type if the substitution map's replacement types contain error types, or if one of the conformances stored in the substitution map is invalid. This is called *substitution failure*, and it indicates an error was diagnosed elsewhere in the user's program. The compiler does not proceed to SILGen if errors were diagnosed, and so error types should not appear after type checking.

Other requirements. Conformance requirements are special, because a substitution map directly records how it fulfills each conformance requirement. Other requirements are instead conditions to check. For example, if the input generic signature states a same-type requirement, then a substitution map for this generic signature must *satisfy* this requirement, in the sense that applying the substitution map to both sides should produce canonically equal substituted types. A substitution map is *well-formed* if it satisfies all derived requirements of its input generic signature. We will discuss the algorithm for checking if requirements are satisfied in Section 9.3.

6.1. Nominal Types

We're now going to look at the relationship between nominal type declarations, nominal types, and substitution maps. Consider these declarations:

```
struct Bacon<T, U> {
  struct Lettuce<V> {
    struct Tomato {
      var t: T
      var u: U
      var v: V
    }
  }
}
```

(We will discuss the stored properties of `Tomato` shortly.) Given these declarations, we can then write down various nominal types, for example the type of "`x`" below:

```
let x: Bacon<Int, Bool>.Lettuce<Float>.Tomato = ...
```

In the implementation, `Bacon<Int, Bool>` is classified as a generic nominal type, while both `Int` and `Bacon<Int, Bool>.Lettuce<Float>.Tomato` are "just" nominal types. However, the latter's *parent* is a generic nominal type. We say that a *specialized type*

is a nominal type that is either itself generic, or has a generic parent. Equivalently, a nominal type is specialized if its declaration has a non-empty generic signature.

If we take the generic arguments from each level of nesting in the specialized type `Bacon<Int, Bool>.Lettuce<Float>.Tomato`, we can form a substitution map for the generic signature of the declaration of `Tomato`, which is `<τ_0_0, τ_0_1, τ_1_0>`:

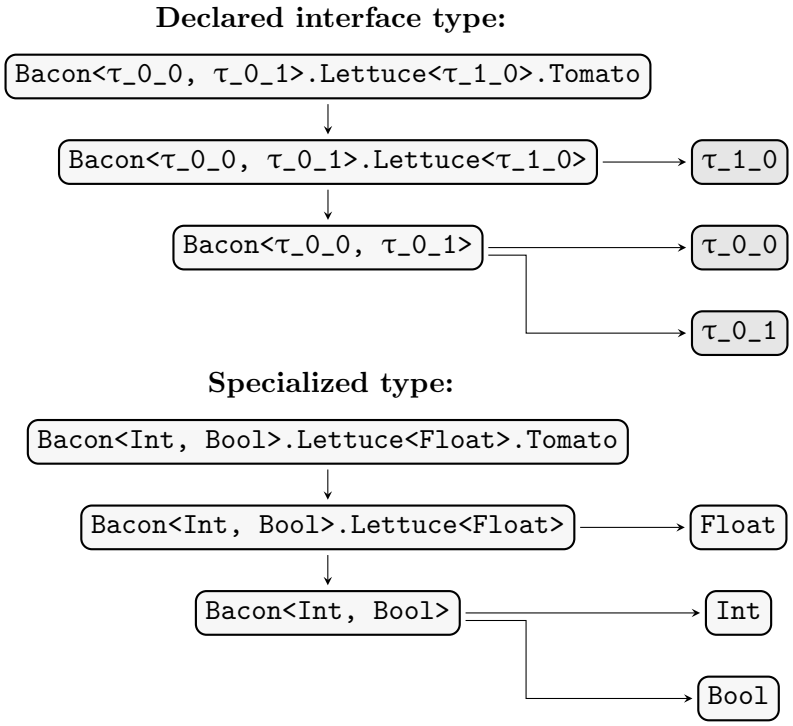
$$\Sigma := \{\texttt{τ_0_0} \mapsto \texttt{Int}, \texttt{τ_0_1} \mapsto \texttt{Bool}, \texttt{τ_1_0} \mapsto \texttt{Float}\}$$

Recall from Chapter 4 that every nominal type declaration has a declared interface type. In the case of `Tomato`, this is `Bacon<τ_0_0, τ_0_1>.Lettuce<τ_1_0>.Tomato`. Now, we see that when we apply our substitution map Σ to the declared interface type of `Tomato`, we get back our specialized type:

$$\begin{aligned}&\texttt{Bacon<τ_0_0, τ_0_1>.Lettuce<τ_1_0>.Tomato} \otimes \Sigma \\ &\quad = \texttt{Bacon<Int, Bool>.Lettuce<Float>.Tomato}\end{aligned}$$

We say that Σ is the *context substitution map* of our specialized type. Figure 6.4 illustrates the transformation.

Figure 6.4.: Applying the context substitution map



Context substitution map. Suppose we have two generic signatures G and H, and a nominal type declaration d with generic signature G. Let $\texttt{X}_d$ denote the declared interface type of d. If we are given some other specialized type formed from d, call it $\texttt{X} \in \textsc{Type}(H)$, then the context substitution map of $\texttt{X}$ is the unique substitution map $\Sigma \in \textsc{Sub}(G, H)$ that satisfies the identity:

$$\texttt{X}_d \otimes \Sigma = \texttt{X}$$

The context substitution map is more than just a mathematical trick, because it explains the type of a member reference expression like "`foo.bar`". Recall that `Tomato` declared three stored properties named `t`, `u`, and `v`, with interface types `τ_0_0`, `τ_0_1`, and `τ_1_0`, respectively. Consider this snippet:

```
let x: Bacon<Int, Bool>.Lettuce<Float>.Tomato = ...
let xt = x.t
let xu = x.u
let xv = x.v
```

To deduce the inferred types of `xt`, `xu`, and `xv`, we can apply our substitution map Σ to the interface type of each stored property:

$$\texttt{τ_0_0} \otimes \Sigma = \texttt{Int}$$
$$\texttt{τ_0_1} \otimes \Sigma = \texttt{Bool}$$
$$\texttt{τ_1_0} \otimes \Sigma = \texttt{Float}$$

Identity substitution map. What is the context substitution map of $\texttt{X}_d$, the declared interface type of d? If Σ satisfies $\texttt{X}_d = \texttt{X}_d \otimes \Sigma$, then in particular, Σ replaces every generic parameter of G with itself. We call this the *identity substitution map* for G and we denote it by 1_G. Note that $1_G \in \textsc{Sub}(G, G)$. For example, if G is the generic signature of `Tomato` from the above, then:

$$1_G := \{\texttt{τ_0_0} \mapsto \texttt{τ_0_0}, \texttt{τ_0_1} \mapsto \texttt{τ_0_1}, \texttt{τ_1_0} \mapsto \texttt{τ_1_0}\}$$

If G states one or more conformance requirements, the identity substitution map for G will also record a series of *abstract conformances* corresponding to each conformance requirement in G. This ensures that applying 1_G to a dependent member type leaves it unchanged. We will discuss abstract conformances in Section 7.4.

For example, if $G :=$ `<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence>`:

$$\begin{aligned} 1_G := \{&\texttt{τ_0_0} \mapsto \texttt{τ_0_0};\\ &[\texttt{τ_0_0: Sequence}] \mapsto [\texttt{τ_0_0: Sequence}],\\ &[\texttt{τ_0_1: Sequence}] \mapsto [\texttt{τ_0_1: Sequence}]\} \end{aligned}$$

In general, if `X` is any interface type in $\textsc{Type}(G)$, then $\texttt{X} \otimes 1_G = \texttt{X}$. In other words, applying the identity substitution map to any interface type of G leaves it unchanged. (This does not hold when the original type is a contextual type that contains archetypes; such a type is not an element of $\textsc{Type}(G)$. We will discuss archetype substitution in Section 8.2, and encounter another substitution map, the *forwarding substitution map*, which plays the role of the identity on contextual types.)

Empty substitution map. A nominal type declaration like `Int`, with an empty generic signature, only declares a single nominal type with no generic arguments. The context substitution map of this type is called the *empty substitution map*, because it does not store any replacement types or conformances. We denote it by $\{\}$.

This is the *only* possible substitution map for the empty generic signature. If G is the empty generic signature, the set of interface types $\textsc{Type}(G)$ is the set of fully-concrete types that do not contain any type parameters. The empty substitution map leaves such a type unchanged, so for example, $\texttt{Int} \otimes \{\} = \texttt{Int}$.

The empty substitution map should not be confused with the identity substitution map of a non-empty generic signature. If the original type contains any type parameters whatsoever, applying the empty substitution map will replace them with error types:

$$\texttt{τ_0_0.Element} \otimes \{\} = \texttt{<<error type>>}$$

6.2. Nested Nominal Types

We've seen that generic signatures and substitution maps use a flat representation that collects all outer generic parameters together, while nominal types have a recursive structure that reflects the lexical nesting of their nominal type declarations. When we build the context substitution map for a nominal type, we must be able to translate between these two representations. In particular, we must be able to recover a generic argument for each generic parameter of the signature. This imposes some restrictions on how nominal type declarations can nest:

1. Structs, enums and classes cannot nest in generic local contexts.
2. Structs, enums and classes cannot nest in protocols or protocol extensions.
3. Protocols cannot nest in other generic contexts.

Types in generic local contexts. This restriction stems from the fact that a nominal type declaration can only encode generic arguments for outer nominal type contexts. If the parent context of a nominal type declaration is a generic context that is not a type, the nominal type declaration only declares a single nominal type without any generic

arguments. This prevents us from being able to properly model the following code, which is rejected today:

```
func f<T>(t: T) {
  struct Nested {  // error
    let t: T

    func printT() {
      print(t)
    }
  }

  Nested(t: t).printT()
}
```

The `Nested` local type declaration has a stored property of type `τ_0_0`, the generic parameter of the outer declaration `f()`. However, `Nested` only declares a single nominal type, also written as `Nested`. This type does not have a parent type or any generic arguments, because it is not nested inside of another nominal type. This means we have to way to fill in the replacement type for `τ_0_0` in the context substitution map:

$$\texttt{Nested} \otimes \{\texttt{τ_0_0} \mapsto ???\} = \texttt{Nested}$$

If we call our function `f()` with two generic argument types, the SIL optimizer may decide to specialize one or both calls to `f()`:

```
func g() {
  f(t: 123)
  f(t: "hello")
}
```

We will form a substitution map replacing `τ_0_0` with `Int` or `String`, respectively, and apply this substitution map to every SIL instruction appearing in the body of `f()`. To ensure the stored property access has the correct substituted type, we must be able to represent the two distinct specializations of `Nested`. Thus, we actually want the declared interface type of `Nested` must to store a generic argument for the outer generic parameter, so notionally, we want to be able to write something like:

$$\texttt{<τ_0_0>.Nested} \otimes \{\texttt{τ_0_0} \mapsto \texttt{Int}\} = \texttt{<Int>.Nested}$$

The representation of runtime type metadata is already designed to store a flat list of generic arguments, just like a generic signature or substitution map. So while lifting this restriction would require some engineering effort on the compiler side, it would be a backward-deployable and ABI-compatible change.

Types in protocol contexts. We do not allow struct, enum, and class declarations to appear inside protocols and protocol extensions today. There are two ways to lift this restriction, and they differ in whether the nominal type declaration should capture the protocol `Self` type. Consider this protocol, and the protocol extension that follows. Note that `Box` has a stored property of type `τ_0_0.Contents`, because the declaration of the `Contents` associated type is visible from `Holder`'s lexical scope:

```
protocol Holder {
  associatedtype Contents
  var contents: Contents { get }
}

extension Holder {
  struct Box {  // error today
    let contents: Contents  // depends on the outer Self
  }

  var box: Box {
    return Box(contents: contents)
  }
}
```

Today, we reject the nested struct `Box`, but one possible interpretation would allow this code. In this model, the declared interface type of `Box` is something like "`τ_0_0.Box`," but that's not a dependent member type; it's a nominal type whose parent type is a *type parameter*. Every nominal type that conforms to `Holder` would gain a new member type declaration named `Box`, and the generic argument for `Self` would be this parent type. That is, if we conform a type `Foo` to `Holder` and call `box()`, we would receive a value of type `Foo.Box`:

```
struct Foo: Holder {
  typealias Contents = Int
}

let x: Foo.Box = Foo.box(123)  // imaginary
```

The context substitution map of `Foo.Box` would then be the protocol substitution map of the conformance:

$$\texttt{τ_0_0.Box} \otimes \{\texttt{τ_0_0} \mapsto \texttt{Foo};\ [\texttt{τ_0_0: Holder}] \mapsto [\texttt{Foo: Holder}]\} = \texttt{Foo.Box}$$

In this model, it would not make sense to reference `Box` as a member of the protocol itself; that is, `Holder.Box` would not be valid.

The alternative would prohibit the nested type declaration from capturing the protocol `Self` type. In this case, the nested type declaration's generic signature would *not* include the protocol `Self` type, so `Box` would be disallowed as written above, because of its stored property. In this model, the protocol would simply act as a namespace, and the nested type would not depend on the protocol in any other way. a nested type could then be referenced as a member of the protocol type itself, like `Holder.Box`.

Protocols in generic contexts. Historically, protocols could only be declared at the top level of a source file. In Swift 5.10 [82], this was relaxed to allow protocols to nest inside other declarations arbitrarily, as long as those declarations are not generic:

```
enum E {
  protocol P {}  // allowed as of SE-0404
}

struct S: E.P {}
```

However, protocols are still prohibited from appearing within generic declarations:

```
struct G<T>(_: T) {
  protocol P {  // error
    func f() -> T  // because what would this mean?
  }
}
```

If a protocol could depend on outer generic parameters in this way, its protocol generic signature would contain those other parameters and their requirements, in addition to `Self`. Haskell calls this a *multi-parameter type class*.

Effectively, each specialization of a generic protocol would be a distinct type, and the same concrete conforming type could conform to multiple specializations of a generic protocol with different implementations of each conformance:

```
struct S: G<Int>.P {...}
struct S: G<String>.P {...}
```

This would be a major change. Today, a conformance requirement [`T: P`] is effectively a unary predicate—a true or false statement—about this type parameter. A conformance to a generic protocol, on the other hand, is a more general kind of relation that relates *multiple* type parameters which then all "participate" in the conformance. At the very least, this entails a complete rethink of the formal system from the previous chapter, as well as most of the material in Part IV. To get another sense of the complexity this feature introduces, see [83].

6.3. Composition

Suppose that we have three generic signatures G, H, I, and a pair of substitution maps $\Sigma_1 \in \textsc{Sub}(G, H)$, $\Sigma_2 \in \textsc{Sub}(H, I)$. If we take any interface type $\texttt{X} \in \textsc{Type}(G)$, we can apply Σ_1 to $\texttt{X}$, and we get $\texttt{X} \otimes \Sigma_1 \in \textsc{Type}(H)$. If we then apply Σ_2 to $\texttt{X} \otimes \Sigma_1$, we get the following element of $\textsc{Type}(I)$:

$$(\texttt{X} \otimes \Sigma_1) \otimes \Sigma_2$$

Now, consider how we might get from $\texttt{X}$ to $(\texttt{X} \otimes \Sigma_1) \otimes \Sigma_2$ in one step.

Definition 6.6. The *composition* of the substitution maps Σ_1 and Σ_2, denoted by $\Sigma_1 \otimes \Sigma_2$, is defined as the unique substitution map which satisfies the following identity for all $\texttt{X} \in \textsc{Type}(G)$:

$$\texttt{X} \otimes (\Sigma_1 \otimes \Sigma_2) := (\texttt{X} \otimes \Sigma_1) \otimes \Sigma_2$$

Since $(\texttt{X} \otimes \Sigma_1) \otimes \Sigma_2 \in \textsc{Type}(I)$, it follows that $\Sigma_1 \otimes \Sigma_2 \in \textsc{Sub}(G, I)$; that is, the input generic signature of $\Sigma_1 \otimes \Sigma_2$ is the input generic signature of Σ_1, and the output generic signature of $\Sigma_1 \otimes \Sigma_2$ is the output generic signature of Σ_2. Substitution map composition gives us this mapping of sets:

$$\textsc{Sub}(G, H) \otimes \textsc{Sub}(H, I) \longrightarrow \textsc{Sub}(G, I)$$

Simply stated, applying the composition of two substitution maps to an interface type produces the same result as first applying the first substitution map, followed by the second. Now we will see how to construct $\Sigma_1 \otimes \Sigma_2$. First, suppose that G, the input generic signature of Σ_1, does not state any conformance requirements, so the behavior of Σ_1 is completely determined by its replacement types:

$$\Sigma_1 := \{\ldots, \texttt{τ_d_i} \mapsto \texttt{τ_d_i} \otimes \Sigma_1, \ldots\}$$

To construct $\Sigma_1 \otimes \Sigma_2$, we apply Σ_2 to each replacement type of Σ_1:

$$\Sigma_1 \otimes \Sigma_2 := \{\ldots, \texttt{τ_d_i} \mapsto \big((\texttt{τ_d_i} \otimes \Sigma_1) \otimes \Sigma_2\big), \ldots\}$$

It is immediate that the following identity now holds all generic parameters $\texttt{τ_d_i}$ of G, which completely determines $\Sigma_1 \otimes \Sigma_2$:

$$\texttt{τ_d_i} \otimes (\Sigma_1 \otimes \Sigma_2) = (\texttt{τ_d_i} \otimes \Sigma_1) \otimes \Sigma_2$$

In the general case where G states one or more conformance requirements, Σ_1 also stores a list of *root conformances*. In this case, substitution map composition also needs to apply Σ_2 to each root conformance of Σ_1 to properly form the new substitution map. Conformance substitution will be explained in Section 7.2.

Example 6.7. Substitution map composition can help us reason about the types of chained member reference expressions, like the type of "`x`" below:

```
struct Outer<T> {
  var inner: Inner<Optional<T>, Bool>
}

struct Inner<T, U> {
  var value: (T, U)
}

let outer: Outer<Int> = ...
let x = outer.inner.value  // What is the type of 'x'?
```

First, consider the type of the "`inner`" stored property of `Outer`. Let's denote its context substitution map by Σ_1:

$$\Sigma_1 := \{\texttt{τ_0_0} \mapsto \texttt{Optional<τ_0_0>}, \texttt{τ_0_1} \mapsto \texttt{Bool}\}$$

Note that the input generic signature of Σ_1 is the generic signature of `Inner`, which is the declaration being referenced, and the output generic signature of Σ_1 is the generic signature of `Outer`, the declaration the reference appears in.

Now, consider the type of the "`outer`" global variable. We will denote its context substitution map by Σ_2:

$$\Sigma_2 := \{\texttt{τ_0_0} \mapsto \texttt{Int}\}$$

The input generic signature of Σ_2 is the generic signature of `Outer`, which is the output generic signature of Σ_1, so we can compose the two substitution maps:

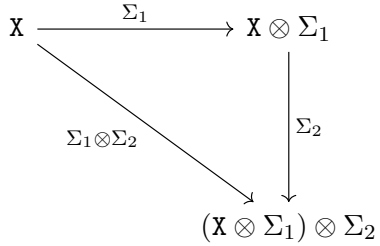
$$\Sigma_1 \otimes \Sigma_2 = \{\texttt{τ_0_0} \mapsto \texttt{Optional<Int>}, \texttt{τ_0_1} \mapsto \texttt{Bool}\}$$

Finally, consider the interface type of the "`value`" stored property of `Inner`. This is the tuple type `(τ_0_0, τ_0_1)`. The type of the expression "`outer.inner.value`" can be derived from this original type in two ways: we can apply each substitution map in turn, or we can apply their composition:

$$\begin{aligned}&(\texttt{(τ_0_0, τ_0_1)} \otimes \Sigma_1) \otimes \Sigma_2\\ &\quad = \texttt{(Optional<τ_0_0>, τ_0_1)} \otimes \Sigma_2\\ &\quad = \texttt{(Optional<Int>, Bool)}\end{aligned}$$

$$\begin{aligned}&\texttt{(τ_0_0, τ_0_1)} \otimes (\Sigma_1 \otimes \Sigma_2)\\ &\quad = \texttt{(τ_0_0, τ_0_1)} \otimes \{\texttt{τ_0_0} \mapsto \texttt{Optional<Int>}, \texttt{τ_0_1} \mapsto \texttt{Bool}\}\\ &\quad = \texttt{(Optional<Int>, Bool)}\end{aligned}$$

Commutative diagrams. A *commutative diagram* exhibits a collection of objects and operations, where each operation is represented by a labeled arrow. When we say this diagram *commutes*, we mean that whenever we have two paths with the same source and destination, we can follow the composition of operations given by either path and arrive at the same result. For example, we can illustrate Definition 6.6 with this diagram:



We will see more interesting commutative diagrams later.

Identity substitution map. We saw that every generic signature has an identity substitution map in Section 6.1. The identity substitution map comes up whenever we forward the generic parameters as arguments to another generic declaration. Now, we consider how the identity behaves under composition. There are two possibilities.

If $\Sigma \in \text{SUB}(G, H)$, we can compose Σ with the identity substitution map 1_G on the left. This applies Σ to each replacement type of 1_G, which projects each replacement type of Σ in turn; we assemble a new substitution map identical to Σ:

$$1_G \otimes \Sigma = \Sigma \tag{1}$$

We can also compose Σ with 1_H on the right. This applies 1_H to each replacement type of Σ. From our earlier discussion of the identity substitution map, we know this leaves the replacement types of Σ unchanged, so again we get:

$$\Sigma \otimes 1_H = \Sigma \tag{1}$$

A remark about primary archetypes, which we won't introduce until Chapter 8. Our definition of $\text{SUB}(G, H)$ was crafted to exclude substitution maps whose replacement types contain primary archetypes. If we also consider such substitution maps, then (1) still holds, but (2) does not. In Section 8.2, we will see that a different substitution map, the *forwarding substitution map*, acts as the identity element on such types, that we will refer to as *contextual types*.

Order of operations. A fact we will state without proof is that substitution map composition is an *associative* operation. That is, if Σ_1, Σ_2, and Σ_3 are any three substitution maps such that all below compositions are defined, then:

$$(\Sigma_1 \otimes \Sigma_2) \otimes \Sigma_3 = \Sigma_1 \otimes (\Sigma_2 \otimes \Sigma_3)$$

For example, this means that when $\mathtt{X}$ is any interface type, all of the following expressions must produce the same substituted type:

$((\mathtt{X} \otimes \Sigma_1) \otimes \Sigma_2) \otimes \Sigma_3$
$(\mathtt{X} \otimes \Sigma_1) \otimes (\Sigma_2 \otimes \Sigma_3)$
$(\mathtt{X} \otimes (\Sigma_1 \otimes \Sigma_2)) \otimes \Sigma_3$
$\mathtt{X} \otimes ((\Sigma_1 \otimes \Sigma_2) \otimes \Sigma_3)$
$\mathtt{X} \otimes (\Sigma_1 \otimes (\Sigma_2 \otimes \Sigma_3))$

Thus, we can omit parentheses without ambiguity in our notation:

$\mathtt{X} \otimes \Sigma_1 \otimes \Sigma_2 \otimes \Sigma_3$

Categorically speaking. We won't need this elsewhere, but it's worth mentioning briefly that a certain mathematical abstraction captures the properties of substitution map composition. A *category* is a collection of *objects* and *morphisms*. A morphism has a *source* and *destination* object, and the collection of morphisms with source A and destination B is denoted $\mathrm{Hom}(A, B)$. The morphisms must satisfy some axioms:

1. For every object A, there is an *identity morphism* $1_A \in \mathrm{Hom}(A, A)$.
2. If $f \in \mathrm{Hom}(A, B)$ and $g \in \mathrm{Hom}(B, C)$, there is a morphism $g \circ f \in \mathrm{Hom}(A, C)$, called the *composition* of g and f. (The conventional notation $g \circ f$ mirrors how we write $g(f(x))$ with function application on the left.)
3. Composition respects the identity: if $f \in \mathrm{Hom}(A, B)$, then $f \circ 1_A = 1_B \circ f = f$.
4. Composition is associative: if $f \in \mathrm{Hom}(A, B)$, $g \in \mathrm{Hom}(B, C)$ and $h \in \mathrm{Hom}(C, D)$, then $h \circ (g \circ f) = (h \circ g) \circ f$.

For example, we could define *the category of interface types* as follows:

- The objects are the sets $\textsc{Type}(G)$, for each generic signature G.
- Substitution maps are morphisms, and Hom is $\textsc{Sub}$.
- The source of a morphism $\Sigma \in \textsc{Sub}(G, H)$ is $\textsc{Type}(G)$.
- The destination of a morphism $\Sigma \in \textsc{Sub}(G, H)$ is $\textsc{Type}(H)$.
- Identity substitution maps are identity morphisms.
- Morphism composition $g \circ f$ is substitution map composition $f \otimes g$.

This happens to be a *concrete* category, where every object is a set and every morphism is a function between sets, but this need not be the case in general. In programming, we often encounter category theory when working with data structures and higher-order functions; an excellent introduction to the topic is [84].

6.4. Subclassing

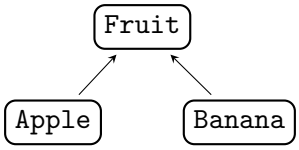
The Swift language supports *subclassing*, or *inheritance*. After some preliminaries, we will use the type substitution algebra to describe the interaction between subclassing and generics. This will be useful when we discuss the semantics of superclass requirements in later chapters. Since superclass requirements are not an essential part of the core Swift generics model, the reader can safely skip this section on a first reading.

An inheritance relationship is established when some class declaration, known as the *subclass*, states a *superclass type* in its inheritance clause. Unlike C++, Swift does not allow multiple inheritance, so a class declaration can have at most one superclass type. Unlike Java, there is no distinguished `Object` class at the root of the hierarchy, so it is common to have class declarations with no superclass type at all.

Example 6.8. First, consider the case where the superclass and the subclass are both non-generic. In the below, `Apple` and `Banana` state the superclass type `Fruit`:

```
class Fruit {
  func eat() {...}
  func juice() {...}
}

class Apple: Fruit {
  override func eat() {...}
}

class Banana: Fruit {}

let f: Fruit = Apple()  // implicit conversion from Apple to Fruit
f.eat()                 // vtable-dispatched call to Apple.eat()
```

In the expression type checker, `Apple` and `Banana` are *subtypes* of `Fruit`. This allows us to assign the result of the `Apple()` constructor call to our variable `f`, whose type is `Fruit`. We can visualize this with a class hierarchy diagram:



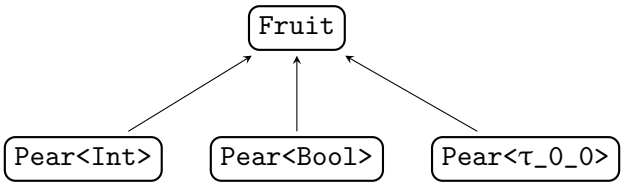
Qualified lookup walks up the class hierarchy, so a qualified lookup into `Apple` or `Banana` will also find members of `Fruit`. Furthermore, subclasses can *override* methods, properties, and subscripts declared in their superclass. In the above, `Apple` overrides `Fruit.eat()` with its own implementation.

An instance of `Fruit` might actually be an `Apple` at run time, so a call to `eat()` on a value type of `Fruit` must be dispatched through a *vtable* stored in the object header. The vtable for `Fruit` points at `Fruit`'s original implementation of `eat()` and `juice()`, while the vtable for `Apple` replaces `eat()` with a pointer to `Apple.eat()`.

Example 6.9. Next, suppose we declare a generic subclass of `Fruit`. Notice how we override `Fruit.eat()` to print the concrete replacement type for `τ_0_0`:

```
class Pear<T>: Fruit {
  override func eat() {
    print(T.self)
  }
}
```

With `Pear` as above, we can form various generic class types, such as `Pear<Int>`, `Pear<Bool>`, and so on; these are all obtained by applying a substitution map to the declared interface type `Pear<τ_0_0>`. The key fact about this example is that every specialization of `Pear` has the same superclass type, `Fruit`, because `Pear`'s superclass type does not depend on its generic parameter `τ_0_0`. Thus, as far as the expression type checker is concerned, all specializations of `Pear` are subtypes of `Fruit`. Our complete class hierarchy diagram is now infinite, but here is one part of it:



This example implements a design pattern known as *type erasure*. If we take a value of type `Pear<X>` for some `X`, we can always convert it to a value of type `Fruit`, effectively *erasing* the generic argument `X` from the static type of the resulting value. This allows the type `X` to vary dynamically. In the below, the static type of `f` is just `Fruit`, but the dynamic type of the value stored within is either `Pear<Int>` or `Pear<String>`, depending on the outcome of an arbitrary check:

```
let someCondition: Bool = ...
let p1 = Pear<Int>()
let p2 = Pear<String>()
let f: Fruit = (someCondition ? p1 : p2)
```

In Chapter 14, we will learn about *existential types*, another more general type erasure mechanism built into the language, based on protocols.

Generic superclasses. The interesting case arises when both subclass and superclass are generic. The superclass type of a class declaration is an interface type, so it can contain type parameters from the generic signature of the subclass. For example:

```
class StoneFruit<T> {}
class Mango<U>: StoneFruit<Array<U>> {}
```

In certain situations, we don't care about generic arguments at all, and we say that the *superclass declaration* of `Mango` is `StoneFruit`. However, the full relationship is expressed by the fact that the superclass *type* of `Mango` is `StoneFruit<Array<τ_0_0>>`. We can also obtain the superclass type of a class *type*, that is, a reference to a class declaration specialized with a list of generic arguments. This is defined in terms of applying a substitution map to the superclass type of a class declaration.

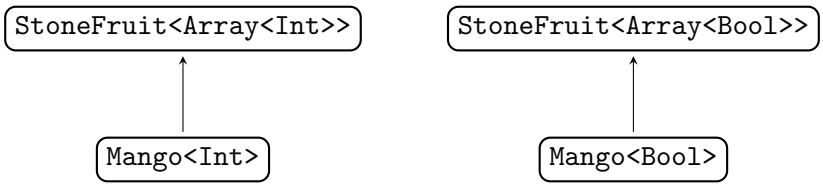
Algorithm 6B (Get superclass type)**.** Takes a class type `C` as input. Returns the superclass type of `C`, or null if `C` does not have a superclass type.

1. Let c be the class declaration of `C`. If c doesn't have a superclass type, return null.
2. Let `S` be the superclass type of c.
3. Let Σ be the context substitution map of `C`.
4. Return $\texttt{S} \otimes \Sigma$.

Example 6.10. The class type `Mango<Int>` is formed by applying the substitution map $\{\texttt{τ_0_0} \mapsto \texttt{Int}\}$ to the declared interface type `Mango<τ_0_0>`. To compute the superclass type of `Mango<Int>`, we apply this substitution map to the superclass type of the declaration, which is `StoneFruit<Array<τ_0_0>>`:

$$\texttt{StoneFruit<Array<τ_0_0>>} \otimes \{\texttt{τ_0_0} \mapsto \texttt{Int}\} = \texttt{StoneFruit<Array<Int>>}$$

The below class hierarchy diagram shows the relationship (or lack hereof) between the two types `Mango<Int>` and `Mango<Bool>`:



Now, consider the superclass type of `Mango<τ_0_0>`. We saw at the beginning of this chapter that the context substitution map of the declared interface type is the identity substitution map. Indeed, since $\texttt{Mango<τ_0_0>} \otimes \{\texttt{τ_0_0} \mapsto \texttt{τ_0_0}\} = \texttt{Mango<τ_0_0>}$, the superclass type of `Mango<τ_0_0>` is `StoneFruit<Array<τ_0_0>>`, which is the same as the superclass type of the declaration of `Mango`.

Suppose that the class type `C` given to Algorithm 6B is an interface type for some generic signature H, so that $\texttt{C} \in \textsc{Type}(H)$. If G is the generic signature of c in Step 2, then $\Sigma \in \textsc{Sub}(G, H)$ in Step 1. We already said that `S`, the superclass type of c, is an element of $\textsc{Type}(G)$, so our result $\texttt{S} \otimes \Sigma \in \textsc{Type}(H)$. That is, the superclass type of `C`, if it exists, is again an interface type for H.

Not every interface type has a superclass type. Thus, for every generic signature H, we get the following *partial* mapping of sets, which we denote by "superclass":

$$\textsf{superclass}\colon \textsc{Type}(G) \to \textsc{Type}(G)$$

If `C` is a class type, $\texttt{C}'$ is the superclass type of `C`, and Σ is a substitution map, then the following diagram commutes:

$$\begin{array}{ccc} \texttt{C} & \xrightarrow{\Sigma} & \texttt{C} \otimes \Sigma \\ {\scriptstyle \textsf{superclass}}\downarrow & & \downarrow{\scriptstyle \textsf{superclass}} \\ \texttt{C}' & \xrightarrow{\Sigma} & \texttt{C}' \otimes \Sigma \end{array}$$

Example 6.11. We can restate our previous example this way:

$$\begin{array}{ccc} \texttt{Mango<τ_0_0>} & \xrightarrow{\{\texttt{τ_0_0} \mapsto \texttt{Int}\}} & \texttt{Mango<Int>} \\ {\scriptstyle \textsf{superclass}}\downarrow & & \downarrow{\scriptstyle \textsf{superclass}} \\ \texttt{StoneFruit<Array<τ_0_0>>} & \xrightarrow{\{\texttt{τ_0_0} \mapsto \texttt{Int}\}} & \texttt{StoneFruit<Array<Int>>} \end{array}$$

Superclass substitution map. If qualified lookup into a class type finds a member one or more levels up the class hierarchy, we need to form the correct substitution map for the member reference expression.

Example 6.12. When we invoke `method()` on a value of type `Top<Int>` below, the substitution map for the call is the context substitution map of `Top<Int>`, because `method()` is a direct member of `Top`:

```
class Top<T> {
  func method() {}
}
Top<Int>().method()
```

If we call `method()` on a subclass of `Top` though, this is no longer the case:

```
class Mid<X, Y>: Top<(Y, X)> {}
class Bot: Mid<Int, Bool> {}
Bot().method()
```

The class type `Bot` is not generic, so it's context substitution map is empty. However, we need a substitution map for `<τ_0_0>`, the generic signature of `method()`. Notice how starting from `Bot`, we reach the parent context of `method()` in two hops; we jump to `Mid`, and then to `Top`. The superclass types of `Mid` and `Bot` are as follows:

$$\mathsf{superclass}(\texttt{Mid<τ_0_0, τ_0_1>}) = \texttt{Top<(τ_0_1, τ_0_0)>} = \texttt{Top<τ_0_0>} \otimes \Sigma_1$$
$$\mathsf{superclass}(\texttt{Bot}) = \texttt{Mid<Int, Bool>} = \texttt{Mid<τ_0_0, τ_0_1>} \otimes \Sigma_2$$

with the two substitution maps below:

$$\Sigma_1 := \{\texttt{τ_0_0} \mapsto \texttt{(τ_0_1, τ_0_0)}\}$$
$$\Sigma_2 := \{\texttt{τ_0_0} \mapsto \texttt{Int}, \texttt{τ_0_1} \mapsto \texttt{Bool}\}$$

The input generic signature of Σ_1 is the generic signature of `Top`, and the output generic signature of Σ_1 is the generic signature of `Mid`. Also, the input generic signature of Σ_2 is the generic signature of `Mid`, and its output generic signature is the generic signature of `Bot`, which is the empty generic signature. If we compose the two substitution maps, we get the following:

$$\Sigma_1 \otimes \Sigma_2 = \{\texttt{τ_0_0} \mapsto \texttt{(Bool, Int)}\}$$

This is the substitution map for the call. In particular, when we call `method()` on a value of type `Bot`, the type of the `self` parameter inside the method is going to be:

$$\texttt{Top<τ_0_0>} \otimes \Sigma_1 \otimes \Sigma_2 = \texttt{Top<(Bool, Int)>}$$

In fact, we can also get the same result if we apply "superclass" twice:

$$\mathsf{superclass}(\mathsf{superclass}(\texttt{Bot})) = (\texttt{Top<τ_0_0>} \otimes \Sigma_1) \otimes \Sigma_2$$

We can generalize this as follows. We are given a class type together with the parent class declaration of some referenced member, so we must walk up the class hierarchy and collect substitutions along the way, until we reach the parent class.

Algorithm 6C (Get superclass type for declaration)**.** As input, takes a class type `C` and a class declaration d. Returns the superclass type of `C` for d.

1. Let c be the class declaration of `C`. If $c = d$, return `C`.
2. Let `S` be the superclass type of c, or signal an invariant violation if it doesn't have one. (That would indicate d is not a superclass of the original class type `C`, in which case members of d should not be visible as members of `C`.)
3. Let Σ be the context substitution map of `C`. Set `C` $\leftarrow$ `S` $\otimes \Sigma$. Go back to Step 1.

Note that this algorithm returns the type of the `self` parameter as seen "inside" the referenced member. To get the substitution map for the reference, we ask this type for its context substitution map. This is known as the *superclass substitution map*.

This wraps up our discussion of subclassing for now. In later sections, we will learn more about this topic, and in particular, we will see how this all intersects with superclass requirements in a generic signature:

- Chapter 7 explains how a subclass inherits conformances from its superclass.
- Section 8.1 explains how archetypes behave when their type parameter is subject to a superclass requirement.
- Section 9.1 explains how type resolution resolves a reference to a member type declared in the superclass of a base type.
- Section 9.3 explains how we can check if a substitution map satisfies a superclass requirement.
- Section 11.2 and Section 11.4 explain what it means for the user's program to state a superclass requirement that is either redundant or in conflict with some other requirement.

6.5. SIL Type Lowering

We end this chapter by taking a brief look at the SIL type system. Recall that SIL is an intermediate representation, constructed from the abstract syntax tree by the compiler's SILGen pass. In this section, the types appearing in the abstract syntax tree will be called *formal types*, to distinguish them from the *lowered types* used by SIL. The translation of formal types into lowered types is the process of *SIL type lowering*.

Our primary focus in this section is the relationship between SIL type lowering and type substitution. We cannot hope to gain a complete understanding of SIL type lowering here, so certain concepts will be omitted or left unexplained. For more details about SIL and SIL types, see [40, 85]. The reader may also choose to skip this section entirely, because this material is not needed in the rest of this book.

Loadable or address-only. Most formal types are also lowered types, and SIL type lowering simply returns the original formal type in this case. One major exception concerns the types of functions. Unlike the function types of Section 3.2, which we will refer to as *formal* function types in this section, the type of a function in SIL is a different kind of type that does not appear in the abstract syntax tree: the *SIL function type*. Compared to formal function types, SIL function types encode more details about the calling convention. We begin by taking a look at one of these details.

A SIL function type makes explicit whether each lowered parameter and result can be passed in registers, or if it must be passed indirectly, via a pointer to a value stored in memory. If the parameter's lowered type is *loadable*, it is passed directly; otherwise, the lowered type is *address-only*, and it must be passed indirectly. Almost all lowered types are loadable. The exceptions are the following:

1. Type parameter types and archetype types are address-only when they are not subject to an `AnyObject` layout constraint (Section 4.2).

   (Values of such type have an unknown size at compile time.)

2. Weak reference types (Section 3.3) are address-only.

   (While a weak reference is just the address of a class instance, the location of each weak reference must be registered with the Swift runtime at all times, and so they can only "exist" in memory.)

3. Struct and enum types with *resilient* declarations are address-only.

   (We briefly talked about the resilience model in Section 2.5, but further discussion is beyond the scope of this book. See [49, 50] for details.)

4. Aggregate types, such as tuples with address-only elements, structs with address-only stored properties, and enums with address-only cases, are address-only.

   (The property of being address-only is transitive with respect to containment.)

5. Existential types (Chapter 14) are address-only when not `AnyObject` constrained.

   (A value of such a type can *contain* a value of address-only type, and so the existential container itself must be address-only.)

We will look at some examples before we discuss the general case.

Example 6.13. The `Int` and `String` types in the standard library are loadable, and so is the below `Loadable` struct type, because its stored properties are loadable:

```
struct Loadable {
  var x: Int
  var y: String
}
```

On the other hand, the below `AddressOnly` struct is address-only, because it contains a weak reference:

```
struct AddressOnly {
  weak var x: AnyObject?
}
```

For generic nominal types, deciding if a type is address-only requires type substitution.

Example 6.14. First, consider the standard library `Optional` enum:

```
public enum Optional<Wrapped> {
  case none
  case some(Wrapped)
}
```

In this section, we will say that a generic nominal type formed from this declaration is an "optional type." (Not to be confused with the optional sugared type that is spelled as `Int?`, from Section 3.2—however, the latter desugars to the former.)

The generic argument of an optional type is called the *payload type* of the optional. In memory, a value of optional type must be large enough to store the payload, together with a tag value indicating if the selected case is `none` or `some`. (The tag is also sometimes encoded *inside* the payload [86].)

An optional type is loadable if and only if its payload type is loadable. Thus, `Optional<Int>` is loadable, while `Optional<AddressOnly>`, with `AddressOnly` as in the previous example, is address-only.

Example 6.15. Next, consider this enum declaration, which is identical to `Optional` except that the "`some`" case is declared to be `indirect`:

```
enum IndirectOptional<Wrapped> {
  case none
  indirect case some(Wrapped)
}
```

Instead of storing the payload inline, an `indirect` enum case is represented by a single reference-counted pointer that refers to a heap-allocated box. For this reason, both `IndirectOptional<Int>` and `IndirectOptional<AddressOnly>` are loadable.

Example 6.16. This generic struct declaration has no stored properties at all:

```
struct Phantom<T> {}
```

Hence, both `Phantom<Int>` and `Phantom<AddressOnly>` are loadable.

Example 6.17. When a type contains type parameters, its category depends on the referencing generic context:

```
func f<T, U: AnyObject>(_ t: T, _ u: U) {
  let x: Optional<T> = ...  // address-only
  let y: Optional<U> = ...  // loadable
}
```

The type of `x` is address-only, because `T` is unconstrained, while the type of `y` is loadable, because `U` is subject to an `AnyObject` layout constraint.

Now, we're going to look at the algorithm. As described here, it accepts both formal and lowered types. In the actual implementation, it operates on lowered types only, and we compute a number of additional properties of the type's layout in the same way.

Algorithm 6D (Decide if type is address-only)**.** Receives a type `X` as input, together with an optional generic signature G in the case where `X` contains type parameters. Returns one of "loadable" or "address-only" as output.

1. If `X` is a **type parameter**, say `X` = `T`: evaluate the `requiresClass`(G, `T`) generic signature query. If the answer to the query is true, return "loadable," otherwise return "address-only."
2. If `X` is an **archetype type** (Chapter 8): proceed as above, but in place of G, use the generic signature stored within the archetype.
3. If `X` is a **weak reference type**, return "address-only."
4. If `X` is a **struct type**, say `X` = $\texttt{X}_d \otimes \Sigma$ where Σ is the context substitution map and $\texttt{X}_d$ is the declared interface type of its struct declaration d, then, for each stored property of d:
   a) Let `Y` be the interface type of the stored property.
   b) Compute the substituted type $\texttt{Y} \otimes \Sigma$.
   c) Recursively check if $\texttt{Y} \otimes \Sigma$ is address-only.
   d) If it is address-only, immediately return "address-only."

   Otherwise, if all substituted stored property types are loadable, return "loadable."
5. If `X` is an **enum type**:
   a) Check if the enum declaration is `indirect`; if so, return "loadable."
   b) Otherwise, proceed as in the struct case, but only consider those enum cases *not* declared as `indirect`.

      (The lowered type of an `indirect` case is always a reference-counted pointer to a heap-allocated box, and such a pointer is loadable. If the enum itself is `indirect`, we proceed as if each case was `indirect`.)
6. If `X` is a **tuple type**, recursively check if each element type of `X` is address-only. If any are address-only, return "address-only." Otherwise, return "loadable."
7. If `X` is an **existential type**, check if the existential type satisfies the `AnyObject` layout constraint. If so, return "loadable," otherwise "address-only."
8. If `X` is a **class type**, **metatype type**, or **(SIL) function type**, return "loadable."

Exploding tuples. A formal function type has zero or more parameter types, and exactly one return type. This return type can be a tuple type, which is how we model the case of having no return value, or more than one return value.

Like a formal function type, a SIL function type has zero or more lowered parameter types, but unlike a formal function type, a SIL function type *also* has zero or more lowered result types. When a tuple type appears in the parameter list or return type of a formal function type, SIL type lowering will "explode" it into multiple lowered parameters or lowered results.

Both lowered parameters and lowered results are annotated with a *convention*, which encodes two things: whether the value is passed directly in registers or indirectly via memory, and also, how the ownership of the value changes during the call:

1. The directness is determined by the lowered parameter or result type; if the type is loadable, we may pass it directly (but sometimes, we must pass it indirectly, as we will see below); when it is address-only, it must be passed indirectly.

2. A discussion of ownership is beyond the scope of this book. See [85] for details.

A SIL function type also has an overall convention specifier. The possible conventions are a superset of those for formal function types, which were listed in Section 3.2. For us, the two that we need are `@convention(thin)`, for functions that don't capture values (such as global functions and methods), and `@convention(thick)`, for functions that do (such as closure expressions). A thick function value consists of a function pointer together with a context, which can be used to store captured values—the caller passes the context as an additional parameter. Finally, a SIL function type has a generic signature.

Example 6.18. Here is a function declaration with two parameters, the second of which is a tuple. The second element of the tuple is our address-only type from Example 6.13:

```
func foo(x: Int, y: (String, AddressOnly)) {}
```

Here is the notation used by the compiler when printing this declaration's SIL function type (for example, put the above declaration together with the `AddressOnly` struct from earlier in a source file, and build it the `-emit-silgen` flag):

```
@convention(thin)
(Int, @guaranteed String, @in_guaranteed AddressOnly) -> ()
```

The `Int` parameter does not have a convention annotation, so it is passed directly. The `Int` type is trivial value, so its values can be moved and copied without ownership concerns. The `String` parameter is also passed directly, but this time, the `@guaranteed` convention indicates that ownership of the string is retained by the caller. Finally, the `AddressOnly` parameter is address-only, so the `@in_guaranteed` convention is used to pass the value indirectly, and once again, the caller retains ownership of the value. Since `foo()` does not state a return type, the SIL function type has zero results.

Example 6.19. Here is a generic function that returns a tuple:

```
func flip<T, U>(t: T, u: U) -> (U, T) {
  return (u, t)
}
```

Here is its SIL function type:

```
@convention(thin)
<T, U> (@in_guaranteed T, @in_guaranteed U) -> (@out U, @out T)
```

Both parameters are passed indirectly, with the caller retaining ownership (so in fact the implementation of the function most copy these values). The SIL function type also returns two results indirectly (at the machine level, the caller provides a pair of return buffers, large enough to hold both results.)

Re-abstraction thunks. So, to get a SIL function type of a function declaration or closure expression, we take its interface type, which is a formal function type. We explode tuples in the parameter list and return type, compute the category of each parameter and result, and do a few other things. Of course, some of those parameter and result types may themselves be function types, and we recursively lower those function types, too. We then form our SIL function type. This is sufficient to explain SILGen's behavior with direct calls of functions and closures, but it is not the whole story.

To understand what happens when a function value is passed between generic contexts, we need to talk about *abstraction patterns*. The operation of SIL type lowering in the compiler receives not just a formal type, but also an abstraction pattern.

This complication arises for the following reason. When a substitution map is applied to a SIL function type, the type parameters in the SIL function type are replaced with concrete types. However, substitution does not change the directness of the SIL function type's parameters and results. If an original type parameter is address-only but its replacement type is loadable, the resulting SIL function type will differ from what we would get if we just lower the substituted formal type instead. Typically, a closure expression is emitted with a "natural" abstraction pattern that is its own formal type. To change the abstraction pattern of a function value, SILGen wraps the function value in a *re-abstraction thunk*. There are two kinds of thunk:

1. A **substituted-to-original thunk** appears when a closure is passed in to a generic function. It wraps a function type lowered with itself as the abstraction pattern, with a function type lowered with a more general abstraction pattern.

2. An **original-to-substituted thunk** appears when a closure is returned from a generic function. It wraps a function type lowered with a more general abstraction pattern, with one lowered with the fully substituted abstraction pattern.

Example 6.20. Consider this rather pointless generic function that applies a closure to a value and returns the result:

```
func apply<T>(_ x: T, _ fn: (T) -> T) -> T {
  return fn(x)
}
```

Say we call our function like this, with the substitution map $\Sigma := \{\texttt{τ_0_0} \mapsto \texttt{String}\}$:

```
let fn: (String) -> String = { $0.uppercased() }
let result = apply("hello world", fn)
```

If we apply Σ to the formal type of the "`fn`" parameter to `apply()`, we get:

$$[\texttt{(τ_0_0) -> τ_0_0}] \otimes \Sigma = \texttt{(String) -> String}$$

The above formal type is identical to the formal type of the closure expression passed in as an argument by the caller. Now, let's take a look at lowered types. Since `String` is loadable, lowering the above formal type gives:

```
@convention(thick) (@guaranteed String) -> (@owned String)
```

On the other hand, the lowered type of the "`fn`" parameter passes the parameter and returns the result indirectly:

```
@convention(thick) <T> (@in_guaranteed T) -> (@out T)
```

Finally, if we apply Σ to this SIL function type, we obtain the following:

```
@convention(thick) (@in_guaranteed String) -> (@out String)
```

To reconcile the difference, SILGen wraps the closure with a substituted-to-original thunk before handing it off to `apply()`. This thunk receives a pointer to a `String`, loads it, and calls the wrapped closure with the loaded value. The closure returns a new `String` directly, which the thunk stores into the indirect result buffer before returning.

More about abstraction patterns. For our purposes, an abstraction pattern simply consists of the unsubstituted formal type for computing the directness of each parameter and result in the SIL function type. (In reality, abstraction patterns also carry a *kind* and some additional information, but we won't need this level of detail here.) So, a re-abstraction thunk is then basically a closure that wraps another closure. The body of the thunk takes each parameter and forwards it to the wrapped closure, then takes the the wrapped closure's result and returns it to the caller of the thunk. In doing so, a re-abstraction thunk can change the convention of the parameters and results.

There is also a useful peephole optimization that we will mention now. Very often, the closure being re-abstracted is a literal closure expression, in which case SILGen is able to directly emit the closure with the expected abstraction pattern, instead of immediately wrapping it in a thunk. This avoids an unnecessary heap allocation.

Opaque abstraction patterns. Type lowering traverses a formal function type "in parallel" with the abstraction pattern when in order to build the SIL function type. The abstraction pattern must have the same shape as the formal type, in the sense that the formal type should be obtainable from the abstraction pattern by substitution. For example, the abstraction pattern `(Int, τ_0_1) -> τ_0_2` is compatible with the formal type `(Int, (Bool, String) -> Float) -> Bool`, but not with the formal type `(Bool, Int, String) -> (Float, Float)`—in the latter, the number of parameters doesn't match up, and neither does the return type.

The implementation of this "parallel decomposition" is mostly obvious, except when the abstraction pattern consists of a single type parameter. This is referred to as an *opaque* abstraction pattern in the implementation (not to be confused with opaque parameters or opaque result types). In this situation, we lower the formal function type in the "most general" way, as if each one of its parameters and results was passed indirectly, without exploding tuples. The next three examples will illustrate this.

Example 6.21. Let's modify the `apply()` function from Example 6.20 to receive an array of closures instead, while making the closure's parameter concrete:

```
func apply1<T>(_ x: String, _ fns: [(String) -> T]) -> [T] {
  return fns.map { fn in fn(x) }
}
```

We can do the same but make the result concrete instead:

```
func apply2<T>(_ x: T, _ fns: [(T) -> String]) -> [String] {
  return fns.map { fn in fn(x) }
}
```

Say we call both functions with an array containing some large number of closures:

```
let fns: [(String) -> String] = [
  { $0.uppercased() },
  /* and more... */
]
apply1("hello ", "world", fns)
apply2("hello ", "world", fns)
```

We don't want to re-abstract every closure in the array before we pass it to `apply1()` and `apply2()`. Instead, when storing a closure in a generic container, we re-abstract it into its most general form, where both the parameter and result are passed indirectly. This is accomplished by wrapping each closure in a substituted-to-original thunk before storing it in the array, and wrapping it again in an original-to-substituted thunk when loading it from the array.

An interesting limitation of this implementation model is that re-abstraction thunks can nest, resulting in potentially unbounded memory usage. This was reported by a Swift developer on the forums [87].

Example 6.22. Suppose we take an array of closures, repeatedly load each element from the array, and store it back without modification:

```
var fns: [(String) -> String] = []

// Initialize the array
for n in 0 ..< 100 {
  array.append({ $0.lowercased() })
}

// Repeat this as many times as necessary
for _ in 0 ..< 1000 {
  for n in 0 ..< array.count {
    let fn = array[n]
    array[n] = fn
  }
}
```

Each loaded element will be wrapped with an original-to-substituted thunk, and this thunk will then be wrapped with a substituted-to-original thunk before being stored back to the same location in the array. Thus, each iteration will allocate a pair of closure contexts on the heap, and repeating will consume an arbitrary amount of memory.

For a simple example like the above, we could detect this situation, and either avoid emitting redundant thunks in SILGen, or clean up redundant thunks with a SIL optimizer pass. A compile-time fix would not completely solve the problem, though, because we can just change the second "`for`" loop into the following:

```
for _ in 0 ..< 1000 {
  for n in 0 ..< array.count {
    let fn = array[n]
    let fn2 = someFunction(fn)
    array[n] = fn2
  }
}
```

Perhaps `someFunction()` just returns its argument unchanged, but if its defined in another module, the compiler has no way to know this, because it cannot analyze its

body, so we would encounter the same problem again. To solve this problem completely would require some kind of run time check when forming a re-abstraction thunk, to "collapse" multiple levels of nested thunks into one.

Our final example shows that a re-abstraction thunk may also need to explode and implode tuple types appearing in the formal function type's parameters and result.

Example 6.23. Consider a closure with formal type `(String, (Int, Bool)) -> ()`. Using its own formal type as the abstraction pattern, we get a SIL function type with three parameters and no results:

```
(@guaranteed String, Int, Bool) -> ()
```

The most general lowered type for this function type, however, is the following:

```
(@in_guaranteed String, @in (Int, Bool)) -> (@out ())
```

The latter lowered type receives the two-element tuple as a single indirect parameter. It also returns an empty tuple indirectly (an empty tuple value does not take up any space in memory, so there is nothing to load or store; however, the calling convention still allots a parameter to serve as the indirect result buffer). Now, if we are asked to emit a substituted-to-original thunk to convert from the first type to the second, the thunk must explode the tuple and load both elements from memory before passing them directly to the wrapped closure. Likewise, an original-to-substituted thunk must implode the last two lowered parameters into a single tuple value to be passed indirectly.

Optional payloads. Most generic nominal types use a uniform representation that does not depend on the abstraction pattern, and type lowering simply returns the original formal type unchanged when given such a formal type, ignoring the abstraction pattern. In particular, the generic arguments of generic nominal types are not themselves lowered.

Optional types are the exception. (Recall the declaration of `Optional` from Example 6.14.) Optional function types are particularly common, and we don't want to pay the cost of allocating a re-abstraction thunk every time we wrap a function type with an optional, or unwrap an optional function type.

For this reason, SIL type lowering treats optional types as a special case. When lowering a formal optional type with an optional type as the abstraction pattern, we lower its payload type with the abstraction pattern's formal type, and form a lowered optional type from the result.

For example, consider the formal type `Optional<(String) -> String>`. Lowering this type with itself as the abstraction pattern yields the following—notice how the generic argument becomes a SIL function type, and this SIL function type receives its parameter and returns its result directly:

```
Optional<@convention(thick) (@guaranteed String) -> (@owned String)>
```

Since the same formal optional type can map to one of several lowered optional types depending on the abstraction pattern, values of optional types may need to be re-abstracted. We accomplish this by generating a conditional branch. If the active case is "`none`" we simply return this empty value. If the active case is "`some`" we wrap the payload in a re-abstraction thunk, and then wrap this thunk in a new optional value.

Finally, we are ready to look at the type lowering algorithm.

Algorithm 6E (Lower type with abstraction pattern)**.** Receives a formal type `X` and an abstraction pattern `Y` as input, together with an optional generic signature G in the case where `Y` contains type parameters. Returns a lowered type. The structure of the abstraction pattern must match the formal type.

1. If `X` is a **function type**:
   a) Walk the formal parameters of `X` and `Y` in parallel:
      i. Lower each formal parameter type using the corresponding formal parameter type of `Y` as the abstraction pattern.
      ii. For any formal parameter of `Y` that is a tuple type, explode the lowered type into multiple lowered parameters.
      iii. Compute a category for each lowered parameter type using Algorithm 6D.
   b) Consider the formal result types of `X` and `Y`:
      i. Lower the formal result type of `X` using the result type of `Y` as the abstraction pattern.
      ii. If the formal result type of `Y` is a tuple type, explode the lowered result type into multiple lowered results.
      iii. Compute a category for each lowered result type using Algorithm 6D.
   c) Collect the lowered parameter types, lowered result types, and their categories, to form a new SIL function type.
2. If `X` is a **tuple type**, walk the elements of `X` and `Y` in parallel, and lower each element. Collect the lowered element types to form a lowered tuple type, and return the result.
3. If `X` is an **optional type**:
   a) Lower the payload type of `X` using the payload type of `Y` as the abstraction pattern.
   b) Form a new optional type from this lowered payload type.
4. If `X` is a **struct type**, **enum type** (except for `Optional`, which was handled above), or **class type**, return `X` itself. (The formal type is already a lowered type in this case, and we ignore the abstraction pattern `Y`.)

Substitution with lowered types. We're now going to investigate the meaning of applying a substitution map to a lowered type. This is a common operation in the SIL optimizer; for example, it is used to generate specializations of generic functions. If the implementation of the function is visible from the call site, the optimizer can generate a specialization by cloning each SIL instruction in the function's body, and applying the substitution map from the call site.

A substitution map's replacement types are always formal types. When we substitute a type parameter appearing in a lowered type, we must sometimes lower the replacement type if it appears in a certain position. We always lower the replacement type with an opaque abstraction pattern. We do this precisely in those positions where Algorithm 6E would lower a formal type that appears there. Such a position within a type is referred to as *lowered position*. Lowered positions are defined recursively. The entire original lowered type is itself in lowered position. Furthermore:

1. A SIL function type can only appear in lowered position, and all of its parameter and result types are also in lowered position.

2. If a tuple type appears in lowered position, its element types are also in lowered position.

3. If an optional type appears in lowered position, its generic argument is also in lowered position.

If a type parameter occurs in any other position, the replacement type is not lowered, and it is substituted as-is.

Let's say that `Y` is an abstraction pattern, `X` is a formal type, and $L(\texttt{Y}, \texttt{X})$ is the lowered type for `X` with respect to `Y`. Finally, let "=" denote canonical type equality. In Example 6.20, we saw that lowering a type with its own abstraction pattern is not, in general, compatible with type substitution—that is, $L(\texttt{X}, \texttt{X}) \otimes \Sigma \neq L(\texttt{X} \otimes \Sigma, \texttt{X} \otimes \Sigma)$. However, there is an identity relating type lowering with type substitution. When the abstraction pattern `Y` remains unsubstituted and we apply a substitution map to both sides, it is always true that $L(\texttt{Y}, \texttt{X}) \otimes \Sigma = L(\texttt{Y}, \texttt{X} \otimes \Sigma)$.

Example 6.24. Let's apply $\{\texttt{τ_0_0} \mapsto \texttt{Int}, \texttt{τ_0_1} \mapsto [\texttt{() -> String}]\}$ to a pair of lowered types. First, consider the generic nominal type `Array<(τ_0_0, τ_0_1)>`. Neither of the two type parameters are in lowered position, so type substitution will replace them without lowering, and we just get `Array<(Int, () -> String)>`.

Now, suppose we apply Σ to `Optional<(τ_0_0, τ_0_1)>`. Both type parameters are in lowered position, so type substitution will lower the replacement types with an opaque abstraction pattern. We see that while `Int` remains unchanged, the formal function type `() -> String` becomes a SIL function type, and we get:

```
Optional<(Int, @convention(thick) () -> (@out String))>
```

History. Swift 3.1 introduced the special type lowering support and re-abstraction for optional payloads; prior to this, optional types were lowered like every other generic nominal type [88]. The most general form of a function type changed in Swift 5, when `-swift-version 3` mode was dropped (these days, the `-swift-version` flag is named `-language-mode`, and the minimum supported language mode at the time of writing is 4). As we saw in Section 3.2, Swift 3 did not distinguish between function types that receive multiple parameters and function types with a single parameter of tuple type. Thus, in Swift 3, the most general form of a function type had to implode all parameters into a single indirect parameter of tuple type, which was somewhat inefficient. Resolving this was a strong impetus for dropping Swift 3 source compatibility [89]. Finally, Swift 5.6 introduced the peephole optimization allowing SILGen to emit a closure expression with an arbitrary abstraction pattern, rather than immediately wrapping it with a substituted-to-original thunk [90].

6.6. Source Code Reference

Key source files:

- `include/swift/AST/SubstitutionMap.h`
- `include/swift/AST/Type.h`
- `include/swift/AST/Types.h`
- `lib/AST/SubstitutionMap.cpp`
- `lib/AST/TypeSubstitution.cpp`

`Type`	*class*

See also Section 3.4.

- `subst()` applies a substitution map to this type and returns the substituted type.

`TypeBase`	*class*

- `getContextSubstitutionMap()` returns this type's context substitution map.

`GenericSignature`	*class*

See also Section 5.7.

- `getIdentitySubstitutionMap()` returns the identity substitution map for this generic signature, that replaces each generic parameter with itself.

`SubstitutionMap` *class*

A substitution map. Just like `Type` and `GenericSignature`, substitution maps are immutable and uniqued, and the representation of a `SubstitutionMap` fits in a single pointer, so they are cheap to pass by value.

The default `SubstitutionMap()` constructor constructs an empty substitution map. The implicit `bool` conversion tests for a non-empty substitution map. We defer our discussion of the entry points that build non-empty substitution maps to Section 7.7, since this involves passing in an array of conformances as well, and we haven't introduced conformances yet.

Accessor methods:

- `empty()` answers if this is the empty substitution map; this is the logical negation of the `bool` implicit conversion.
- `getGenericSignature()` returns the substitution map's input generic signature.
- `getReplacementTypes()` returns an array of `Type`.
- `hasAnySubstitutableParams()` answers if this substitution map's input generic signature contains at least one generic parameter that is not fixed to a concrete type (see the `GenericSignatureImpl::areAllParamsConcrete()` method from Section 5.7).

Recursive properties of replacement types:

- `hasPrimaryArchetypes()` answers if any of the replacement types contain primary archetypes.
- `hasOpenedExistential()` answers if any of the replacement types contain an existential archetype.
- `hasDynamicSelf()` answers if any of the replacement types contain the dynamic Self type.

A substitution map is *canonical* if all replacement types are canonical types and all conformances are canonical conformances. A substitution map is canonicalized by constructing a new substitution map from the original substitution map's canonicalized replacement types and conformances.

- `isCanonical()` answers if the replacement types and conformances stored in this substitution map are canonical.
- `getCanonical()` constructs a new substitution map by canonicalizing the replacement types and conformances of this substitution map.

Substitution map composition (Section 6.3):

- `subst()` returns the composition of this substitution map on the left with the given substitution map on the right.

As with types, substitution maps have two levels of equality; two substitution maps are equal pointers if their replacement types and conformances are equal pointers. Two substitution maps are canonically equal if their canonical substitution maps are equal pointers; or equivalently, if their replacement types and conformances are canonically equal. The `operator==` overloads implements pointer equality. Canonical equality can be tested by first canonicalizing both sides:

```
if (subMap1.getCanonical() == subMap2.getCanonical())
  ...;
```

However, unlike types, checking two substitution maps for equality is rare. The fact that they are uniquely-allocated is more of a performance optimization than anything else.

Subclassing

`ClassDecl`	*class*

See also Section 4.6.

- `getSuperclass()` returns this class declaration's superclass type.

`Type`	*class*

- `getSuperclass()` returns this class type's superclass type. This is Algorithm 6B.
- `getSuperclassDecl()` returns this class type's superclass declaration. This is Algorithm 6C.

SIL Types

Key source files:

- `include/swift/AST/Types.h`
- `include/swift/SIL/SILType.h`
- `lib/SIL/IR/SILType.cpp`
- `lib/SIL/IR/SILFunctionType.cpp`

A lowered type is represented as a `CanType`, just like a formal type. Be forewarned that the distinction between the two is not enforced statically, and care must be taken not to get them mixed up.

`TypeBase` *class*

See also Section 3.4. Two methods distinguish formal types from lowered types:

- `isLegalFormalType()` checks if this is a legal formal type.
- `isLegalSILType()` checks if this is a legal lowered type.

`SILValueCategory` *enum class*

In the implementation, whether a type is loadable or address-only is referred to as the lowered type's "category."

- `SILValueCategory::Object` is the category of loadable types.
- `SILValueCategory::Address` is the category of address-only types.

`SILType` *class*

Combines a lowered type together with its category. SIL types can be taken apart:

- `getASTType()` returns the SIL type's lowered type as a `CanType`.
- `getCategory()` returns the SIL type's category.

New SIL types can be constructed from a lowered type and a category as follows:

- `SILType::getPrimitiveObjectType()` is a static factory method that returns a new object SIL type for the given lowered type. The lowered type must be loadable.
- `SILType::getPrimitiveAddressType()` is a static factory method that returns a new address SIL type for the given lowered type.

One can also call the `SILType` constructor with a lowered type and a `SILValueCategory`.

Note that all of the above requires already having a lowered type on hand; attempting to construct a SIL type from a formal type that is not a lowered type will result in a run time assertion. To construct a SIL type from a formal type, the formal type must be lowered first, as discussed below.

SIL types support down-casting to the various `TypeBase` subclasses that represent different kinds of lowered types. The `SILType` class declares three template methods, `is<>()`, `getAs<>`, and `castTo<>`. These are equivalent to calling `getASTType()` and then

passing the result to the top-level `isa<>`, `dyn_cast<>`, and `cast<>` template functions discussed in Section 3.4.

The printed representation of a SIL type is `$type` for a loadable type and `$*type` for an address-only type, where `type` is the printed representation of its lowered type. This printed representation shows up in `-emit-silgen` output, for example.

`SILFunctionType` *class*

A `TypeBase` subclass representing a SIL function type.

- `getParameters()` returns an `ArrayRef<SILParameterInfo>` with the function's lowered parameters.
- `getResults()` returns an `ArrayRef<SILResultInfo>` with the function's lowered results.
- `getInvocationGenericSignature()` returns the function's generic signature.

In reality, generic SIL function types have a more complex representation than what we've described; see [91].

SIL Type Lowering

Key source files:

- `include/swift/SIL/AbstractionPattern.h`
- `include/swift/SIL/TypeLowering.h`
- `lib/SIL/IR/AbstractionPattern.cpp`
- `lib/SIL/IR/TypeLowering.cpp`

`AbstractionPattern` *class*

An abstraction pattern.

- `getKind()` returns the abstraction pattern's kind. See the source code for an explanation of the possible kinds.
- `isTypeParameterOrOpaqueArchetype()` returns if this is an opaque abstraction pattern.
- `getType()` returns the abstraction pattern's formal type.

`TypeConverter` *class*

A singleton object responsible for SIL type lowering.

- `getLoweredType()` takes a `CanType` representing a formal type, and returns a `SILType` which encodes the lowered type together with its category.
- `getTypeLowering()` takes a `CanType` representing a formal type, and returns a `TypeLowering` object, which encodes the lowered type together with additional recursive properties computed by type lowering, such as whether the type is trivial.

Re-abstraction Thunks

Key source files:

- `lib/SILGen/SILGenPoly.cpp`
- `lib/SILGen/SILGenThunk.cpp`

`SILGenFunction` *class*

A class containing various entry points used by SILGen to emit the SIL functions. The following two entry points emit re-abstraction thunks:

- `emitOrigToSubstValue()` re-abstracts a value from a given abstraction pattern to the "natural" abstraction pattern for its formal type. If the value has function type, this emits an original-to-substituted thunk.
- `emitSubstToOrigValue()` re-abstracts a value from the "natural" abstraction pattern for its formal type to a given abstraction pattern. If the value has function type, this emits an substituted-to-original thunk.

SIL Type Substitution

Key source files:

- `lib/SIL/IR/SILTypeSubstitution.cpp`

`SILType` *class*

SIL type substitution:

- `subst()` applies a substitution map to a SIL type.

`SILTypeSubstituter` *class*

A `CanTypeVisitor` implementation used to implement the above.

7. Conformances

ONFORMANCES RELATE types with the protocols they conform to. To be precise, a *conformance* describes how its conforming type *witnesses* the requirements of a protocol. We will start by looking at the representation of conformances, and then discuss conformance lookup. Finally, conformances also play an important role in type substitution, so we will fill the gaps in the previous chapter's discussion of substitution maps, by showing how we use conformances to substitute dependent member types.

Conformances come in three distinct varieties:

1. A **concrete conformance** records that a nominal type conforms to a protocol, in which case we know the original declaration of the conformance, and thus all of the witnesses.
2. An **abstract conformance** records that a type parameter or archetype satisfies a conformance requirement, but we don't know "how" (Section 7.4).
3. The **invalid conformance** records that some type does not conform.

Concrete conformances break down further into four kinds:

1. A **normal conformance** represents the actual declaration of a conformance on a nominal type, or extension.
2. A **specialized conformance** represents a normal conformance with a substitution map applied (Section 7.2).
3. An **inherited conformance** represents a conformance inherited by a subclass from its superclass.
4. A **self conformance** represents the conformance of an existential type to its own protocol. This is possible only in special circumstances (Section 14.3).

Normal conformances. In the language, a normal conformance is declared by stating the name of a protocol in the inheritance clause of a nominal type or extension:

```
struct Horse: Animal {...}

struct Cow {...}
extension Cow: Animal {...}
```

When $\texttt{X}_d$ is the declared interface type of a nominal type declaration d that conforms to `P`, we denote the normal conformance by [$\texttt{X}_d$`: P`]. On the previous page, we declared two normal conformances, [`Horse: Animal`] and [`Cow: Animal`].

The constituent parts of a normal conformance are the following:

- The **conforming type**, which is the declared interface type of the nominal type declaration.
- The **protocol declaration** being conformed to.
- The **conforming context** where this normal conformance is declared, either the nominal type declaration itself, or one of its extensions. This is a declaration context (Chapter 4).
- The **generic signature** of the conforming context. If the conformance context is a constrained extension, this generic signature has additional requirements not present in the nominal type's generic signature; we say that we have a *conditional conformance* (Section 10.4).
- A **type witness** for each associated type declared by the protocol. This is an interface type written using the generic signature of the conformance (Section 7.3).
- An **associated conformance** for each associated conformance requirement of the protocol. This is another conformance with the appropriate subject type. Among other things, given a conformance to a derived protocol, this allows us to recover the conformance of the same type to a base protocol (Section 7.5).
- A **value witness** for each value requirement of the protocol. This is a reference to a value declaration, together with a substitution map. The value declaration is either a member of the conforming nominal type, an extension, a superclass, or a protocol extension.

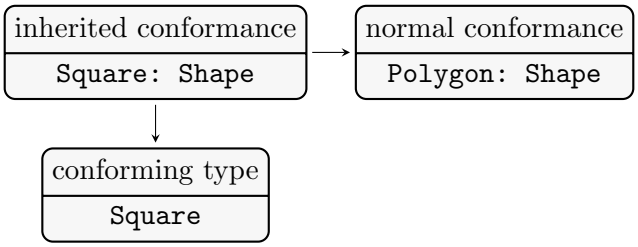
Every nominal type and extension declaration has a list of *local conformances*, which are normal conformances stated in the declaration's inheritance clause. In our example, [`Horse: Animal`] is a local conformance of the nominal type declaration `Horse`, while [`Cow: Animal`] is a local conformance of the extension of `Cow`. In SILGen, we generate a witness table for each local conformance of each declaration in each primary file.

In addition, every nominal type declaration also has a *conformance lookup table*. This is how global conformance lookup will *find* a conformance, in the next section. In this table, we collect the local conformances from the nominal type itself, and all of its extensions. In Swift, conformances are inherited, so the conformance lookup table for a class has an additional behavior; it also collects all conformances from its superclass, and its superclass, and so on. To ensure that such a conformance inherited from a superclass has the correct conforming type, we introduce a little bit of bookkeeping.

Inherited conformances. In the below, `Square` is a subclass of `Polygon`, so it inherits the normal conformance [`Polygon: Shape`]:

```
protocol Shape {}
class Polygon: Shape {}
class Square: Polygon {}
```

When building the conformance lookup table for a subclass, we wrap each superclass conformance in a new *inherited conformance* structure, which consists of the subclass type, together with the superclass conformance:



We denote this inherited conformance by [`Square: Shape`]. It behaves identically to the superclass conformance [`Polygon: Shape`], except if ask for its conforming type, we get back `Square` instead of `Polygon`. In Example 7.2, we will see that more complex behaviors can manifest when the superclass declaration has generic parameters.

7.1. Conformance Lookup

The principal mechanism to find the conformance of a given type to a protocol is called *global conformance lookup*. We implement it with a case analysis.

Algorithm 7A (Global conformance lookup)**.** Takes a type `X` and a protocol `P` as input. Outputs a conformance [`X: P`] if one exists, the invalid conformance otherwise.

- If `X` is the declared interface type of a nominal type declaration d: look for a normal or inherited conformance in the conformance lookup table of d, and return it if present, otherwise return the invalid conformance.
- If `X` is some other specialized type: recursively look up the conformance of its declared interface type $\texttt{X}_d$ to `P` first, and then apply the context substitution map of `X` to this conformance [$\texttt{X}_d$`: P`] (Section 7.2).
- If `X` is a type parameter: form the abstract conformance for `X` and `P` (Section 7.4).
- If `X` is any other kind of type: return the invalid conformance.

We can use global conformance lookup to answer the "yes/no" question of whether a type actually conforms, by performing a lookup and checking if the result is anything but an invalid conformance. We can also pick any valid conformance [`X: P`] apart, and get back the conforming type `X` and the protocol `P`. We relate this destructuring to global conformance lookup in the form of a commutative diagram:

$$\texttt{X} \underset{\text{get conforming type}}{\overset{\text{look up conformance}}{\rightleftarrows}} [\texttt{X: P}]$$

If we start with a type `X`, look up the conformance to `P`, and get the conforming type out of this conformance, we end up with a type canonically equivalent to `X`. (The two types might be distinct pointers, because conformance lookup does not always preserve type sugar.)

The opposite direction is more subtle. If we start with a conformance [`X: P`], get the conforming type `X`, and look up the conformance of `X` to `P`, we would like to find the *same* conformance. We say that global conformance lookup is *coherent* if this property holds. We follow the simplest approach, and only promise coherence if there is *at most* one conformance for every possible pairing of type with protocol.

Coherence. When a conformance is stated on an extension, we allow the extended type, the conformed protocol, and the extension itself to be in different modules. There are three possibilities. First, we can conform our own type to a protocol from another module:

```
struct MyType {...}
extension MyType: Hashable {...}
```

We can also conform a type from another module to our own protocol:

```
protocol MyProtocol {...}
extension Int: MyProtocol {...}
```

Finally, both the conforming type and conformed protocol can be declared in modules separate from ours. This is called a *retroactive conformance*, and the compiler warns about this as of Swift 6 [92]:

```
extension String.UTF8View: Hashable {...}
// warning: extension declares a conformance of imported type
// 'UTF8View' to imported protocol 'Hashable'; this will not
// behave correctly if the owners of 'Swift' introduce this
// conformance in the future
```

To bypass the warning, the user must indicate that this is really what they want, by annotating the conformance with the `@retroactive` attribute:

```
extension String.UTF8View: @retroactive Hashable {...}
```

When used incorrectly, retroactive conformances can create a situation where we have two *overlapping* conformances for the same conforming type and protocol. This can lead to inconsistent behavior.

Suppose we declare a concrete `MyKey` type in a common module, and then two other modules both import `MyKey` and declare a retroactive conformance of `MyKey` to `Hashable`. If a downstream module then imports all three modules, both normal conformances of `MyKey` to `Hashable` will be visible. The compiler will pick the first one it finds, but this may not be consistent between invocations.

We run into the same problem at run time, with library evolution. Suppose a vendor publishes `MyKey` in a shared library, and a third party defines a retroactive conformance of `MyKey` to `Hashable` in their binary. If the vendor then updates their library to include a conformance of `MyKey` to `Hashable` and the client runs their binary without recompiling, both witness tables will be visible to the Swift runtime. A dynamic cast of `MyKey` to `Hashable` may end up using the witness table for either conformance.

Avoiding the use of `@retroactive` mostly eliminates this possibility, but does not foreclose it entirely, because in combination with library evolution, subclassing can also introduce overlapping conformances. Suppose that a vendor publishes a shared library declaring an open class and a protocol:

```
public protocol Crop {}

open class Hay {}
```

A client can declare their own subclass of `Hay`, because it is `open`. They can also conform their subclass of `Hay` to `Crop` without stating the `@retroactive` attribute on the conformance, because the conforming type is the subclass in the client module:

```
import Farm

class Alfalfa: Hay {}
extension Alfalfa: Crop {}
```

Now, if the vendor subsequently adds a conformance of `Hay` to `Crop` in a future version of the library and the client runs their binary without recompiling, the client binary will now observe two overlapping conformances of `Alfalfa` to `Crop` at run time: the conformance inherited from `Hay`, and the local conformance of `Alfalfa` to `Crop`.

Future directions. Extending the language implementation to guarantee correctness in the presence of overlapping conformances is a major undertaking, as it would require several architectural changes in the compiler and runtime.

The global conformance lookup operation would need a disambiguation rule, perhaps by taking additional source location information into account. Parts of the compiler that rely on coherence would need to be redesigned to perform each lookup exactly once and save the result for later, instead of freely performing another lookup and assuming the same conformance will be found.

For example, when we resolve a generic nominal type written in source, we perform global conformance lookup to ensure that the generic arguments satisfy the conformance requirements of the nominal type. If we later need the context substitution map of this generic nominal type, we currently call global conformance lookup again, to populate the substitution map's conformances. We could avoid the second lookup by changing the representation of a generic nominal type to store a substitution map, instead of just the generic arguments.

Dynamic casts at runtime pose similar challenges. One valid design is to exclude retroactive conformances from dynamic cast behavior entirely. Another would be to introduce a new runtime entry point for dynamic casting that takes the reified lexical environment into account, allowing a shared library to use its own private conformances without changing the behavior of binaries that link against it.

7.2. Conformance Substitution

In this section, we will see that we can apply substitution maps to normal, specialized, and inherited conformances. In Section 7.4, we will extend this to abstract conformances. This operation is called *conformance substitution*, by analogy with type substitution.

To begin, we will incorporate global conformance lookup into the type substitution algebra with a new form of the $\otimes$ operator. Given a type $\texttt{X}$ and a protocol $\texttt{P}$, we denote the result of global conformance lookup by $\langle\texttt{P}\rangle \otimes \texttt{X}$. Recall that if d is a nominal type declaration conforming to $\texttt{P}$, and $\texttt{X}_d$ is the declared interface type of d, we defined Algorithm 7A to output the normal conformance of $\texttt{X}_d$ to $\texttt{P}$:

$$\langle\texttt{P}\rangle \otimes \texttt{X}_d := [\texttt{X}_d\texttt{:}\ \texttt{P}]$$

Normal conformances. Now, we suppose that $\texttt{X}$ is a generic nominal type with declaration d, but with arbitrary generic arguments, so $\texttt{X}$ is not necessarily equal to $\texttt{X}_d$. We will use the type substitution algebra to figure out $\langle\texttt{P}\rangle \otimes \texttt{X}$. From Section 6.1, we know we can split up $\texttt{X}$ into the declared interface type $\texttt{X}_d$, and a substitution map Σ, called the context substitution map of $\texttt{X}$. Thus, we have:

$$\langle\texttt{P}\rangle \otimes \texttt{X} = \langle\texttt{P}\rangle \otimes (\texttt{X}_d \otimes \Sigma)$$

In order to reduce the right-hand side further, we will define it in a way that makes $\otimes$ into an associative operation. This is a reasonable assumption, because we already saw that $\otimes$ is associative when operating on types and substitution maps. Indeed, recall that if $\mathtt{X}$ is a type, and Σ_1, Σ_2, Σ_3 are substitution maps, we were able to show that order of operations does not matter in the expressions below:

$$\mathtt{X} \otimes (\Sigma_1 \otimes \Sigma_2) = (\mathtt{X} \otimes \Sigma_1) \otimes \Sigma_2$$
$$\Sigma_1 \otimes (\Sigma_2 \otimes \Sigma_3) = (\Sigma_1 \otimes \Sigma_2) \otimes \Sigma_3$$

To get the corresponding identity for global conformance lookup, we *define* that:

$$\langle \mathtt{P} \rangle \otimes (\mathtt{X}_d \otimes \Sigma) := (\langle \mathtt{P} \rangle \otimes \mathtt{X}_d) \otimes \Sigma$$

In the above, we know that $\langle \mathtt{P} \rangle \otimes \mathtt{X}_d = [\mathtt{X}_d\mathtt{:}\ \mathtt{P}]$ is the normal conformance, so now we see that global conformance lookup must apply the context substitution map of $\mathtt{X}$ to the normal conformance $[\mathtt{X}_d\mathtt{:}\ \mathtt{P}]$, which is exactly what Algorithm 7A does:

$$\langle \mathtt{P} \rangle \otimes \mathtt{X} = [\mathtt{X}_d\mathtt{:}\ \mathtt{P}] \otimes \Sigma$$

When we apply a non-identity substitution map Σ to the normal conformance $[\mathtt{X}_d\mathtt{:}\ \mathtt{P}]$, we form a *specialized conformance* from $[\mathtt{X}_d\mathtt{:}\ \mathtt{P}]$ and Σ. We denote it by $[\mathtt{X}\mathtt{:}\ \mathtt{P}]$, where $\mathtt{X} = \mathtt{X}_d \otimes \Sigma$. We say that Σ is the *conformance substitution map* of $[\mathtt{X}\mathtt{:}\ \mathtt{P}]$.

In the case where Σ is the identity substitution map for the generic signature of the conforming context, we don't need to form a specialized conformance. Instead, we just return the original normal conformance; that is, $[\mathtt{X}_d\mathtt{:}\ \mathtt{P}] \otimes 1_G := [\mathtt{X}_d\mathtt{:}\ \mathtt{P}]$.

Output generic signatures. Before we proceed, we introduce a new concept. We associate an *output generic signature* with each conformance:

- The output generic signature of a normal conformance is the generic signature of the conformance context (the nominal itself, or an extension).
- The output generic signature of a specialized conformance is the output generic signature of its conformance substitution map.
- The output generic signature of an inherited conformance is the output generic signature of its underlying superclass conformance.

The set of conformances with output generic signature G is denoted by $\textsc{Conf}(G)$. This is related to the notation of Chapter 6 as follows. The above definition implies that if $[\mathtt{X}\mathtt{:}\ \mathtt{P}] \in \textsc{Conf}(G)$, then $\mathtt{X} \in \textsc{Type}(G)$. Conversely, if some $\mathtt{X} \in \textsc{Type}(G)$ conforms to $\mathtt{P}$, then $\langle \mathtt{P} \rangle \otimes \mathtt{X} \in \textsc{Conf}(G)$, by definition of Algorithm 7A.

We've only seen substitution maps applied to normal conformances so far, but shortly we will generalize this and define $[\mathtt{X}\mathtt{:}\ \mathtt{P}] \otimes \Sigma \in \textsc{Conf}(H)$ for any concrete conformance $[\mathtt{X}\mathtt{:}\ \mathtt{P}] \in \textsc{Conf}(G)$ and substitution map $\Sigma \in \textsc{Sub}(G, H)$.

Example 7.1. The standard library declares a conformance of `Array` to `Sequence`. If we look up the conformance of `Array<τ_0_0>` to `Sequence`, we get this normal conformance back, because `Array<τ_0_0>` is the declared interface type of `Array`:

$$\langle \texttt{Sequence}] \otimes \texttt{Array<}\tau\texttt{_0_0>} = [\texttt{Array<}\tau\texttt{_0_0>: Sequence}]$$

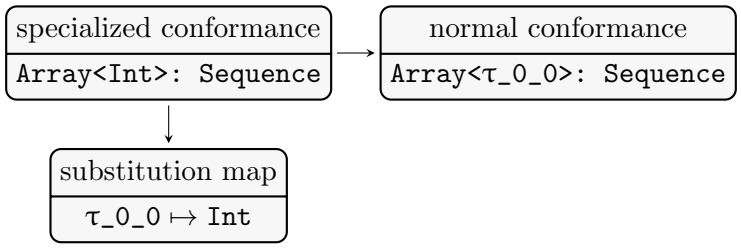
Next, we consider how `Array<Int>` conforms to `Sequence`. We observe that:

$$\texttt{Array<Int>} = \texttt{Array<}\tau\texttt{_0_0>} \otimes \{\tau\texttt{_0_0} \mapsto \texttt{Int}\}$$

Global conformance lookup applies this substitution map to the normal conformance:

$$\begin{aligned}
&\langle \texttt{Sequence}] \otimes \texttt{Array<Int>} \\
&\quad = \langle \texttt{Sequence}] \otimes \big(\texttt{Array<}\tau\texttt{_0_0>} \otimes \{\tau\texttt{_0_0} \mapsto \texttt{Int}\}\big) \\
&\quad = \big(\langle \texttt{Sequence}] \otimes \texttt{Array<}\tau\texttt{_0_0>}\big) \otimes \{\tau\texttt{_0_0} \mapsto \texttt{Int}\} \\
&\quad = [\texttt{Array<}\tau\texttt{_0_0>: Sequence}] \otimes \{\tau\texttt{_0_0} \mapsto \texttt{Int}\} \\
&\quad = [\texttt{Array<Int>: Sequence}]
\end{aligned}$$

We denote this specialized conformance by [`Array<Int>: Sequence`], but the in-memory structure looks like this:



Specialized conformances. Now, suppose we're given a specialized conformance $[\texttt{X}_d\texttt{: P}] \otimes \Sigma_1 \in \textsc{Conf}(H)$ where $\Sigma_1 \in \textsc{Sub}(G, H)$, and we wish to apply some other substitution map $\Sigma_2 \in \textsc{Sub}(H, I)$ to this specialized conformance. To implement this, we form a new specialized conformance from $[\texttt{X}_d\texttt{: P}]$ and $\Sigma_1 \otimes \Sigma_2 \in \textsc{Sub}(G, I)$, so that $[\texttt{X}_d\texttt{: P}] \otimes (\Sigma_1 \otimes \Sigma_2) \in \textsc{Conf}(I)$. This relates conformance substitution with substitution map composition, and continues with the trend of $\otimes$ being an associative operation:

$$([\texttt{X}_d\texttt{: P}] \otimes \Sigma_1) \otimes \Sigma_2 := [\texttt{X}_d\texttt{: P}] \otimes (\Sigma_1 \otimes \Sigma_2)$$

Inherited conformances. Finally, we can apply a substitution map to an inherited conformance. Suppose that $[\texttt{C: P}]$ is an inherited conformance with conforming type $\texttt{C} \in \textsc{Type}(G)$ and superclass conformance $[\texttt{C}'\texttt{: P}] \in \textsc{Conf}(G)$, where $\texttt{C}'$ is a superclass type of `C`. If $\Sigma \in \textsc{Sub}(G, H)$ is a substitution map, then $[\texttt{C: P}] \otimes \Sigma \in \textsc{Conf}(H)$ is the inherited conformance built from $\texttt{C} \otimes \Sigma$ and $[\texttt{C}'\texttt{: P}] \otimes \Sigma$.

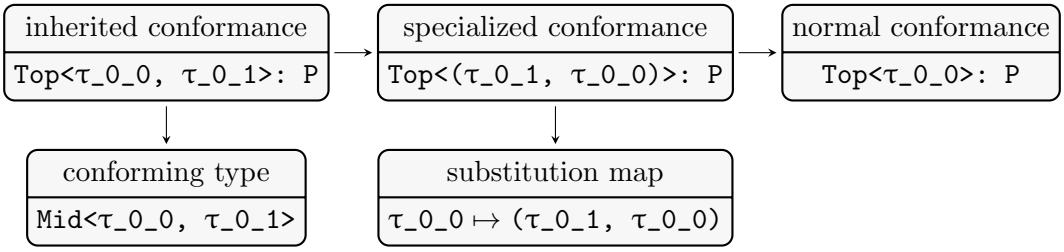
Example 7.2. Consider `Mid` below, which inherits the conformance to `P` from `Top`:

```
protocol P {}
class Top<T>: P {}
class Mid<X, Y>: Top<(Y, X)> {}
```

The superclass type of `Mid` is `Top<(τ_0_1, τ_0_0)>`, with context substitution map:

$$\Sigma_1 := \{\texttt{τ_0_0} \mapsto \texttt{(τ_0_1, τ_0_0)}\}$$

When we build the conformance lookup table of `Mid`, we look up the conformance of the superclass type `Top<(τ_0_1, τ_0_0)>` to `P`, which gives us a specialized conformance with the above substitution map. We then wrap this specialized conformance in an inherited conformance, to give it the correct conforming type `Mid<τ_0_0, τ_0_1>`. We record the resulting structure in the table:



Suppose we ask global conformance lookup for the conformance of `Mid<Int, Bool>` to `P`. We need the context substitution map of `Mid<Int, Bool>`:

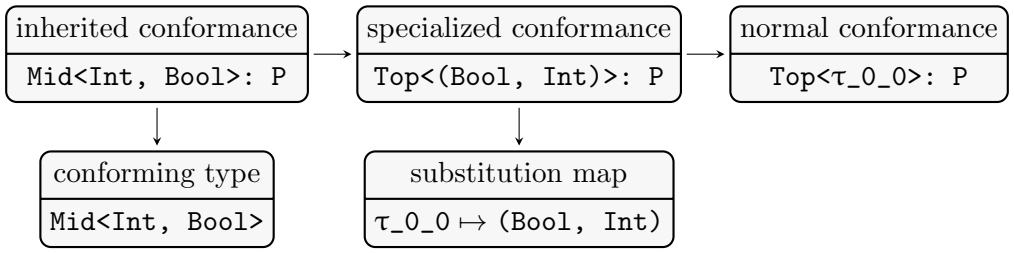
$$\Sigma_2 := \{\texttt{τ_0_0} \mapsto \texttt{Int}, \texttt{τ_0_1} \mapsto \texttt{Bool}\}$$

We first find the inherited conformance [`Top<τ_0_0, τ_0_1>: P`] from the conformance lookup table of `Mid`, and then we apply Σ_2 to this inherited conformance, which will in turn apply Σ_2 to its subclass type `Mid<τ_0_0, τ_0_1>` and specialized conformance [`Top<(τ_0_1, τ_0_0)>: P`]. For the latter, we note $\Sigma_1 \otimes \Sigma_2 = \{\texttt{τ_0_0} \mapsto \texttt{(Bool, Int)}\}$:

$$\texttt{Mid<τ_0_0, τ_0_1>} \otimes \Sigma_2 = \texttt{Mid<Int, Bool>}$$

$$[\texttt{Top<(τ_0_1, τ_0_0)>: P}] \otimes \Sigma_2 = [\texttt{Top<(Bool, Int)>: P}]$$

Our final result is the rather intricate inherited conformance [`Mid<Int, Bool>: P`]:



Summary. Let's recap the various forms of the $\otimes$ operator we've worked with so far. We can substitute types (Chapter 6):

$$\textsc{Type}(G) \otimes \textsc{Sub}(G, H) \longrightarrow \textsc{Type}(H)$$

We can substitute conformances (the present section):

$$\textsc{Conf}(G) \otimes \textsc{Sub}(G, H) \longrightarrow \textsc{Conf}(H)$$

We can compose substitution maps (Section 6.3):

$$\textsc{Sub}(G, H) \otimes \textsc{Sub}(H, I) \longrightarrow \textsc{Sub}(G, I)$$

We can look up conformances, where $\textsc{Proto}$ is the set of all protocols (Section 7.1):

$$\textsc{Proto} \otimes \textsc{Type}(G) \longrightarrow \textsc{Conf}(G)$$

Furthermore, we saw that the $\otimes$ operator is associative, so parentheses are never required to disambiguate the result of an expression.

7.3. Type Witnesses

In order to conform to a protocol with associated types, a nominal type declaration must declare a *type witness* for each associated type. There are four ways to declare a type witness in the source language:

1. With a **member type declaration**—that is, a nested nominal type or type alias declaration—having the same name as the associated type. This member type might be a child of the conforming nominal type, one of its extensions, or if the conforming type is a class, one of its superclasses.
2. With **associated type inference**, which deduces type witnesses by considering the candidate value witnesses for each value requirement in the protocol.
3. With a **generic parameter** having the same name as the associated type.
4. With a **default type witness** on the associated type declaration, if one is present. This is the fallback if all else fails.

Example 7.3. We will look at four nominal type declarations that conform to the protocol below. Each one will demonstrate one of the four cases above:

```
protocol Pet {
  associatedtype Toy = Int
  func play(_: Toy)
}
```

Before we begin, we add a default witness for `play()` that can be used with an arbitrary `Toy`, so conforming types are not required to provide their own:

```
extension Pet {
  func play(_: Toy) {}
}
```

Our first type, `Chicken`, explicitly declares a member type named `Toy`, while relying on the default witness for `play()`. This is Case 1 above:

```
struct Chicken: Pet {
  struct Toy {}
}
```

The `Cat` on the other hand does not explicitly witness `Toy`, but unlike `Chicken`, it has its own implementation of `play()` taking a `String`. We rely on associated type inference to figure out that the type witness must be `String`. This is Case 2:

```
struct Cat: Pet {
  // synthesized: typealias Toy = String
  func play(_: String) {}
}
```

Associated type inference synthesizes a type alias member for each inferred associated type, so we can refer to `Cat.Toy` elsewhere in the program, as if we explicitly declared the type alias `Toy` as a member of `Cat`. Now, consider `Dog`, which witnesses `Pet`'s associated type with a generic parameter named `Toy`. This is Case 3:

```
struct Dog<Toy>: Pet {
  // synthesized: typealias Toy = Toy
}
```

When we check the conformance [`Dog: Pet`], we synthesize a type alias member named `Toy`, whose underlying type is `Dog`'s generic parameter, also named `Toy`. While generic parameter declarations are not themselves visible as *members*, in this case we have a type alias that just happens to have the same name, so in fact the type representation `Dog<Ball>.Toy` resolves to `Ball`:

```
struct Ball {}

// same as play(_: Ball):
func play(_: Dog<Ball>.Toy)
```

(However, if `Dog` did not state a conformance to `Pet`, there would be no such type alias, and type resolution would diagnose an error while resolving `Dog<Ball>.Toy`. The interaction between type resolution and associated type inference is discussed further in Section 9.2.)

Last but not least, `Horse` does not declare a member type named `Toy`, nor does it implement `play()`. We use the default type witness for `Toy`, which was declared to be `Int` inside `Pet`. This is Case 4:

```
struct Horse: Pet {
  // synthesized: typealias Toy = Int
}
```

Normal conformances. Every normal conformance contains a table that maps the protocol's associated type declarations to the type witnesses of the conformance. This table is lazily populated by the **type witness request**, which attempts to resolve a single type witness using qualified lookup, which handles Case 1.

If no such member type declaration exists, we then evaluate the **type witnesses request**, which attempts to simultaneously resolve all type witnesses in the normal conformance via associated type inference. This code path implements Case 2, but also Case 3 and Case 4, which users might not think of as being part of associated type inference. We will discuss associated type inference in Section 7.6.

When a conformance is declared in a secondary file of a frontend job, we resolve its type witnesses only if we need to while type checking something else. On the other hand, when a conformance is declared in a primary file, we also run the *conformance checker* as part of the **type-check primary file request**. The conformance checker resolves all type and value witnesses and performs various additional checks. If the conformance checker does not emit any diagnostics, we know the conformance provides a complete set of type and value witnesses, and we can move on to code generation.

Projection. We incorporate type witnesses into the type substitution algebra with a new form of the $\otimes$ operator. Suppose $[\texttt{X}_d\texttt{: P}]$ is a normal conformance, and `P` declares an associated type `A`. We introduce the notation $\langle\texttt{P}|\texttt{A}$ for the associated type declaration, and then we denote the type witness for `A` with the following expression:

$$\langle\texttt{P}|\texttt{A} \otimes [\texttt{X}_d\texttt{: P}]$$

Example 7.4. We can summarize Example 7.3 using our new notation:

$$\langle\texttt{Pet}|\texttt{Toy} \otimes [\texttt{Chicken: Pet}] = \texttt{Chicken.Toy}$$
$$\langle\texttt{Pet}|\texttt{Toy} \otimes [\texttt{Cat: Pet}] = \texttt{String}$$
$$\langle\texttt{Pet}|\texttt{Toy} \otimes [\texttt{Dog<}\tau\texttt{_0_0>: Pet}] = \tau\texttt{_0_0}$$
$$\langle\texttt{Pet}|\texttt{Toy} \otimes [\texttt{Horse: Pet}] = \texttt{Int}$$

Specialized conformances. The type witnesses of a specialized conformance become completely determined once we require that $\otimes$ is an associative operation. Suppose that $\texttt{X} = \texttt{X}_d \otimes \Sigma$, and $[\texttt{X: P}] = [\texttt{X}_d\texttt{: P}] \otimes \Sigma$ is a specialized conformance. To project the type witness for some `A` from the specialized conformance, we apply the conformance substitution map Σ to the type witness for `A` in the underlying normal conformance:

$$\langle \texttt{P}|\texttt{A} \otimes \big([\texttt{X}_d\texttt{: P}] \otimes \Sigma\big) := \big(\langle \texttt{P}|\texttt{A} \otimes [\texttt{X}_d\texttt{: P}]\big) \otimes \Sigma$$

Example 7.5. The standard library's `[Array<τ_0_0>: Sequence]` normal conformance witnesses the `Element` and `Iterator` associated types of `Sequence` as follows:

$$\begin{aligned}
&\langle \texttt{Sequence}|\texttt{Element} \otimes [\texttt{Array<τ_0_0>: Sequence}] \\
&\quad = \texttt{τ_0_0} \\
&\langle \texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Array<τ_0_0>: Sequence}] \\
&\quad = \texttt{IndexingIterator<Array<τ_0_0>>}
\end{aligned}$$

Recall the specialized conformance `[Array<Int>: Sequence]` from Example 7.1. To get its type witnesses, we apply the conformance substitution map $\{\texttt{τ_0_0} \mapsto \texttt{Int}\}$:

$$\begin{aligned}
&\langle \texttt{Sequence}|\texttt{Element} \otimes [\texttt{Array<Int>: Sequence}] \\
&\quad = \texttt{τ_0_0} \otimes \{\texttt{τ_0_0} \mapsto \texttt{Int}\} \\
&\quad = \texttt{Int} \\
&\langle \texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Array<Int>: Sequence}] \\
&\quad = \texttt{IndexingIterator<Array<τ_0_0>>} \otimes \{\texttt{τ_0_0} \mapsto \texttt{Int}\} \\
&\quad = \texttt{IndexingIterator<Array<Int>>}
\end{aligned}$$

An important invariant is that any type parameters appearing in the type witness of a normal or specialized conformance are valid in the output generic signature of the conformance. In other words, if $[\texttt{X: P}] \in \textsc{Conf}(G)$ and `A` is an associated type of `P`, then $\langle \texttt{P}|\texttt{A} \otimes [\texttt{X: P}] \in \textsc{Type}(G)$. (When we get to abstract conformances, we will see the property holds there as well.) To summarize all of the above, if we take $\textsc{AssocType}_\texttt{P}$ to mean the set of all associated type declarations of a fixed protocol `P`, and $\textsc{Conf}_\texttt{P}(G)$ as the subset of $\textsc{Conf}(G)$ containing only conformances to this `P`, then for each protocol `P`, type witness projection gives us this additional form of the $\otimes$ operation:

$$\textsc{AssocType}_\texttt{P} \otimes \textsc{Conf}_\texttt{P}(G) \longrightarrow \textsc{Type}(G)$$

Finally, we consider the relationship between global conformance lookup and type witness projection. If $\texttt{X} = \texttt{X}_d \otimes \Sigma$ conforms to `P`, and `P` declares an associated type `A`, one way to resolve the type representation "`X.A`" is via global conformance lookup followed by type witness projection:

$$\langle \texttt{P}|\texttt{A} \otimes \langle \texttt{P}] \otimes \texttt{X}_d \otimes \Sigma$$

Of the four possible ways to parenthesize this expression, there are three that correspond to valid combinations of $\otimes$:

$$\Big(\big(\langle \texttt{P}|\texttt{A} \quad \otimes \quad \big(\langle \texttt{P}] \quad \otimes \quad \texttt{X}_d\big)\big) \otimes \quad \Sigma\Big) \tag{1}$$

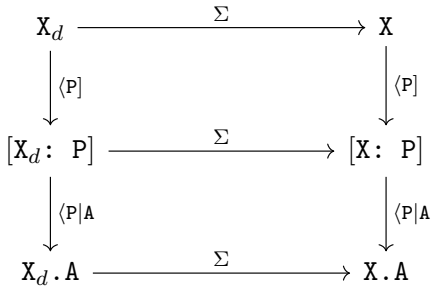
$$\Big(\ \langle \texttt{P}|\texttt{A} \quad \otimes \big(\big(\langle \texttt{P}] \quad \otimes \quad \texttt{X}_d\big) \quad \otimes \quad \Sigma\big)\Big) \tag{2}$$

$$\Big(\ \langle \texttt{P}|\texttt{A} \quad \otimes \quad \big(\langle \texttt{P}] \otimes \big(\texttt{X}_d \quad \otimes \quad \Sigma\big)\big)\Big) \tag{3}$$

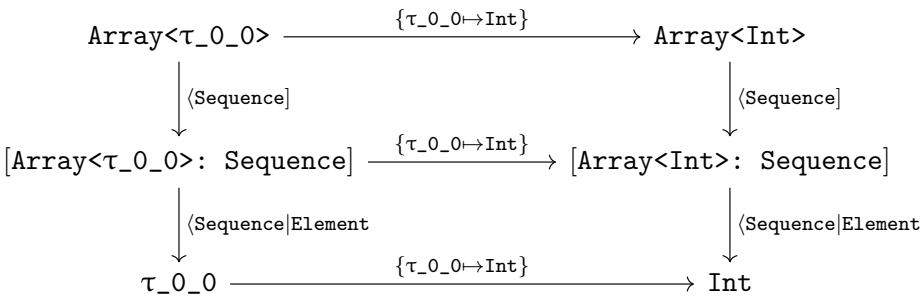
Let's say that $\texttt{X}_d\texttt{.A}$ denotes the type witness for `A` in $[\texttt{X}_d\texttt{: P}]$, and `X.A` denotes the type witness of `A` in `[X: P]`. Then, all three combinations above must output `X.A`, because we defined $\otimes$ to be associative in each case:

1. We can look up the conformance of $\texttt{X}_d$ to `P`, project the type witness $\texttt{X}_d\texttt{.A}$ from this normal conformance, and apply Σ to this type witness to get `X.A`.
2. We can look up the conformance of $\texttt{X}_d$ to `P`, apply Σ to this normal conformance to get a specialized conformance, and project the type witness `X.A`.
3. We can apply Σ to $\texttt{X}_d$, look up the conformance of `X` to `P`, and project the type witness `X.A`.

We can also exhibit this with a commutative diagram. Each of the three evaluation orders corresponds to one of the three unique paths from $\texttt{X}_d$ to `X.A`:



To make this concrete, with the `Element` type witness of `[Array<Int>: Sequence]` from Example 7.5, we get the following, because "`Array<Int>.Element`" resolves to `Int`:



7.4. Abstract Conformances

An *abstract conformance* represents the conformance of a type parameter to a protocol. If `T` is a type parameter and `P` is a protocol, global conformance lookup forms an abstract conformance when given `T` and `P`:

$$\langle \texttt{P} \rangle \otimes \texttt{T} := [\texttt{T: P}]$$

We say that an abstract conformance `[T: P]` is *valid* in a generic signature G if the conformance requirement `[T: P]` can be derived from G (Section 5.2). Thus, there is no ambiguity in our two respective formalisms using the same notation `[T: P]` to denote both conformance requirements and abstract conformances.

We will use abstract conformances to fill in the part of Algorithm 6A we have not explained yet, namely, how to apply a substitution map to a bound dependent member type. Before we can do this, we must define type witness projection and conformance substitution with an abstract conformance. Suppose that $G \vdash$ `[T: P]`, and `P` declares an associated type `A`. In Section 5.4, we saw that we can derive the bound dependent member type `T.[P]A` by applying the AssocDecl inference rule to a derivation of `[T: P]`:

1. `[T: P]` (...)
2. `T.[P]A` (AssocDecl 1)

In the type substitution algebra, `[T: P]` is also a valid abstract conformance, and quite naturally, we define the dependent member type `T.[P]A` to be the type witness for `A` in this conformance:

$$\langle \texttt{P|A} \rangle \otimes [\texttt{T: P}] := \texttt{T.[P]A}$$

We can write every bound dependent member type in this way, which reveals a close connection between the derived requirements formalism and type substitution.

Example 7.6. Recall the `firstTwoEqual()` function from Example 5.1. Let G be the generic signature of this function:

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
                    τ_0_0.[Sequence]Element: Equatable,
                    τ_0_0.[Sequence]Element == τ_0_1.[Sequence]Element>
```

We can express each one of `τ_0_0.[Sequence]Element` and `τ_0_1.[Sequence]Element` as the type witness of an abstract conformance:

$$\langle \texttt{Sequence} \rangle \otimes \texttt{τ_0_0} = [\texttt{τ_0_0: Sequence}]$$
$$\langle \texttt{Sequence|Element} \rangle \otimes [\texttt{τ_0_0: Sequence}] = \texttt{τ_0_0.[Sequence]Element}$$

$$\langle \texttt{Sequence} \rangle \otimes \texttt{τ_0_1} = [\texttt{τ_0_1: Sequence}]$$
$$\langle \texttt{Sequence|Element} \rangle \otimes [\texttt{τ_0_1: Sequence}] = \texttt{τ_0_1.[Sequence]Element}$$

Factoring a dependent member type in this manner allows us to understand what happens when we apply a substitution map to such a type:

$$\texttt{T.[P]A} \otimes \Sigma := \big(\langle \texttt{P}|\texttt{A} \otimes [\texttt{T: P}]\big) \otimes \Sigma$$

The right-hand side has but one interpretation, since we require $\otimes$ to be associative:

$$\big(\langle \texttt{P}|\texttt{A} \otimes [\texttt{T: P}]\big) \otimes \Sigma := \langle \texttt{P}|\texttt{A} \otimes \big([\texttt{T: P}] \otimes \Sigma\big)$$

We're applying the substitution map Σ to the abstract conformance $[\texttt{T: P}]$, and then we get the final result by projecting a type witness from this substituted conformance. This new form of conformance substitution is called *local conformance lookup*, by analogy with global conformance lookup, because it plays a similar role.

While we won't investigate the implementation of local conformance lookup until Chapter 12, we can completely *specify* its behavior now. Suppose that $[\texttt{T: P}]$ and Σ are as above. Since $[\texttt{T: P}] = \langle \texttt{P} \rangle \otimes \texttt{T}$, we also have:

$$[\texttt{T: P}] \otimes \Sigma = (\langle \texttt{P} \rangle \otimes \texttt{T}) \otimes \Sigma = \langle \texttt{P} \rangle \otimes (\texttt{T} \otimes \Sigma)$$

In other words, local conformance lookup of $[\texttt{T: P}]$ in Σ must find the *same* conformance as global conformance lookup of $\texttt{T} \otimes \Sigma$ to $\texttt{P}$.

Substitution maps. Given any valid abstract conformance, local conformance lookup must be able to recover the corresponding substituted conformance from the substitution map. Indeed, in the previous chapter, we mentioned that in addition to replacement types, substitution maps also store conformances, when their input generic signature has conformance requirements. We can expand upon this now.

Suppose we have a substitution map $\Sigma \in \textsc{Sub}(G, H)$. The abstract conformance for an *explicit* conformance requirement $[\texttt{T: P}]$ of G is called a *root abstract conformance*. For each root abstract conformance of G, the substitution map Σ contains a conformance with subject type $\texttt{T} \otimes \Sigma$ and protocol $\texttt{P}$; we call these the *root conformances* of Σ. Local conformance lookup with a root abstract conformance is easy to describe. Applying Σ to a root abstract conformance of G projects the corresponding root conformance from Σ:

$$[\texttt{T: P}] \otimes \{\ldots, [\texttt{T: P}] \mapsto [\texttt{X: P}], \ldots\} := [\texttt{X: P}]$$

When a generic declaration is referenced in the language, its generic arguments are either specified explicitly, or inferred. Once we have this list of replacement types, we augment it with a list of root conformances to form a substitution map. To get each root conformance, we take the subject type of each conformance requirement in the declaration's generic signature, substitute in our replacement types, and perform a global conformance lookup. This construction guarantees that each root conformance has the same output generic signature as that which describes the replacement types of Σ. Thus, if Σ is in $\textsc{Sub}(G, H)$, every root conformance of Σ must be in $\textsc{Conf}(H)$.

Setting aside the question of what local conformance lookup should do when the abstract conformance is more generally not a root abstract conformance, we can now describe the missing case of Algorithm 6A:

Algorithm 7B (Substitute dependent member type)**.** Takes a dependent member type `T.[P]A` and a substitution map Σ as input. The dependent member type must be a valid type parameter in the substitution map's input generic signature. Outputs the substituted type $\texttt{T.[P]A} \otimes \Sigma$.

1. Let `A` be the associated type declaration referenced by this dependent member type. (The algorithm does not accept unbound dependent member types.)
2. Let `P` be the protocol containing this associated type declaration.
3. Let `T` be the base type parameter of this dependent member type.
4. Perform the local conformance lookup $[\texttt{T: P}] \otimes \Sigma$ using Algorithm 12A.
5. Project the type witness for `A` from this conformance, and return it.

We also generalize substitution map composition $\Sigma_1 \otimes \Sigma_2$ to the case where Σ_1 has root conformances, by applying the Σ_2 to each root conformance of Σ_1:

$$\Sigma_1 \otimes \Sigma_2 := \{\ldots, [\texttt{T: P}] \mapsto [\texttt{X: P}] \otimes \Sigma_2, \ldots\}$$

That is, if Σ_1 sends the root abstract conformance $[\texttt{T: P}]$ to some conformance $[\texttt{X: P}]$, we must have $[\texttt{T: P}] \otimes (\Sigma_1 \otimes \Sigma_2) = [\texttt{X: P}] \otimes \Sigma_2$. Finally, we remark that for any generic signature G, the root conformances of the identity substitution map 1_G are exactly the root *abstract* conformances of G:

$$1_G := \{\ldots, [\texttt{T: P}] \mapsto [\texttt{T: P}], \ldots\}$$

Example 7.7. To continue with Example 7.6, suppose we call `firstTwoEqual()` with `Array<Int>` and `Set<Int>`:

```
func doIt(_ s1: Array<Int>, _ s2: Set<Int>) {
  if firstTwoEqual(s1, s2) {...}
}
```

Here is the substitution map Σ for the call; it has three root conformances:

$$\begin{aligned}\Sigma := \{&\texttt{τ_0_0} \mapsto \texttt{Array<Int>},\\ &\texttt{τ_0_1} \mapsto \texttt{Set<Int>};\\ &[\texttt{τ_0_0: Sequence}] \mapsto [\texttt{Array<Int>: Sequence}], &(1)\\ &[\texttt{τ_0_1: Sequence}] \mapsto [\texttt{Set<Int>: Sequence}], &(2)\\ &[\texttt{τ_0_0.[Sequence]Element: Equatable}] \mapsto [\texttt{Int: Equatable}]\} &(3)\end{aligned}$$

Let's discuss how we populate the root conformances when we construct Σ. The first two conformance requirements have subject type τ_0_0 and τ_0_1 respectively, so we look up the conformance of $\tau_0_0 \otimes \Sigma$ and $\tau_0_1 \otimes \Sigma$ to `Sequence`:

$$\langle \texttt{Sequence} \rangle \otimes (\tau_0_0 \otimes \Sigma) = [\texttt{Array<Int>: Sequence}] \quad (1)$$

$$\langle \texttt{Sequence} \rangle \otimes (\tau_0_1 \otimes \Sigma) = [\texttt{Set<Int>: Sequence}] \quad (2)$$

The third conformance requirement has subject type $\tau_0_0.[\texttt{Sequence}]\texttt{Element}$. We can write this dependent member type as the type witness of a root abstract conformance:

$$\tau_0_0.[\texttt{Sequence}]\texttt{Element} = \langle \texttt{Sequence} | \texttt{Element} \otimes [\tau_0_0\texttt{: Sequence}]$$

To substitute this dependent member type, we project the type witness for `Element` from root conformance (1), which gives us `Int`. We then look up its conformance to `Equatable` to get the final root conformance (3):

$$\begin{aligned}
&\langle \texttt{Equatable} \rangle \otimes \tau_0_0.[\texttt{Sequence}]\texttt{Element} \otimes \Sigma \\
&\quad = \langle \texttt{Equatable} \rangle \otimes \langle \texttt{Sequence} | \texttt{Element} \otimes ([\tau_0_0\texttt{: Sequence}] \otimes \Sigma) \\
&\quad = \langle \texttt{Equatable} \rangle \otimes \langle \texttt{Sequence} | \texttt{Element} \otimes [\texttt{Array<Int>: Sequence}] \\
&\quad = \langle \texttt{Equatable} \rangle \otimes (\tau_0_0 \otimes \{\tau_0_0 \mapsto \texttt{Int}\}) \\
&\quad = \langle \texttt{Equatable} \rangle \otimes \texttt{Int} \\
&\quad = [\texttt{Int: Equatable}] \quad (3)
\end{aligned}$$

The type parameters $\tau_0_0.\texttt{Element}$ and $\tau_0_1.\texttt{Element}$ are equivalent in G, so we would expect that $\tau_0_1.[\texttt{Sequence}]\texttt{Element} \otimes \Sigma = \texttt{Int}$ also. Indeed, once we know that the type witness for `Element` in $[\texttt{Set<}\tau_0_0\texttt{>: Sequence}]$ is τ_0_0, we see that:

$$\begin{aligned}
&\tau_0_1.[\texttt{Sequence}]\texttt{Element} \otimes \Sigma \\
&\quad = \langle \texttt{Sequence} | \texttt{Element} \otimes ([\tau_0_1\texttt{: Sequence}] \otimes \Sigma) \\
&\quad = \langle \texttt{Sequence} | \texttt{Element} \otimes [\texttt{Set<Int>: Sequence}] \\
&\quad = \tau_0_0 \otimes \{\tau_0_0 \mapsto \texttt{Int}\} \\
&\quad = \texttt{Int}
\end{aligned}$$

Our substitution map satisfies the same-type requirement. If the replacement type for τ_0_1 was `Set<String>` instead for example, then this would no longer hold. We will discuss the problem of checking if a substitution map satisfies the requirements of its input generic signature in Section 9.3. Finally, while G requires the two `Sequences` have the same `Element`, there is no such requirement for their `Iterator` types. Indeed, they are mapped to different concrete types by Σ:

$$\tau_0_0.[\texttt{Sequence}]\texttt{Iterator} \otimes \Sigma = \texttt{IndexingIterator<Array<Int>>}$$

$$\tau_0_1.[\texttt{Sequence}]\texttt{Iterator} \otimes \Sigma = \texttt{IndexingIterator<Set<Int>>}$$

Protocol substitution maps. In Chapter 5, we saw that a protocol declaration `P` has the generic signature `<Self where Self: P>`, called the protocol generic signature, or $G_{\texttt{P}}$ for short. We will sometimes need to specify a substitution map for $G_{\texttt{P}}$. This is called a *protocol substitution map*, and it consists of a single replacement type `X` and a conformance of this type `X` to `P`. We denote it by $\Sigma_{[\texttt{X: P}]}$:

$$\Sigma_{[\texttt{X: P}]} := \{\texttt{Self} \mapsto \texttt{X};\ [\texttt{Self: P}] \mapsto [\texttt{X: P}]\}$$

Applying $\Sigma_{[\texttt{X: P}]}$ to `Self` outputs the conforming type, whereas applying $\Sigma_{[\texttt{X: P}]}$ to a dependent member type `Self.[P]A`, where `A` is some associated type of `P`, outputs the type witness for `A` in the conformance:

$$\texttt{Self} \otimes \Sigma_{[\texttt{X: P}]} = \texttt{X}$$
$$\texttt{Self.[P]A} \otimes \Sigma_{[\texttt{X: P}]} = \langle \texttt{P} | \texttt{A} \otimes [\texttt{Self: P}] \otimes \Sigma = \langle \texttt{P} | \texttt{A} \otimes [\texttt{X: P}]$$

7.5. Associated Conformances

In addition to providing a type witness for each associated type, a normal conformance must also satisfy each associated requirement in the protocol's requirement signature (Section 5.1). We check associated requirements in the conformance checker, when we visit each normal conformance declared in a primary file. We will discuss the general case of how requirements are checked and diagnosed in Section 9.3, while this section will focus on associated conformance requirements in particular, which are of great importance to type substitution.

Suppose that d is a nominal type declaration that conforms to `P`, so we have a normal conformance $[\texttt{X}_d\texttt{: P}]$, and `P` states an associated conformance requirement $[\texttt{Self.U: Q}]_{\texttt{P}}$. To check the requirement, we must replace `Self` with $\texttt{X}_d$ in the subject type `Self.U` of our requirement, and then check that the substituted type conforms to `Q`. The type parameter `Self.U` is written with respect to the protocol generic signature of `P`, so we apply the protocol substitution map $\Sigma_{[\texttt{X}_d\texttt{: P}]}$ to `Self.U` and then follow with global conformance lookup:

$$\langle \texttt{Q} \rangle \otimes \left(\texttt{Self.U} \otimes \Sigma_{[\texttt{X}_d\texttt{: P}]}\right)$$

The result is the *associated conformance* that witnesses the associated conformance requirement $[\texttt{Self.U: Q}]_{\texttt{P}}$. Inside each normal conformance, we record a table of its associated conformances, indexed by the associated conformance requirements of its protocol. We lazily populate this table from the **associated conformance request**. The request receives a normal conformance and an associated conformance requirement as input, and then proceeds as above; first applying the protocol substitution map to the requirement's subject type, and then performing a global conformance lookup.

Projection. In the type substitution algebra, we denote an associated conformance requirement by $\langle \texttt{Self.U: Q}]$. Associated conformance projection gives us this new form of the $\otimes$ operator. With a normal conformance, projection evaluates the associated conformance request, as above:

$$\langle \texttt{Self.U: Q}] \otimes [\texttt{X}_d\texttt{: P}] := \langle \texttt{Q}] \otimes \big(\texttt{Self.U} \otimes \Sigma_{[\texttt{X}_d\texttt{: P}]}\big)$$

The associativity of $\otimes$ determines associated conformance projection with a specialized conformance $[\texttt{X: P}] = [\texttt{X}_d\texttt{: P}] \otimes \Sigma$. First, we project the associated conformance from the underlying normal conformance $[\texttt{X}_d\texttt{: P}]$, and then we apply Σ to this conformance:

$$\begin{aligned}
&\langle \texttt{Self.U: Q}] \otimes [\texttt{X: P}] \\
&\quad = \langle \texttt{Self.U: Q}] \otimes \big([\texttt{X}_d\texttt{: P}] \otimes \Sigma\big) \\
&\quad = \big(\langle \texttt{Self.U: Q}] \otimes [\texttt{X}_d\texttt{: P}]\big) \otimes \Sigma
\end{aligned}$$

We also define associated conformance projection from an abstract conformance. Recall that if $G \vdash [\texttt{T: P}]$ and $[\texttt{Self.U: Q}]_{\texttt{P}}$ is an associated requirement of $\texttt{P}$, we can apply the AssocConf inference rule to derive the conformance requirement $[\texttt{T.U: Q}]$, where $\texttt{T.U}$ denotes the type parameter obtained from $\texttt{Self.U}$ by replacing $\texttt{Self}$ with $\texttt{T}$:

1. $[\texttt{T: P}]$ $(\ldots)$
2. $[\texttt{T.U: Q}]$ (AssocConf 1)

Thus, $[\texttt{T.U: Q}]$ is a valid abstract conformance, and in fact it is the associated conformance for $\langle \texttt{Self.U: Q}]$ in the abstract conformance $[\texttt{T: P}]$, so we define:

$$\langle \texttt{Self.U: Q}] \otimes [\texttt{T: P}] := [\texttt{T.U: Q}]$$

Let $\text{AssocConf}_{\texttt{P}}$ denote the set of all associated conformance requirements of $\texttt{P}$, and recall that $\text{Conf}_{\texttt{P}}(G)$ is the subset of $\text{Conf}(G)$ consisting of conformances to $\texttt{P}$. Associated conformance projection gives us this new form of the $\otimes$ binary operation:

$$\text{AssocConf}_{\texttt{P}} \otimes \text{Conf}_{\texttt{P}}(G) \longrightarrow \text{Conf}(G)$$

One final variant of $\otimes$ is described in Section 9.3. A complete summary of the type substitution algebra appears in Appendix C.

Protocol inheritance. Recall that when a protocol $\texttt{P}$ inherits from another protocol $\texttt{Q}$, we represent this with an associated conformance requirement $[\texttt{Self: Q}]_{\texttt{P}}$ (Section 4.3). In this case, given any conformance to the derived protocol $[\texttt{X: P}]$, we can project the conformance with the same conforming type, but to the base protocol $\texttt{Q}$:

$$\langle \texttt{Self: Q}] \otimes [\texttt{X: P}] = [\texttt{X: Q}]$$

Example 7.8. We described the `Collection` protocol, which inherits from `Sequence`, in Example 5.13. The standard library `Array` also conforms to `Collection`, and not just `Sequence`. Consider the specialized conformance [`Array<Int>: Collection`]. We can project ⟨`Self: Sequence`] from [`Array<Int>: Collection`], which takes us back to the specialized conformance [`Array<Int>: Sequence`] of Example 7.1:

$$\begin{aligned}&\langle\texttt{Self: Sequence}] \otimes [\texttt{Array<Int>: Collection}]\\ &\quad= \langle\texttt{Self: Sequence}] \otimes [\texttt{Array<}\tau\texttt{_0_0>: Collection}] \otimes \{\tau\texttt{_0_0} \mapsto \texttt{Int}\}\\ &\quad= [\texttt{Array<}\tau\texttt{_0_0>: Sequence}] \otimes \{\tau\texttt{_0_0} \mapsto \texttt{Int}\}\\ &\quad= [\texttt{Array<Int>: Sequence}]\end{aligned}$$

Now, let's say that Σ is the protocol substitution map for [`Array<Int>: Collection`], so we're replacing `τ_0_0` with `Array<Int>`. We're going to apply Σ to the type parameter `τ_0_0.[Sequence]Element` of $G_{\texttt{Collection}}$:

$$\begin{aligned}&\tau\texttt{_0_0.[Sequence]Element} \otimes \Sigma\\ &\quad= \langle\texttt{Sequence|Element} \otimes [\tau\texttt{_0_0: Sequence}] \otimes \Sigma\end{aligned}$$

This time, [`τ_0_0: Sequence`] is not an explicit requirement of our generic signature, so [`Array<Int>: Sequence`] is not a root conformance of Σ. Local conformance lookup must take one extra step. Here is a derivation of $G_{\texttt{Collection}} \vdash$ [`τ_0_0: Sequence`]:

1. [`τ_0_0: Collection`] (Conf)
2. [`τ_0_0: Sequence`] (AssocConf 1)

In the type substitution algebra, this is a statement about abstract conformances:

$$\langle\texttt{Self: Sequence}] \otimes [\tau\texttt{_0_0: Collection}] = [\tau\texttt{_0_0: Sequence}]$$

We've found a *conformance path* for [`τ_0_0: Sequence`] in $G_{\texttt{Collection}}$. A conformance path tells local conformance lookup how to get the substituted conformance by projecting a series of associated conformances from a root conformance in the substitution map:

$$\begin{aligned}&\tau\texttt{_0_0.[Sequence]Element} \otimes \Sigma\\ &\quad= \langle\texttt{Sequence|Element} \otimes \underbrace{\langle\texttt{Self: Sequence}] \otimes [\tau\texttt{_0_0: Collection}]}_{\text{conformance path}} \otimes \Sigma\end{aligned}$$

We evaluate [`τ_0_0: Collection`] $\otimes \Sigma$ first, to get the root conformance from Σ. Then, we project the associated conformance [`Array<τ_0_0>: Sequence`], and from there, the type witness for `Element`, to get `Int`:

$$\tau\texttt{_0_0.[Sequence]Element} \otimes \Sigma = \texttt{Int}$$

Chapter 12 will show that we can always find a conformance path for an arbitrary abstract conformance.

Example 7.9. In Example 7.5, we saw that [Array<τ_0_0>: Sequence] witnesses the Iterator associated type with IndexingIterator, specialized at Array<τ_0_0>:

$$\langle \texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Array<}\tau\texttt{_0_0>: Sequence}] = \texttt{IndexingIterator<Array<}\tau\texttt{_0_0>>}$$

Now, recall that the Sequence protocol states the associated conformance requirement $[\texttt{Self.Iterator: IteratorProtocol}]_{\texttt{Sequence}}$. If we project this requirement from our conformance, we get the specialized conformance of IndexingIterator<Array<τ_0_0>> to IteratorProtocol:

$$\begin{aligned}&\langle \texttt{Self.Iterator: IteratorProtocol}] \otimes [\texttt{Array<}\tau\texttt{_0_0>: Sequence}] \\ &= \langle \texttt{IteratorProtocol}] \otimes \texttt{IndexingIterator<Array<}\tau\texttt{_0_0>>} \\ &= [\texttt{IndexingIterator<Array<}\tau\texttt{_0_0>>: IteratorProtocol}]\end{aligned}$$

As always, this specialized conformance consists of the following:

1. The normal conformance [IndexingIterator<τ_0_0>: IteratorProtocol].
2. The context substitution map of IndexingIterator<Array<τ_0_0>>.

Let's look at (1) first. The standard library's IndexingIterator type is interesting, because it implements the requirements of IteratorProtocol in terms of an abstract conformance to Collection; that is, it can serve as the Iterator type witness for any Collection:

```
struct IndexingIterator<Elements: Collection>: IteratorProtocol {
  typealias Element = Elements.Element
  mutating func next() -> Element? {
    ...
  }
}
```

The generic signature of IndexingIterator is $G_{\texttt{Collection}}$; it has a single conformance requirement. The type witness for Element in (1) is a dependent member type:

$$\langle \texttt{IteratorProtocol}|\texttt{Element} \otimes [\texttt{IndexingIterator<}\tau\texttt{_0_0>: IteratorProtocol}] = \tau\texttt{_0_0.[Sequence]Element}$$

Now, consider the substitution map (2), which we will call Σ. Its input generic signature is $G_{\texttt{Collection}}$, and its output generic signature is that of Array, so just <τ_0_0>:

$$\Sigma := \{\tau\texttt{_0_0} \mapsto \texttt{Array<}\tau\texttt{_0_0>}; [\tau\texttt{_0_0: Collection}] \mapsto [\texttt{Array<}\tau\texttt{_0_0>: Collection}]\}$$

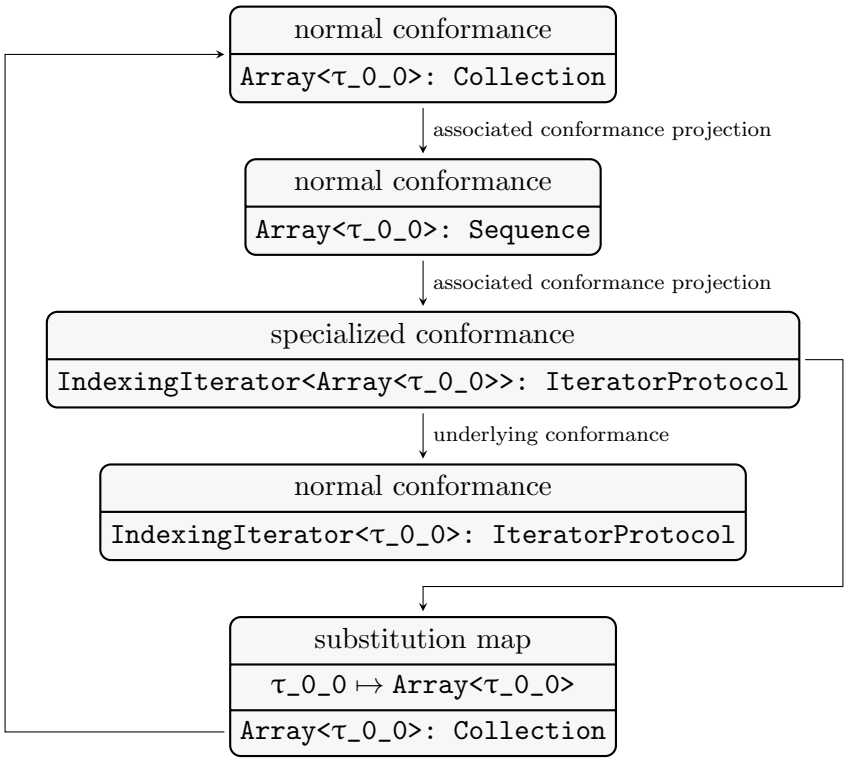
Let's project the type witness for `Element` from our associated conformance (1):

$$\begin{aligned}&\langle\texttt{IteratorProtocol}|\texttt{Element}\\&\quad\otimes[\texttt{IndexingIterator<Array<}\tau\texttt{_0_0>>: IteratorProtocol}]\\&\quad=\tau\texttt{_0_0.[Sequence]Element}\otimes\Sigma\end{aligned}$$

This is like the previous example, except we're replacing τ`_0_0` with `Array<`τ`_0_0>`, and not `Array<Int>`, but otherwise we can use the same conformance path:

$$\begin{aligned}&\tau\texttt{_0_0.[Sequence]Element}\otimes\Sigma\\&\quad=\langle\texttt{Sequence}|\texttt{Element}\otimes\langle\texttt{Self: Sequence}]\otimes[\tau\texttt{_0_0: Collection}]\otimes\Sigma\\&\quad=\langle\texttt{Sequence}|\texttt{Element}\otimes[\texttt{Array<}\tau\texttt{_0_0>: Sequence}]\\&\quad=\tau\texttt{_0_0}\end{aligned}$$

In terms of the input and output generic signature of Σ, this states that the `Element` associated type in the `Iterator` associated conformance of `Array` is witnessed by the `Element` generic parameter of `Array`. While deriving this, we saw that the conformance substitution map of this `Iterator` conformance contains [`Array<`τ`_0_0>: Collection`], which is related to [`Array<`τ`_0_0>: Sequence`] by an associated conformance projection, closing the cycle:



We observed a structural correspondence between generic signatures and requirement signatures in Section 5.1. There is a similar correspondence between substitution maps and conformances, if we consider the constituent parts of both:

Substitution map	Conformance
Replacement types: $\texttt{τ_0_0} \otimes \Sigma$	Type witnesses: $\langle \texttt{P|A} \otimes [\texttt{X: P}]$
Root conformances: $[\texttt{T: P}] \otimes \Sigma$	Associated conformances: $\langle \texttt{Self.U: Q}] \otimes [\texttt{X: P}]$

Hence the four fundamental semantic objects in the Swift generics implementation: generic signatures, requirement signatures, substitution maps, and conformances.

7.6. Associated Type Inference

The final section of this chapter is about *associated type inference*. A more accurate name for it would be "type witness inference," because we're actually inferring the type witnesses in a conformance; but the current name is what we use. What follows is not a complete description of this language feature, but it is sufficient to serve as a quick reference, and also to demonstrate an important theoretical result.

In Section 7.3, we listed four ways to declare a type witness in the language; associated type inference handles cases 2, 3, and 4 in that list. Therefore, we end up here if we are unable to resolve at least one type witness of a normal conformance with a qualified lookup into the conforming type. We proceed as follows:

1. We build a data structure to describe our problem.
2. We find all solutions to the problem described by this data structure.
3. We analyze each solution that we found, and try to pick the best one.
4. If we are successful, we now have all of the type witnesses in the conformance.

The problem instance. We start by looking at the protocol's *value requirements*—those are its function, subscript, and variable (or property) members. The interface type of a value requirement is written with respect to the protocol generic signature `<Self where Self: P>`. We only need to consider a value requirement if its interface type mentions a dependent member type `Self.[P]A`, where `A` is one of the associated types whose type witness we're trying to infer. According to the rules of the Swift language, the conforming type must witness each value requirement with some member declaration that has the same name and kind as the value requirement. The interface

type of the witness must also match the interface type of the value requirement, if we substitute the conforming type in place of `Self`. In particular, if the interface type of the value requirement involves a dependent member type `Self.[P]A`, then the interface type of the value witness must contain the type witness for `A` in the same position.

In associated type inference, we don't know what the mapping from value requirements to value witnesses will be yet, and of course we don't even know all of the type witnesses either. However, at the very least, for each value requirement, we can perform a qualified lookup to find members of the conforming type that *could* witness this value requirement, if their interface type would match. This gives us a set of *candidate value witnesses* for each value requirement.

Example 7.10. Our first example involves this protocol:

```
protocol Meal {
  associatedtype Chef
  associatedtype Main
  associatedtype Desert

  func eat(_: Main, _: Desert)
  func prepare(_: Chef) -> Desert
}
```

In the below conformance, we are asked to infer all three associated types:

```
struct Lunch: Meal {
  func eat(_: Int, _: Bool) {...}
  func eat(_: Bool, _: Float) {...}
  func eat(_: Int, _: Float) -> String {...}

  func prepare(_: Void) -> String {...}
  func prepare(_: String) -> Bool {...}
}
```

Each candidate value witness gives us a *partial solution*, or a list of "associated type $\mapsto$ type witness" pairs, as follows. We walk the interface type of the requirement in parallel with the interface type of the witness. If they differ at any position, we reject the pairing, unless the corresponding child of the requirement's interface type is a dependent member type `Self.[P]A`; in this case, we add a *type witness assignment* to our partial solution. We collect the partial solutions from each value requirement into a *disjunction*. The list of disjunctions obtained from all value requirements is our *problem instance*:

$$\text{problem instance} = \text{list of disjunctions} = \text{list of lists of partial solutions}$$

A *solution* to our problem instance is then just a very special kind of partial solution. We say that a solution is *complete* if every associated type appears *at least* once on the left-hand side of "$\mapsto$", and *consistent* if every associated type appears *at most* once on the left-hand side of "$\mapsto$". A solution *covers* a disjunction if it contains one of the partial solutions from the disjunction as a subset. We state our goal as thus: given a problem instance, *we must find a complete and consistent solution that covers all disjunctions.*

Example 7.11. Given the declarations of Example 7.10, the below table shows the interface type of each value requirement and candidate value witness:

Name	Requirement	Witnesses

`eat(_:_:)`	`(Self.Main, Self.Desert) -> ()`	`(Int, Bool) -> ()`
		`(Bool, Float) -> ()`
		`(Int, Float) -> String`

`prepare()`	`(Self.Chef) -> Self.Desert`	`(Void) -> String` `(String) -> Bool`

Consider the first candidate for `eat(_:_:)`. If it were the witness, then replacing `Self` with `Meal` in the type of the value requirement should give us `(Int, Bool) -> ()`, which could only happen if the type witness for `Main` was `Int`, and the type witness for `Desert` was `Bool`. We denote this partial solution by $\{\texttt{Main} \mapsto \texttt{Int}, \texttt{Desert} \mapsto \texttt{Bool}\}$. On the other hand, the third candidate value witness for `eat(_:_:)` doesn't match the return type of the requirement, so it does not contribute a partial solution to the disjunction. After we analyze all candidate value witnesses, we see that our problem instance has two disjunctions, with two partial solutions in each disjunction, and two type witness assignments in each partial solution:

$$\{\{\texttt{Main} \mapsto \texttt{Int}, \texttt{Desert} \mapsto \texttt{Bool}\}, \{\texttt{Main} \mapsto \texttt{Bool}, \texttt{Desert} \mapsto \texttt{Float}\}\}$$
$$\{\{\texttt{Chef} \mapsto \texttt{Void}, \texttt{Desert} \mapsto \texttt{String}\}, \{\texttt{Chef} \mapsto \texttt{String}, \texttt{Desert} \mapsto \texttt{Bool}\}\}$$

We can solve this instance by hand, by observing that `Desert` must be one of `Bool`, `Float`, or `String`. However, `Bool` is the only one that appears in both disjunctions, so in fact `Desert` is `Bool`. There is only one solution, formed from the first partial solution in the first disjunction and the second partial solution in the second disjunction:

$$\{\texttt{Main} \mapsto \texttt{Int}, \texttt{Desert} \mapsto \texttt{Bool}\} \cup \{\texttt{Chef} \mapsto \texttt{String}, \texttt{Desert} \mapsto \texttt{Bool}\}$$
$$= \{\texttt{Chef} \mapsto \texttt{String}, \texttt{Main} \mapsto \texttt{Int}, \texttt{Desert} \mapsto \texttt{Bool}\}$$

This determines all type *and* value witnesses in the conformance [`Lunch: Meal`].

We can observe more subtle behaviors when a candidate value witness is a member of a protocol extension, because now dependent member types can appear on the *right-hand side* of a match. We will not attempt to describe the full range of possibilities here, and instead just give one example.

Example 7.12. We will determine which of the three candidate value witnesses for `P.f(_:_:)` is the actual witness, based on the possible type witness assignments:

```
protocol P {
  associatedtype A
  associatedtype B
  func f(_: A, _: B)
  func g(_: B)
}

struct S: P {
  func f(_: String, _: Float) {}                                    // (1)
  func g(_: Bool) {}
}

extension P {
  func f(_: Int, _: B) {}                                           // (2)
  func f(_: Void, _: Array<B>) {}                                   // (3)
}
```

This table shows the interface type of each requirement and candidate witness:

Name	Requirement	Witnesses

`f(_:_:)`	`(Self.A, Self.B) -> ()`	`(String, Float) -> ()`
		`(Int, Self.B) -> ()`
		`(Void, Array<Self.B>) -> ()`

`g(_:)`	`(Self.B) -> ()`	`(Bool) -> ()`

Consider the second candidate for `f(_:_:)`. If it were the witness, then `A` would have to be `Int`, but we cannot say anything about `B`. We omit the *tautological* assignment $\texttt{B} \mapsto \texttt{B}$, so our partial solution is just $\{\texttt{A} \mapsto \texttt{Int}\}$. The third candidate appears similar, but unlike the second, it does not contribute a partial solution to the disjunction at all. We cannot represent the assignment $\texttt{B} \mapsto \texttt{Array<Self.[P]B>}$ because it really involves the infinite type `Array<Array<Array<...>>>`. Thus, we do not need to consider the possibility that $\texttt{A} \mapsto \texttt{Void}$, either. The disjunction for `foo(_:_:)` therefore only has two partial solutions. Here is our problem instance:

$$\{\{\texttt{A} \mapsto \texttt{String}, \texttt{B} \mapsto \texttt{Float}\}, \{\texttt{A} \mapsto \texttt{Int}\}\}$$
$$\{\{\texttt{B} \mapsto \texttt{Bool}\}\}$$

The only solution is $\{\texttt{A} \mapsto \texttt{Int}\} \cup \{\texttt{B} \mapsto \texttt{Bool}\} = \{\texttt{A} \mapsto \texttt{Int}, \texttt{B} \mapsto \texttt{Bool}\}$. Thus, (2) is the actual witness for `P.f(_:_:)` in this conformance.

The solver. Suppose that we have a consistent partial solution that covers some, but not all, of the disjunctions in our problem instance. To make progress on the problem, we can pick a disjunction that has not been covered yet, and pick a partial solution from this disjunction. If this partial solution is consistent with ours, we can extend our solution to get a larger solution. At this point, we've either found a complete solution, or we've reduced the size of the problem, so that one fewer disjunction remains. On the other hand, if all partial solutions in our chosen disjunction turn out to be inconsistent with our solution, we have made a bad choice at some point, so we must *backtrack* and make different choices. This is the basic idea behind the associated type inference solver. Note that we do not stop after finding one solution, because we wish to visit all solutions and rank them, as discussed below.

Algorithm 7C (Associated type inference)**.** The algorithm consists of a recursive helper procedure, and the outer procedure. The recursive helper takes a list of disjunctions D, a partial solution S, and the index i of the next disjunction to be covered. It visits all consistent solutions that contain S. The outer procedure invokes the recursive procedure with $S \leftarrow \varnothing$ and $i \leftarrow 0$; so it visits every consistent solution that covers all of D.

1. If i equals the length of D, we have a solution S. Call the visitor and return.
2. Otherwise, let $D[i]$ be the ith disjunction in D. Set $j \leftarrow 0$.
3. If j equals the length of $D[i]$, there are no more solutions that contain S. Return.
4. Otherwise, let $D[i][j]$ be the jth partial solution in $D[i]$. If $S \cup D[i][j]$ is also a consistent partial solution, recurse with $S \leftarrow S \cup D[i][j]$ and $i \leftarrow i + 1$.
5. Set $j \leftarrow j + 1$. Go back to Step 3.

Incomplete solutions. The solver cycles through every consistent solution that covers every disjunction, but certain solutions might not assign every associated type, in which case we will get a solution that is consistent but not complete. In this case, we try a few more things to fill in the missing type witnesses before we give up on this solution:

1. We analyze the protocol's associated same-type requirements to see if this type witness is equivalent to some other known type witness.
2. We check if the conforming nominal type declares a generic parameter with the same name as the associated type.
3. We look for a default type witness, either on the associated type declaration we are attempting to infer, or some other associated type declaration with the same name, in some other protocol the conforming type conforms to.

We will now look at an example of each behavior.

Example 7.13. The first behavior handles a rather common scenario. Suppose we have a `Sequence` conformance whose `Element` type is just some fixed concrete type. We could write it as follows:

```
struct Fibonacci: Sequence {
  struct Iterator: IteratorProtocol {
    mutating func next() -> Int? {...}
  }
  func makeIterator() -> Iterator {...}
}
```

Consider the nested type declaration first. We deduce the type witness for `Element` from `next()` in the `IteratorProtocol` conformance:

$$\langle \texttt{IteratorProtocol}|\texttt{Element} \otimes [\texttt{Fibonacci.Iterator: IteratorProtocol}] = \texttt{Int}$$

Now consider the outer type declaration. The `Sequence` protocol has a single value requirement, which only determines the type witness for `Iterator`:

$$\langle \texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Fibonacci: Sequence}] = \texttt{Fibonacci.Iterator}$$

We cannot deduce the `Element` type witness in `[Fibonacci: Sequence]` from the value requirements of `Sequence`, so instead we consider the associated same-type requirement $[\texttt{Self.Element == Self.Iterator.Element}]_{\texttt{Sequence}}$. The left-hand side is exactly the type witness we're trying to infer, and the right-hand side can be resolved given the type witnesses known so far:

$$\langle \texttt{IteratorProtocol}|\texttt{Element} \otimes \langle \texttt{Self.Iterator: IteratorProtocol}] \\ \otimes [\texttt{Fibonacci: Sequence}] = \texttt{Int}$$

This allows us to conclude that $\langle \texttt{Sequence}|\texttt{Element} \otimes [\texttt{Fibonacci: Sequence}] = \texttt{Int}$.

Example 7.14. The second behavior is peculiar, because we only attempt to resolve a type witness to a generic parameter *after* we've already exhausted the possibility of inferring it from a candidate value witness. This was not immediately apparent in Example 7.3, where the `Dog` struct just happened to witness the associated type `Toy` with the generic parameter `Toy`. However, we might have an associated type with the same name as a generic parameter, but a different type witness assigned, from a candidate value witness. Here, we have a hypothetical iterator that produces all permutations from a set of elements:

```
struct Permutations<Element>: IteratorProtocol {
  // synthesized: typealias Element = Array<Element>
  func next() -> [Element]? {...}
}
```

The `Element` type witness in the conformance is `Array<τ_0_0>`, and not `τ_0_0`, despite what the generic parameter's name may suggest. Thus, `Permutations<Int>.Element` resolves to `Array<Int>` below:

```
func inspect(_: Permutations<Int>.Element) {...}
```

Example 7.15. For the third behavior, we recall the `IndexingIterator` type from Example 7.9. We saw that `IndexingIterator` has the interesting property that it can witness the `Iterator` associated type of `Sequence` for an *arbitrary* `Collection`, so it should be the default. However, it cannot be the default for all `Sequence` conformances, because it's generic argument must be a `Collection`. The standard library models this by having `Collection` re-state `Iterator` to give it a default type witness:

```
associatedtype Iterator = IndexingIterator<Self>
```

Re-stating an inherited associated type declaration does not change the equivalence class structure of a generic signature, but it allows a derived protocol to provide a more suitable default type witness for an inherited associated type. Thus, if we're asked to infer the `Iterator` associated type in a `Sequence` conformance, and the conforming type *also* conforms to `Collection`, we use the above default.

Finishing up. If the above produces a complete solution, we still need to check that our type witness assignment actually satisfies the protocol's associated requirements, using Algorithm 9B from Section 9.3. If the algorithm accepts, we have a *valid* solution. Just like in the expression type checker (Section 3.2), we collect all valid solutions, and consider the three possibilities:

- **One solution**—there is just one unique way to assign a type witness to each associated type in the conformance.
- **No solutions**—no possible assignment of type witnesses could be deduced from the conformance.
- **Multiple solutions**—there is more than one possible consistent assignment of type witnesses in the conformance.

If there is one solution, we record the type witnesses in the conformance, and return successfully. In the case of multiple solutions, we first rank the solutions using a heuristic. For example, all else being equal, we prefer a solution that does not involve a candidate value witness from a protocol extension, because such candidates are typically intended to be used as fallback implementations. If the ranking does not produce a clear winner, we diagnose an ambiguity error.

Example 7.16. In the below, we find two valid solutions $\{\texttt{A} \mapsto \texttt{Int}\}$ and $\{\texttt{A} \mapsto \texttt{Bool}\}$. There is no reason to prefer one or the other, so we diagnose an error:

```
protocol P {
  associatedtype A
  func f(_: A)
}

struct S: P {  // error
  func f(_: Int) {}
  func f(_: Bool) {}
}
```

Example 7.17. A limitation of the current associated type inference implementation is that we only consider a single conformance at a time, so we miss certain deductions that would be possible if the problem instance could describe multiple conformances at once. For example, we are unable to deduce that `Food` must be `Int` here:

```
protocol Base { associatedtype Food }

protocol Derived: Base { func eat(_: Food) }

struct Snack: Derived {
  func eat(_: Int) {}
}
```

This is a long-standing bug; the workaround is to re-declare `Food` in `Derived` [93].

Boolean satisfiability. A *Boolean formula* is an expression built up from variables $x_1, \ldots, x_n$ and three operations: $X \vee Y$ (disjunction, "or"), $X \wedge Y$ (conjunction, "and"), $\neg X$ (negation, "not"). Given an *assignment* of a 1 (true) or 0 (false) value to each variable, we can *evaluate* a Boolean formula by repeated application of the below rules:

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

$\neg$	
0	1
1	0

A Boolean formula is *satisfiable* if there exists at least one *satisfying assignment* under which the formula evaluates to 1. The *satisfiability problem*, or SAT, asks if a given Boolean formula is satisfiable. We will now show that associated type inference can solve SAT, by encoding a Boolean formula as a conformance, where a valid assignment of type witnesses corresponds to a satisfying assignment of truth values.

We only need to consider a restricted form of the SAT problem. Let's say that a single variable x or its negation $\neg x$ is a *literal*, while a *clause* is a disjunction of one or more literals, for example $(\neg x_1 \vee x_2 \vee \neg x_3)$. A formula is in *conjunctive normal form*, or CNF, if it is a conjunction of clauses:

$$(\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

The above formula is more precisely in 3CNF, meaning it has the additional property that no clause contains more than three literals. In fact, an arbitrary Boolean formula can always be converted first into CNF, and then 3CNF, in a way that preserves satisfiability (or lack thereof). Thus, for our purposes it suffices to solve 3SAT, as the restriction of the SAT problem to 3CNF is known.

We will encode the above 3CNF formula as an instance of associated type inference, but it will be obvious that this encoding works for any 3CNF formula. First, we need a pair of nominal types to represent true and false:

```
struct T {}
struct F {}
```

We then declare a nominal type named `Solver` that is common to all 3CNF formulas, and an `Instance` protocol that encodes each clause of our specific 3CNF formula. Finally, associated type inference will kick in and solve 3SAT when we state the conformance of `Solver` to `Instance`. We will look at `Solver` first:

```
struct Solver: Instance {
```

We have two overloads of `literal(_:_:)`:

```
  func literal(_: T, _: F) {}
  func literal(_: F, _: T) {}
```

We also have $2^3 - 1 = 7$ overloads of `clause(_:_:_:)`, one for each triple of true or false such that not all three are false:

```
  func clause(_: T, _: F, _: F) {}
  func clause(_: F, _: T, _: F) {}
  func clause(_: T, _: T, _: F) {}
  func clause(_: F, _: F, _: T) {}
  func clause(_: T, _: F, _: T) {}
  func clause(_: F, _: T, _: T) {}
  func clause(_: T, _: T, _: T) {}
```

This completes the `Solver`:

```
}
```

Now, we fill out the `Instance` protocol, in which we encode the variables and clauses of our 3CNF formula. For each variable x_i, we declare an associated type `XiP` to represent the literal x_i, and an associated type `XiN` to represent the literal $\neg x_i$:

```
protocol Instance {
  associatedtype X1P; associatedtype X1N
  associatedtype X2P; associatedtype X2N
  associatedtype X3P; associatedtype X3N
```

We also add a series of methods named `literal(_:_:)` to our protocol, one for each variable x_i, to encode the fact that x_i and $\neg x_i$ are complementary literals:

```
  func literal(_: X1P, _: X1N)
  func literal(_: X2P, _: X2N)
  func literal(_: X3P, _: X3N)
```

Finally, we add a series of methods named `clause(_:_:_:)`, one for each clause. To get the three parameter types, we translate each literal x_i or $\neg x_i$ appearing in the clause into a parameter of type `XiP` or `XiN`, and if a clause has fewer than three literals, we just repeat a literal. For example, the first clause in our formula is $(\neg x_1 \vee x_2 \vee x_3)$, which becomes `(_: X1N, _: X2P, _: X3P)`, and so on:

```
  func clause(_: X1N, _: X2P, _: X3P)
  func clause(_: X1N, _: X2P, _: X3N)
  func clause(_: X2N, _: X3P, _: X3P)
  func clause(_: X1P, _: X2P, _: X2P)
  func clause(_: X1N, _: X2N, _: X2N)
}
```

The above encodes a series of disjunctions for Algorithm 7C:

1. Each `Instance.literal(_:_:)` has the two `Solver.literal(_:_:)` overloads as candidates, so each `XiP` must be `T` or `F`, and `XiN` must be the negation.

2. Each `Instance.clause(_:_:_:)` has seven `Solver.clause(_:_:_:)` overloads as candidates, so we get a disjunction where each partial solution is a satisfying assignment for the clause, because at least one literal in the clause is true.

In our example, associated type inference finds a unique solution, and we can even recover the satisfying assignment, by looking at the type witnesses for each `XiP`. For example, the following will print `(F, T, T)`:

```
print((Solver.X1P, Solver.X2P, Solver.X3P).self)
```

A quick calculation confirms that $\{x_1 := 0, x_2 := 1, x_3 := 1\}$ is a satisfying assignment:

$$(\neg 0 \vee 1 \vee 1) \wedge (\neg 0 \vee 1 \vee \neg 1) \wedge (\neg 1 \vee 1) \wedge (0 \vee 1) \wedge (\neg 0 \vee \neg 1) = 1$$

There is one caveat with how we interpret the output of our "SAT solver." Formally, SAT is a *decision problem* with a true or false answer; it just asks if at least one satisfying assignment *exists*. On the other hand, associate type inference will enumerate *all* valid solutions and attempt to pick the best one. Our example formula was not only satisfiable, but it happened to have a unique satisfying assignment. In general, if associated type inference finds a unique solution, *or* if it diagnoses an ambiguity because there is more than one valid solution, we can conclude that our Boolean formula is satisfiable. If there is no solution, there is no satisfying assignment. But what does this all *mean*?

Non-deterministic polynomial time. We've arrived at one of the most celebrated ideas in theoretical computer science. A classic text on this topic is [94], while [95] has a gentler introduction. We can start by describing a *non-deterministic* algorithm to solve satisfiability. We fix an encoding of truth assignments as strings of symbols; for example, we can write down the 0 and 1 values assigned to each of our n variables, in some fixed order. Our non-deterministic algorithm then simultaneously "guesses" all 2^n truth assignments, and "checks" each assignment in parallel with all others, by evaluating the formula to obtain a true or false result. The overall algorithm outputs success if at least one thread outputs success, otherwise it outputs failure.

As far as non-deterministic algorithms go, this one is extremely efficient. If someone hands us an assignment of truth values, we can *check* if it is a satisfying assignment in $O(n)$ steps, where n is the length of the formula. More generally, a decision problem belongs to the class of *non-deterministic polynomial time problems*, also known as NP for short, if we can check that a given solution satisfies the problem instance with an algorithm that always terminates in at most $O(n^k)$ steps, where n is the size of the instance, and k is a fixed constant.

We require that the checking phase terminates on all inputs, which implies it only uses a finite amount of memory. In a 1971 paper, Stephen A. Cook used this observation to show that the execution of a non-deterministic algorithm can be modeled by a Boolean formula (the general idea here is almost like assembling a finite digital circuit). In other words, *every* NP problem can be translated into a Boolean satisfiability problem—and importantly, this encoding itself takes only polynomial time and space [96].

Boolean satisfiability was the first of many *NP-complete* problems to be discovered. The NP-complete problems are "the hardest" problems in NP, because they have every *other* problem in NP as a special case, and they are all as "equally hard" as each other. More generally, if our problem can encode an NP-complete problem, but our problem is itself not necessarily in NP, we just say that our problem is *NP-hard*:

Theorem 7.18. Swift associated type inference is NP-hard.

But why are these problems "hard"? The reason is very simple. No machine is capable of unlimited parallelism, so we cannot actually *run* a non-deterministic algorithm as specified, unless of course, we simulate every possible thread of execution sequentially. In particular, there is no apparent way to translate a non-deterministic *polynomial time* algorithm into an equally efficient deterministic algorithm for a real computer.

Notice that the running time of Algorithm 7C is worst case *exponential*, due to the use of backtracking search. We can make various improvements to reduce the size of the search space, so the running time need not be exponential on *every* instance. However, if we could actually devise an algorithm that could *always* solve associated type inference in polynomial time, then by Theorem 7.18, our algorithm would be quite remarkable indeed—because it could solve *every* NP problem in polynomial time. On the other hand, if only someone could prove that *at least one* NP problem cannot be solved by a deterministic polynomial time algorithm, then we could immediately conclude the same about *every* NP-hard problem, so in particular, it would preclude the possibility of solving associated type inference in polynomial time.

This is, of course, the famous "P $\stackrel{?}{=}$ NP" problem, which remains unsolved at the present time. For a survey of the problem, see [97].

Overload resolution. In realistic Swift programs, associated type inference is only ever presented with a handful of requirements and candidate witnesses at a time, and the search space explored by Algorithm 7C is not large. However, this matching of requirements against candidate witnesses is just a special case of *overload resolution*, performed by the expression type checker. Swift programmers sometimes run into the NP-hardness of that problem, when `the compiler is unable to type-check this expression in reasonable time`. It can also be shown that overload resolution is NP-hard in the C♯ programming language [98], using an encoding of the SAT problem quite similar to ours.

We will not explain it here, but in fact associated type inference can be seen as an instance of an *exact cover with colors*, or XCC problem. XCC is also NP-complete of course, but just as with SAT, clever algorithms have been devised that can solve many instances of XCC efficiently. This is a possible future direction if the time spent in associated type inference ever becomes non-trivial in a real-world project. Algorithms for solving XCC and SAT are discussed in [99]. Finally, another good reference about the SAT problem is [100].

* * *

7.7. Source Code Reference

Global Conformance Lookup

Key source files:

- include/swift/AST/ConformanceLookup.h
- lib/AST/ConformanceLookup.cpp
- lib/AST/ConformanceLookupTable.h
- lib/AST/ConformanceLookupTable.cpp
- include/swift/AST/DeclContext.h

`lookupConformance()` *function*

Performs global conformance lookup of a type to a protocol. Note that if this is a conditional conformance, this does not check conditional requirements. To check conditional requirements, use `checkConformance()` from Section 10.5 instead.

Conformance Lookup Table

Key source files:

- lib/AST/ConformanceLookupTable.h
- lib/AST/ConformanceLookupTable.cpp
- include/swift/AST/DeclContext.h

`ConformanceLookupTable` *class*

A conformance lookup table for a nominal type declaration. Every `NominalTypeDecl` has a conformance lookup table, but it is not exposed outside of the global conformance lookup implementation.

`IterableDeclContext` *class*

Base class inherited by `NominalTypeDecl` and `ExtensionDecl`.

- `getLocalConformances()` returns a list of conformances directly declared on this nominal type or extension.

`NominalTypeDecl` *class*

See also Section 4.6.

- `getAllConformances()` returns a list of all conformances declared on this nominal type, its extensions, and inherited from its superclass, if any.

Operations on Conformances

Key source files:

- include/swift/AST/ProtocolConformanceRef.h
- include/swift/AST/ProtocolConformance.h
- lib/AST/ProtocolConformanceRef.cpp
- lib/AST/ProtocolConformance.cpp

`ProtocolConformanceRef` *class*

A protocol conformance. The representation fits in a single pointer, so values of this type are cheap to pass by value. An invalid conformance is encoded using a null pointer, and it is an error to call most of the below operations on an invalid conformance.

Usually, instances of `ProtocolConformanceRef` are obtained by substitution or global conformance lookup. It is also possible to construct them directly:

- To get an invalid conformance, call the default constructor, or equivalently, the `ProtocolConformanceRef::forInvalid()` static method.
- To wrap a concrete conformance, call the one-argument constructor that takes a `ProtocolConformance *`.
- To get an abstract conformance, call the `forAbstract()` static method.

A `ProtocolConformanceRef` can be taken apart:

- `isInvalid()` checks if this conformance is invalid.
- `isAbstract()` checks if this conformance is abstract.
- `getAbstract()` returns the stored `AbstractConformance *` if this conformance is abstract, otherwise asserts.
- `isConcrete()` checks if this conformance is concrete.
- `getConcrete()` returns the stored `ProtocolConformance *` if this conformance is concrete, otherwise asserts.

Often, we do not care if a protocol conformance is abstract or concrete, because we instead use the following methods of `ProtocolConformanceRef` that work on both:

- `getType()` returns the conforming type.

- `getProtocol()` returns the `ProtocolDecl` being conformed to.
- `getTypeWitness()` returns the type witness for the given associated type declaration.
- `getAssociatedConformance()` returns the associated conformance for an associated conformance requirement of the conformed protocol.
- `subst()` returns a new protocol conformance obtained by applying a substitution map to this conformance.

Like `Type`, `GenericSignature`, and `SubstitutionMap`, conformances are immutable and uniquely allocated. Thus, conformances can be tested for equality using the `operator==` overload. This depends on type sugar, unless the conformance is *canonical*.

- `isCanonical()` answers if this conformance is canonical.
- `getCanonical()` returns the canonical conformance equivalent to this one.

`AbstractConformance` *class*

An abstract protocol conformance. This class is rarely used directly, because both of its operations are available on `ProtocolConformanceRef` itself.

- `getType()` returns the conforming type.
- `getProtocol()` returns the `ProtocolDecl` being conformed to.

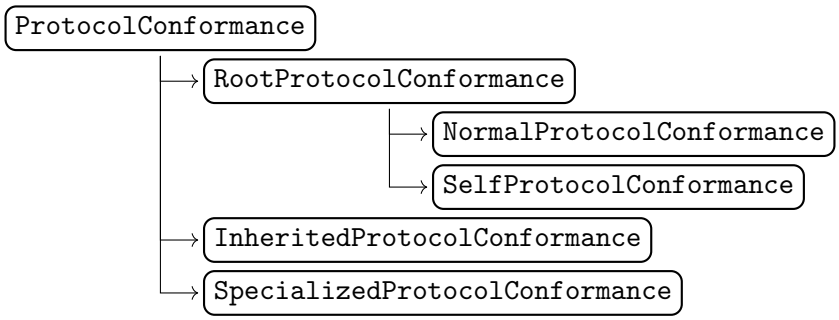
Abstract conformances with the same conforming type and protocol are equal as pointers. An abstract conformance is canonical if its conforming type is a canonical type.

`ProtocolConformance` *class*

A concrete protocol conformance. Concrete protocol conformances are always passed by pointer. Concrete conformances can have conditional requirements; this is documented in Section 10.4 and Section 10.5.

- `getType()` returns the conforming type.
- `getProtocol()` returns the conformed protocol.
- `getTypeWitness()` returns the type witness for an associated type.
- `getAssociatedConformance()` returns the associated conformance for a conformance requirement in the protocol's requirement signature.
- `subst()` returns the protocol conformance obtained by applying a substitution map to this conformance.

Figure 7.1.: The `ProtocolConformance` class hierarchy



The `ProtocolConformance` class is the root of a class hierarchy shown in Figure 7.1.

`RootProtocolConformance` *class*

Abstract base class for `NormalProtocolConformance` and `SelfProtocolConformance`. Inherits from `ProtocolConformance`.

`NormalProtocolConformance` *class*

A normal protocol conformance. Subclass of `RootProtocolConformance`.

- `getDeclContext()` returns the conforming declaration context, either a nominal type declaration or extension.
- `getGenericSignature()` returns the generic signature of the conforming context.

A normal conformance is always canonical.

`InheritedProtocolConformance` *class*

An inherited protocol conformance. Subclass of `ProtocolConformance`.

- `getInheritedConformance()` returns the base conformance, which must be normal or specialized.

An inherited conformance is canonical if its conforming type is a canonical type, and its base conformance is a canonical conformance.

`SpecializedProtocolConformance` *class*

A specialized protocol conformance. Subclass of `ProtocolConformance`.

- `getGenericConformance()` returns the underlying normal conformance.
- `getSubstitutionMap()` returns the conformance substitution map.

A specialized conformance is canonical if its conformance substitution map is canonical. To canonicalize a specialized conformance, we canonicalize the elements of its substitution map, and form a new specialized conformance.

Type Substitution

Key source files:

- include/swift/AST/SubstitutionMap.h
- lib/AST/TypeSubstitution.cpp

`SubstitutionMap` *class*

We discussed substitution maps in Section 6.6. Recall that a substitution map stores a list of conformances, one for each conformance requirement in its input generic signature. Three overloads of the `get()` static method construct substitution maps. They differ in how the replacement types and conformances are specified:

`get(GenericSignature, ArrayRef<Type>, ArrayRef<ProtocolConformanceRef>)`:
Builds a new substitution map from an input generic signature, an array of replacement types, and array of conformances. The first array's elements are in one-to-one correspondence with the signature's generic parameters, and the second array's elements are in one-to-one correspondence with the signature's conformance requirements.

`get(GenericSignature, ArrayRef<Type>, LookupConformanceFn)`:
Builds a new substitution map from an input generic signature, an array of replacement types, and array of conformances. Instead of providing an array of conformances, this form takes a callback, which is invoked on each conformance requirement.

`get(GenericSignature, TypeSubstitutionFn, LookupConformanceFn)`:
Builds a new substitution map by invoking a pair of callbacks to produce each replacement type and conformance. This overload takes two callbacks, which are invoked to produce each replacement type and conformance.

Finally, the `getProtocolSubstitutions()` static method builds a protocol substitution map for a protocol generic signature, given a conformance to this protocol.

`TypeSubstitutionFn` *type alias*

The type of a replacement type callback for the third form of `SubstitutionMap::get()`.

```
using TypeSubstitutionFn
  = llvm::function_ref<Type(SubstitutableType *dependentType)>;
```

The parameter type is always a `GenericTypeParamType *` when the callback is used with `SubstitutionMap::get()`.

`LookupConformanceFn` *type alias*

The type signature of a conformance lookup callback for `SubstitutionMap::get()`.

```
using LookupConformanceFn = llvm::function_ref<
    ProtocolConformanceRef(InFlightSubstitution &IFS,
                           CanType origType,
                           ProtocolDecl *proto)>;
```

The `origType` and `proto` are the subject type and protocol declaration of a conformance requirement in the input generic signature of the substitution map being constructed. If desired, the `InFlightSubstitution` instance can be used to recover the substituted subject type as follows:

```
Type substType = origType.subst(IFS);
```

`LookUpConformanceInModule` *struct*

A callback intended to be used with `SubstitutionMap::get()` as a conformance lookup callback. Overloads `operator()` with the signature of `LookupConformanceFn` to perform a global conformance lookup with the given requirement's substituted subject type and protocol. An instance of this callback is constructed without arguments. For example:

```
auto subMap = SubstitutionMap::get(genericSig, replacementTypes,
                                   LookUpConformanceInModule());
```

`LookUpConformanceInSubstitutionMap` *struct*

A callback intended to be used with `SubstitutionMap::get()` as a conformance lookup callback. Overloads `operator()` with the signature of `LookupConformanceFn` to perform a local conformance lookup into another substitution map (Section 7.4). Constructed with another `SubstitutionMap`.

For example, if `genericSig` is the same as the input generic signature of `subMap` except that it drops some requirements, we can construct a substitution map for `genericSig` as follows:

```
auto newMap = SubstitutionMap::get(
    genericSig,
    subMap.getReplacementTypes(),
    LookUpConformanceInSubstitutionMap{subMap});
```

`TypeSubstituter::transformDependentMemberType()` *method*

Implements Algorithm 7B.

Associated Type Inference

Key source file:

- lib/Sema/AssociatedTypeInference.cpp

`TypeWitnessRequest`	*class*

Request evaluator request for lazily resolving a type witness in a normal conformance. Attempts a qualified lookup, and if that fails, evaluates a `TypeWitnessesRequest`.

`TypeWitnessesRequest`	*class*

Request evaluator request for resolving all type witnesses in a normal conformance, first by attempting lookup and then associated type inference.

`AssociatedConformanceRequest`	*class*

Request evaluator request for lazily looking up an associated conformance of a normal conformance that was parsed from source. Computes the substituted subject type of the requirement and calls global conformance lookup.

`InferredAssociatedTypesByWitness`	*struct*

A partial solution.

`InferredAssociatedTypesByWitnesses`	*type alias*

A disjunction, represented as an `llvm::SmallVector` of partial solutions.

`InferredAssociatedTypesByWitnesses`	*type alias*

The problem instance, represented as a `llvm::SmallVector` of pairs, where the first element in each pair is a value requirement and the second is a disjunction.

`AssociatedTypeInference`	*class*

The associated type inference solver.

- `inferTypeWitnessesViaValueWitnesses()` builds the problem instance, consisting of a disjunction for each value requirement that mentions the given set of associated types.
- `getPotentialTypeWitnessesFromRequirement()` builds a disjunction for a value requirement, by finding all candidate value witnesses for this requirement and constructs a partial solution for each one.
- `getPotentialTypeWitnessesByMatchingTypes()` builds a partial solution from a pairing of a value requirement with candidate value witness.
- `solve()` implements Algorithm 7C.

8. Archetypes

An archetype encapsulates a reduced type parameter and a generic signature, two semantic objects we met in Chapter 5. This representation is self-describing, so it can answer questions about protocol conformance and such without further context. An archetype thus behaves like a concrete type in many ways; it is the "most general" concrete type that satisfies the requirements of its type parameter. (Recall that a type parameter is essentially a *name* that requires a separate generic signature to interpret.) Archetypes are created by mapping type parameters into a *generic environment*, an object derived from a generic signature. Multiple environments can be instantiated from a single generic signature, each one generating a distinct family of archetypes. Exactly one of those is the *primary* generic environment, whose archetypes are called *primary archetypes*. We will discuss these first.

A primary generic environment defines a correspondence between type parameters and primary archetypes:

- **Mapping into an environment** recursively replaces type parameters in the given interface type with their archetypes from the given primary generic environment.
- **Mapping out of an environment** recursively replaces primary archetypes with their reduced type parameters.

Mapping into an environment produces a type which no longer contains type parameters, but instead may contain primary archetypes. We call this a *contextual type*.

Contextual types and primary archetypes are used in place of interface types inside the expressions that appear in generic function bodies, but archetypes don't surface as a distinct concept in the language model; it's a representation change. SILGen lowers expressions to SIL instructions which consume and produce SIL values, so the types of SIL values are also contextual types. Finally, when IRGen generates code for a generic function, archetypes become *values* representing the runtime type metadata provided by the caller of the generic function.

We will see that various operations on interface types, such as generic signature queries and type substitution, have analogs in the world of contextual types. As a practical matter, understanding these is important for compiler developers. The representation of archetypes will also motivate the *type parameter graph* in Section 8.3, which will give us a new way to think about generic signatures. Finally, in Section 8.4 we give a historical sketch of the Swift 3 generics implementation, based on the construction of a finite type parameter graph where archetypes were the source of truth.

Nesting. A nested generic declaration with a distinct generic signature (because it introduced new generic parameters or requirements) also introduces a fresh primary generic environment; we do not "inherit" primary archetypes from the outer scope. All primary archetypes, including those that represent outer generic parameters, are instantiated from the innermost environment. (If the inner declaration does not declare new generic parameters or requirements, it has the same generic signature, and thus the same generic environment, as its parent context.)

This allows correct modeling of nested declarations that impose new requirements on outer generic parameters, which we discussed in Section 4.2. This was not supported prior to Swift 3 because outer primary archetypes were re-used. This simple program demonstrates this behavior:

```
struct Box<T> {
  var contents: T?

  mutating func take() -> T {
    let value: T = self.contents!
    self.contents = nil
    return value
  }

  func compare(_ other: T) -> Bool where T: Equatable {
    guard let value: T = self.contents else { return false }
    return value == other
  }
}
```

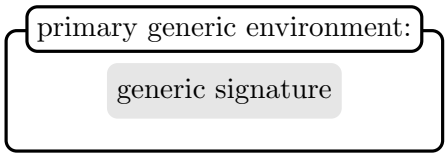
The `take()` and `compare()` methods have different generic signatures, with `take()` inheriting the generic signature of struct `Box`, and `compare()` adding a conformance requirement. The type representation "`T`" resolves to the *same* generic parameter type `T` (or `τ_0_0`) in all source locations where it appears. However, this generic parameter maps to two different archetypes in the generic environment of each function. Thus, the type of the expression "`self.contents`" is actually different in `take()` and `compare()`. Let's use the notation $[\![\texttt{T}]\!]_d$ for the archetype obtained by mapping the type parameter `T` into the generic environment of some declaration d. Here, we have two archetypes:

1. $[\![\texttt{T}]\!]_{\text{take}}$, which does not conform to any protocols.
2. $[\![\texttt{T}]\!]_{\text{compare}}$, which conforms to `Equatable`.

The expression type checker is aware of this fact, without needing to plumb the generic signature of the current context through manually, because it operates on archetypes instead of type parameters.

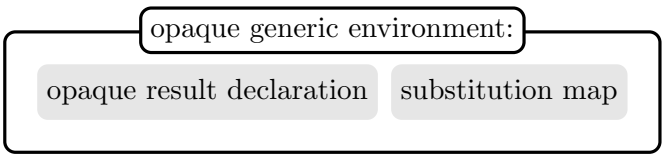
Generic environment kinds. There are three generic environment kinds. Instances of each kind are uniquely allocated for each combination of input parameters:

- As we already said, every generic signature has exactly one **primary generic environment** of primary archetypes.



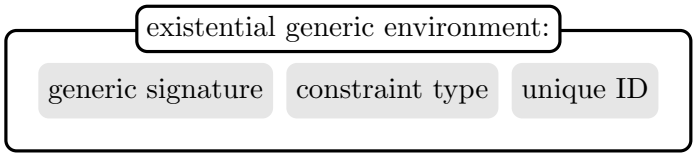
  Primary generic environments preserve the sugared names of generic parameters for the printed representation of an archetype, so a primary generic environment depends on the pointer identity of its generic signature; two generic signatures which are canonically equal but differ in type sugar will instantiate distinct primary generic environments.

- When a declaration has an opaque result type, we can create an **opaque generic environment** parameterized by a substitution map for the owner declaration's generic signature:



  Unlike primary archetypes, *opaque archetypes* are not scoped to the lexical scope of their owner declaration; they can appear anywhere that their owner declaration is visible. They also behave differently under substitution. It is legal for an interface type to contain an opaque archetype, in particular the return type of the owner declaration itself will contain one. Details are in Chapter 13.

- An **existential generic environment** is created when an existential value is opened at a call expression. An *existential archetype* instantiated from such an environment represents the concrete payload stored inside the existential value.



  Every opening expression gets a fresh unique ID, and therefore a new existential generic environment. In the AST, existential archetypes cannot "escape" from their opening expression; in SIL, they are introduced by an opening instruction and are similarly scoped by the dominance relation on the control flow graph. We will discuss existential types in Chapter 14.

Archetype equality. Some notes about contextual types:

1. Every primary archetype is a contextual type. More generally, a contextual type is a concrete type that contains a primary archetype, like `Array<⟦T⟧>` formed from the primary archetype ⟦`T`⟧.
2. A fully-concrete type like `Array<Int>` is both an interface type and a contextual type.
3. An interface type like `Array<τ_0_0>` is *not* a contextual type.
4. A contextual type will only ever contain primary archetypes from a single generic environment.
5. Archetypes from non-primary generic environments may appear in both interface types and contextual types, but we won't worry about those for now.

In the following, let G be a generic signature, and write $\textsc{Type}(G)$ for the set of all interface types of G, as before. Now, let's say that $\textsc{Type}(⟦G⟧)$ is the set of all contextual types that contain primary archetypes of G. We can understand mapping into and out of an environment as giving us a pair of functions:

$$\mathsf{in}_G \colon \textsc{Type}(G) \longrightarrow \textsc{Type}(⟦G⟧)$$
$$\mathsf{out}_G \colon \textsc{Type}(⟦G⟧) \longrightarrow \textsc{Type}(G)$$

In Section 5.3, we defined the reduced type equality relation on the valid type parameters of G, and in Section 5.6, we extended this to all of $\textsc{Type}(G)$ via the recursive definition of a reduced type. The fact that an archetype's type parameter is a reduced type means the following when it comes to equivalence of types:

1. If $G \vdash$ [`T == U`] for type parameters `T` and `U`, then in_G(`T`) and in_G(`U`) give two equal pointers to the same archetype. The three relations of reduced type equality, canonical type equality and type pointer equality coincide on archetypes.
2. If $G \vdash$ [`T == X`] for some type parameter `T` and concrete type `X`, then we define in_G(`T`) := in_G(`X`). A type parameter fixed to a concrete type is not represented by an archetype; instead, we map it into the environment by recursively mapping the concrete type.
3. If `T` is some interface type, then $\mathsf{out}_G(\mathsf{in}_G$(`T`)) is canonically equal to the reduced type of `T` when computed with Algorithm 5E.
4. If two interface types are equivalent under reduced type equality, the corresponding contextual types are equivalent under canonical type equality.

Taken together, the operations of in_G and canonical type equality faithfully represent the equivalence class structure of a generic signature. We will see how archetypes subsume the remaining generic signature queries next.

8.1. Local Requirements

The behavior of a type parameter $\texttt{T} \in \textsc{Type}(G)$ can be understood by considering all conformance, superclass, and layout requirements with a subject type of `T` that can be derived from G. We can recover this using the generic signature queries from Section 5.6, which take a G and a `T`; that is, the caller must keep track of the G. If instead we have an archetype that represents `T` in a generic signature G, we can directly consult a list of *local requirements* recorded within the archetype itself. We pretend we have a primary archetype $[\![\texttt{T}]\!] \in \textsc{Type}([\![G]\!])$ below, but everything we say in this section is also true of the other archetype kinds as well. An archetype $[\![\texttt{T}]\!]$ records the following:

- **Required protocols:** A list of all protocols `P` such that $G \vdash [\texttt{T: P}]$. This is the result of the `getRequiredProtocols`() generic signature query.
- **Superclass bound:** A class type `C` such that $G \vdash [\texttt{T: C}]$. This is the result of the `getSuperclassBound`() generic signature query.
- **Requires class flag:** True if $G \vdash [\texttt{T: AnyObject}]$. This is the result of the `requiresClass`() generic signature query.
- **Layout constraint:** A more fine-grained view than the previous, to differentiate between Objective-C and Swift-native reference counting. This is the result of the `getLayoutConstraint`() generic signature query.

When we create an archetype, we use the `getLocalRequirements`() generic signature query to gather all of the above information in one shot.

Qualified name lookup. An archetype can serve as the base type for a qualified name lookup (Section 2.1). The visible members are those of the archetype's required protocols, superclass declaration, and finally, the concrete protocol conformances of the superclass. This explains how we type check a member reference expression where the base type is an archetype.

Global conformance lookup. Consider the two declarations below, together with the generic signature `<Box, Elt where Box: Base<Elt>, Box: Proto>`:

```
protocol OtherProto { func otherRequirement() }
protocol Proto { func requirement() }
class Base<Elt>: OtherProto { func otherRequirement() }
```

If we have a value of type $[\![\texttt{Box}]\!]$, a call to `requirement()` must dispatch through the witness table; whereas a call to `otherRequirement()` can directly invoke the concrete witness, even though we don't have the concrete replacement type for $[\![\texttt{Box}]\!]$.

We want global conformance lookup to return an abstract conformance when given the archetype $[\![\texttt{Box}]\!]$ and `Proto`, but a concrete conformance with `OtherProto`:

$$\langle\texttt{Proto}] \otimes [\![\texttt{Box}]\!] = [[\![\texttt{Box}]\!]\texttt{: Proto}]$$
$$\langle\texttt{OtherProto}] \otimes [\![\texttt{Box}]\!] = [\texttt{Base<}[\![\texttt{Elt}]\!]\texttt{>: OtherProto}]$$

Thus, the subject type of an abstract conformance can be an archetype, and not just a type parameter. Also notice how the second conformance is formed from the normal conformance $[\texttt{Base<}\tau\texttt{_0_0>: OtherProto}]$ with this conformance substitution map:

$$[\texttt{Base<}\tau\texttt{_0_0>: OtherProto}] \otimes \{\tau\texttt{_0_0} \mapsto [\![\texttt{Elt}]\!]\}$$
$$= [\texttt{Base<}[\![\texttt{Elt}]\!]\texttt{>: OtherProto}]$$

We will discuss substitution maps with contextual replacement types shortly, but first let's formalize the above behavior of global conformance lookup.

Recall that with a type parameter, Algorithm 7A returns an abstract conformance (Section 7.4), without checking if the conformance actually holds. We will now extend the algorithm to handle archetypes, which gives us a more useful behavior that distinguishes several cases. Let $[\![\texttt{T}]\!]$ be an archetype with generic signature G, and let `P` be a protocol:

1. If $G \vdash [\texttt{T: C}]$ for some class type `C`, and `C` conforms to `P`, then the archetype *conforms concretely* to `P`. The class type `C` may contain type parameters, so global conformance lookup recursively calls itself on $\mathsf{in}_G(\texttt{C})$:

   $$\langle\texttt{P}] \otimes [\![\texttt{T}]\!] := \langle\texttt{P}] \otimes \mathsf{in}_G(\texttt{C}) = [\mathsf{in}_G(\texttt{C})\texttt{: P}]$$

   The result is actually wrapped in an *inherited conformance*, which we recall from Chapter 7 exists to give it the conforming type $[\![\texttt{T}]\!]$ rather than $\mathsf{in}_G(\texttt{C})$.

2. If $G \vdash [\texttt{T: P}]$, the archetype *conforms abstractly*, and thus global conformance lookup returns an abstract conformance:

   $$\langle\texttt{P}] \otimes [\![\texttt{T}]\!] := [[\![\texttt{T}]\!]\texttt{: P}]$$

3. Otherwise, $[\![\texttt{T}]\!]$ does not conform to `P`, and global conformance lookup returns an invalid conformance.

When the subject type of an abstract conformance is an archetype, we define type witness and associated conformance projection to map the result into the archetype's generic environment. To recap, when the subject type of an abstract conformance is a type parameter `T`, the type witness for `[P]A` is a dependent member type `T.[P]A`, and the associated conformance for $[\texttt{Self.U: Q}]_\texttt{P}$ is the abstract conformance $[\texttt{T.U: Q}]$. So, when the subject type is an archetype $[\![\texttt{T}]\!]$, we must also map this type witness or associated conformance into the environment:

$$\langle\texttt{P|A} \otimes [[\![\texttt{T}]\!]\texttt{: P}] := \mathsf{in}_G(\texttt{T.[P]A}) = [\![\texttt{T.[P]A}]\!]$$
$$\langle\texttt{Self.U: Q}] \otimes [[\![\texttt{T}]\!]\texttt{: P}] := \langle\texttt{Q}] \otimes \mathsf{in}_G(\texttt{T.U}) = [[\![\texttt{T.U}]\!]\texttt{: Q}]$$

8.2. Primary Archetypes

Recall that $\textsc{Sub}(G, H)$ is set of substitution maps with input generic signature G and output generic signature H, that is, their replacement types are interface types. To understand how type substitution behaves when the original type is a contextual type, we define a new form of our $\otimes$ operator:

$$\textsc{Type}([\![G]\!]) \otimes \textsc{Sub}(G, H) \longrightarrow \textsc{Type}(H)$$

To apply a substitution map Σ to a contextual type $\mathtt{Y} \in \textsc{Type}([\![G]\!])$, we first map the contextual type out of its environment, and then apply the substitution map to this interface type:

$$\mathtt{Y} \otimes \Sigma = \mathsf{out}_G(\mathtt{Y}) \otimes \Sigma$$

If the replacement types of Σ are interface types, we always get an interface type back, regardless of whether the original type was an interface type or a contextual type.

Substitution maps can also have contextual replacement types. This will be important in Section 9.3, so we will introduce the notation now. We write $\textsc{Sub}(G, [\![H]\!])$ for the set of substitution maps whose replacement types are drawn from $\textsc{Type}([\![H]\!])$. Now, if Σ has contextual replacement types, we always get a contextual type back, regardless of whether the original type was an interface type or contextual type:

$$\textsc{Type}(G) \otimes \textsc{Sub}(G, [\![H]\!]) \longrightarrow \textsc{Type}([\![H]\!])$$
$$\textsc{Type}([\![G]\!]) \otimes \textsc{Sub}(G, [\![H]\!]) \longrightarrow \textsc{Type}([\![H]\!])$$

Substitution map composition generalizes like so:

$$\textsc{Sub}(F, G) \otimes \textsc{Sub}(G, [\![H]\!]) \longrightarrow \textsc{Sub}(F, [\![H]\!])$$
$$\textsc{Sub}(F, [\![G]\!]) \otimes \textsc{Sub}(G, [\![H]\!]) \longrightarrow \textsc{Sub}(F, [\![H]\!])$$

Forwarding substitution map. In the expression type checker, SILGen, and other places that contextual types appear, a special substitution map often comes in handy. If G is a generic signature, the *forwarding substitution map* of G, denoted $1_{[\![G]\!]}$, sends each generic parameter $\mathtt{\tau_d_i}$ of G to the corresponding contextual type $\mathsf{in}_G(\mathtt{\tau_d_i})$ in the primary generic environment of G:

$$1_{[\![G]\!]} := \{\ldots, \mathtt{\tau_d_i} \mapsto \mathsf{in}_G(\mathtt{\tau_d_i}), \ldots\}$$

Note that $1_{[\![G]\!]} \in \textsc{Sub}(G, [\![G]\!])$. The forwarding substitution map looks similar to the identity substitution map $1_G \in \textsc{Sub}(G, G)$ from Section 6.3. We recall this substitution map sends every generic parameter to itself:

$$1_G := \{\ldots, \mathtt{\tau_d_i} \mapsto \mathtt{\tau_d_i}, \ldots\}$$

If $\mathtt{X} \in \textsc{Type}(G)$ is an interface type and $\mathtt{Y} \in \textsc{Type}(\llbracket G \rrbracket)$ is a contextual type, we can apply 1_G and $1_{\llbracket G \rrbracket}$ to each one:

$$\mathtt{X} \otimes 1_G = \mathtt{X}$$
$$\mathtt{X} \otimes 1_{\llbracket G \rrbracket} = \mathsf{in}_G(\mathtt{X})$$
$$\mathtt{Y} \otimes 1_G = \mathsf{out}_G(\mathtt{Y}) \otimes 1_G = \mathsf{out}_G(\mathtt{Y})$$
$$\mathtt{Y} \otimes 1_{\llbracket G \rrbracket} = \mathsf{out}_G(\mathtt{Y}) \otimes 1_{\llbracket G \rrbracket} = \mathsf{in}_G(\mathsf{out}_G(\mathtt{Y})) = \mathtt{Y}$$

Or in other words:

- The **identity substitution map** 1_G leaves an interface type unchanged, while it maps a contextual type out of the environment.
- The **forwarding substitution map** $1_{\llbracket G \rrbracket}$ leaves a contextual type unchanged, while it maps an interface type into the environment.

We can use this to convert between substitution maps with contextual and interface replacement types, by composing the substitution map with the identity or forwarding substitution map on the right. If $\Sigma \in \textsc{Sub}(G, H)$ and $\Sigma' \in \textsc{Sub}(G, \llbracket H \rrbracket)$, we have:

$$\Sigma \otimes 1_{\llbracket H \rrbracket} \in \textsc{Sub}(G, \llbracket H \rrbracket)$$
$$\Sigma' \otimes 1_H \in \textsc{Sub}(G, H)$$

Furthermore, if $\mathtt{X} \in \textsc{Type}(G)$,

$$\mathtt{X} \otimes (\Sigma \otimes 1_{\llbracket H \rrbracket}) = \mathsf{in}_H(\mathtt{X} \otimes \Sigma)$$
$$\mathtt{X} \otimes (\Sigma' \otimes 1_H) = \mathsf{out}_H(\mathtt{X} \otimes \Sigma')$$

Finally, substitution maps support a **map replacement types out of environment** operation which is more direct than composing with the identity substitution map:

$$\mathsf{out}_H \colon \textsc{Sub}(G, \llbracket H \rrbracket) \longrightarrow \textsc{Sub}(G, H)$$

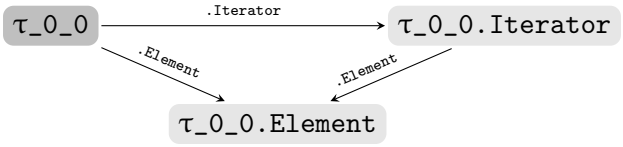
Invariants. In the implementation, a pair of predicates distinguish interface types from contextual types:

- `hasTypeParameter()` tests if the type contains type parameters.
- `hasPrimaryArchetype()` tests if the type contains primary archetypes.

These predicates are used to check preconditions in cases where a piece of compiler logic only expects to be called with an interface type or a contextual type, but not vice versa. Generally, the predicate asserted is the negation of the *opposite* condition: logic that expects only interface types should assert that the given type does not contain archetypes, while logic that expects only contextual types should assert that the type does not contain a type parameter.

8.3. The Type Parameter Graph

Let G be the generic signature `<τ_0_0 where τ_0_0: Sequence>`, and consider the type parameter `τ_0_0.Element`. One way to think about the operation $\mathsf{in}_G(\texttt{τ_0_0.Element})$ is that we start from the archetype $[\![\texttt{τ_0_0}]\!]$, and then "jump" to $[\![\texttt{τ_0_0.Element}]\!]$ in one step, labeled "`.Element`". Similarly, to find $\mathsf{in}_G(\texttt{τ_0_0.Iterator.Element})$, we start from $[\![\texttt{τ_0_0}]\!]$, take a step to $[\![\texttt{τ_0_0.Iterator}]\!]$, and then take one more step, at which point we again end up at $[\![\texttt{τ_0_0.Element}]\!]$, because `τ_0_0.Element` is the reduced type of `τ_0_0.Iterator.Element`:



We will define an object called the *type parameter graph* of a generic signature, such that valid type parameters are *paths* in this graph, and equivalence classes of type parameters, or archetypes, are the *vertices*. Two equivalent type parameters define a pair of paths with the same destination vertex. We will have occasion to study other directed graphs later, so as usual we begin with the abstract definitions.

Definition 8.1. A *directed graph* is a pair (V, E) consisting of a set of vertices V together with a set of edges E, where every edge $e \in E$ has an associated *source* vertex and a *destination* vertex, denoted $\mathrm{src}(e)$ and $\mathrm{dst}(e)$, respectively.

Some books (for example, [81]) define directed graphs such that an edge is exactly an ordered pair of vertices, $(\mathrm{src}(e), \mathrm{dst}(e))$. This disallows graphs where two edges share the same source and destination. Our formulation is more general because the source and destination are merely *properties* of an edge, which may also have an additional *label* of its own. This is called a *directed multi-graph* in [101].

A finite directed graph can be visualized by plotting each vertex as a point, and each edge as an arrow pointing from the source towards the destination. If either V or E is infinite, then we say that (V, E) is an *infinite graph*. We can still visualize some finite *subgraph* (V', E'), where $V' \subseteq V$ and $E' \subseteq E$, if the remaining structure is understood somehow.

Definition 8.2. Let (V, E) be a directed graph. A *path* $p := (v, (e_1, \ldots, e_n))$ starts at a source vertex $v \in V$ and follows a sequence of zero or more edges $(e_1, \ldots, e_n)$, not necessarily distinct, which must satisfy the following conditions:

1. If the path contains at least one edge, the source of the first edge must equal the source vertex: $\mathrm{src}(e_1) = v$.
2. If the path contains at least two edges, the source of each subsequent edge must equal the destination of the preceding edge: $\mathrm{src}(e_{i+1}) = \mathrm{dst}(e_i)$ for $0 < i \leq n$.

Each path also has a *source* and *destination*. If $p := (v, (e_1, \ldots, e_n))$ as above, its source is v, and the destination is either the source vertex v again if the sequence of edges is empty, or the destination of the final edge otherwise:

$$\mathrm{src}(p) := v$$
$$\mathrm{dst}(p) := \begin{cases} v & (\text{if } n = 0) \\ \mathrm{dst}(e_n) & (\text{if } n > 0) \end{cases}$$

The *length* of a path is the number of edges in the path. Every vertex $v \in V$ defines an empty path (of length zero), denoted 1_v, with source vertex v followed by an empty sequence of edges; by the above definitions, we have $\mathrm{src}(1_v) = \mathrm{dst}(1_v) = v$. Every edge e also defines a one-element path (of length one), with source vertex $\mathrm{src}(e)$ and destination vertex $\mathrm{dst}(e)$. We can also denote the one-element path as e, because $\mathrm{src}(e)$ and $\mathrm{dst}(e)$ have the same meaning whether we interpret e as an edge or a path.

Definition 8.3. We use the following terminology throughout:

- A *cycle* is a non-empty path with the same source and destination vertex.
- A directed graph is *acyclic* if it does not contain cycles.
- A *tree* is a directed graph with a distinguished root vertex, and the property that there is exactly one unique path from the root to every other vertex. Note that every tree is acyclic.

We observe that this definition of a tree matches the concept familiar to programmers.

We now have enough graph theory to define our main object of interest.

Definition 8.4. Let G be a generic signature. The *type parameter graph* of G is the directed graph constructed as follows:

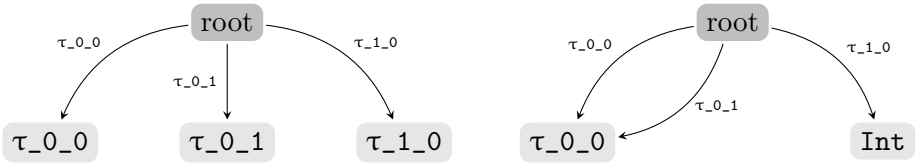
- The vertex set contains a distinguished root vertex.
- The vertex set also contains a vertex for each equivalence class of type parameters of G. Each such vertex is labeled with the reduced type of the equivalence class (which might be a concrete type).
- The edge set contains an edge for each generic parameter `τ_d_i` of G. The source vertex of each such edge is the root, and the destination vertex the equivalence class of the corresponding generic parameter type.
- Additional edges join those pairs of vertices where the source is the equivalence class of some type parameter `U`, and the destination is the equivalence class of a dependent member type `U.A` with base type `U`. Each such edge is labeled with the name of the associated type "`.A`".

The key fact about this graph is that every valid type parameter `τ_d_i.A`$_1$`...A`$_n$ defines a path from the root vertex to the type parameter's equivalence class, if we follow the edges labeled "`τ_d_i`" and "`.A`$_1$" and so on. Notice that two type parameters belong to the same equivalence class if and only if the corresponding paths end at the same vertex. If a type parameter is fixed to a concrete type, the type parameter's path ends at a vertex labeled by this concrete type.

Example 8.5. Consider these two generic signatures:

```
<τ_0_0, τ_0_1, τ_1_0>
<τ_0_0, τ_0_1, τ_1_0 where τ_0_0 == τ_0_1, τ_1_0 == Int>
```

The first signature has no requirements at all, and each generic parameter is in its own equivalence class. Its type parameter graph is a tree, shown on the left. The second signature collapses the first two and fixes the third to a concrete type. This time, the graph is not a tree, because we have two paths from the root that end at the same vertex.

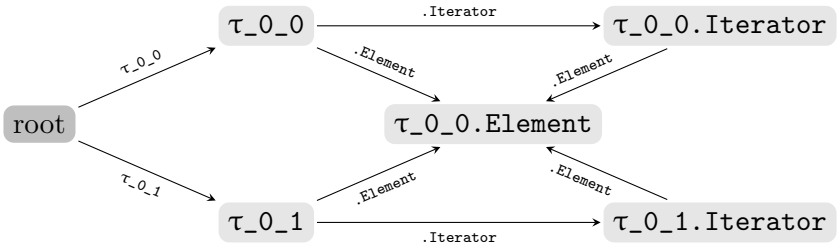


Without conformance requirements, there isn't much more we can do, as our supply of vertices is limited. We can generate more vertices by adding conformance requirements to protocols with associated types.

Example 8.6. We're going to revisit the generic signature from Example 5.2, but let's drop the [`τ_0_0.Element: Equatable`] requirement (the `Equatable` protocol does not declare any associated types, so this requirement does not generate any new vertices or edges in our graph):

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
                    τ_0_0.Element == τ_0_1.Element>
```

We already studied the equivalence class structure of this generic signature extensively. The type parameter graph gives us a new way to see the member type relationships:

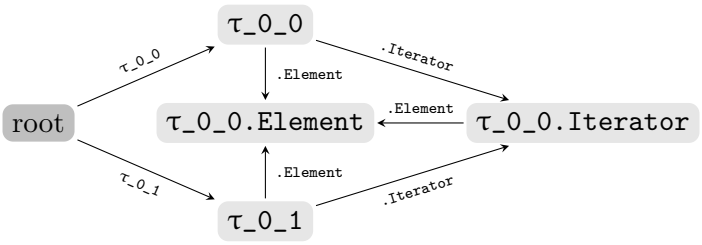


Notice how the paths corresponding to each one of the type parameters `τ_0_0.Element`, `τ_0_1.Element`, `τ_0_0.Iterator.Element`, and `τ_0_1.Iterator.Element` all have the same destination, because they belong to the same equivalence class ⟦`τ_0_0.Element`⟧.

Example 8.7. Now, recall this generic signature from Example 5.11:

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
                    τ_0_0.Iterator == τ_0_1.Iterator>
```

Here is its type parameter graph:



From the above graph, we see that the type parameters `τ_0_0.Element`, `τ_0_1.Element`, `τ_0_0.Iterator.Element`, and `τ_0_1.Iterator.Element` are also equivalent in this signature, just as in the previous example. Here though, the paths for `τ_0_0.Iterator` and `τ_0_1.Iterator` also end at the same vertex.

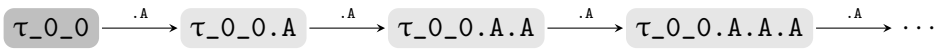
In each of the remaining examples, our generic signature only has a single generic parameter `τ_0_0`, so to simplify the presentation, we omit the distinguished root vertex from the graph. Instead, we say that the equivalence class of `τ_0_0` is the root. This changes our formulation slightly, for now the path corresponding to a type parameter has one fewer step; there is no edge corresponding to `τ_0_0`.

The next example exhibits an infinite type parameter graph. In any given compilation session, the set of archetypes that are actually instantiated will form an arbitrarily large but finite subgraph of the generic signature's type parameter graph.

Example 8.8. Recall this protocol from Example 5.12:

```
protocol N {
  associatedtype A: N
}
```

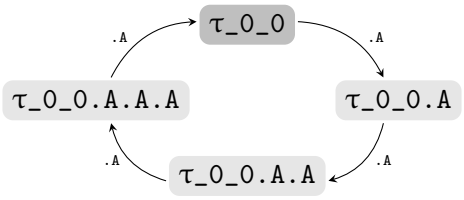
We saw that the protocol generic signature $G_{\texttt{N}}$ (that is, `<τ_0_0 where τ_0_0: N>`) defines an infinite set of equivalence classes. Its type parameter graph is called a *ray*, or an infinite path:



Example 8.9. Now let's change `N` by adding an associated same-type requirement:

```
protocol Z4 {
  associatedtype A: Z4 where Self == Self.A.A.A.A
}
```

Just like $G_{\texttt{N}}$, the protocol generic signature $G_{\texttt{Z4}}$ still defines an infinite set of valid type parameters, but the set of equivalence classes is now finite, so we're back in the world of finite graphs. The type parameter graph is the *cycle graph* of order 4:

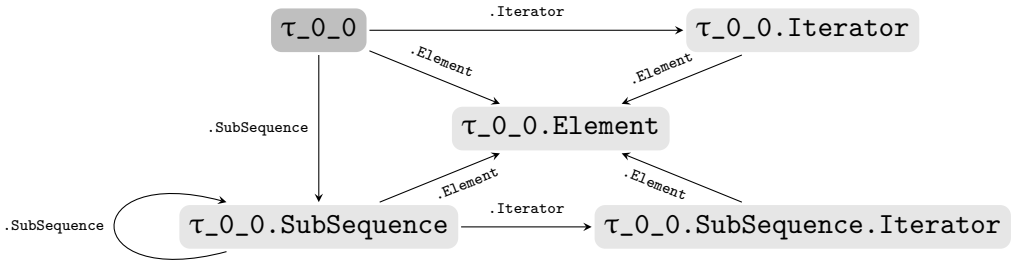


For example, `τ_0_0.A` and `τ_0_0.A.A.A.A.A` are equivalent in this generic signature. Notice how if a generic signature has an infinite theory but only finitely many equivalence classes, a simple counting argument shows that at least one equivalence class must be infinite. In $G_{\texttt{Z4}}$, *every* equivalence class is infinite.

Example 8.10. Recall the simplified `Collection` protocol from Example 5.13:

```
protocol Collection: Sequence {
  associatedtype SubSequence: Collection
      where Element == SubSequence.Element
      where SubSequence == SubSequence.SubSequence
}
```

When we studied the protocol generic signature $G_{\texttt{Collection}}$, we saw that it defines an infinite theory, but only a finite set of equivalence classes. The type parameter graph looks like this:



The equivalence class $[\![\texttt{τ_0_0.SubSequence}]\!]$ contains infinitely many type parameters, and just like $G_{\texttt{Z4}}$, the type parameter graph of $G_{\texttt{Collection}}$ contains a cycle of length 1 (sometimes known as a *loop*), which we can take any number of times to generate more type parameters in this equivalence class:

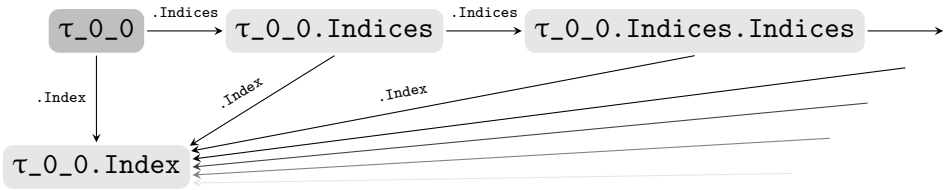
```
τ_0_0.SubSequence
τ_0_0.SubSequence.SubSequence
τ_0_0.SubSequence.SubSequence.SubSequence
...
```

We get the other infinite equivalence classes by taking the cycle zero or more times, followed by some other edge. In general, if a generic signature has an infinite theory but a finite set of equivalence classes, the type parameter graph must contain a cycle. However, this is no longer the case if the set of equivalence classes is itself infinite.

Example 8.11. The real `Collection` protocol declares two more associated types, `Index` and `Indices`, and imposes some requirements on them, which boil down to:

```
protocol Indexable {
  associatedtype Index
  associatedtype Indices: Indexable
    where Index == Indices.Index
}
```

The protocol generic signature $G_{\texttt{Indexable}}$ defines an infinite set of equivalence classes, and the equivalence class of `τ_0_0.Index` is infinite, and yet the graph is acyclic:



If v and w are vertices in some directed graph (V, E), we say that v is a *successor* of w if there is an edge $e \in E$ having $\mathrm{src}(e) = v$ and $\mathrm{dst}(e) = w$. Likewise, w is a *predecessor* of v. A vertex in the type parameter graph always has a finite set of successors, because all edges that share a source vertex correspond to distinct associated type declarations. The `Indexable` example above shows that we cannot make the same claim about the *predecessor* set of a vertex, because the vertex $[\![\texttt{τ_0_0.Index}]\!]$ has an infinite set of predecessors in $G_{\texttt{Indexable}}$.

Definition 8.12. A directed graph (V, E) is *locally finite* if every vertex has a finite set of successors, so for every $v \in V$, there are finitely many $e \in E$ with $\mathrm{src}(e) = v$.

In Section 12.2, we will meet another locally finite directed graph that we can associate with a generic signature, called the *conformance path graph*.

8.4. The Archetype Builder

A generic environment faithfully represents the equivalence classes of a generic signature: the in_G operation encodes the reduced type equality relation, and the local requirements of its archetypes encode everything else. Today, a generic environment is a *derived* representation, in the sense that it sits on top of generic signature queries. This was not always the case, and back in Swift 3, archetypes were the fundamental primitives. In this section, we will describe the `ArchetypeBuilder` algorithm that was used to build them. While now obsolete and no longer part of the implementation, the algorithm has a certain elegance to it, and understanding it will motivate the material in Part IV.

We can group generic signatures into three families of increasing complexity:

1. Generic signatures with a *finite theory*, that is a finite set of derived requirements and valid type parameters. Example 8.6 is a member of this family.
2. Generic signatures with an infinite theory, but a *finite set of equivalence classes* of type parameters. Example 8.9 and Example 8.10 belong to this family.
3. Generic signatures with an *infinite set of equivalence classes*. This family includes Example 8.8 and Example 8.11, and we will later see it is the most difficult.

Swift 3 could only accept the first family, because protocols were prohibited from stating recursive conformance requirements on their associated types (so today's `Collection` protocol could not be expressed). In general, the associated requirements of a protocol took on a much simpler form; protocols could state inheritance relationships, and they could impose conformance requirements on their associated types, but associated same-type requirements were not supported. Also, while Swift 3 did allow superclass and concrete same-type requirements, we're going to ignore them because they only involve a little bit of additional bookkeeping and are not central to the algorithm.

Potential archetypes. The algorithm represents a partially-formed equivalence class of type parameters using a data structure called a *potential archetype*. In memory, a potential archetype t stores these four fields:

`FORWARD`(t)	An optional pointer to another potential archetype.
`TYPE`(t)	A type parameter. This field is immutable after creation.
`PROTO`(t)	A set of protocol types.
`MEMBERS`(t)	A set of (identifier, pointer to potential archetype) pairs.

A potential archetype t always represents a fixed type parameter `T`, so that `TYPE`(t) = `T` for the lifetime of t. In the initial state, we have `FORWARD`(t) = null, `PROTO`$(t) = \varnothing$, and `MEMBERS`$(t) = \varnothing$. Potential archetypes are allocated sequentially while the algorithm runs, and then freed all at once when the entire archetype builder instance is freed.

The archetype builder establishes the following invariant. Any potential archetype with a null forwarding pointer represents the reduced type parameter of its equivalence class. In this case, $\texttt{PROTO}(t)$ is the set of all protocols this equivalence class conforms to, and $\texttt{MEMBERS}(t)$ lists all of the successors of the equivalence class in the type parameter graph. Otherwise, if a potential archetype t has a non-null forwarding pointer, it points to some other potential archetype in the same equivalence class, and the other potential archetype precedes it in type parameter order; that is, $\texttt{TYPE}(\texttt{FORWARD}(t)) < \texttt{TYPE}(t)$. By following this chain, we eventually end up at the reduced type parameter.

We now describe how this invariant is established, then show how the resulting data structure can implement generic signature queries.

Eager expansion. The archetype builder begins by creating a potential archetype for each declared generic parameter; these are called *root* potential archetypes. Potential archetypes that correspond to dependent member types come into existence from the *expansion* of conformance requirements. The expansion of a conformance requirement is the first of the two principal operations on potential archetypes. A protocol can impose conformance requirements on its associated types, so this expansion process is recursive. Notice how in Step 6b, we must catch and diagnose recursive conformance requirements to protocols we've already seen, to guarantee termination; this is of course the major limitation with the "eager expansion" approach.

Algorithm 8A (Expand conformance requirement)**.** Takes a potential archetype t and a protocol `P` as input. Has side effects.

1. If $\texttt{FORWARD}(t)$ is non-null, load $t \leftarrow \texttt{FORWARD}(t)$, and repeat if necessary.
2. If $\texttt{P} \in \texttt{PROTO}(t)$, return.
3. Otherwise, set $\texttt{PROTO}(t) \leftarrow \texttt{PROTO}(t) \cup \{\texttt{P}\}$.
4. For each protocol `Q` appearing in the inheritance clause of the declaration of `P`, recursively apply this algorithm to t and `Q`.
5. Let $\texttt{T} := \texttt{TYPE}(t)$.
6. For each associated type declaration `A` of `P`:
   a) If $\texttt{MEMBERS}(t)$ does not have an entry for `A`, create a new potential archetype v with $\texttt{TYPE}(v) = \texttt{T.A}$. Add the ordered pair $(\texttt{A}, v)$ to $\texttt{MEMBERS}(t)$.

   Otherwise, $\texttt{MEMBERS}(t)$ contains an existing entry $(\texttt{A}, v)$ with $\texttt{TYPE}(v) = \texttt{T.A}$.

   b) For each protocol `Q` listed in the inheritance clause of the declaration of `A`, first check if `Q` already appears among the set of all protocols we've visited in prior recursive calls; if so, emit a diagnostic. Otherwise, add `Q` to the visited set, and recursively apply this algorithm to v and `Q`.

The second principal operation is that of merging two potential archetypes, to form a larger equivalence class. This implements same-type requirements.

Algorithm 8B (Merge potential archetypes)**.** Receives a pair of potential archetypes t and u as input. Has side effects.

1. If either $\texttt{FORWARD}(t)$ or $\texttt{FORWARD}(u)$ is non-null, follow the forwarding pointers.
2. If t and u are equal pointers to the same potential archetype, return.
3. If $\texttt{TYPE}(u) < \texttt{TYPE}(t)$ under Algorithm 5D, first swap t and u. (This step is what eventually guarantees that $\texttt{FORWARD}(t) = \text{null}$ if and only if $\texttt{TYPE}(t)$ is a reduced type parameter.)
4. Set $\texttt{FORWARD}(u) \leftarrow t$.
5. Set $\texttt{PROTO}(t) \leftarrow \texttt{PROTO}(t) \cup \texttt{PROTO}(u)$.
6. Let $\texttt{T} := \texttt{TYPE}(t)$.
7. For each entry $(\texttt{A}, w) \in \texttt{MEMBERS}(u)$,
   a) If $\texttt{MEMBERS}(t)$ does not contain a potential archetype for $\texttt{A}$, create a new potential archetype v such that $\texttt{TYPE}(v) = \texttt{T.A}$. Add $(\texttt{A}, v)$ to $\texttt{MEMBERS}(t)$.
   b) Otherwise, $\texttt{MEMBERS}(t)$ contains an existing entry $(\texttt{A}, v)$ with $\texttt{TYPE}(v) = \texttt{T.A}$.
   c) In either case, invoke this algorithm recursively to merge v and w.

We need one more utility subroutine to actually resolve an arbitrary type parameter to a potential archetype, starting from the roots:

Algorithm 8C (Resolve potential archetype)**.** Receives the array of root potential archetypes, and a type parameter $\texttt{T}$. Returns the potential archetype for $\texttt{T}$, or null if one has not been created.

1. If $\texttt{T}$ is a generic parameter, let t be the corresponding root potential archetype. If $\texttt{FORWARD}(t)$ is non-null, load $t \leftarrow \texttt{FORWARD}(t)$, and repeat if necessary. Return t.
2. Otherwise, $\texttt{T}$ must be a dependent member type $\texttt{U.A}$ with some base type $\texttt{U}$ and identifier $\texttt{A}$. First, invoke the algorithm recursively to resolve $\texttt{U}$ to a potential archetype u.
3. If $\texttt{MEMBERS}(u)$ contains a potential archetype for $\texttt{A}$, let t be this potential archetype. If $\texttt{FORWARD}(t)$ is non-null, load $t \leftarrow \texttt{FORWARD}(t)$, and repeat if necessary. Return t.
4. Otherwise, return null. The type parameter $\texttt{T}$ is either invalid, or we haven't yet expanded a conformance requirement involved in its derivation.

We're now ready to look at the main algorithm. We visit each requirement written in source, expanding conformance requirements and merging pairs of potential archetypes corresponding to same-type requirements. There is a minor complication: requirements can be written in any order, so for example, if we process [`T.Element: Equatable`] before [`T: Sequence`], the potential archetype for `T.Element` may not have been created yet. This is allowed, and if Algorithm 8C returns null, we add the requirement to the "delayed requirements" list and re-process it again after attempting all other requirements.

Algorithm 8D (Historical `ArchetypeBuilder` algorithm)**.** Receives a list of generic parameters, requirements, and protocol declarations as input. Outputs an array of root potential archetypes. There are three intermediate data structures: a list of pending requirements, a list of delayed requirements, and a flag to record forward progress.

1. (Initialize) Add all requirements to the pending list. Set the flag to false. Create a root potential archetype for each generic parameter.
2. (Reprocess) If the pending list is empty and the flag is set, move any requirements from the delayed list to the pending list, and clear the flag.
3. (Check) If the pending list is still empty, go to Step 8.
4. (Resolve) Remove a requirement from the pending list.
5. (Conformance) For a conformance requirement [`T: P`], invoke Algorithm 8C to resolve `T` to a potential archetype. If this potential archetype does not exist, add [`T: P`] to the delayed list and set the flag. Otherwise, invoke Algorithm 8A to expand the conformance requirement.
6. (Same-type) For a same-type requirement [`T == U`], invoke Algorithm 8C to resolve each one of `T` and `U` to an potential archetype. If either potential archetype does not exist, add [`T == U`] to the delayed list and set the flag. Otherwise, invoke Algorithm 8B to merge the two potential archetypes.
7. (Repeat) Go back to Step 2.
8. (Diagnose) If we end up here, there are no more pending requirements. Any requirements remaining on the delayed list reference type parameters which cannot be resolved; diagnose them as invalid.

Once the algorithm returns, the potential archetype structure now encodes the finite type parameter graph built from the input requirements, assuming we didn't diagnose an invalid recursive conformance. At this point, no more expansion or merging takes place; the remaining potential archetypes, those without forwarding pointers, are "frozen" into immutable archetype *types* that describe this generic signature to the rest of the compiler. While generic signature queries did not exist back then, we can reconcile this algorithm with our present model by defining each generic signature query as follows:

- `isValidTypeParameter`(G, `T`): Resolve `T` to a potential archetype t, and check if t is non-null.
- `getReducedType`(G, `T`): Resolve `T` to a potential archetype t, and return `TYPE`(t).
- `areReducedTypeParametersEqual`(G, `T`, `U`): Resolve both `T` and `U` to potential archetypes t and u, and check if t and u are equal pointers to the same potential archetype.
- `requiresProtocol`(G, `T`, `P`): Resolve `T` to a potential archetype t, and check if `P` $\in$ `PROTO`(t).
- `getRequiredProtocols`(G, `T`): Resolve `T` to a potential archetype t, and return `PROTO`(t).

The archetype builder is an example of a *union-find* or *disjoint set* data structure (see, for example, Section 2.3.3 of [102]). An instance of this data structure defines an equivalence relation. Starting from a collection of singleton equivalence classes, the two fundamental operations are *union*, to merge two equivalence classes, and *find*, to resolve an element to its equivalence class. In our case, Algorithm 8B is *union*, and Algorithm 8C is *find* (whereas the expansion in Algorithm 8A is our own thing).

Our description of these algorithms omits some standard techniques for optimizing these operations; for example, when *find* follows a forwarding pointer, we usually want to write the pointer back to its original storage location as well, to eliminate repeated traversals on subsequent lookups.

This kind of data structure first appeared in the early history of compilers in the implementation of "`EQUIV` statements" for declaring that two symbols ought to share a storage location. These statements thus define an equivalence relation on symbols, and the performance gains from a careful implementation were noted in [103]. A survey of later techniques appears in [104].

Lazy expansion. From a theoretical point of view, the archetype builder's approach amounts to exhaustive enumeration of all derived requirements and valid type parameters of a generic signature, made slightly more efficient by the choice of data structure (the asymmetry in the handling of member types in Algorithm 8B means we skip parts of the search space that would yield nothing new).

The eager expansion model survived the introduction of protocol `where` clauses in Swift 4 [77], and thus associated requirements, with only relatively minor changes. The introduction of recursive conformances in Swift 4.1 [105] necessitated a larger overhaul. Once the type parameter graph becomes infinite, the eager conformance requirement expansion of Algorithm 8A no longer makes sense. The `ArchetypeBuilder` was renamed to the `GenericSignatureBuilder` as part of a re-design where the recursive expansion was now performed as needed, within the lookup of Algorithm 8C itself [106].

In the lazy expansion model, the potential archetype structure was highly mutable, as generic signature queries would create new potential archetypes and merge existing potential archetypes while exploring new subgraphs of the type parameter graph. The on-going mutation prevented sharing of structure between `GenericSignatureBuilder` instances, and every generic signature that depends on a complicated standard library protocol, such as `RangeReplaceableCollection`, would eventually construct its own copy of a large graph of potential archetypes. This would significantly impact compiler memory usage and performance.

Lazy expansion also suffered from correctness problems. Even the second family of generic signatures, those with an infinite theory but a finite set of equivalence classes, did not always work properly. Example 8.9 happened to be one such instance, and we will later see another in Example 18.10. An even more fundamental problem was discovered later. As we will see in Section 16.4, the third family of generic signatures, those having an infinite set of equivalence classes, actually contains generic signatures where the question of reduced type equality is *undecidable*. Such generic signatures must be rejected by any correct implementation of Swift. By virtue of its design, the `GenericSignatureBuilder` pretended to be able to accept any generic signature and answer queries about it. Lazy expansion was a dead end! This seemed surprising at the time, because in the absence of recursive conformances, eager expansion completely "solved" Swift generics in a straightforward way.

These problems motivated the search for a sound and decidable foundation on top of which Swift generics should be built, which became the Requirement Machine. Instead of incrementally constructing finite subgraphs of the type parameter graph, the correct approach is to construct a *convergent rewriting system*. While more abstract, this is actually much *simpler* than lazy expansion of the type parameter graph. Just like with the original eager expansion design, the rewriting system is a finite data structure that is built once and then remains immutable after construction. Unlike eager expansion, a rewriting system can describe an infinite set of equivalence classes, and in many cases, it can also encode a *finite* set without exhaustive enumeration. We will investigate the contemporary approach and study a correctness proof in Part IV.

8.5. Source Code Reference

`GenericEnvironment` *class*

A generic environment. Instances are allocated in the AST context, and passed by pointer. A flat array maps generic parameter types to archetypes, and a separate side table is allocated to store the mapping for dependent member types.

- `getGenericSignature()` returns this generic environment's generic signature.

- `mapTypeIntoContext()` returns the contextual type obtained by mapping an interface type into this generic environment.
- `getForwardingSubstitutionMap()` returns a substitution map for mapping each generic parameter to its contextual type—an archetype, or a concrete type if the generic parameter is fixed to a concrete type via a same-type requirement.
- `getOrCreateArchetypeFromInterfaceType()` is the private entry point which maps a single type parameter to an archetype, creating the archetype the first time. This method uses the `getLocalRequirements`() generic signature query.

`GenericSignature` *class*

See also Section 5.7.

- `getGenericEnvironment()` returns the primary generic environment associated with this generic signature.

`TypeBase` *class*

See also Section 3.4.

- `mapTypeOutOfContext()` returns the interface type obtained by mapping this contextual type out of its generic environment.

`SubstitutionMap` *class*

See also Section 6.6.

- `mapReplacementTypesOutOfContext()` returns the substitution map obtained by mapping this substitution map's replacement types and conformances out of their generic environment.

`ProtocolConformanceRef` *class*

See also Section 7.7.

- `mapConformanceOutOfContext()` returns the protocol conformance obtained by mapping this protocol conformance out of its generic environment.

`DeclContext` *class*

See also Section 4.6.

- `getGenericEnvironmentOfContext()` returns the generic environment of the innermost generic declaration containing this declaration context.

- `mapTypeIntoContext()` Maps an interface type into the primary generic environment for the innermost generic declaration. If at least one outer declaration context is generic, this is equivalent to:

```
dc->getGenericEnvironmentOfContext()->mapTypeIntoContext(type);
```

  For convenience, the `DeclContext` version of `mapTypeIntoContext()` also handles the case where no outer declaration is generic. In this case, it returns the input type unchanged, after asserting that it does not contain any type parameters (since type parameters appearing outside of a generic declaration are nonsensical).

`ArchetypeType` *class*

An archetype.

- `getName()` returns the name of this archetype, which is either the name of its generic parameter type or associated type declaration.
- `getFullName()` returns the "dotted" name of this archetype, which looks like a string representation of its type parameter.

Taking an archetype apart:

- `getInterfaceType()` returns the reduced type parameter of this archetype.
- `getGenericEnvironment()` returns the archetype's generic environment.
- `isRoot()` answers if the reduced type parameter is a generic parameter type.

Local requirements (Section 8.1):

- `getConformsTo()` returns the archetype's required protocols. This set does not include inherited protocols. To actually check if an archetype conforms to a specific protocol, use global conformance lookup (Section 7.7) instead of looking through this array.
- `getSuperclass()` returns the archetype's superclass bound, or the empty `Type` if there isn't one.
- `requiresClass()` answers with the requires class flag.
- `getLayoutConstraint()` returns the layout constraint, or the empty layout constraint if there isn't one.

`PrimaryArchetypeType` *class*

Subclass of `ArchetypeType` representing a primary archetype.

9. Type Resolution

Type resolution transforms the syntactic type representations produced by the parser into the semantic types of Chapter 3. Type representations have a tree structure. The leaf nodes are *identifier type representations* without generic arguments, such as `Int`. Nodes with children include *member type representations* which recursively store a base type representation, such as `T.Element`. There are also type representations for function types, metatypes, tuples, and existentials; they have children and follow the same shape as the corresponding kind of type. Finally, identifier and member type representations may have children in the form of generic arguments, such as `Array<Int>`. One of our main goals in this chapter is to understand how type resolution forms a generic nominal type from a reference to a type declaration and a list of generic arguments.

Type resolution builds the *resolved type* by consulting the type representation itself, as well as contextual information describing where the type representation appears:

1. Identifier type representations are resolved by unqualified lookup of their identifier; this depends on the type representation's source location. For example, a type representation might name a generic parameter declared in the current scope.

2. Resolution of certain type representations is also sensitive to semantic position. For example, a function type representation appearing in parameter position resolves to a non-escaping function type unless annotated with the `@escaping` attribute; in any other position, a function type representation always resolves to an escaping function type. This behavior was introduced in Swift 3 [107].

We encode contextual information in the *type resolution context*, consisting of the below:

1. The *declaration context* where the type representation is written. This is passed to unqualified lookup for identifier type representations. Fully resolving dependent member types also requires the generic signature of this declaration context.

2. A *type resolution context* and a set of *type resolution flags* together encode the semantic position of the type representation inside its declaration context.

3. A *type resolution stage* specifies if type resolution may query the generic signature of this declaration context. We use type resolution to build the generic signature, but type resolution uses the generic signature to resolve dependent member types and check generic arguments. The staged resolution breaks the request cycle by delaying certain semantic checks until the generic signature is available.

The two type resolution stages are *structural* resolution stage and *interface* resolution stage; we say we resolve a type *in* the given stage:

1. Structural resolution stage does not use the current declaration context's generic signature, and so it doesn't validate type parameters or check generic arguments.

2. Interface resolution stage requests the current context's generic signature first, and issues generic signature queries against this signature to perform semantic checks.

The **generic signature request** resolves various type representations so that it can build a generic signature from user-written requirements, as we will see in Chapter 11. This must be done in the structural resolution stage.

The **interface type request** resolves type representations in the interface resolution stage, to form a value declaration's interface type from semantically well-formed types. By resolving types in the interface stage, the **interface type request** depends on the **generic signature request**.

Structural resolution stage differs from interface resolution stage in two respects:

- References to associated types resolve to unbound dependent member types in the structural resolution stage, and references to invalid member types are not diagnosed. We describe this in Section 9.2.

- Generic arguments are not checked to satisfy the requirements of a generic nominal type in the structural resolution stage. We'll describe checking generic arguments in Section 9.3.

All invocations of type resolution "downstream" of the generics implementation must use the interface resolution stage, to not admit invalid types. To emit the full suite of diagnostics for type representations resolved in the structural stage, in particular inheritance clauses and trailing `where` clauses of generic declarations, the **type-check primary file request** revisits these type representations and resolves them again in the interface resolution stage.

While the structural resolution stage skips some semantic checks, it can still produce diagnostics; name lookup can fail to resolve an identifier, and certain simpler semantic invariants are still enforced, such as checking that the *number* of generic arguments is correct. For these reasons, care must be taken to not emit the same diagnostics twice if the same invalid type representation is resolved in both stages.

The general mechanism for this is to have each type representation store an "invalid" flag; after diagnosing an error, type resolution sets the invalid flag and returns an error type as the resolved type. If an invalid type representation is resolved again, type resolution immediately returns another error type without visiting the type representation or emitting any new diagnostics. A type representation can only transition from valid to invalid, and never back again.

9.1. Identifier Type Representations

An *identifier type representation* is a single identifier that names a type declaration in some outer scope. We find the *resolved type declaration* via unqualified lookup, starting from the source location of the type representation (Section 2.1). We then form the resolved type, which will be a nominal type, type alias type, generic parameter type, or dependent member type, depending on the type declaration's kind. We show some examples before describing the general principle.

Nominal types. A top-level non-generic nominal type declaration declares a single type, which we referred to as the declared interface type in Chapter 4. Here, the type representation resolves to the struct type `Int` declared by the standard library:

```
var x: Int = ...
```

If a nominal type declaration has a generic parameter list, it instead declares a new type for every possible list of generic arguments. Both identifier and member type representations may contain a list of generic arguments; we discuss how generic arguments are applied and checked in Section 9.3.

We allow the generic arguments to be omitted when the resolved type declaration's generic parameters are visible from the type representation's lexical scope; the generic *arguments* of the resolved type are taken to be the generic *parameters* of the resolved type declaration. In other words, the resolved type becomes the nominal type declaration's declared interface type, without a substitution map applied. For example:

```
struct Outer<T> {
  // Interface type of ‘x’ is ‘Optional<Outer<T>.Inner>’
  var x: Inner?

  class Inner {
    // Interface type of ‘y’ is ‘Outer<T>’
    var y: Outer
  }

  struct GenericInner<U> {
    // Return type of ‘f’ is ‘Outer<T>.GenericInner<U>’
    func f() -> GenericInner {}
  }
}
```

Section 9.4 describes another special case where generic arguments can be omitted when referencing a generic nominal type.

Recall from Section 2.1 and Chapter 4 that unqualified lookup visits each outer scope in turn, and if that scope is a nominal type declaration, we attempt a qualified lookup with this nominal type as the base type. If this base type is a class type, qualified lookup also walks up the superclass hierarchy.

Thus, our identifier type representation might refer to a member type of a superclass of some outer nominal type declaration. In this case, the declared interface type will be written using type parameters that are not visible in the current scope. To get the final resolved type, we apply a substitution map to the declared interface type.

If we found a member of the base type's immediate superclass, we use the context substitution map of the superclass type. (In the general case, we use Algorithm 6C from Section 6.4). In the below example, unqualified lookup of `Inner` inside `Derived` finds the member of `Base`. The generic parameter `T` of `Base` is always `Int` in `Derived`, so the superclass substitution map we apply to members of `Base` when seen from `Derived`, is $\{\texttt{T} \mapsto \texttt{Int}\}$:

```
class Base<T> {
  struct Inner {}
}

class Derived: Base<Int> {
  // Interface type of ‘x’ is ‘Base<Int>.Inner’
  var x: Inner = ...
}
```

Note that qualified lookup also visits conformed protocols. While we do not allow nominal type declarations to nest inside protocols and their extensions, we can observe associated type and type alias declarations this way. We will see how this is expressed with a substitution map in the next section.

It is also possible that a protocol or protocol extension states a superclass requirement on `Self`. In this case, the members of the superclass are again visible to unqualified lookup. The superclass bound of `Self` cannot involve itself or its member types, so it must be a fully-concrete type. We then proceed as if we did a qualified lookup into this concrete type instead. To continue our example, we can reference `Base.Inner` from a protocol extension of `Proto` where `Self` inherits from `Derived`:

```
protocol Proto: Derived {}

extension Proto {
  // Return type of ‘f’ is ‘Base<Int>.Inner’
  func f() -> Inner {}
}
```

Generic parameters. Unqualified lookup will find generic parameter declarations if any of the outer declaration contexts have generic parameter lists.

```
struct G<T> {
  func f<U>(...) {
    var x: T = ...    // Canonical type of 'x' is τ_0_0
    var y: U = ...    // Canonical type of 'y' is τ_1_0
  }
}
```

The resolved type is the declared interface type of the generic parameter declaration, which is the corresponding generic parameter type.

The identifier Self. Speaking of `Self`, inside a protocol or protocol extension this refers to the implicit generic parameter named `Self`, also known as `τ_0_0` (Section 4.3), which we resolve like a reference to any other named generic parameter:

```
protocol Proto {
  // Return type of 'f' is the 'Self' generic parameter of 'Proto'
  func f() -> Self
}
```

Inside the source range of a struct or enum declaration or an extension thereof, `Self` is shorthand for the declared interface type of this nominal type declaration. This is not a generic parameter type at all, but rather a nominal type, generic or non-generic:

```
struct Outer<T> {
  // Return type of 'f' is 'Outer<T>'
  func f() -> Self {}
}
```

Inside the source range of a class declaration or an extension of one, `Self` stands for the dynamic `Self` type, wrapping the declared interface type of the class:

```
extension Outer {
  class InnerClass {
    // Return type of 'f' is the dynamic Self type of
    // 'Outer<T>.InnerClass'
    func f() -> Self {}
  }
}
```

We previously described the dynamic `Self` type in Section 3.3. Historically, Swift only had `Self` in protocols and dynamic `Self` in classes, and the latter could only appear in the return type of a method. Swift 5.1 introduced the ability to state dynamic `Self` in more positions, and also refer to "static" `Self` inside struct and enum declarations [108].

Type aliases. Identifier type representations can also refer to type alias declarations, generalizing the behavior described for nominal type declarations above. Once again, we take the declared interface type of the type alias declaration, and possibly apply a substitution map. While the declared interface type of a nominal type declaration is a nominal type, the declared interface type of a type alias declaration is a type alias type. This is a sugared type, canonically equal to the underlying type of the type alias declaration.

If the named type alias declaration is in a local context, the resolved type is the declared interface type, with no substitution map applied:

```
func f<T>() {
  typealias A = (Int, T)

  // Interface type of ‘a’ is canonically equal to ‘(Int, T)’
  let a: A = ...
}
```

If the resolved type alias declaration is a member of a superclass of some outer type declaration, we must apply a substitution map, just like we do when resolving a nominal type found inside a superclass:

```
class Base<T> {
  typealias InnerAlias = T?
}

class Derived: Base<Int> {
  // Return type of ‘f’ is canonically equal to ‘Optional<Int>’
  func f() -> InnerAlias {}
}
```

As explained in Section 6.2, a nominal type declaration cannot be a member of a protocol or protocol extension, but a type alias declaration can. We say it's a *protocol type alias*. Unqualified lookup will find such a type alias declaration from within the scope of any nominal type declaration that conforms to this protocol. We discuss protocol type aliases in the next section when we talk about member type representations.

Associated types. If the identifier type representation is located within the source range of a protocol or protocol extension, the protocol's associated type declarations are visible to unqualified lookup. The resolved type is the declared interface type of the associated type declaration, which is a dependent member type around "`Self`":

```
protocol Pair {
  associatedtype A
  associatedtype B

  // Interface type of ‘a’ is ‘Self.[P]A’
  var a: A { get }

  // Interface type of ‘b’ is ‘Self.[P]B’
  var b: B { get }
}
```

Associated type declarations are also visible from within the protocol's conforming types. Recall that associated types can be witnessed by generic parameters, member type declarations, or inference (Section 7.3). When the type witness is a generic parameter or member type, unqualified lookup will always find the witness *before* the associated type declaration. However, if the type witness is inferred, unqualified lookup will find the associated type declaration:

```
struct S: Pair {
  // Explicit type witness:
  typealias A = Int

  // Inferred type witness:
  // typealias B = String

  var a: A  // resolved type is canonically equal to ‘Int’
  var b: String  // ‘B == String’ is inferred from ‘b’

  // Return type of ‘f’ and ‘g’ is canonically equal to ‘String’
  func f() -> B {}
  func g() -> B {}
}
```

Note that `f()` and `g()` have the same return type, but the two type representations are resolved in a slightly different manner. Let's assume that we compute the interface type of `f()` first (but the other order might also arise, depending on how the remainder of the program is structured).

Resolving the return type of `f()`, we find the associated type declaration `B` of `Pair`. Type resolution performs a global conformance lookup to find the concrete conformance [`S: Pair`], then projects the type witness for `B` from this conformance. This evaluates the **type witness request**, which *synthesizes* the type alias `B` inside `S`. The declared interface type of `S.B`, canonically equal to `String`, is returned as the type witness.

Resolving the return type of `g()`, we now find the type alias `S.B` we just synthesized. This shows that associated type inference has a side effect on the member lookup table of `S`, but it effectively acts as a form of lazy caching; the first lookup triggers the synthesis of this member. See Section 7.6 for further details about associated type inference.

Modules. A module declaration is a special kind of type declaration that can only be used as the base of a member type representation. Otherwise a bare module is an error:

```
_ = Swift.self  // error: expected module member name after module name
```

Of course `import` declarations are usually followed by a bare module name, but they are resolved by different means than type resolution.

Summary. If the resolved type declaration is in local context, or at the top level of a source file, the resolved type is just its declared interface type. Otherwise, we found the resolved type declaration by performing a qualified lookup into some outer nominal type or extension. In this case, the resolved type declaration might be a *direct* member of the outer nominal, or a member of a superclass or protocol. In the direct case, or if we started from a protocol and found a member of another protocol, we again return the member's declared interface type. Otherwise, we build a substitution map whose output generic signature is the generic signature of the outer nominal type or extension. We apply it to the member's declared interface type to get the resolved type:

- In the **superclass case**, we take the outer nominal's superclass bound and the member's parent class declaration, and build the superclass substitution map.
- In the **protocol case**, we take the outer nominal's declared interface type and the member's parent protocol, and build the protocol substitution map from the conformance. (We described this as projecting a type witness from a conformance instead of applying a substitution map; we'll say more about the protocol case in the next section, when we generalize these concepts further.)

Source ranges. In the scope tree, a nominal type or extension declaration actually defines *two* scopes, one nested within the other. The smaller source range contains the *body* only, from "`{`" to "`}`". The larger source range starts with the declaration's opening keyword, such as "`class`" or "`extension`", and continues until the "`}`". In particular,

the `where` clause is inside the larger source range, but outside of the smaller source range. Within a protocol or protocol extension, the protocol's associated type members (and type alias members, too) are always visible in both scopes:

```
// This is OK; 'Element' resolves to 'Self.[Collection]Element'
extension Collection where Element == Int {...}
```

Within a nominal type declaration, generic parameter declarations are visible in the larger scope, but member types can only be seen in the body:

```
// 'A' cannot be referenced here:
struct G<T> where A: Equatable {
  typealias A = T

  // 'A' can be referenced here:
  var a: A = ...
}
```

Prior discussion of name lookup from protocol contexts appeared in Section 4.3.

9.2. Member Type Representations

A *member type representation* consists of a *base* type representation together with an identifier, joined by "." in the concrete syntax. The base might be an identifier type representation, or recursively, another member type representation. The base may also have generic arguments. The general procedure for resolving a member type representation is the following. We start by recursively resolving the base type representation; then, we issue an qualified lookup (Section 2.1) to look for a member type declaration with the given name, inside the resolved base type. This finds the resolved type declaration, from which we compute the resolved type by applying a substitution map.

The resolved types obtained this way include nominal types, dependent member types, and type alias types. We classify the various behaviors by considering each kind of base type in turn, and describing its member types.

Module base. When the base is an identifier naming a module, the member type representation refers to a top-level type declaration within this module:

```
var x: Swift.Int = ...
```

The type declarations referenced this way are those that can appear at the top level of a module, namely nominal types and type aliases. The resolved type is the declared interface type of the declaration.

Type parameter base. As we alluded at the beginning of this chapter, the behavior of member type resolution with a type parameter base depends on the type resolution stage. We will begin with a description of the interface resolution stage, even though it happens last, because it implements the "complete" behavior.

Interface resolution stage. In this stage we interpret the type parameter relative to the current declaration context's generic signature. The requirements imposed on this type parameter give a list of protocols, and possibly a superclass bound, or a concrete type. The member type declarations of these types are the member type declarations of our type parameter. This includes:

1. Associated type declarations from all conformed protocols.
2. Type alias declarations from all conformed protocols and their extensions.
3. Member types of the concrete superclass type, if any.
4. If the type parameter is fixed to a concrete type via a same-type requirement, the members of this concrete type.

We get the list of types that qualified lookup must look inside by collecting the results of the `getRequiredProtocols`(), `getSuperclassBound`(), and `getConcreteType`() generic signature queries (Section 5.6).

The first case, where qualified lookup finds an associated type declaration, is extremely important; this is how we form type parameters recursively in type resolution. If some type parameter `T` conforms to a protocol, say `Provider`, and this protocol declares an associated type `Entity`, then type resolution of "`T.Entity`" will find the associated type declaration `Entity` by performing a qualified lookup into `Provider`.

```
protocol Provider {
  associatedtype Entity
}

struct G<T: Provider> {
  var x: T.Entity = ...
}
```

To obtain the resolved type `T.[Provider]Entity` from `Self.[Provider]Entity` (the declared interface type of the `Entity` associated type), we must replace `Self` with `T`. We do this by applying the protocol substitution map, which satisfies the protocol generic signature $G_{\texttt{Provider}}$ with the abstract conformance [`T: Provider`]:

$$\Sigma_{[\texttt{T: Provider}]} := \{\texttt{Self} \mapsto \texttt{T};\ [\texttt{Self: Provider}] \mapsto [\texttt{T: Provider}]\}$$

Applying $\Sigma_{\texttt{[T: Provider]}}$ to `Self.[Provider]Entity` projects the type witness from this conformance, with an equation we've seen a few times now, for example in Section 7.4:

$$\begin{aligned}\texttt{Self.[Provider]Entity} \otimes \Sigma_{\texttt{[T: Provider]}} &\\ &= \langle \texttt{Provider} | \texttt{Entity} \otimes [\texttt{Self: Provider}] \otimes \Sigma_{\texttt{[T: Provider]}} \\ &= \langle \texttt{Provider} | \texttt{Entity} \otimes [\texttt{T: Provider}] \\ &= \texttt{T.[Provider]Entity}\end{aligned}$$

The second case, where we found a protocol type alias as a member of the base, is closely related. The resolved type is again obtained by replacing all occurrences of `Self` in the underlying type of the type alias with the resolved base type of the member type representation. We're also going to make things slightly more interesting by using a dependent member type `T.Element` as the base type, rather than the generic parameter `T`:

```
protocol Subscriber {
  associatedtype Parent: Provider
  typealias Content = Self.Parent.Entity  // or just Parent.Entity
}

func process<T: Sequence>(_: T) where T.Element: Subscriber {
  // Interface type of 'x' is canonically equal to
  // 'T.Element.Parent.Entity'
  let x: T.Element.Content = ...
}
```

The above behavior can be explained by applying the protocol substitution map to the declared interface type of the type alias declaration:

$$\begin{aligned}\texttt{Self.[Subscriber]Parent.[Provider]Entity} \otimes \Sigma_{\texttt{[T.Element: Subscriber]}} &\\ &= \texttt{T.[Sequence]Element.[Subscriber]Parent.[Provider]Entity}\end{aligned}$$

In this example, the underlying type of the protocol type alias was a type parameter, but it can also be a concrete type that recursively contains type parameters, too.

References to associated type declarations and protocol type aliases are resolved by the same rule: we apply a protocol substitution map to the declared interface type of the resolved type declaration. We assumed the base type is a type parameter, so we form a protocol substitution map from an abstract conformance. We'll see shortly that when the base type is concrete, we can still reference associated type declarations and protocol type aliases, and we form a protocol substitution map from a concrete conformance instead.

Structural resolution stage. Before we talk about type parameter bases subject to superclass or concrete type requirements, let's take a moment to describe the structural resolution stage. In the structural resolution stage, we don't have a generic signature, so we cannot perform generic signature queries or qualified lookup. Instead, a member type representation with a type parameter base always resolves to an unbound dependent member type formed from the base type and identifier. First, we need an example where structural resolution actually takes place. We can take the `Provider` protocol declared prior, and write a function with a trailing `where` clause:

```
func f<T: Provider>(_: T) where T.Entity: Equatable {...}
```

The **generic signature request** resolves the type representation `T.Entity` in the structural resolution stage, to obtain the unbound dependent member type `T.Entity`. We have two user-written requirements, from which we build our generic signature:

{[`T: Provider`], [`T.Entity: Equatable`]}

Requirement minimization rewrites the second requirement into one that contains the bound dependent member type `T.[Provider]Entity`, and we get the following generic signature:

```
<T where T: Provider, T.[Provider]Entity: Equatable>
```

At this point, `f()` has a generic signature, so type representations appearing inside the function can be resolved in the interface resolution stage. The rewriting of requirements that use unbound dependent member types into requirements that use bound dependent member type swill be completely justified in Section 11.4.

We discussed bound and unbound dependent member types in Section 5.4. An unbound dependent member type from the structural resolution stage can be converted into a bound dependent member type by first checking the `isValidTypeParameter`() generic signature query, followed by `getReducedType`(). The first check is needed since an unbound dependent member type, being a syntactic construct, might not name a valid member type at all. However, the usual situation is that the entire type representation is resolved again in the interface resolution stage, at which point an invalid type parameter is resolved to an error type and a diagnostic is emitted. In particular, type representations in the trailing `where` clause of each declaration are re-visited by the **type-check primary file request**, which walks all top-level declarations in source order and emits further diagnostics.

Suppose now we change `f()`, to add the invalid requirement [`T.Foo: Equatable`]:

```
func f<T: Provider>(_: T)
    where T.Entity: Equatable, T.Foo: Equatable {...}
```

The invalid requirement will be dropped by requirement minimization, and the **generic signature request** does not emit any diagnostics. Instead, the invalid member type representation `T.Foo` will be diagnosed by **type-check primary file request**, because qualified lookup would fail to find a member type named `Foo` in `Provider` when we revisit the `where` clause in interface resolution stage. We will resume the discussion of how invalid requirements are diagnosed in Section 11.3.

Next, we look at this function `g()`, which references our previously-seen protocol type alias `Content` from its trailing `where` clause.

```
func g<T: Sequence>(_: T)
    where T.Element: Subscriber, T.Element.Content: Equatable {...}
```

Here, we form the requirement [`T.Element.Content: Equatable`] whose subject type is the unbound dependent member type `T.Element.Content`. Indeed, this is not a type alias type at all. As we will learn in Section 17.5, protocol type aliases introduce rewrite rules, and the computation of reduced types replaces the unbound dependent member type with the underlying type of the protocol type alias. In this case, we get a generic signature with a requirement having quite the long subject type,

[`T.[Sequence]Element.[Subscriber]Parent.[Provider]Entity: Equatable`].

Type aliases can also appear in protocol extensions. However, such aliases cannot be referenced from positions resolved in the structural resolution stage, and in particular, trailing `where` clauses. We will see later that type aliases from protocol extensions do not define rewrite rules, so the reduction described above cannot be performed. This situation is detected after the fact, when the type representation is revisited in the interface resolution stage:

```
extension Provider {
  typealias Object = Entity
}

// error: 'Object' was defined in extension of protocol 'Provider'
// and cannot be referenced from a 'where' clause
struct G<T: Provider> where T.Object: Equatable {}
```

Remaining cases. We now wrap up the case where we have a type parameter base, the type parameter is subject to a concrete same-type or superclass requirement, and the resolved type declaration is a member of this concrete type. We compute the resolved type from this type declaration by proceeding as if the base type was this concrete type or superclass bound instead of the type parameter spelled by the user. Once again, various limitations surface if this type representation is also resolved in structural resolution stage. The "concrete contraction" pass allows certain cases to work (Section 20.5).

Concrete base. Now, consider a member type representation where the base resolves to something that's not a type parameter. The only interesting possibilities are when the base type is a nominal type or dynamic `Self` type, because function types and such do not have member types. (The dynamic `Self` type is simply unwrapped to its underlying nominal type, allowing member types of the innermost class declaration to be referenced by "`Self.Foo`", which is perhaps slightly more explicit than the almost equivalent "`Foo`".)

Unlike the type parameter case, here we immediately perform a qualified lookup into the base type without consulting the generic signature of the current context. When both the base and the member type declaration are non-generic nominal types, and the member type is a direct member of the base, the resolved type is just the declared interface type of the member. The member might also be found in a superclass, and not as a direct member of the base. We saw an almost identical example in the previous section, where the reference to `Inner` was instead an identifier type representation inside the body of `Derived`:

```
class Base<T> {
  struct Inner {}
}

class Derived: Base<Int> {}

// Interface type of 'x' is Base<Int>.Inner
var x: Derived.Inner = ...
```

The superclass is generic, so we apply the superclass substitution map to the declared interface type of `Inner` to get the resolved type for `Inner` as a member of `Derived`:

$$\texttt{Base<T>.Inner} \otimes \{\texttt{T} \mapsto \texttt{Int}\} = \texttt{Base<Int>.Inner}$$

Next, suppose we wish to reference `Inner` directly, as a member of `Base`. The `Base` class is generic, and we haven't seen how generic arguments are applied when resolving a type representation yet, but for now assume we know how to resolve the base type here:

```
var y: Base<String>.Inner = ...
```

Given the generic nominal type `Base<String>`, we resolve the member type representation written above by applying the context substitution map of `Base<String>` to the declared interface type of `Inner`, which we can do because `Inner` has the same generic signature as `Base`:

$$\texttt{Base<T>.Inner} \otimes \{\texttt{T} \mapsto \texttt{String}\} = \texttt{Base<String>.Inner}$$

The substitutions above are trivial in a sense, because the referenced member is a nominal type declaration, and to compute the resolved type, we simply transfer over the generic arguments from the base type. However, when the member is a type alias declaration, we can actually encode an arbitrary substitution. Each choice of base type defines a possible substitution map, which is then applied to the underlying type of the type alias, which can be any valid interface type for its generic signature. When we introduced substitution maps in Chapter 6, we saw that resolving the type of a type alias member was one situation where substitution maps naturally appear in the language. We're going to look at some examples of type alias members now.

The type alias might be declared in a constrained extension of the base. If the constrained extension introduces new conformance requirements, the underlying type of the type alias declaration may reference new type parameters not present in the extended type's generic signature. Recalling that `Optional` type has a single generic parameter `Wrapped`, we declare the following constrained extension:

```
extension Optional where Wrapped: Sequence {
  typealias OptionalElement = Optional<Wrapped.Element>
}

// Interface type of ‘x’ is ‘Optional<Int>’
var x: Optional<Array<Int>>.OptionalElement = ...
```

The context substitution map of `Optional<Array<Int>>` is $\{$`Wrapped` $\mapsto$ `Array<Int>`$\}$, which is not a valid substitution map for the constrained extension's generic signature, because there's no conformance for the requirement [`Wrapped: Sequence`]. Applying this substitution map to the underlying type of our type alias would signal substitution failure and return the error type, because the substituted type for the type parameter `Wrapped.[Sequence]Element` cannot be resolved without this conformance:

$$\texttt{Optional<Wrapped.[Sequence]Element>} \otimes \{\texttt{Wrapped} \mapsto \texttt{Array<Int>}\} = \texttt{<<error type>>}$$

Instead we build the context substitution map of our base type, but using the generic signature of our constrained extension. This uses global conformance lookup to resolve the conformance:

$$\Sigma := \{\texttt{Wrapped} \mapsto \texttt{Array<Int>};\ [\texttt{Wrapped: Sequence}] \mapsto [\texttt{Array<Int>: Sequence}]\}$$

Now, applying Σ to the underlying type of our type alias outputs the expected result by projecting the type witness from the conformance:

$$\texttt{Optional<Wrapped.[Sequence]Element>} \otimes \Sigma = \texttt{Optional<Int>}$$

If instead we had a `Wrapped` type that does not conform to `Sequence`, for example as in `Optional<Float>.OptionalElement`, we again get a substitution failure so the resolved type becomes an error type. We will see in the next section how such type representations are diagnosed.

To complete the discussion of a member type representation with a concrete base, we finally consider the possibility that the named member is an associated type or protocol type alias declared in some conformed protocol. When resolving a protocol member with a type parameter base, we used the protocol substitution map constructed from the abstract conformance of the type parameter to the protocol. With a concrete base, we do the same thing, except we form the protocol substitution map from a *concrete* conformance.

In the below, `Tomato` conforms to `Plant`, so we can access the protocol type alias `Food` as a member of `Tomato`:

```
struct Ketchup {}
struct Pasta<Flavor> {}

protocol Plant {
  associatedtype Sauce
  typealias Food = Pasta<Sauce>
  consuming func process() -> Sauce
}

struct Tomato: Plant {
  consuming func process() -> Ketchup {...}
}

// Interface type of ‘x’ is canonically equal to ‘Ketchup’
var x: Tomato.Sauce = ...

// Interface type of ‘y’ is canonically equal to ‘Pasta<Ketchup>’
var y: Tomato.Food = ...
```

When we resolve the interface type of “`x`” and then “`y`”, we see that in both cases the resolved type declaration is a member of `Plant`, so we build the protocol substitution map $\Sigma_{[\texttt{Tomato: Plant}]}$ from the normal conformance [`Tomato: Plant`]:

$$\Sigma_{[\texttt{Tomato: Plant}]} := \{\texttt{Self} \mapsto \texttt{Tomato};\ [\texttt{Self: Plant}] \mapsto [\texttt{Tomato: Plant}]\}$$

To resolve the interface type of “`x`”, we apply $\Sigma_{[\texttt{Tomato: Plant}]}$ to `Self.[Plant]Sauce`, the declared interface type of `Sauce`. This is equivalent to projecting the type witness

for `Sauce` from this conformance, giving us the resolved type `Ketchup`:

$$\begin{aligned}&\texttt{Self.[Plant]Sauce} \otimes \Sigma_{[\texttt{Tomato: Plant}]} \\ &\quad = \langle \texttt{Plant} | \texttt{Sauce} \otimes [\texttt{Self: Plant}] \otimes \Sigma_{[\texttt{Tomato: Plant}]} \\ &\quad = \langle \texttt{Plant} | \texttt{Sauce} \otimes [\texttt{Tomato: Plant}] \\ &\quad = \texttt{Ketchup}\end{aligned}$$

To resolve the interface type of "`y`", where we have the type representation `Tomato.Food`, we apply $\Sigma_{[\texttt{Tomato: Plant}]}$ to `Pasta<Self.[Plant]Sauce>`, the underlying type of `Food`, which recursively transforms the type parameter contained therein:

$$\begin{aligned}&\texttt{Pasta<Self.[Plant]Sauce>} \otimes \Sigma_{[\texttt{Tomato: Plant}]} \\ &\quad = \texttt{Pasta<Ketchup>}\end{aligned}$$

Protocol base. This is really just a funny edge case. A protocol type alias is visible as a member type of the protocol type *itself* when its underlying type does not depend on the protocol `Self` type. The `Food` protocol type alias above cannot be referenced as a member of the protocol type `Plant`, because the underlying type of `Food` contains `Self`, and `Plant` is not a valid replacement type for `Self`; `Plant` does not conform to `Plant`! Type resolution must reject the member type representation "`Plant.Food`":

```
// error: cannot access type alias 'Food' from 'Plant'; use a
// concrete type or generic parameter base instead
var x: Plant.Food = ...
```

Similarly, an associated type member can *never* be referenced with a protocol base, like "`Plant.Sauce`"; its declared interface type *always* contains `Self`. However, if the underlying type of some protocol type alias does not contain `Self`, the type alias is just a shortcut for another globally-visible type that does not depend on the conformance. We allow the reference because no substitution needs to be performed:

```
protocol Pet {
  typealias Age = Int
}

// 'Pet.Age' is just another spelling for 'Int'
func celebratePetBirthday(_ age: Pet.Age) {}
```

Due to peculiarities of type substitution, protocol type aliases that are also generic are always considered to depend on `Self`, even if their underlying type does not reference `Self`, so they cannot be referenced with a protocol base. (In structural resolution stage, a generic type alias cannot be referenced with a type parameter base, either. Perhaps it is best not to stick generic type aliases inside protocols, at all.)

General principle. Let's say that H is the generic signature of the current context, and `T` is the resolved base type of our member type representation, obtained via a recursive call to type resolution. We perform a qualified lookup after considering the base type `T`:

- If `T` is a type parameter, we look through a list of nominal type declarations discovered by generic signature queries against `T` and H.
- If `T` is a nominal type, we look into the nominal type declaration of `T`.

If qualified lookup fails or finds more than one type declaration, we diagnose an error. Otherwise, let d be the resolved type declaration with declared interface type $\texttt{X}_d$ and generic signature G. We construct a substitution map $\Sigma \in \textsc{Sub}(G, H)$, then compute $\texttt{X}_d \otimes \Sigma$ to get the final resolved type. We build Σ from the base type `T` and information about the parent context of d. There are three cases to handle:

- In the **direct case**, d is a direct member of the nominal type declaration of `T`, and Σ is the context substitution map of `T`.
- In the **superclass case**, d is a member of a superclass of `T`, and Σ is the superclass substitution map formed from `T` and the parent class of d.
- In the **protocol case**, d is a member of some protocol `P`, either via conformance (if `T` is concrete) or protocol inheritance (if `T` is a type parameter), and Σ is the protocol substitution map $\Sigma_{[\texttt{T}:\ \texttt{P}]}$ for the conformance of `T` to `P`, found via global conformance lookup.

If the base type `T` is a type parameter subject to a concrete same-type requirement or a superclass requirement, we replace `T` with the corresponding concrete type obtained by a generic signature query against H before proceeding to compute Σ above.

In all three cases above, d might be defined in a constrained extension that imposes further conformance requirements. When building Σ, we resolve any of these additional conformances via global conformance lookup (Section 7.1).

This is called a *context substitution map for a declaration context*. This concept generalizes the context substitution map of a type from Section 6.1, which was an inherent property of a type, without reference to a declaration context. If `T` is a nominal type and d is a direct member of the nominal type declaration of `T`, the context substitution map of `T` for the parent context of d is simply the context substitution map of `T`.

The context substitution map of a type for a declaration context also arises when type checking member reference expressions, like "`foo.bar`", that appear in a function body. In this case, qualified lookup will find any member named `bar`, not just a type declaration, and the type of the expression is computed by applying the corresponding context substitution map to the interface type of `bar`.

History. In pre-evolution Swift, associated type declarations in protocols were declared with the `typealias` keyword, and protocol type aliases did not exist. Swift 2.2 added the distinct `associatedtype` keyword for declaring associated types, to make space for protocol type aliases in the language [109]. Protocol type aliases were then introduced as part of Swift 3 [110].

Caching the type declaration. Having computed the resolved type of an identifier or member type representation, we stash the resolved type declaration within our type representation, as a sort of cache. If the type representation is resolved again (perhaps once in the structural stage, and then again in the interface stage), we skip name lookup and proceed directly to computing the resolved type from the stored type declaration. The optimization was more profitable in the past, when type resolution actually had *three* stages. The third stage would resolve interface types to archetypes, but it has since been subsumed by the **map type into environment** operation on generic environments. We also pre-populate this cache when parsing textual SIL, by assigning a type declaration to certain type representations. Name lookup would otherwise not find these declarations, because of SIL syntax oddities that we're not going to discuss here.

9.3. Generic Arguments

Identifier and member type representations may be equipped with generic arguments, where each generic argument is recursively another type representation:

```
Array<Int>
Dictionary<String, Array<Int>>
Big<Int>.Small<String>
```

To resolve a type representation with generic arguments, we begin by finding the resolved type declaration using name lookup, and then perform a few additional steps not present in the non-generic case. Let's say that d is the resolved type declaration, G is the generic signature of d, and G' is the generic signature of the parent context of d. If the parent generic signature G' is empty, then we're referencing a top-level generic declaration, like `Array<Int>` for example. (If G is *also* empty, then d is not actually generic at all; we will diagnose below.) Notice there are several new semantic checks here, each of which can diagnose errors and return an error type:

1. We check that d has a generic parameter list, and ensure we have the correct number of generic arguments.

2. We recursively resolve all generic argument type representations, to form an array of generic argument types.

3. We form a substitution map Σ for G from these generic argument types.
4. We check that Σ satisfies all explicit requirements of G.
5. We compute the substituted type $\texttt{X}_d \otimes \Sigma$, where $\texttt{X}_d$ is the declared interface type of d, to get the final resolved type.

Let's say that H is the generic signature of the declaration context where the type representation appears. The check in Step 1 consults the generic parameter list of d; this is a syntactic construct describing the innermost generic parameters of G we met in Section 4.1. This does not require knowledge of either G or H, so we perform this check in both interface resolution stage and structural resolution stage. Checking requirements in Step 4 requires knowledge of both G and H, and we only do it in the interface resolution stage. How this checking is done is a topic unto itself; we will turn our attention to it shortly.

We want our resolved type to be an interface type for H, so $\texttt{X}_d \otimes \Sigma \in \textsc{Type}(H)$. Since $\texttt{X}_d \in \textsc{Type}(G)$, we wish to construct a $\Sigma \in \textsc{Sub}(G, H)$. In the previous section, we dealt with the case where d does not introduce any new generic parameters or requirements, but might be in a generic context. Back then, the resolved type was $\texttt{X}_d \otimes \Sigma'$ where Σ' is the context substitution map for `T` with respect to the declaration context of d. In the general case, $\Sigma' \in \textsc{Sub}(G', H)$ and not $\textsc{Sub}(G, H)$, but the "difference" between the two is that a substitution map for G must also fix d's innermost generic parameters and witness any new conformance requirements; so we form $\Sigma \in \textsc{Sub}(G, H)$ by adding to Σ' the generic argument types from Step 2, and resolving their conformances.

That is all a fancy way to say, the recursive nesting of type declarations gives rise to a recursive nesting of type representations. Type resolution forms a substitution map at each nesting level by adding new generic arguments to the previous substitution map.

Example 9.1. Suppose we're resolving the interface type of "`x`" below:

```
struct Big<T> {
  struct Small<U, V> {}
}

struct From<X: Sequence> {
  var x: Big<X.Iterator>.Small<X.Element, Int> = ...
}
```

To illustrate the connection between nested type declarations and generic parameter depth, we're going to work with canonical types. We are resolving a type representation inside `From`, and indeed the generic arguments contain type parameters from the generic signature of `From`, canonically `<τ_0_0 where τ_0_0: Sequence>`.

We first resolve the base type representation `Big<X.Iterator>`. The resolved type declaration is `Big`. Since the parent context of `Base` has an empty generic signature, the generic parameter of `Big` has the canonical type `τ_0_0`. We form a substitution map $\{\texttt{τ_0_0} \mapsto \texttt{τ_0_0.Iterator}\}$ from our generic argument. Substitution gives us the resolved base type:

$$\texttt{Big<τ_0_0>} \otimes \{\texttt{τ_0_0} \mapsto \texttt{τ_0_0.Iterator}\} = \texttt{Big<τ_0_0.Iterator>}$$

Next, we resolve the member type declaration `Small` via qualified lookup into the base type. The generic parameters `U` and `V` of `Small` appear at depth 1, so they declare the generic parameter types `τ_1_0` and `τ_1_1`. We take the context substitution map of the base type, which we saw is $\{\texttt{τ_0_0} \mapsto \texttt{τ_0_0.Iterator}\}$, and insert our generic arguments as the replacement types for `τ_1_0` and `τ_1_1`:

$$\begin{aligned}\Sigma := \{&\texttt{τ_0_0} \mapsto \texttt{τ_0_0.Iterator},\\ &\texttt{τ_1_0} \mapsto \texttt{τ_0_0.Element},\\ &\texttt{τ_1_1} \mapsto \texttt{Int}\}\end{aligned}$$

Substitution gives us the final resolved type:

$$\begin{aligned}&\texttt{Big<τ_0_0>.Small<τ_1_0, τ_1_1>} \otimes \Sigma\\ &\qquad = \texttt{Big<τ_0_0.Iterator>.Small<τ_0_0.Element, Int>}\end{aligned}$$

Note that the `τ_0_0` on the left-hand side is interpreted relative to the generic signature of the resolved type declaration, while `τ_0_0` on the right is relative to the generic signature of `From`.

Checking generic arguments. Returning to Step 4 from the beginning of the present section, we're given a substitution map $\Sigma \in \textsc{Sub}(G, H)$ and we must decide if Σ satisfies the requirements of G, the generic signature of the referenced type declaration. We apply Σ to each explicit requirement of G to get a series of *substituted requirements*. A substituted requirement is a statement about concrete types that is either true or false; we proceed to check each one by global conformance lookup, canonical type comparison, and so on, as will be described below. If any substituted requirements are unsatisfied, we diagnose an error. This checking of substituted requirements is a generally useful operation, used not only by type resolution; we discuss other applications at the end of the section.

Our generic argument types can contain type parameters of H (the generic signature of the current context), and checking a substituted requirement may raise questions about H. To avoid separately passing in H, we require that a substituted requirement's types are expressed in terms of *archetypes* instead (see Section 8.2 for a refresher). Thus, we begin by mapping Σ into the primary generic environment of H, and assume henceforth that $\Sigma \in \textsc{Sub}(G, [\![H]\!])$.

Definition 9.2. We denote by $\textsc{Req}(G)$ the set of all requirements whose left-hand and right-hand side types contain type parameters of G (all explicit and derived requirements of G are also elements of $\textsc{Req}(G)$, but there are many more). Similarly, let $\textsc{Req}(\llbracket H \rrbracket)$ denote the set of requirements written using the primary archetypes of H.

We then define *requirement substitution* as a new "overload" of $\otimes$:

$$\textsc{Req}(G) \otimes \textsc{Sub}(G, \llbracket H \rrbracket) \to \textsc{Req}(\llbracket H \rrbracket)$$

Requirement substitution must apply Σ to every type parameter appearing in a given requirement R by considering the requirement kind. In all of the below, `T` is the subject type of the requirement, so $\texttt{T} \in \textsc{Type}(G)$:

- For a **conformance requirement** `[T: P]`, we apply Σ to `T`. The protocol type `P` remains unchanged because it does not contain any type parameters:

  $$[\texttt{T: P}] \otimes \Sigma := [(\texttt{T} \otimes \Sigma)\texttt{: P}]$$

- For a **superclass requirement** `[T: C]`, we apply Σ to `T` as well as the superclass bound `C`, which might be a generic class type containing type parameters of G:

  $$[\texttt{T: C}] \otimes \Sigma := [(\texttt{T} \otimes \Sigma)\texttt{: } (\texttt{C} \otimes \Sigma)]$$

- For a **same-type requirement** `[T == U]`, we apply Σ to both sides; `U` is either a type parameter or a concrete type that may contain type parameters of G:

  $$[\texttt{T == U}] \otimes \Sigma := [(\texttt{T} \otimes \Sigma)\texttt{ == } (\texttt{U} \otimes \Sigma)]$$

- For a **layout requirement** `[T: AnyObject]`, we apply Σ to `T`. The right-hand side remains invariant under substitution:

  $$[\texttt{T: AnyObject}] \otimes \Sigma := [(\texttt{T} \otimes \Sigma)\texttt{: AnyObject}]$$

Having obtained a substituted requirement, we can check that it is satisfied. Before describing how this is done, we look at a few examples to help motivate what follows.

Example 9.3. Let's begin by looking at how type resolution proceeds to resolve the underlying type of `A` and `B`:

```
struct Concat<T: Sequence, U: Sequence> where T.Element == U.Element {}

// (1) all requirements satisfied
typealias A = Concat<String, Substring>

// (2) 'T.Element == U.Element' unsatisfied
typealias B = Concat<Array<Int>, Set<String>>
```

When resolving the underlying type of type alias `A`, we form this substitution map:

$$\begin{aligned}\Sigma_a := \{&\texttt{τ_0_0} \mapsto \texttt{String},\\ &\texttt{τ_0_1} \mapsto \texttt{Substring};\\ &[\texttt{τ_0_0: Sequence}] \mapsto [\texttt{String: Sequence}],\\ &[\texttt{τ_0_1: Sequence}] \mapsto [\texttt{Substring: Sequence}]\}\end{aligned}$$

We apply Σ_a to each explicit requirement in the generic signature of `Concat`:

$$[\texttt{τ_0_0: Sequence}] \otimes \Sigma_a = [\texttt{String: Sequence}]$$
$$[\texttt{τ_0_1: Sequence}] \otimes \Sigma_a = [\texttt{Substring: Sequence}]$$
$$[\texttt{τ_0_0.Element == τ_0_1.Element}] \otimes \Sigma_a = [\texttt{Character == Character}]$$

The first two substituted requirements claim their subject types conform to `Sequence`. We can check these by performing a global conformance lookup, which returns a valid concrete conformance in both cases, so we conclude both requirements are satisfied. The final substituted requirement is a statement that `Character` and `Character` are the same type, which also appears to be true. Thus, Σ_a satisfies the generic signature of `Concat`, and we successfully form the resolved type `Concat<String, Substring>`.

When resolving the underlying type of type alias `B`, we see it is invalid:

$$\begin{aligned}\Sigma_b := \{&\texttt{τ_0_0} \mapsto \texttt{Array<Int>},\\ &\texttt{τ_0_1} \mapsto \texttt{Set<String>};\\ &[\texttt{τ_0_0: Sequence}] \mapsto [\texttt{Array<Int>: Sequence}],\\ &[\texttt{τ_0_1: Sequence}] \mapsto [\texttt{Set<String>: Sequence}]\}\end{aligned}$$

The substitution map Σ_b does not satisfy the same-type requirement because the conformances of `Array<Int>` and `Set<String>` to `Sequence` have different type witnesses for `Element`:

$$[\texttt{τ_0_0.Element == τ_0_1.Element}] \otimes \Sigma_b = [\texttt{Int == String}]$$

We diagnose the failure and reject the declaration of `B`.

Example 9.4. Let's take `Concat` as above, but reference it with generic arguments from a generic context. Let G be the generic signature of `Concat`, and H the generic signature of `OuterGeneric`. We are resolving the interface type of "`x`":

```
struct OuterGeneric<C: Collection> {
  var x: Concat<C, C.SubSequence> = ...
}
```

We build $\Sigma \in \textsc{Sub}(G, [\![H]\!])$ by mapping our generic arguments into the primary generic environment of H:

$$\begin{aligned}\Sigma := \{&\texttt{τ_0_0} \mapsto [\![\texttt{C}]\!],\\ &\texttt{τ_0_1} \mapsto [\![\texttt{C.SubSequence}]\!];\\ &[\texttt{τ_0_0: Sequence}] \mapsto [[\![\texttt{C}]\!]\texttt{: Sequence}],\\ &[\texttt{τ_0_1: Sequence}] \mapsto [[\![\texttt{C.SubSequence}]\!]\texttt{: Sequence}]\}\end{aligned}$$

Note that our original requirements contained type parameters of G, but now involve the primary archetypes of H after substitution:

$$\begin{aligned}&[\texttt{τ_0_0: Sequence}] \otimes \Sigma = [[\![\texttt{C}]\!]\texttt{: Sequence}]\\ &[\texttt{τ_0_1: Sequence}] \otimes \Sigma = [[\![\texttt{C.SubSequence}]\!]\texttt{: Sequence}]\\ &[\texttt{τ_0_0.Element == τ_0_1.Element}] \otimes \Sigma = [[\![\texttt{C.Element}]\!]\texttt{ == }[\![\texttt{C.Element}]\!]]\end{aligned}$$

We check the first two requirements by performing a global conformance lookup on each archetype. Recalling Section 8.1, this is implemented as a generic signature query against the archetype's generic signature. In fact, from the associated requirements [Self: Sequence] and [Self.SubSequence: Collection] of Collection, we can derive $H \vdash$ [C: Sequence] and $H \vdash$ [C.SubSequence: Sequence]. Global conformance lookup succeeds and outputs a pair of abstract conformances:

$$\begin{aligned}&\langle\texttt{Sequence}] \otimes [\![\texttt{C}]\!] = [[\![\texttt{C}]\!]\texttt{: Sequence}]\\ &\langle\texttt{Sequence}] \otimes [\![\texttt{C.SubSequence}]\!] = [[\![\texttt{C.SubSequence}]\!]\texttt{: Sequence}]\end{aligned}$$

(In fact, we could have also checked that Σ satisfies a conformance requirement with a local conformance lookup into Σ itself.)

The third requirement is a same-type requirement, so we apply Σ to both sides; being dependent member types, we project a type witness from each abstract conformance:

$$\begin{aligned}&\texttt{τ_0_0.Element} \otimes \Sigma = \langle\texttt{Sequence}|\texttt{Element} \otimes [[\![\texttt{C}]\!]\texttt{: Sequence}]\\ &\texttt{τ_0_1.Element} \otimes \Sigma = \langle\texttt{Sequence}|\texttt{Element} \otimes [[\![\texttt{C.SubSequence}]\!]\texttt{: Sequence}]\end{aligned}$$

The two type witness projections construct the dependent member types τ_0_0.Element and τ_0_0.SubSequence.Element, and map them into the generic environment of H. Since $H \vdash$ [τ_0_0.Element == τ_0_0.SubSequence.Element], both map to the same reduced type in H, so they also define the same archetype:

$$\begin{aligned}&\langle\texttt{Sequence}|\texttt{Element} \otimes [[\![\texttt{C}]\!]\texttt{: Sequence}] = [\![\texttt{C.Element}]\!]\\ &\langle\texttt{Sequence}|\texttt{Element} \otimes [[\![\texttt{C.SubSequence}]\!]\texttt{: Sequence}] = [\![\texttt{C.Element}]\!]\end{aligned}$$

Both sides of our substituted same-type requirement are canonically equal, so we can conclude that all requirements of G are satisfied, and our type representation is valid.

Algorithm 9A (Check requirement)**.** Takes a substituted requirement $R \in \textsc{Req}(\llbracket H \rrbracket)$ as input, where the generic signature H is not given explicitly; R may contain primary archetypes of H, but not type parameters. Returns true if R is satisfied, false otherwise. In the below, `X` is the concrete subject type of R, so $\texttt{X} \in \textsc{Type}(\llbracket H \rrbracket)$. We handle each requirement kind as follows:

- For a **conformance requirement** [`X: P`], we perform the global conformance lookup $\texttt{X} \otimes \texttt{T}$. There are three possible outcomes:
  1. If we get an abstract conformance, it must be that `X` is an archetype of H whose type parameter conforms to `P`. Return true.
  2. If we get a concrete conformance, it might be conditional (Section 10.4). These conditional requirements are also substituted requirements of $\textsc{Req}(\llbracket H \rrbracket)$, and we check them by recursively invoking this algorithm. If all conditional requirements are satisfied (or if there aren't any), return true.
  3. If we get an invalid conformance, or if the conditional requirement check failed above, return false.
- For a **superclass requirement** [`X: C`], we proceed as follows:
  1. If `X` is a class type canonically equal to `C`, return true.
  2. If `X` and `C` are two distinct generic class types for the same class declaration, return false.
  3. If `X` does not have a superclass type (Section 6.4), return false.
  4. Otherwise, let $\texttt{X}'$ be the superclass type of `X`. Recursively apply the algorithm to the superclass requirement [$\texttt{X}'$`: C`].
- For a **layout requirement** [`X: AnyObject`], we check if `X` is a class type, an archetype satisfying the `AnyObject` layout constraint, or an `@objc` existential, and if so, we return true. Otherwise, we return false. (We'll discuss representation of existentials in Chapter 14.)
- For a **same-type requirement** [`X == Y`], we check if `X` and `Y` are canonically equal.

Contextually-generic declarations. A type declaration with a trailing `where` clause but no generic parameter list was called a contextually-generic declaration in Section 4.2. A non-generic declaration inside a constrained extension is conceptually similar; we'll meet constrained extensions in Section 10.3. In both cases, the generic signature G of the referenced declaration and the generic signature of the parent context G' share the same generic parameters, but G has additional requirements not present in G'. While there are no generic arguments to apply, we still proceed to check that Σ satisfies the requirements of G.

Example 9.5. The `Inner` type below demonstrates the first case:

```
struct Outer<T: Sequence> {
  struct Inner where T.Element == Int {}
}

// all requirements are satisfied
typealias A = Outer<Array<Int>>.Inner

// 'T.Element == Int' is unsatisfied
typealias B = Outer<Array<String>>.Inner
```

The second type alias `B` is ultimately invalid. While the base `Outer<Array<String>>` resolves without error, the member type representation fails because this substituted requirement is unsatisfied, with Σ as the context substitution map of the base type:

$$[\texttt{τ_0_0.Element == Int}] \otimes \Sigma = [\texttt{String == Int}]$$

Example 9.6. There is one more detail left to discuss. Using our `Concat` type from Example 9.3, we now consider the following type alias:

```
// (3) 'T: Sequence' unsatisfied;
// 'T.Element == U.Element' substitution failure
typealias C = Concat<Float, Set<Int>>
```

We get the following substitution map; note that it contains an invalid conformance:

$$\begin{aligned}\Sigma := \{&\texttt{τ_0_0} \mapsto \texttt{Float},\\ &\texttt{τ_0_1} \mapsto \texttt{Set<Int>};\\ &[\texttt{τ_0_0: Sequence}] \mapsto (\text{invalid}),\\ &[\texttt{τ_0_1: Sequence}] \mapsto [\texttt{Set<Int>: Sequence}]\}\end{aligned}$$

We apply this substitution map to each requirement of our generic signature:

$$[\texttt{τ_0_0: Sequence}] \otimes \Sigma = [\texttt{Float: Sequence}]$$
$$[\texttt{τ_0_1: Sequence}] \otimes \Sigma = [\texttt{Set<Int>: Sequence}]$$
$$[\texttt{τ_0_0.Element == τ_0_1.Element}] \otimes \Sigma = [\texttt{<<error type>> == Int}]$$

The first conformance requirement is unsatisfied and will be diagnosed. The same-type requirement contains error types after substitution, and does not need to be diagnosed; in fact, its failure is a consequence of the first conformance requirement being unsatisfied.

After applying our substitution map, but before checking the substituted requirement via the above algorithm, we look for error types. The presence of error types in the substituted requirement is indicative of one of several problems:

1. One of the generic argument types contained an error type, meaning an error was diagnosed earlier by type resolution.
2. The original type was a dependent member type, and the substituted base type does not conform to the member type's protocol. This means that an earlier conformance requirement was unsatisfied, and hence already diagnosed.
3. The declaration of a normal conformance may itself contain error types for any invalid or missing type witnesses, in which case projecting a type witness may output an error type; again, we will have diagnosed an error earlier, when checking the conformance.

In all cases, a diagnostic was already emitted, thus the requirement itself need not be diagnosed. We called this a *substitution failure* in Chapter 6.

Algorithm 9B (Check substitution map)**.** Takes two inputs:

1. A substitution map $\Sigma \in \textsc{Sub}(G, [\![H]\!])$.
2. Some list of elements of $\textsc{Req}(G)$. (When checking the generic arguments of a type representation, these are the explicit requirements of the generic signature G.)

As output, returns an *unsatisfied* list, and a *failed* list. If both output lists are empty, all input requirements are satisfied by Σ.

1. (Setup) Initialize the two output lists, initially empty.
2. (Done) If the input list is empty, return.
3. (Next) Remove the next requirement R from the input list and compute $R \otimes \Sigma$.
4. (Failed) If $R \otimes \Sigma$ contains error types, move R to the failed list.
5. (Check) Invoke Algorithm 9A with $R \otimes \Sigma$. If the requirement is unsatisfied, move R to the unsatisfied list.
6. (Loop) Go back to Step 2.

If any requirements appear in the unsatisfied list, type resolution diagnoses a series of errors at the source location of the generic type representation, one for each unsatisfied requirement. Requirements on the failed list are dropped, because another diagnostic will have been emitted, as explained previously. If at least one requirement was unsatisfied or failed, the resolved type becomes the error type; this enforces the invariant that clients of type resolution will not encounter any types that do not satisfy generic requirements—with the important exception of the error type itself!

There's more. Other places where we use Algorithm 9B:

1. When checking the declaration of a normal conformance [$\texttt{X}_d$`: P`] where $\texttt{X}_d$ is the declared interface type of some nominal type declaration d, we must decide if the given set of type witnesses satisfy the associated requirements of `P`. In other words, we take the protocol substitution map $\Sigma_{[\texttt{T: P}]}$, and apply it to each associated requirement of `P` (Section 5.1).

2. When a concrete type `X` conforms to a protocol `P` via a conditional conformance, we check if the context substitution map of `X` satisfies the conditional requirements of [`X: P`]. We will describe this in Section 10.4.

3. The conditional requirements of a conditional conformance are also computed via the same algorithm when the declaration of the conformance is type checked. We ask which requirements in the generic signature of the constrained extension are *not* satisfied by the generic signature of the extended type.

4. Checking if a subclass method is a well-formed override of a superclass method asks whether the generic signature of the subclass method satisfies each requirement of the generic signature of the superclass method (Chapter 11).

There are also two related problems which follow different code paths but reason about requirements in the same way as above:

1. The expression type checker translates generic requirements to constraints when type checking a reference to a generic function; these constraints are then solved by the constraint solver and a substitution map is formed for the call. This is entirely analogous to what happens in type resolution when referencing a generic type declaration.

2. Requirement inference is the step in building a new generic signature where we *add* requirements to ensure that certain substituted requirements will be satisfied (Section 11.1).

9.4. Unbound Generic Types

Introduced in Section 3.3, the *unbound generic type* represents a reference to a generic type declaration without generic arguments, and a *placeholder type* represents a specific missing generic argument.

Unbound generic types and placeholder types only appear when permitted by the type resolution context. These contexts are those syntactic positions where missing generic arguments can be filled in by some other mechanism, for example by using the expression type checker to infer the type of an expression.

The following type resolution contexts allow unbound generic types or placeholder types, with each context handling them in some specific manner:

1. The type annotation of a variable declaration with an initial value expression may have either unbound generic types or placeholder types in various nested positions:

   ```
   let callback: () -> (Array) = { return [1, 2, 3] }
   let dictionary: Dictionary<Int, _> = { 0: "zero" }
   ```

   The missing generic arguments are inferred from the initial value expression when computing the variable declaration's interface type.

2. Types written in expressions, such as metatype values (`GenericType.self`), the callees of constructor invocations (`GenericType(...)`), and target types of casts (`x as GenericType`). In all cases, the generic arguments are inferred from the surrounding expression.

3. The extended type of an extension is typically written as an unbound generic type:

   ```
   struct GenericType<T: Sequence> {...}
   extension GenericType {...}
   ```

   We will see in Section 10.3 that writing generic arguments for the extended type also has meaning, as a shorthand for a series of same-type requirements. Placeholder types cannot appear here.

4. The underlying type of a type alias may contain an unbound generic type (but not a placeholder type). This is a shorthand for a generic type alias that forwards its generic arguments, so the below are equivalent, given `GenericType` as above:

   ```
   typealias GenericAlias = GenericType
   typealias GenericAlias<T: Sequence> = GenericType<T>
   ```

   If the underlying type of a type alias is an unbound generic type, it cannot have its own generic parameter list, and vice versa. Additionally, only the outermost type representation can resolve to an unbound generic type. Both of the following are thus invalid:

   ```
   typealias WrongGenericAlias<T> = GenericType
   typealias WrongFunctionAlias = () -> (GenericType)
   ```

Notice how the first two contexts allow any type representation to resolve to an unbound generic type even if it appears in a nested position, while in the last two, only the topmost type representation can resolve to an unbound generic type.

A limitation. An unbound generic type can refer to either a nominal type declaration, or a type alias declaration. However, an unbound generic type referring to a type alias declaration cannot be the parent type of a nominal type. It is an error to access a member type of a generic type alias without providing generic arguments, even when an unbound generic type referencing a nominal type declaration can appear in the same position:

```
struct GenericType<T> {
  struct Nested {
    let value: T
  }
}

typealias GenericAlias = GenericType

_ = GenericType.Nested(value: 123)        // OK
_ = GenericAlias<Int>.Nested(value: 123)  // OK

_ = GenericAlias.Nested(value: 123)       // error
_ = GenericAlias<Int>.Nested(value: 123)  // OK
```

A future direction. If we think of type representations as syntactic, and types as semantic, unbound generic types occupy a weird point in between. They are produced by type resolution and refer to type declarations, but they do not actually survive type checking. When all is said and done, generic arguments in expressions will either be resolved or replaced with error types. We could eliminate unbound generic types from the implementation by reworking type resolution to take a callback that fills in missing generic arguments. Each context where an unbound generic type or placeholder type can appear would supply its own callback. For example, the expression type checker would provide returns a fresh type variable type. This callback model is partially implemented today, but all existing callbacks are trivial; the callback's presence simply communicates to type resolution that an unbound generic type is permitted in this position.

9.5. Source Code Reference

Key source files:

- include/swift/AST/TypeResolutionStage.h
- lib/Sema/TypeCheckType.h

- lib/Sema/TypeCheckType.cpp

A client of type resolution must first create a `TypeResolutionOptions`, and then a `TypeResolution` instance from that. The latter defines a `resolveType()` method to resolve a `TypeRepr *` to a `Type`. We start with `TypeResolutionOptions` and its constituent parts, `TypeResolverContext` and `TypeResolutionFlags`.

`TypeResolutionOptions` *class*

Encodes the semantic behavior of type resolution, consisting of a base context, a current context, and flags. The base context and context are initially identical. When type resolution recurses on a nested type representation, it preserves the base context, possibly changes the current context, and clears the `Direct` flag. Thus, the base context encodes the overall syntactic position of the type representation, while the context encodes the role of the current type representation.

- `TypeResolutionOptions(TypeResolutionContext)` creates a new instance with the given base context, and no flags.
- `getBaseContext()` returns the base `TypeResolverContext`.
- `getContext()` returns the current `TypeResolverContext`.
- `getFlags()` returns the `TypeResolutionFlags`.

`TypeResolverContext` *enum class*

The type resolution context, which encodes the position of a type representation:

- `TypeResolverContext::None`: no special type handling is required.
- `TypeResolverContext::GenericArgument`: generic arguments of a generic nominal type.
- `TypeResolverContext::ProtocolGenericArgument`: generic arguments of a parameterized protocol type.
- `TypeResolverContext::TupleElement`: elements of a tuple element type.
- `TypeResolverContext::AbstractFunctionDecl`: the base context of a function declaration's parameter list.
- `TypeResolverContext::SubscriptDecl`: the base context of a subscript declaration's parameter list.
- `TypeResolverContext::ClosureExpr`: the base context of a closure expression's parameter list.

- `TypeResolverContext::FunctionInput`: the type of a parameter in a function type or declaration.
- `TypeResolverContext::VariadicFunctionInput`: the type of a variadic parameter in a function type or declaration. This differs from the above in that nested function types become `@escaping` by default.
- `TypeResolverContext::InoutFunctionInput`: the type of an `inout` parameter in a function type or declaration.
- `TypeResolverContext::FunctionResult`: the result type of a function type or declaration.
- `TypeResolverContext::PatternBindingDecl`: the type of a pattern in a pattern binding declaration.
- `TypeResolverContext::ForEachStmt`: the type of a variable in a `for` statement.
- `TypeResolverContext::ExtensionBinding`: the extended type of an extension declaration.
- `TypeResolverContext::InExpression`: a type appearing in expression context.
- `TypeResolverContext::ExplicitCastExpr`: the target type of an `as` cast expression.
- `TypeResolverContext::EnumElementDecl`: the base context of an enum element parameter list.
- `TypeResolverContext::EnumPatternPayload`: the payload type of an enum element pattern. Tweaks the behavior of tuple element labels.
- `TypeResolverContext::TypeAliasDecl`: the underlying type of a non-generic type alias.
- `TypeResolverContext::GenericTypeAliasDecl`: the underlying type of a generic type alias.
- `TypeResolverContext::ExistentialConstraint`: the constraint type of an existential type (Chapter 14);
- `TypeResolverContext::GenericRequirement`: the constraint type of a conformance requirement in a `where` clause.
- `TypeResolverContext::SameTypeRequirement`: the subject type or constraint type of a same-type requirement in a `where` clause.

- `TypeResolverContext::ProtocolMetatypeBase`: the instance type of a protocol metatype, like `P.Protocol`.
- `TypeResolverContext::MetatypeBase`: the base type of a concrete metatype, like `T.Type`.
- `TypeResolverContext::ImmediateOptionalTypeArgument`: the payload type of an optional sugared type, like `T?`. This just tailors some diagnostics.
- `TypeResolverContext::EditorPlaceholderExpr`: the type of an editor placeholder.
- `TypeResolverContext::Inherited`: the inheritance clause of a concrete type.
- `TypeResolverContext::GenericParameterInherited`: the inheritance clause of a generic parameter.
- `TypeResolverContext::AssociatedTypeInherited`: the inheritance clause of an associated type.
- `TypeResolverContext::CustomAttr`: the type of a custom attribute referencing a property wrapper or result builder.

`TypeResolutionFlags` *enum class*

A set of flags to control various esoteric behaviors:

- `TypeResolutionFlags::AllowUnspecifiedTypes`: allows type resolution to succeed if no type representation was actually provided. Used when resolving type annotations which can be omitted, such as the types of variable declarations and the types of closure parameters.
- `TypeResolutionFlags::SILType`: used when resolving a position where lowered SIL types, such as SIL function types, can appear.
- `TypeResolutionFlags::SILMode`: used when resolving a position where ordinary AST types appear, but in textual SIL, which enables special syntax, for example `@dynamic_self` for the dynamic `Self` type.
- `TypeResolutionFlags::FromNonInferredPattern`: when resolving an explicitly-stated type annotation in a pattern binding, muffles warnings where the type of a variable binding is inferred as `()` or `Never`, which suggests a programmer mistake unless the type was explicitly stated.

- `TypeResolutionFlags::Direct`: set when resolving the top-level type representation, and cleared when type resolution recurses on a child. Allows certain type representations which cannot be nested inside other type representations, such as `inout T` for an `inout` parameter type.
- `TypeResolutionFlags::SilenceErrors`: disables diagnostics, simply returning an error type if type resolution fails. Used in pattern resolution when it is not known if a pattern refers to a type or some other member.
- `TypeResolutionFlags::AllowModule`: allows an identifier type representation with a single component to resolve to a `ModuleType`, which is usually prohibited.
- `TypeResolutionFlags::AllowUsableFromInline`: makes internal type declarations annotated with the `@usableFromInline` attribute visible, which is used in the implementation of the `@_specialize` attribute.
- `TypeResolutionFlags::DisallowOpaqueTypes`: prevents opaque parameter declarations from appearing in contexts where they would otherwise be allowed, such as the generic arguments of a parameterized protocol type `any P<some Q>`.
- `TypeResolutionFlags::Preconcurrency`: when resolving a reference to a type alias marked with the `@preconcurrency` attribute, strips off `@Sendable` and global actor attributes from function types appearing in the underlying type of the type alias.

`TypeResolution` *class*

After initializing options, a client must create an instance of this class by calling one of two factory methods, depending on the desired type resolution stage. Each method takes a `DeclContext`, `TypeResolutionOptions`, and a pair of callbacks for resolving placeholder types and unbound generic types:

- `forStructural()` creates a type resolution in structural resolution stage.
- `forInterface()` creates a type resolution in interface resolution stage.

The main thing you can do with an instance is, of course, resolve types:

- `resolveType()` takes a `TypeRepr *` and returns a `Type`.

Another way to use this class is to call a static utility method that creates a new type resolution in interface resolution stage, resolves a type representation, and maps it into a generic environment to get a contextual type. This is used by the expression type checker to resolve types appearing in expression context:

- `resolveContextualType()`

The actual implementation of `resolveType()` uses these getter methods:

- `getDeclContext()` returns the declaration context where this type representation appears.
- `getASTContext()` returns the global AST context singleton.
- `getStage()` returns the current `TypeResolutionStage`.
- `getOptions()` returns the current `TypeResolutionOptions`.
- `getGenericSignature()` returns the current `GenericSignature` in interface resolution stage.

`TypeResolutionStage` *enum class*

Encodes the type resolution stage:

- `TypeResolutionStage::Structural`
- `TypeResolutionStage::Interface`

`ResolveTypeRequest` *class*

The implementation of the `TypeResolution::resolveType()` entry point evaluates the `ResolveTypeRequest`. While this request is not cached, it uses the request evaluator infrastructure to detect and diagnose circular type resolution. The evaluation function destructures the given type representation by kind using a visitor class.

`TypeResolver` *class*

A recursive visitor, internal to the implementation of type resolution. The methods of this visitor implement type resolution for each kind of type representation.

Type Representations

Key source files:

- include/swift/AST/TypeRepr.h
- lib/Sema/TypeCheckType.cpp

`TypeRepr` *class*

Abstract base class for type representations. This class hierarchy resembles `TypeBase`, `Decl`, `ProtocolConformance`, and others.

Type representations store a source location and kind:

- `getLoc()`, `getSourceRange()` returns the source location and source range of this type representation.
- `getKind()` returns the `TypeReprKind`.

Each `TypeReprKind` corresponds to a subclass of `TypeRepr`. Instances of subclasses support safe downcasting via the `isa<>`, `cast<>` and `dyn_cast<>` template functions,

```
if (auto *indentRepr = dyn_cast<IdentTypeRepr>(typeRepr))
  ...

auto *funcRepr = cast<FunctionTypeRepr>(typeRepr);
...

assert(isa<ArrayTypeRepr>(typeRepr));
```

Identifier Type Representations

`DeclRefTypeRepr` *class*

Abstract base class for identifier and member type representations.

- `getNameRef()` returns the identifier.
- `getGenericArgs()` returns the array of generic argument type representations, if any were written.

Recall that type representations cache the type declaration found by name lookup to speed up the case where both type resolution stages resolve the same type representation:

- `getBoundDecl()` returns the cached type declaration.
- `getDeclContext()` for the first component only, returns the outer declaration context where unqualified lookup found the type declaration.
- `setValue()` records a cached type declaration and the declaration context from which it was found.

Note that `getDeclContext()` is not always the same as the declaration context of the type declaration, because the type declaration might be a member of a conformed protocol or superclass of the declaration context, having been reached indirectly. However, the declaration context is always one of the outer declaration contexts in which the type representation appears.

`IdentTypeRepr` *class*

Subclass of `DeclRefTypeRepr`. Represents something like `Int` or `Array<Int>`.

Unqualified Lookup

Key source files:

- lib/Sema/TypeCheckType.cpp
- lib/Sema/TypeCheckNameLookup.cpp

`resolveTopLevelIdentTypeComponent()` *function*

Resolves the first component of an identifier type representation. Performs an unqualified lookup, disambiguates results, handles the special case of a `Self` reference, applies generic arguments if they were provided, and diagnoses errors.

`TypeChecker::lookupUnqualifiedType()` *function*

Wrapper around unqualified lookup which only considers type declarations.

`getSelfTypeKind()` *function*

Determines if `Self` in the given context refers to the innermost nominal type declaration, the dynamic `Self` type of the innermost class declaration, or if it is invalid to utter `Self`.

`resolveTypeDecl()` *function*

Resolves an unqualified reference to a type declaration to a type. This handles the behavior where omitted generic arguments when referencing type within its own context are deduced to be the corresponding generic parameters. It also handles various edge-case behaviors where the referenced type declaration was a nested type of a different nominal type, such as a superclass or conformed protocol, and requires substitutions.

Member Type Representations

`MemberTypeRepr` *class*

Subclass of `DeclRefTypeRepr` representing a member type representation.

- `getBase()` returns the base type representation.

Qualified Lookup

Key source files:

- lib/Sema/TypeCheckType.cpp
- lib/Sema/TypeCheckNameLookup.cpp

`resolveNestedIdentTypeComponent()` *function*

Resolves subsequent components of an identifier type representation. Performs a qualified lookup, disambiguates results, applies generic arguments if they were provided, and diagnoses errors.

`TypeResolution::resolveDependentMemberType()` *function*

Implements the special behavior where a member type of a type parameter directly resolves to an unbound dependent member type in the structural resolution stage, or via name lookup to a bound dependent member type or type alias type in the interface resolution stage.

`TypeChecker::lookupMemberType()` *function*

Wrapper around qualified lookup which only considers type declarations, and resolves type witnesses of concrete types when it finds an associated type declaration with a concrete base type.

`TypeChecker::isUnsupportedMemberTypeAccess()` *function*

Determines if a member type access is invalid. This includes such oddities as accessing a member type of an unbound generic type referencing a type alias (Section 9.4), or accessing a dependent protocol type alias where the base type is the protocol itself (Section 9.2).

Applying Generic Arguments

- `include/swift/AST/Requirement.h`
- `lib/AST/Requirement.cpp`
- `lib/Sema/TypeCheckGeneric.cpp`

`applyGenericArguments()` *function*

Constructs a nominal type or type alias type, given a type declaration and a list of generic arguments. In the interface resolution stage, also checks that the generic arguments satisfy the requirements of the type declaration's generic signature.

`TypeResolution::applyUnboundGenericArguments()` *method*

Factor of `applyGenericArguments()` to build a substitution map from the base type and generic arguments and check requirements of a generic declaration.

`TypeResolution::checkContextualRequirements()` *method*

Factor of `applyGenericArguments()` to build the substitution map from the base type and check requirements of contextually-generic declarations.

`Requirement` *class*

See also Section 5.7. Requirement substitution:

- `subst()` applies a substitution map to this requirement, returning the substituted requirement.

Checking requirements:

- `checkRequirement()` answers if a single requirement is satisfied, implementing Algorithm 9A. Any conditional requirements are returned via a `SmallVector` supplied by the caller, who must then check these requirements recursively.
- `checkRequirements()` checks each requirement in an array; if any have their own conditional requirements, those are checked too. This implements Algorithm 9B for when the caller needs to check a condition without emitting diagnostics. For example, this is used by `checkConformance()` to check conditional requirements in Section 10.5.

`checkGenericArgumentsForDiagnostics()` *function*

Uses `Requirement::checkRequirement()` to implements a variant of Algorithm 9B that records additional information for diagnostics. This is used by type resolution and the conformance checker.

Part III.

Subtleties

10. Extensions

Extensions add members to existing nominal type declarations. We refer to this nominal type declaration as the *extended type* of the extension. The extended type can originate from the same source file, another source file of the main module, or most generally, some other module. Extensions themselves are declarations, but they are *not* value declarations in the sense of Chapter 4, meaning the extension itself cannot be referenced by name. Instead, the members of an extension become visible as members of the extended type to qualified name lookup.

Consider a module containing a pair of struct declarations, `Outer` and `Outer.Middle`:

```
public struct Outer<T> {
  public struct Middle<U> {}
}
```

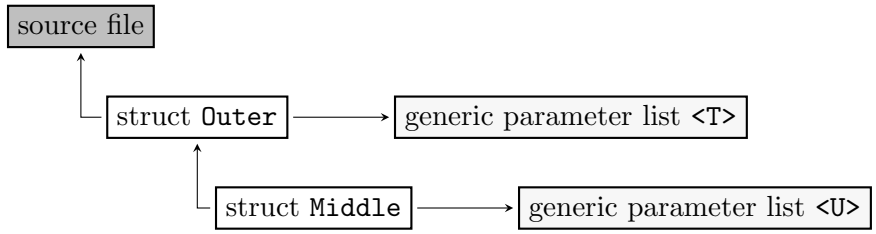
A second module can declare an extension of `Outer.Middle` to add a method named `foo()` and a nested type named `Outer.Middle.Inner`:

```
extension Outer.Middle {
  func foo() {}
  struct Inner<V> {}
}
```

If a third module subsequently imports both the first and second module, it will see the members `Outer.Middle.foo()` and `Outer.Middle.Inner` just as if they were defined inside `Outer.Middle` itself.

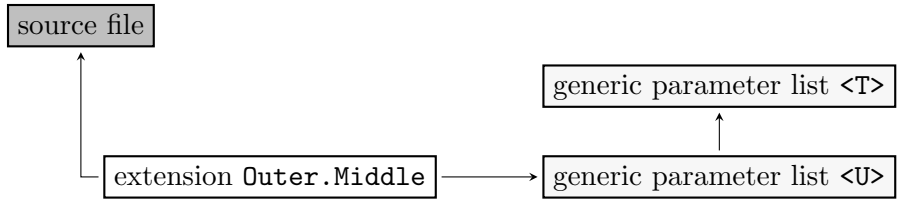
Extensions and generics. The generic parameters of the extended type are visible in the scope of the extension's body. Each extension has a generic signature, which describes the interface types of its members. An *unconstrained extension* has the same generic signature as the extended type. Extensions can also impose additional requirements on their generic parameters via a `where` clause; this declares a *constrained extension* with its own generic signature (Section 10.3). Finally, an extension can state a conformance to a protocol, which as we recall, declares a normal conformance visible to global conformance lookup. If the extension is unconstrained, this is essentially equivalent to stating a conformance on the extended type. If the extension is constrained, the conformance becomes a *conditional conformance* (Section 10.4).

Let's begin by taking a closer look at generic parameter lists of extensions, by way of our nested type `Outer.Middle` and its extension above. Recall how generic parameters of nominal types work from Section 4.1: their names are lexically scoped to the body of the type declaration, and each generic parameter is uniquely identified by its depth and index. We can represent the declaration context nesting and generic parameter lists of our nominal type declaration `Outer.Middle` with a diagram like the following:



We create the fiction that each generic parameter in the scope of the extended type is also visible inside the extension, by *cloning* generic parameter declarations. The cloned declarations have the same name, depth, and index as the originals, but they are parented to the extension. This ensures that looking up a generic parameter inside an extension finds a generic parameter with the same depth and index as the one with the same name in the extended type.

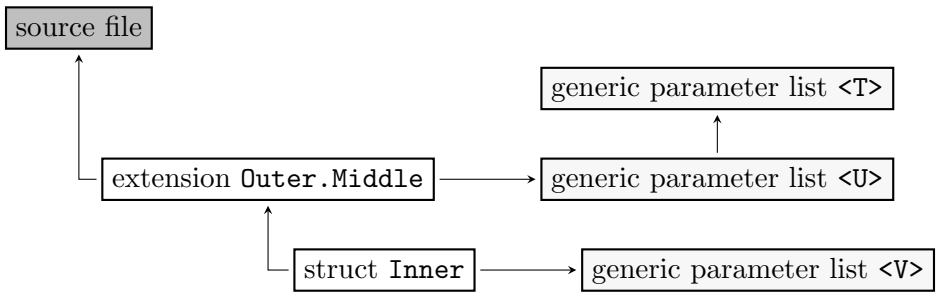
Since all generic parameters in a single generic parameter list have the same depth, an extension may have multiple generic parameter lists, one for each level of depth (that is, generic context nesting) of the extended type. We link the cloned generic parameter lists together using an "outer list" pointer. The innermost generic parameter list is "the" generic parameter list of the extension, and the head of the list; it is cloned from the extended type. Its outer list is cloned from the extended type's parent generic context, if any, and so on. The outermost generic parameter list has a null outer list. In our extension of `Outer.Middle`, this looks like so:



Unqualified name lookup traverses the outer list pointer when searching for generic parameter declarations by name. Now, consider the nested type `Inner` from our example, which declares a generic parameter list of its own:

```
extension Outer.Middle {
  struct Inner<V> {}
}
```

If an extension declares a nested generic type, the depth of the nested type's generic parameters becomes one greater than the depth of the extension's innermost generic parameter list. Thus, the generic parameter `V` of `Inner` has depth 2 and index 0. All three generic parameter lists are visible inside the body of `Outer.Middle.Inner`:



The declared interface type of an extension is the declared interface type of the extended type; the self interface type of an extension is the self interface type of the extended type. The two concepts coincide except when the extended type is a protocol, in which case the declared interface type is the protocol type, whereas the self interface type is the protocol `Self` type.

Other behaviors. An extension member generally behaves as if were declared inside the extended type itself, with some differences:

- Members of *protocol extensions* are not requirements imposed on the concrete type in the way that members of a protocol declaration are; they actually have a body.

  A protocol extension member with the same name and interface type as a protocol requirement acts as a *default witness*, to be used if the conforming type does not provide its own witness for this requirement:

```
protocol P {
  func f()
}

extension P {
  func f() {
    print("default witness for P.f()")
  }
}

struct S: P {}

S().f()  // calls the default witness for P.f()
```

- Extensions of struct and class declarations cannot add stored properties, only computed properties:

```
struct S {
  var x: Int
}

extension S {
  var y: Int  // stored property: error
  var flippedX: Int { return -x }  // computed property: okay
}
```

  All stored properties must be initialized by all declared constructors; extensions being able to introduce stored properties hitherto-unknown to the original module would break this invariant. Another reason is that the stored property layout of a struct or class is computed when the struct or class declaration is emitted, and there is no mechanism for extensions to alter this layout after the fact.

- Extensions cannot add new cases to enum declarations, for similar reasons as stored properties; doing so would complicate both the static exhaustiveness checking for `switch` statements and the in-memory layout computation of the enum.

- When the extended type is a class, the methods of the extension are implicitly `final`, and the extension's methods are not permitted to override methods from the superclass. Calls to non-`final` class methods are dispatched through a *vtable*, or table of function pointers, attached to the runtime metadata of the class, and there is no mechanism for extensions to add new entries or replace existing entries in the vtable.

- The rules for types nested in extensions are the same as those for types nested in nominal types (Section 6.2). An extension of a struct, enum, or class may contain a nested struct, enum, or class, while a protocol extension cannot, just as a protocol cannot. Extensions themselves must be at the top level of a source file, but the extended type can be nested inside of another nominal type (or extension).

10.1. Extension Binding

The extended type of an extension is specified by the type representation that follows the `extension` keyword. Extension members are made available to qualified lookup by the process of *extension binding*, which resolves the type representation of an extension to its extended type, and adds the extension's members to the extended type's name lookup table.

A complication is that the extended type of an extension may *itself* be declared inside of another extension. Extension binding cannot simply visit all extension declarations in a single pass in source order, because of ordering dependencies between extensions and nested types. Instead, multiple passes are performed; a failure to bind an extension is not a fatal condition, and instead failed extensions are revisited later after subsequent extensions are successfully bound. This process iterates until fixed point.

Algorithm 10A (Bind extensions)**.** Takes a list of all extensions in the main module as input, in any order.

1. Enqueue all extension declarations on the pending list.
2. Clear the delayed list.
3. Clear the flag.
4. (Check) If the pending list is empty, go to Step 6.
5. (Resolve) Remove an extension from the pending list and attempt to resolve its extended type. If resolution succeeds, associate the extension with the resolved nominal type declaration and set the flag. If resolution fails, add the extension to the delayed list but do not emit any diagnostics and leave the flag unchanged. Go back to Step 4.
6. (Retry) If the flag is set, move the entire contents of the delayed list to the pending list, clear the delayed list, clear the flag, and go to Step 4. Otherwise, return.

The worklist-driven extension binding algorithm was introduced in Swift 5. Older compiler releases attempted to bind extensions in a single pass, something that could either succeed or fail depending on declaration order. This incorrect behavior was one of the most frequently reported bugs of all time [111].

Invalid extensions. If extension binding fails to resolve the extended type of an extension, it simply remains on the delayed list without any diagnostics emitted. Invalid extensions are diagnosed later, when the **type-check primary file request** visits all primary files and attempts to resolve the extended types of any extensions again.

Extension binding uses a more limited form of type resolution, because we only need to resolve the type representation to a *type declaration* and not a *type*. This type declaration must be a nominal type declaration, so the extended type is typically written as an *identifier* or *member* type representation (Sections 9.1 and 9.2). Extension binding runs early in the type checking process, immediately after parsing and import resolution. We cannot build generic signatures or check conformances in extension binding, because those requests assume that extension binding has already taken place, freely relying on qualified lookup to find members of arbitrary extensions.

In particular, extension binding cannot find declarations synthesized by the compiler, including type aliases created by associated type inference. Also, it cannot perform type substitution; this rules out extensions of generic type aliases whose underlying type is itself a type parameter.

If extension binding fails while the **type-check primary file request** successfully resolves the extended type when it visits the extension later, we emit a special diagnostic, tailored by a couple of additional checks. If ordinary type resolution returned a type alias type `Foo` whose underlying type is a nominal type `Bar`, the type checker emits the "extension of type `Foo` must be declared as an extension of `Bar`" diagnostic. Even though we know what the extended type should be at this point, we must still diagnose an error; it is too late to bind the extension, because other name lookups may have already been performed, potentially missing out on finding members of this extension.

Example 10.1. An invalid extension of a synthesized type alias:

```
protocol Animal {
  associatedtype FeedType
  func eat(_: FeedType)
}

struct Horse: Animal {
  func eat(_: Hay) {}
}

// error: extension of type 'Horse.FeedType' must be declared as an
// extension of 'Hay'
extension Horse.FeedType {...}
```

Extension binding fails to resolve `Horse.FeedType`, because this type alias was not written in source, so it does not exist when extension binding runs. However, when the **type-check primary file request** visits this extension later, type resolution will trigger associated type inference, which will synthesize the type alias from the witness `Horse.eat(_:)`. We produce a diagnostic directing the user to state the correct extended type that can be found at extension binding time.

Example 10.2. An invalid extension of a type alias with an underlying type that is a type parameter:

```
typealias G<T: Sequence> = T.Element

// error: extension of type 'G<Array<Int>>' must be declared as an
// extension of 'Int'
extension G<Array<Int>> {...}
```

Extension binding fails to resolve `G<Array<Int>>`, because this requires performing a type substitution that depends on the conformance [`Array: Sequence`]:

$$\texttt{T.Element} \otimes \{\texttt{T} \mapsto \texttt{Array<Int>};\ [\texttt{T: Sequence}] \mapsto [\texttt{Array<Int>: Sequence}]\}$$
$$= \texttt{Int}$$

The other case where extension binding fails but **type-check primary file request** is able to resolve the extended type is when this type is not actually a nominal type. In this situation, the type checker emits the fallback "non-nominal type cannot be extended" diagnostic:

```
typealias Fn = () -> ()

// error: you wish
extension Fn {
  ...
}
```

The extension binding algorithm is quadratic in the worst case, where each pass over the pending list is only able to bind exactly one extension. However, only unrealistic code examples trigger this pathological behavior. Indeed, the first pass binds all extensions of types not nested inside of other extensions, and the second pass binds extensions of types nested inside extensions that were bound by the first pass, which covers the vast majority of cases in reasonable user programs.

Example 10.3. The following order of extensions requires four passes to bind; by adding more nested types, the number of iterations can be increased arbitrarily:

```
struct Outer {}

extension Outer.Middle.Inner {}  // bound in 3rd pass

extension Outer.Middle {  // bound in 2nd pass
  struct Inner {}
}

extension Outer {  // bound in 1st pass
  struct Middle {}
}

extension DoesNotExist {}  // remains on delayed list
```

In the first pass, the extensions of `Outer.Middle.Inner` and `Outer.Middle` both fail, but we do successfully bind the extension of `Outer`. The second pass once again fails to bind the extension of `Outer.Middle.Inner`, but it does bind the extension of `Outer.Middle`. Finally, the third pass binds the extension of `Outer.Middle.Inner`. Notice how in each of the first three passes, we are able to bind at least one extension. The invalid extension of `DoesNotExist` remains on the delayed list after each of the first three passes. Since the fourth pass does not make any forward progress, the algorithm stops. Later we end up in the diagnostic path, which resolves `DoesNotExist` via ordinary type resolution, surfacing the "unknown type" error.

Local types. Because extensions can only appear at the top level of a source file, the extended type must ultimately be visible from the top level. This allows extensions of types nested inside other top-level types, but precludes extensions of local types nested inside of functions or other local contexts, because there is ultimately no way to name a local type from the top level of a source file. (As a curious consequence, local types cannot conditionally conform to protocols, since the only way to declare a conditional conformance is to write an extension!)

10.2. Direct Lookup

Now we're going to take closer look at *direct lookup*, the primitive operation that looks for a value declaration with the given name inside of a nominal type and its extensions. This was first introduced in Section 2.1 as the layer below qualified name lookup, which performs direct lookups into the base type, its conformed protocols, and superclasses.

Nominal type declarations and extensions are *iterable declaration contexts*, meaning they contain member declarations. Before discussing direct lookup, let's consider what happens if we ask an iterable declaration context to list its members. This is a lazy operation which triggers work the first time it is called:

- Iterable declaration contexts parsed from source are populated by delayed parsing; when the parser first reads a source file, the bodies of iterable declaration contexts are skipped, and only the source range is recorded. Asking for the list of members goes and parses the source range again, constructing declarations from their parsed representation (Section 2.2).
- Iterable declaration contexts from binary and imported modules are equipped with a *lazy member loader* which serves a similar purpose. Asking the lazy member loader to list all members will build the corresponding Swift declarations from deserialized records or imported Clang declarations. The lazy member loader can also find just those declarations with a *specific* name, as explained below; this is the more common operation, since it is much more efficient.

Member lookup table. Every nominal type declaration has an associated *member lookup table*, which is used for direct lookup. This table maps each identifier to a list of value declarations with that name (multiple value declarations can share a name because Swift allows type-based overloading). The declarations in a member lookup table are understood to be members of one or more iterable declaration contexts, which are exactly the type declaration itself and all of its extensions. These iterable declaration contexts might originate from a mix of different module kinds. For example, the nominal type itself might be an Objective-C class from an imported Objective-C module, with one extension declared in a binary Swift module, and another extension defined in the main module, parsed from source.

The lookup table is populated lazily, kind of like a state machine. Say we're asked to perform a direct lookup for some given name `X`. If this is the first lookup into this table, we begin by populating the table with *all* members from any *parsed* iterable declaration contexts, which might trigger delayed parsing. Each entry in the member lookup table also stores a "complete" bit. The "complete" bit of these initially-populated entries is *not* set, meaning that each entry only contains those members that were parsed from source. Next, if any iterable declaration contexts originate from binary or imported modules, direct lookup asks each lazy member loader to selectively load only those members named `X`. (Parsed declaration contexts do not offer this level of granularity, because there is no way to find a specific member without parsing them all.) After the lazy member loaders do their work, the lookup table entry for `X` is now complete, so we set its "complete" bit. If a subsequent direct lookup encounters a member lookup table entry whose "complete" bit is already set, the list of declarations stored in the entry is returned immediately, without querying the lazy member loaders.

This *lazy member loading* mechanism ensures that only those members which are actually referenced in a compilation session are loaded from serialized and imported iterable declaration contexts.

Example 10.4. Listing 10.1 shows an example of lazy member loading. Observe the following:

- The `NSHorse` class itself is declared in Objective-C in a header file. Suppose this header file is part of the `HorseKit` module, imported by the two Swift source files.
- The Swift source file `b.swift` declares an extension of `NSHorse`. Let's say that this is a secondary file in the current frontend job.
- The Swift source file `c.swift` declares a function which calls some methods on `NSHorse`. For our example this needs to be a primary file of this frontend job.
- The `trot()` method in the class has a different interface type than the `trot()` method in the extension, so they are two distinct overloaded methods with the same name.

Listing 10.1.: A class implemented in Objective-C, with an extension written in Swift

```
// a.h
@interface NSHorse: NSObject
- (void) trot: (int) x;
- (void) canter: (float) x;
@end
```

```
// b.swift
import HorseKit

extension NSHorse {
  func walk(_: Float) {}
  func trot(_: Float) {}
}
```

```
// c.swift
import HorseKit

func ride(_ horse: NSHorse) {
  horse.walk(1.0)
  horse.trot(2.0)
  horse.walk(1.0)
}
```

The compiler begins by parsing both Swift source files, skipping over the extension body with delayed parsing because it appears in a secondary file. Next, extension binding is performed, which associates the extension of `NSHorse` with the imported class declaration of `NSHorse`. Finally, we type check the body of the `ride()` function, since it appears in a primary file. Type checking the expressions in the body performs three direct lookups into `NSHorse`, first with the name `walk`, then `trot`, and finally `walk` again.

(`walk`) The table is initially empty, so it must be populated with the members of all parsed iterable declaration contexts first. This triggers delayed parsing of the extension, which adds two entries to our member lookup table:

Name	Declarations	Complete?
`walk`	`NSHorse.walk` in extension of `NSHorse`	×
`trot`	`NSHorse.trot` in extension of `NSHorse`	×

At this point, neither entry is complete, because we haven't looked for members inside the class itself, which was imported and not parsed. Direct lookup does that next, by asking the lazy member loader associated with the class declaration to load any members named `walk`. The class does not define such a member, so we simply mark the member lookup table entry as complete:

Name	Declarations	Complete?
`walk`	`NSHorse.walk` in extension of `NSHorse`	✓
`trot`	`NSHorse.trot` in extension of `NSHorse`	×

Direct lookup returns `NSHorse.walk()` to the type checker.

(`trot`) While the member lookup table already has an entry named `trot`, it is not complete, so the lazy member loader for the class declaration is consulted again. This time, the class contains a member with this name also. The `trot` method is imported from Objective-C and added to the member lookup table:

Name	Declarations	Complete?
`walk`	`NSHorse.walk` in extension of `NSHorse`	✓
`trot`	`NSHorse.trot` in extension of `NSHorse` `NSHorse.trot` in class `NSHorse`	✓

Direct lookup now returns both overloads of `NSHorse.trot()` to the type checker.

(`walk`) The third lookup finds that the corresponding member lookup table entry is already complete, so we immediately return the existing declaration stored there without consulting the lazy member loader. Notice that the `canter` method of `NSHorse` was never referenced, so it did not need to be imported at all.

History. Lazy member loading was introduced in Swift 4.1 to avoid wasted work in deserializing or importing members that are never referenced. The speedup is most pronounced when multiple frontend jobs perform direct lookup into a common set of deserialized or imported nominal types. The overhead of loading all members of this common set of types was multiplied across frontend jobs prior to the introduction of lazy member loading.

10.3. Constrained Extensions

An extension can impose its own requirements on the generic parameters of the extended type; we say this as a *constrained extension*. These requirements are always additive; the generic signature of a constrained extension is built from the generic signature of the extended type, together with the new requirements. The members of a constrained extension are then only available on specializations of the extended type that satisfy these requirements. There are three ways to declare a constrained extension:

1. using a `where` clause,
2. by writing a generic nominal type with generic arguments as the extended type,
3. by writing a generic type alias type as the extended type, with some restrictions.

Case 1 is the most general form; Case 2 and Case 3 can also be expressed by writing the appropriate requirements in a `where` clause. An extension that does not fall under one of these three cases is sometimes called an *unconstrained extension* when it is important to distinguish it from a constrained extension. The generic signature of an unconstrained extension is the same as the generic signature of the extended type.

Here is a constrained extension of `Set` which constrains the `Element` type to `Int`:

```
extension Set where Element == Int {...}
```

The generic signature of `Set` is `<Element where Element: Hashable>`. Adding the same-type requirement [`Element == Int`] makes the requirement [`Element: Hashable`] redundant since `Int` conforms `Hashable`, so the extension's generic signature becomes `<Element where Element == Int>`.

This example demonstrates that while the requirements of the constrained extension abstractly imply those of the extended type, they are not a "syntactic" subset—the [`Element: Hashable`] requirement does not appear in the constrained extension's generic signature, because it became redundant when [`Element == Int`] was added.

In the early days of Swift, this extension was not supported at all, because the compiler did not permit same-type requirements between generic parameters and concrete types; only dependent member types could be made concrete. This restriction was lifted in Swift 3, and many concepts described in this book, most importantly substitution maps and generic environments, were introduced as part of this effort.

Extending a generic nominal type. An extension of a generic nominal type with generic arguments is shorthand for a `where` clause that constrains each generic parameter to a concrete type. The generic argument types must be fully concrete; they cannot reference the generic parameters of the extended type declaration. The previous example can be more concisely expressed using this syntax:

```
extension Set<Int> {...}
```

A non-generic type alias type whose underlying type is a generic nominal type can also be used in the same manner:

```
typealias StringMap = Dictionary<String, String>
extension StringMap {...}
```

This shorthand was introduced in Swift 5.7 [112].

Extending a generic type alias. A generic type alias is called a "pass-through type alias" if it satisfies the following three conditions:

1. the underlying type of the generic type alias must be a generic nominal type,
2. the type alias must have the same number of generic parameters as the underlying type declaration,
3. the type alias must substitute each generic parameter of the underlying type declaration with the corresponding generic parameter of the type alias (where the correspondence means they have the same depth and index).

A pass-through type alias essentially equivalent to the underlying generic nominal type, except that it may introduce additional requirements via a `where` clause. An extension of a pass-through type alias is equivalent to a constrained extension of the underlying nominal type which states these additional requirements. Note that the pass-through

type alias is not required to give its generic parameters the same names as its underlying type, which is slightly confusing, because the names used by the type alias declaration do not matter to the extension at all; the generic parameter list of the extension is always cloned from the extended nominal type, and not the type alias.

Example 10.5. While it is a generally useful feature, the ability to extend a pass-through type alias was motivated by the desire to maintain source compatibility after a specific standard library change.

The standard library defines a generic struct named `Range` with a single `Bound` generic parameter conforming to `Comparable`:

```
struct Range<Bound: Comparable> {...}
```

Prior to Swift 4.2, the standard library also had a separate `CountableRange` type with a different set of requirements:

```
struct CountableRange<Bound: Stridable>
    where Bound.Stride: SignedInteger {}
```

The `Stridable` protocol inherits from `Comparable`, so a generic argument satisfying the requirements of `CountableRange` also satisfies the requirements of `Range`. The only difference between the two types was that `CountableRange` conformed to more protocols than `Range`. After conditional conformances were added to the language, `CountableRange` no longer made sense to keep around as a separate type. To preserve source compatibility, Swift 4.2 replaced `CountableRange` with a generic type alias:

```
typealias CountableRange<Bound: Stridable> = Range<Bound>
    where Bound.Stride: SignedInteger
```

In Swift 4.1, the following were two extensions of different nominal types:

```
extension Range {...}

extension CountableRange {...}
```

As of Swift 4.2, the second extension is interpreted as a constrained extension of `Range` via the pass-through type alias `CountableRange`:

```
extension Range
    where Bound: Stridable,
          Bound.Stride: SignedInteger {...}
```

Example 10.6. The following type aliases are not pass-through type aliases.

1. The underlying type of this type alias is not a nominal type:

   ```
   typealias A<T> = () -> T
   ```

2. This type alias has the wrong number of generic parameters:

   ```
   typealias B<Value> = Dictionary<AnyHashable, Value>
   ```

3. This type alias does not substitute the second generic parameter of the underlying type with the second generic parameter of the type alias:

   ```
   typealias C<Key, Value> = Dictionary<Key, (Value) -> Int>
   ```

10.4. Conditional Conformances

A conformance written on a nominal type or unconstrained extension implements the protocol's requirements for all specializations of the nominal type, and we call this an *unconditional* conformance. A conformance declared on a *constrained* extension is what's known as a *conditional conformance*, which implements the protocol requirements only for those specializations of the extended type which satisfy the requirements of the extension. Conditional conformances were introduced in Swift 4.2 [113].

For example, arrays have a natural notion of equality, defined in terms of an equality operation on the element type. However, we don't require that every array has an `Equatable` element type. Instead, we declare a conditional conformance of `Array` to `Equatable`, for those arrays whose element type is `Equatable`:

```
struct Array<Element> {...}

extension Array: Equatable where Element: Equatable {
  func ==(lhs: Self, rhs: Self) -> Bool {
    guard lhs.count == rhs.count else { return false }

    for i in 0..<lhs.count {
      // 'Element: Equatable' conditional requirement is used here
      guard lhs[i] == rhs[i] else { return false }
    }

    return true
  }
}
```

More complex conditional requirements can also be written. We previously discussed overlapping conformances and coherence in Section 7.1, and conditional conformances inherit an important restriction. A nominal type can still only conform to a protocol once, even if that conformance is conditional, so in particular, overlapping conditional conformances are not supported and we emit a diagnostic:

```
struct G<T> {}

protocol P {}

extension G: P where T == Int {}
extension G: P where T == String {}  // error
```

Computing conditional requirements. A conditional conformance stores a list of *conditional requirements*. If G is the generic signature of the constrained extension where the conformance was declared, and H is the generic signature of the conforming type, then certainly G satisfies all requirements of H. If the converse is also true, we have an unconditional conformance. Otherwise, the conditional requirements of the conformance are precisely those requirements of G not satisfied by H. A simple example is the conditional conformance of `Dictionary` to `Equatable`:

```
extension Dictionary: Equatable where Value: Equatable {...}
```

The generic signature of `Dictionary` is `<Key, Value where Key: Hashable>`. The generic signature of our constrained extension has two requirements:

```
<Key, Value where Key: Hashable, Value: Equatable>
```

The first requirement is already satisfied by the generic signature of `Dictionary` itself; the second requirement is the conditional requirement of our conformance.

We compute the conditional requirements by handing Algorithm 9B the requirements of G together with the forwarding substitution map $1_{[\![H]\!]}$. The algorithm outputs a list of failed and unsatisfied requirements. They're not really "failed" or "unsatisfied" though; instead, the concatenation of these two lists is precisely the list of conditional requirements in our normal conformance.

The "failed" case corresponds to a conditional requirement whose subject type is not even a valid type parameter of H. The [`T.Element: Hashable`] requirement below has the subject type `T.Element`, which was defined by [`T: Sequence`], which is itself a conditional requirement, so [`G: Sequence`] has two conditional requirements:

```
struct G<T> {}

extension G: Sequence where T: Sequence, T.Element: Hashable {...}
```

Specialized conditional conformances. Building upon Section 7.1, we now describe how substitution relates to conditional conformances. If $\texttt{X}_d$ is the declared interface type of some nominal type declaration d that *unconditionally* conforms to `P`, and $\texttt{X} = \texttt{X}_d \otimes \Sigma$ is a specialized type of d for some substitution map Σ, then looking up the conformance of `X` to `P` returns a specialized conformance with conformance substitution map Σ:

$$\langle \texttt{P} \rangle \otimes \texttt{X} = [\texttt{X}_d\texttt{: P}] \otimes \Sigma$$

If $[\texttt{X}_d\texttt{: P}]$ is conditional though, we cannot take Σ to be the context substitution map of `X`. The type witnesses of $[\texttt{X}_d\texttt{: P}]$ might contain type parameters of the constrained extension, and not just the conforming type; however the context substitution map of `X` has the generic signature of the conforming type. We instead define Σ to be the context substitution map of `X` for the generic signature of the constrained extension, as in Section 9.2. Indeed, we find ourselves in the same situation as in member type resolution, when the referenced type declaration is declared in a constrained extension; we must perform some additional global conformance lookups to completely populate the substitution map.

The conditional requirements of the specialized conformance `[X: P]` are the substituted requirements obtained by applying Σ to each conditional requirement of $[\texttt{X}_d\texttt{: P}]$. This makes the following diagram commute for each conditional requirement R:

$$\begin{array}{ccc} [\texttt{X}_d\texttt{: P}] & \xrightarrow{\text{apply } \Sigma} & [\texttt{X: P}] \\ \Big\downarrow{\scriptstyle \text{get conditional requirement}} & & \Big\downarrow{\scriptstyle \text{get conditional requirement}} \\ R & \xrightarrow{\text{apply } \Sigma} & R \otimes \Sigma \end{array}$$

Consider the `[Array<τ_0_0>: Equatable]` conformance from the standard library. The generic signature of `Array` is `<τ_0_0>`, with no requirements, while the conformance context is the constrained extension with signature `<τ_0_0 where τ_0_0: Equatable>`. The context substitution map of `Array<Int>` for the constrained extension is:

$$\begin{aligned}\Sigma := \{&\texttt{τ_0_0} \mapsto \texttt{Int}; \\ &[\texttt{τ_0_0: Equatable}] \mapsto [\texttt{Int: Equatable}]\}\end{aligned}$$

We compose this substitution map with the normal conformance, and obtain a specialized conditional conformance, denoted `[Array<Int>: Equatable]`:

$$[\texttt{Array<τ_0_0>: Equatable}] \otimes \Sigma = [\texttt{Array<Int>: Equatable}]$$

The normal conformance has a single conditional requirement, `[τ_0_0: Equatable]`. We apply Σ to get the conditional requirement of the specialized conformance:

$$[\texttt{τ_0_0: Equatable}] \otimes \Sigma = [\texttt{Int: Equatable}]$$

Global conformance lookup. Global conformance lookup purposely does not check conditional requirements, allowing callers to answer two different questions:

1. Does this specialized type with its specific list of generic arguments conform to the protocol?
2. What requirements should a list of generic arguments satisfy to make the type conform?

When the first interpretation is desired, a convenience entry point can be used that combines global conformance lookup with Algorithm 9A to check any conditional requirements.

Checking conditional requirements is more subtle than checking for the presence of invalid conformances in the conformance substitution map. It is true that their presence indicates a conditional requirement is unsatisfied; for example, `Array<AnyObject>` does not conditionally conform to `Equatable`, since an `AnyObject` is not `Equatable`, and we get the following specialized conformance:

$$\begin{aligned}
&\langle\texttt{Equatable}] \otimes \texttt{Array<AnyObject>} \\
&= [\texttt{Array<}\tau\texttt{_0_0>: Equatable}] \otimes \{\tau\texttt{_0_0} \mapsto \texttt{AnyObject}; \\
&\qquad\qquad\qquad\qquad\qquad\qquad [\tau\texttt{_0_0: Equatable}] \mapsto \text{invalid}\} \\
&= [\texttt{Array<AnyObject>: Equatable}]
\end{aligned}$$

However, looking at the substitution map cannot tell us if other kinds of conditional requirements are unsatisfied, such as same-type, superclass, and layout requirements. For example, the conformance substitution map of [`Pair<Int, String>: Diagonal`] does not contain any invalid conformances, but it does not satisfy the conditional requirement [τ`_0_0 == `τ`_0_1`]:

```
protocol Diagonal {}
struct Pair<T, U> {}
extension Pair: Diagonal where T == U {}
```

Thus, conditional requirements must always be checked by Algorithm 9A, and not in some "ad-hoc" manner by digging through the substitution map.

Protocol inheritance. Protocol inheritance is modeled as an associated conformance requirement on `Self`, so for instance `Derived` has an associated conformance requirement [`Self: Base`]:

```
protocol Base {...}
protocol Derived: Base {...}
```

When checking a conformance to `Derived`, the conformance checker ensures that the conforming type satisfies the [`Self: Base`] requirement. When the conformance to `Base` is unconditional, this always succeeds, because the conformance declaration also *implies* an unconditional conformance to the base protocol:

```
struct Pair<T, U> {}
extension Pair: Derived {...}
// implies 'extension Pair: Base {}'
```

The nominal type's conformance lookup table synthesizes these implied conformances and makes them available to global conformance lookup. With conditional conformances, such implied conformances are not synthesized, because there is no way to guess what the conditional requirements should be. The conformance checker still checks the associated conformance requirement on `Self` though, so the user must first explicitly declare a conformance to each base protocol when writing a conditional conformance.

Suppose we wish for `Pair` to conform to `Derived` when [`τ_0_0 == Int`]:

```
extension Pair: Derived where T == Int {...}
```

The compiler will diagnose an error unless there is also an *explicit* conformance of `Pair` to `Base`. There are several possible ways to declare a conformance of `Pair` to `Base`. The simplest way is to conform unconditionally:

```
extension Pair: Base {...}
```

Things get more interesting if the conformance to `Base` is also conditional, because then the conformance to `Derived` only makes sense if the conditional requirements of [`Pair: Derived`] imply the conditional requirements of [`Pair: Base`]. We establish this condition as follows. There are three generic signatures in play here:

1. The signature of `Pair`; call it H.
2. The signature of the extension declaring conformance to `Base`; we call it G_1.
3. The signature of the extension declaring conformance to `Derived`; we call it G_2.

The conformance [`Pair: Derived`] satisfies the [`Self: Base`] associated conformance requirement of `Derived` if any specialization of `Pair` which satisfies the conditional requirements of (3) also satisfies the conditional requirements of (2). In the conformance checker, this falls out from the general case of checking associated requirements.

To each associated requirement, we apply the protocol substitution map for the normal conformance, followed by the forwarding substitution map for the generic signature of the conformance. In our example, this gives us the following substituted requirement:

$$[\texttt{Self: Base}] \otimes \Sigma_{[\texttt{Pair: Derived}]} \otimes 1_{[\![G_2]\!]} = [\texttt{Pair<}[\![\texttt{T}]\!]\texttt{, Int>: Base}]$$

Next, we ask Algorithm 9A if the substituted requirement is satisfied. This performs a global conformance lookup and checks its conditional requirements:

$$\langle \texttt{Base} \rangle \otimes \texttt{Pair<}[\![\texttt{T}]\!]\texttt{, Int>} = [\texttt{Pair<}[\![\texttt{T}]\!]\texttt{, Int>: Base}]$$

Since $\texttt{Pair<}[\![\texttt{T}]\!]\texttt{, Int>} \in \textsc{Type}([\![G_2]\!])$, we get $\langle \texttt{Base} \rangle \otimes \texttt{Pair<}[\![\texttt{T}]\!]\texttt{, Int>} \in \textsc{Conf}([\![G_2]\!])$. The conditional requirements of this conformance are the requirements of G_1 substituted with the primary archetypes of G_2. We must check if they are satisfied to determine the validity of the conformance to `Derived`. We now look at three different definitions of the conformance [`Pair: Base`], and see how each one impacts the validity of the conformance [`Pair: Derived`].

(★ Case 1.) The conditional requirements of the `Base` conformance might be identical to those of `Derived`:

```
extension Pair: Base where T == Int {...}
```

When checking the `Derived` conformance, we substitute the conditional requirement [`τ_0_0 == Int`] of `Base`:

$$[\texttt{τ_0_0 == Int}] \otimes \{\texttt{τ_0_0} \mapsto \texttt{Int}, \texttt{τ_0_1} \mapsto [\![\texttt{U}]\!]\} = [\texttt{Int == Int}]$$

The substituted requirement is satisfied, so the conformance [`Pair: Derived`] is valid.

(★ Case 2.) The conditional requirements of the `Base` conformance are allowed to be less constrained than `Derived`:

```
extension Pair: Base where T: Equatable {...}
```

When checking the `Derived` conformance, we substitute the conditional requirement [`τ_0_0: Equatable`] of `Base`:

$$[\texttt{τ_0_0: Equatable}] \otimes \{\texttt{τ_0_0} \mapsto \texttt{Int}, \texttt{τ_0_1} \mapsto [\![\texttt{U}]\!]\} = [\texttt{Int: Equatable}]$$

This is also satisfied, so again the conformance [`Pair: Derived`] is valid.

(★ Case 3.) However, the conditional requirements of `Base` are *not* permitted to be *more* constrained than `Derived`:

```
extension Pair: Base where U: Equatable {...}
```

When checking the `Derived` conformance, we substitute the conditional requirement [`τ_0_1: Equatable`] of `Base`:

$$[\texttt{τ_0_1: Equatable}] \otimes \{\texttt{τ_0_0} \mapsto \texttt{Int}, \texttt{τ_0_1} \mapsto [\![\texttt{U}]\!]\} = [[\![\texttt{U}]\!]\texttt{: Equatable}]$$

In the generic signature of the extension where the `Derived` conformance was declared, the archetype $[\![\texttt{U}]\!]$ does *not* conform to `Equatable`. So if [`Pair: Base`] is declared as above, the compiler must reject the conformance [`Pair: Derived`].

Termination. Conditional conformances can express non-terminating computation at compile time. The below code is taken from a bug report which remains unfixed for the time being [114]:

```
protocol P {}

protocol Q {
  associatedtype A
}

struct G<T: Q> {}

extension G: P where T.A: P {}

struct S: Q {
  typealias A = G<S>
}

func takesP<T: P>(_: T.Type) {}
takesP(G<S>.self)  // called here
```

The `takesP()` function has the generic signature `<τ_0_0 where τ_0_0: P>`. The last line calls this function with `G<S>` as the generic argument for `τ_0_0`. This substitution map for this call also needs to store the conformance [`G<S>: P`], but this conformance cannot actually be constructed. The normal conformance [`G<τ_0_0>: P`] is declared on a constrained extension with the following generic signature:

```
<τ_0_0 where τ_0_0: Q, τ_0_0.[Q]A: P>
```

To construct the specialized conformance [`G<S>: P`] from this normal conformance, we must first construct the conformance substitution map. The substitution map should map `τ_0_0` to `S`, and store a pair of conformances:

$$\begin{aligned}\Sigma := \{&\texttt{τ_0_0} \mapsto \texttt{S};\\ &[\texttt{τ_0_0: Q}] \mapsto [\texttt{S: Q}],\\ &[\texttt{τ_0_0.[Q]A: P}] \mapsto ???\}\end{aligned}$$

The conforming type of the first conformance is $\texttt{τ_0_0} \otimes \Sigma = \texttt{S}$, so the first conformance is the normal conformance [`S: Q`]. The conforming type of the second conformance is $\langle \texttt{Q|A} \otimes [\texttt{S: Q}] = \texttt{G<S>}$; so the second conformance is exactly [`G<S>: P`], the specialized conformance we're currently constructing! However, in the current model, a substitution map cannot form a cycle with a specialized conformance it contains.

Now, suppose we *could* represent a circular substitution map of this kind:

$$\begin{aligned}\Sigma := \{&\texttt{τ_0_0} \mapsto \texttt{S};\\ &[\texttt{τ_0_0: Q}] \mapsto [\texttt{S: Q}],\\ &[\texttt{τ_0_0.[Q]A: P}] \mapsto [\texttt{G<τ_0_0>: P}] \otimes \Sigma\}\end{aligned}$$

Unfortunately, this would still not allow our example to type check. The next issue we encounter is checking the conditional requirement, [τ_0_0.[Q]A: P]. Applying Σ gives us the substituted conditional requirement:

$$[\texttt{τ_0_0.[Q]A: P}] \otimes \Sigma = [\texttt{G<S>: P}]$$

So G<S> conditionally conforms to P if the conditional requirement [G<S>: P] is satisfied; this conditional requirement is satisfied if G<S> conditionally conforms to P. At this point, we are stuck in a loop, again. For now, the compiler crashes on this example while constructing the conformance substitution map; hopefully a future update to this book will describe the eventual resolution of this bug by imposing an iteration limit.

Our example only encodes an infinite loop, but conditional conformances can actually express arbitrary computation. This is shown for the Rust programming language, which also has conditional conformances, in [115]. We will study a different but related encoding of non-terminating compile-time computation in Section 12.3.

Soundness. In a valid program, we expect every substitution map to correctly model its input generic signature.

Definition 10.7. Let Σ be a substitution map with input generic signature G, and whose replacement types are fully concrete, so they do not contain any type parameters. We say that Σ is *well-formed*, if for each derived requirement R of G, the substituted requirement $R \otimes \Sigma$ is satisfied according to Algorithm 9A.

(We remark that the restriction on replacement types is not a real limitation; as we saw in Section 9.3, we can first compose Σ with the forwarding substitution map $1_{[\![H]\!]}$ for some generic signature H, to replace type parameters with archetypes.)

If G has an infinite theory of derived requirements, we cannot directly check if Σ is well-formed. Algorithm 9B only checks if Σ satisfies each *explicit* requirement of G, and this by itself is not sufficient. An immediate counterexample appears when the substituted requirement depends on a conformance that does not satisfy the associated requirements of its protocol. For example, we might declare a `Bad` type conforming to `Sequence` whose `Iterator` type witness does not conform to `IteratorProtocol`:

```
struct Bad: Sequence {
  typealias Iterator = Int  // error
}
```

The substitution map $\Sigma := \{\texttt{τ_0_0} \mapsto \texttt{Bad}; [\texttt{τ_0_0: Sequence}] \mapsto [\texttt{Bad: Sequence}]\}$ satisfies all of the explicit requirements of its generic signature, but it does not satisfy the derived requirement $[\texttt{τ_0_0.[Sequence]Iterator: IteratorProtocol}]$. However, this is not a problem, because we still diagnose an error when we check the conformance, so the program is rejected anyway.

Unfortunately, we can write down a substitution map that is not well-formed, and yet Algorithm 9B does not produce any diagnostics at all, including during conformance checking. This reveals a soundness hole with conditional conformances that remains unfixed for the time being. We start with two protocols:

```
protocol Bar {
  associatedtype Beer
  func brew() -> Beer
}

protocol Pub {
  associatedtype Beer
  func pour() -> Beer
}
```

We declare a function generic over types that conform to both `Bar` and `Pub`, so it has the generic signature `<τ_0_0 where τ_0_0: Bar, τ_0_0: Pub>`:

```
func both<T: Bar & Pub>(_ t: T) -> (T.Beer, T.Beer) {
  return (t.brew(), t.pour())
}
```

Now, we declare a `BrewPub` struct, and pass an instance of `BrewPub<Int>` to `both()`:

```
struct BrewPub<T> {}
let result = both(BrewPub<Int>())
```

For this call to type check, `BrewPub` must conform to `Bar` and `Pub`; assume for a moment these conformances already exist. Here is the substitution map for the call:

$$\begin{aligned}\Sigma := \{&\texttt{τ_0_0} \mapsto \texttt{BrewPub<Int>};\\ &[\texttt{τ_0_0: Bar}] \mapsto [\texttt{BrewPub<Int>: Bar}],\\ &[\texttt{τ_0_0: Pub}] \mapsto [\texttt{BrewPub<Int>: Pub}]\}\end{aligned}$$

Now, applying Σ to $\texttt{τ_0_0.[Bar]Beer}$ and $\texttt{τ_0_0.[Pub]Beer}$ projects the type witness for `Beer` from each conformance:

$$\texttt{τ_0_0.[Bar]Beer} \otimes \Sigma = \langle \texttt{Bar} | \texttt{Beer} \otimes [\texttt{BrewPub<Int>: Bar}]$$
$$\texttt{τ_0_0.[Pub]Beer} \otimes \Sigma = \langle \texttt{Pub} | \texttt{Beer} \otimes [\texttt{BrewPub<Int>: Pub}]$$

For Σ to be well-formed, both type witnesses must be canonically equal, because in the generic signature of `both()`, the bound dependent member types `τ_0_0.[Bar]Beer` and `τ_0_0.[Pub]Beer` are both equivalent to the unbound dependent type `τ_0_0.Beer`.

If `BrewPub` conforms to these protocols unconditionally, the redeclaration checking rules prevent us from declaring the two conformances to have distinct type witnesses:

```
extension BrewPub: Bar {
  typealias Beer = Float
  func brew() -> Float { return 0.0 }
}

extension BrewPub: Pub {
  typealias Beer = String  // error: invalid redeclaration
  func pour() -> String { return "" }
}
```

If the conformances are conditional though, neither type alias is visible from the other constrained extension, so they are not rejected as redeclarations. All that remains is to pick conditional requirements such that our generic argument type `Int` satisfies both:

```
extension BrewPub: Bar where T: Equatable {
  typealias Beer = Float
  func brew() -> Float { return 0.0 }
}

extension BrewPub: Pub where T: ExpressibleByIntegerLiteral {
  typealias Beer = String
  func pour() -> String { return "" }
}
```

Inside the body of `both()`, we're assuming the calls to `brew()` and `pour()` return the same type. However, what actually happens is that one returns a `Float` while the other returns a `String`, which results in undefined behavior. There are two potential solutions:

1. We could tighten the rules for redeclaration checking of type aliases in constrained extensions, requiring that all such type aliases have canonically equal underlying types. This would *reject the conditional conformances* above as invalid.
2. We could extend Algorithm 9B to detect conformances having incompatible type witnesses in the given substitution map. This would instead *reject the call* to `both()` written above as invalid.

The second approach is more desirable, and we will revisit this problem in Chapter 20.

10.5. Source Code Reference

Key source files:

- include/swift/AST/Decl.h
- lib/AST/Decl.cpp

`ExtensionDecl` *class*

Represents an extension declaration.

- `getExtendedNominal()` returns the extended type declaration. Returns `nullptr` if extension binding failed to resolve the extended type. Asserts if extension binding has not yet visited this extension.
- `computeExtendedNominal()` actually evaluates the request which resolves the extended type to a nominal type declaration. Only used by extension binding.
- `getExtendedType()` returns the written extended type, which might be a type alias type or a generic nominal type. This uses ordinary type resolution, so it only happens after extension binding. This is used to implement the syntax sugar described at the start of Section 10.3.
- `getDeclaredInterfaceType()` returns the declared interface type of the extended type declaration.
- `getSelfInterfaceType()` returns the self interface type of the extended type declaration. Different than the declared interface type for protocol extensions, where the declared interface type is the protocol type, but the self interface type is the protocol `Self` type.

Extension Binding

Key source files:

- include/swift/AST/NameLookup.h
- lib/AST/Decl.cpp
- lib/AST/NameLookup.cpp
- lib/Sema/TypeChecker.cpp

`bindExtensions()` *function*

Takes a `ModuleDecl *`, which must be the main module, and implements Algorithm 10A.

`ExtendedNominalRequest` *class*

The request evaluator request which resolves the extended type to a nominal type declaration. This calls out to a restricted form of type resolution which does not apply generic arguments or perform substitution.

`directReferencesForTypeRepr()` *function*

Takes a `TypeRepr *` and the extension's parent declaration context, which is usually a source file. Returns a vector of `TypeDecl *`. Some of the type declarations might be type alias declarations; the next entry point fully resolves everything to a list of nominal types.

`resolveTypeDeclsToNominal()` *function*

Takes a list of `TypeDecl *` and outputs a list of `NominalTypeDecl *`, by resolving any type alias declarations to nominal types, using the same restricted form of type resolution recursively.

If the output list is empty, type resolution failed and the extension cannot be bound. If the output list contains more than one entry, type resolution produced an ambiguous result. For now, we always take the first nominal type declaration from the output list in that case.

Direct Lookup and Lazy Member Loading

Key source files:

- `lib/AST/NameLookup.cpp`
- `include/swift/AST/LazyResolver.h`

`DirectLookupRequest` *class*

The request evaluator request implementing direct lookup. The entry point is the `NominalTypeDecl::lookupDirect()` method, which was introduced in Section 2.6. This request is uncached, because the member lookup table effectively implements caching outside of the request evaluator.

To understand the implementation of `DirectLookupRequest::evaluate()`, one can start with the following functions:

- `prepareLookupTable()` adds all members from extensions without lazy loaders, and members loaded so far from extensions with lazy loaders, without marking any entries as complete.
- `populateLookupTableEntryFromLazyIDCLoader()` asks a lazy member loader to load a single entry and adds it to the member lookup table.
- `populateLookupTableEntryFromExtensions()` adds all members from extensions that do not have lazy member loaders.

`MemberLookupTable` *class*

Every `NominalTypeDecl` has an instance of `MemberLookupTable`, which maps declaration names to lists of `ValueDecl`. The most important methods:

- `find()` returns an iterator pointing at an entry, which might be missing or incomplete.
- `isLazilyComplete()` answers if an entry is complete.
- `markLazilyComplete()` marks an entry as complete.

`LazyMemberLoader` *class*

Abstract base class implemented by different kinds of modules to look up top-level declarations and members of types and extensions. For the main module, this consults lookup tables built from source, for serialized modules this deserializes records and builds declarations from them, for imported modules this constructs Swift declarations from Clang declarations.

Constrained Extensions

Key source files:

- `lib/Sema/TypeCheckDecl.cpp`
- `lib/Sema/TypeCheckGeneric.cpp`

The `GenericSignatureRequest` was previously introduced in Section 11.5. It delegates to a pair of utility functions to implement special behaviors of extensions.

`collectAdditionalExtensionRequirements()` *function*

Collects an extension's requirements from the extended type, which handles extensions of pass-through type aliases (`extension CountableRange {...}`) and extensions of generic nominal types (`extension Array<Int> {...}`).

`isPassthroughTypealias()` *function*

Answers if a generic type alias satisfies the conditions that allow it to be the extended type of an extension.

Conditional Conformances

Key source files:

- `include/swift/AST/GenericSignature.h`
- `include/swift/AST/ProtocolConformance.h`
- `lib/AST/GenericSignature.cpp`
- `lib/AST/ProtocolConformance.cpp`
- `lib/Sema/TypeCheckProtocol.cpp`

`checkConformance` *function*

A utility function that first calls `lookupConformance()` (Section 7.7), and then checks any conditional requirements using `checkRequirements()`, described in Section 9.5.

The `NormalProtocolConformance` and `SpecializedProtocolConformance` classes were previously introduced in Section 7.7.

`NormalProtocolConformance` *class*

- `getConditionalRequirements()` returns an array of conditional requirements; this is non-empty exactly when this is a conditional conformance.

`SpecializedProtocolConformance` *class*

- `getConditionalRequirements()` applies the conformance substitution map to each conditional requirement of the underlying normal conformance.

`GenericSignatureImpl` *class*

See also Section 5.7.

- `requirementsNotSatisfiedBy()` returns an array of those requirements of this generic signature not satisfied by the given generic signature. This is used for computing the conditional requirements of a `NormalProtocolConformance`.

11. Building Generic Signatures

Building a generic signature from user-written requirements is something we glossed over before, and it's time to detail it now. We're going to fill in missing steps between the syntax for declaring generic parameters and stating requirements of Section 4.1 and Section 4.2, and the generic signature of Chapter 5, a semantic representation of the generic parameters and requirements of a declaration.

The requirements in a generic signature must be reduced, minimal, and ordered in a certain way (we will see the formal definitions in Section 11.4). To build a generic signature then, we must convert user-written requirements into an *equivalent* set of requirements that satisfy these additional invariants. We're going to start from the entry points for building generic signatures and peel away the layers:

- The **generic signature request** lazily builds a generic signature for a declaration written in source. This is a multi-step process; we call out to type resolution to construct the user-written requirements from syntactic representations, and then process them in stages to obtain minimal requirements. We will see how this request decomposes the declaration's syntactic forms before delegating to the **inferred generic signature request** to do most of the work.

- The **abstract generic signature request** builds a generic signature from a list of requirements that already "exists" but may not be minimal. This skips the name lookup and type resolution, but otherwise processes requirements in a similar way.

- All generic signatures are ultimately created by the **primitive constructor**, which takes generic parameters and requirements known to already obey the necessary invariants, and simply allocates and initializes the semantic object.

  Outside of the generics implementation, the primitive constructor is used in a handful of cases when it is already known the requirements satisfy the necessary conditions. When deserializing a generic signature from a serialized module for example, we know it satisfied the invariants when it was serialized.

  We also use the primitive constructor to build the protocol generic signature `<Self where Self: P>` for a protocol `P`, because we know it satisfies the invariants by definition. Anywhere else we need to build a generic signature "from scratch," we use the **abstract generic signature request** instead.

- The **requirement signature request** lazily builds a requirement signature for a protocol written in source. The flow resembles the inferred generic signature request; we start by resolving user-written requirements, then perform minimization. There's no "abstract" equivalent of the requirement signature request; a requirement signature is always attached to a specific named protocol declaration.

Now we look at each involved request in detail.

Generic signature request. Two easy cases are handled first:

1. If the declaration is a protocol or an unconstrained protocol extension, we build the protocol generic signature `<Self where Self: P>` using the primitive constructor.
2. A declaration having neither a generic parameter list nor a trailing `where` clause simply inherits the generic signature from its parent context. If the declaration is at the top level of a source file, we return the empty generic signature. Otherwise, we recursively evaluate the **generic signature request** against the parent.

In every other case, the generic signature request kicks off the lower-level **inferred generic signature request**, handing it a list of arguments:

1. The generic signature of the parent context, if any.

   The generic signature of a nested declaration extends that of the parent with additional generic parameters and requirements.

2. The current generic context's generic parameter list, if any.

   At least one of the first two inputs must be specified; absent a parent signature or any generic parameters to add, the resulting generic signature is necessarily empty, and the caller handles this case by not evaluating the request at all.

3. The current generic context's trailing `where` clause, if any.

   The inferred generic signature request will call out to type resolution to resolve the requirement representations written here into requirements.

4. Any additional requirements to add.

   This enables a shorthand syntax for declaring a constrained extension by writing a generic nominal type or a "pass-through" generic type alias as the extended type (Section 10.3).

5. A list of types eligible for requirement inference.

   If we're building the generic signature of a function or subscript declaration, this consists of the declaration's parameter and return types; otherwise, it is empty (Section 11.1).

6. A source location for diagnostics.

Inferred generic signature request. A possibly apocryphal story says the name "inferred generic signature request" was chosen because "requirement inference" is one of the steps below, but this does not infer the generic signature in any real sense. Rather, this request transforms user-written requirements into a minimal, reduced form via a multi-step process shown in Figure 11.1:

1. **Requirement resolution** builds user-written requirements, from constraint types stated in generic parameter inheritance clauses and requirement representations in the trailing `where` clause. This happens in structural resolution stage (Chapter 9), so the resolved requirements may contain unbound dependent member types, which will reduce to bound dependent member types in requirement minimization.

   Diagnostics are emitted here if type resolution fails.

2. **Requirement inference** (Section 11.1) allows certain requirements to be omitted in source if they can be inferred. These inferred requirements are added to the list of user-written requirements.

3. **Decomposition and desugaring** (Section 11.2) transforms requirements collected by the first two stages, rewriting conformance and same-type requirements into a simpler form, and detecting trivial requirements that are always or never satisfied.

   Diagnostics are emitted here if a requirement is always unsatisfied as written.

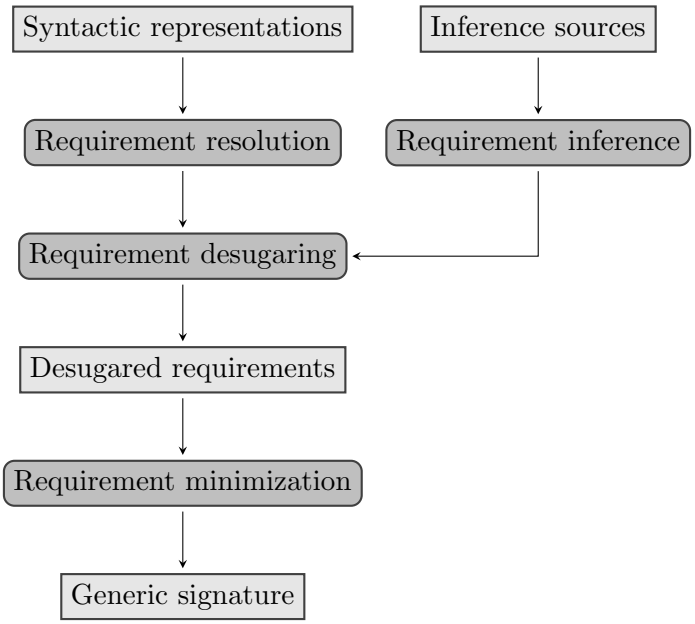
4. **Requirement minimization** is where the real magic happens; we will talk about the invariants established by this in Section 11.4, and pick up the topic again when we get to the construction of a requirement machine from desugared requirements in Chapter 15.

   Diagnostics are emitted here if no substitution map can satisfy the generic signature as written; the signature is said to contain *conflicting requirements* in this case.

The diagnostics mentioned above are emitted at the source location of the declaration, which is given to the request. This source location is used for one more diagnostic, an artificial restriction of sorts. Once we have a generic signature, we ensure that every innermost generic parameter is a reduced type. If a generic parameter is not reduced, it must be equivalent to a concrete type or an earlier generic parameter; it serves no purpose and should be removed. We diagnose an error in `-language-mode 6`, or a warning in earlier language modes:

```
// error: same-type requirement makes generic parameter 'T' non-generic
func add<T>(_ lhs: T, _ rhs: T) -> T where T == Int {
  return lhs + rhs
}
```

Figure 11.1.: Overview of the inferred generic signature request



The restriction only concerns innermost generic parameters, so we allow this:

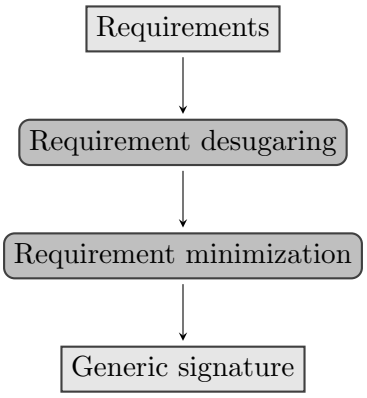
```
struct Outer<T> {
  func f() where T == Int {...}
}
```

And this:

```
extension Outer where Element == Int {
  func f() {...}
}
```

Abstract generic signature request. This request builds a generic signature from a list of generic parameters and requirements provided by the caller. The flow here is simpler, as shown in Figure 11.2. As input, it takes an optional parent generic signature, a list of generic parameter types to add, and a list of requirements to add. At least one of the first two parameters must be specified; if there is no parent generic signature and there are no generic parameters to add, the result would be the empty generic signature, which the caller is expected to handle by not evaluating this request at all.

Figure 11.2.: Overview of the abstract generic signature request



Like the inferred generic signature request, the abstract generic signature request decomposes, desugars and minimizes requirements; for this reason, it is preferred over using the primitive constructor. Often this request is invoked with a list of substituted requirements obtained by applying a substitution map to each requirement of some original generic signature. This request is used in various places:

- Computing the generic signature of an opaque type declaration (Chapter 13).
- Computing the generic signature for an opened existential type (Section 14.1).
- Checking that a method override declared by a subclass satisfies the generic requirements of the overridden method in the superclass.

This request does not perform requirement inference. No diagnostics are emitted either; instead, an error value is returned which is checked by the caller.

Requirement signature request. This request builds the requirement signature for a given *protocol component*, or a collection of one or more mutually-recursive protocols, as we will see in Section 15.1. The evaluation function begins by evaluating two subordinate requests to collect the user-written requirements of each protocol:

- The **structural requirements request** collects associated requirements from the protocol's inheritance clause, associated type inheritance clauses, any `where` clauses on the protocol's associated types, and the `where` clause on the protocol itself. Refer to Section 4.3 for a description of the syntax.
- The **type alias requirements request** collects protocol type aliases and converts them to same-type requirements. These are discussed further in Section 17.5.

The requirement signature request takes these user-written associated requirements, and decomposes, desugars and minimizes them, much like the requirements of a generic signature. There is also a primitive constructor for requirement signatures. It is the last step of the requirement signature request. We also use it after reading a protocol from a serialized module, because the requirements were already minimal when serialized.

Protocol inheritance clauses. When searching for a member of a protocol, name lookup must also visit inherited protocols. Consider this example:

```
protocol Base {
  associatedtype Other: Base
  typealias Salary = Int
}

protocol Good: Base {
  typealias Income = Salary
}
```

The underlying type of `Income` in `Good` refers to `Salary` in `Base`. Protocol inheritance relationships are encoded in the protocol's requirement signature. A concrete type that conforms to `Good` must also conform to `Base`, so `Good` has an associated conformance requirement [`Self: Base`]. Name lookup cannot issue generic signature queries though, because building the requirement signature depends on type resolution, which depends on name lookup. To avoid request cycles, name lookup must interpret the inheritance clause of a protocol directly, without interfacing with generics. This introduces a minor limitation, which we now describe.

A general fact is that the protocol inheritance relation is transitive, so `Most` also inherits from `Base` here, because `Most` inherits from `Good`, and `Good` inherits from `Base`:

```
protocol Most: Good {}
```

We can understand this behavior by writing down a derivation for the requirement [`τ_0_0: Base`] in the protocol generic signature $G_{\texttt{Most}}$. We start with [`τ_0_0: Most`] and apply the associated conformance requirements [`Self: Good`]$_{\texttt{Most}}$ and [`Self: Base`]$_{\texttt{Good}}$:

1. [`τ_0_0: Most`] (Conf)
2. [`τ_0_0: Good`] (AssocConf 1)
3. [`τ_0_0: Base`] (AssocConf 2)

When the protocol inheritance relationship is a syntactic consequence of the inheritance clause, we can write down a derivation like the above where every step has a subject type

of `τ_0_0`. This is why a name lookup into `Good` knows to look into `Base`. On the other hand, it is possible to devise a protocol declaration where [`τ_0_0: Base`] is a non-trivial consequence of a same-type requirement between `τ_0_0` and some other type parameter. For example, here we have $G_{\texttt{Bad}} \vdash$ [`τ_0_0: Base`], which is not immediately apparent, because nothing is stated in the protocol's inheritance clause:

```
protocol Bad {
  associatedtype Tricky: Base where Self == Tricky.Other
  typealias Income = Salary  // error
}
```

We diagnose an error, because we are unable to resolve `Salary` as a member of `Bad`. To understand why, notice that the derivation $G_{\texttt{Bad}} \vdash$ [`τ_0_0: Base`] involves the associated same-type requirement $[\texttt{Self == Self.Tricky.Other}]_{\texttt{Bad}}$, so it is not a consequence of the protocol's syntactic inheritance clause:

1. [`τ_0_0: Bad`] (Conf)
2. [`τ_0_0.Tricky: Base`] (AssocConf 1)
3. [`τ_0_0.Tricky.Other: Base`] (AssocConf 2)
4. [`τ_0_0 == τ_0_0.Tricky.Other`] (AssocSame 1)
5. [`τ_0_0: Base`] (SameConf 3 4)

A special code path detects this problem and diagnoses a warning to explain what is going on. After building a protocol's requirement signature, the **type-check primary file request** verifies that any conformance requirements known to be satisfied by `Self` actually appear in the protocol's inheritance clause, or `where` clause entries with a subject type of `Self`.

In our example, we discover the unexpected derived requirement [`τ_0_0: Base`] of $G_{\texttt{Bad}}$, but at this point it is too late to attempt the failed name lookup of `Salary` again. The compiler instead suggests that the user should explicitly state the inheritance from `Base` in the inheritance clause of `Bad`:

```
$ swiftc bad.swift

bad.swift:8:22: error: cannot find type ‘Salary’ in scope
  typealias Income = Salary
                     ^~~~~~
bad.swift:6:10: warning: protocol ‘Bad’ should be declared to refine
‘Base’ due to a same-type constraint on ‘Self’
protocol Bad {
         ^
```

A mistake of this sort is baked into the standard library's `SIMDScalar` protocol. The `Self` type of `SIMDScalar` must conform to all of `Equatable`, `Hashable`, `Encodable` and `Decodable`, via an associated same-type requirement:

```
public protocol SIMDScalar {
  ...
  associatedtype SIMD2Storage: SIMDStorage
    where SIMD2Storage.Scalar == Self
  ...
}

public protocol SIMDStorage {
  associatedtype Scalar: Codable, Hashable
  ...
}
```

However, these requirements are not stated in `SIMDScalar`'s inheritance clause. Since the inheritance clause of a protocol is part of the Swift ABI, this omission cannot be fixed at this point, so the type checker specifically muffles the warning when building the standard library.

11.1. Requirement Inference

Consider a function that returns the unique elements of a collection:

```
func uniqueElements<S: Sequence>(_ seq: S) -> Set<S.Element>
    where S.Element: Hashable {...}
```

When resolving the function's return type, we must establish that the generic argument `τ_0_0.Element` satisfies the requirements in the generic signature of the `Set` type declared in the standard library:

```
struct Set<Element: Hashable> {...}
```

The sole substituted requirement is satisfied (Section 9.3) because the generic signature of `uniqueElements()` contains the requirement [`τ_0_0.Element: Hashable`]. In fact, this requirement *has* to be part of our function's generic signature, for the return type, and thus the overall function declaration, to be valid at all. The *requirement inference* feature allows us to omit this requirement:

```
func uniqueElements<S: Sequence>(_ seq: S) -> Set<S.Element>
    /* where S.Element: Hashable */ {...}
```

These *inferred requirements* are not the derived requirements in the sense of Section 5.2. They appear in the generic signature, along with user-written requirements; they are not consequences of other requirements. Both declarations of `uniqueElements()` above really are identical, and in particular they have the *same* generic signature with an explicit conformance requirement [`τ_0_0.Element: Hashable`]. Consumers of generic signatures do not know or care about requirement inference.

Inferred requirements. When building the generic signature of a declaration, we collect inferred requirements by visiting type representations written in the following specific positions:

1. The parameter and return types of function and subscript declarations, if the generic declaration we're given is one of those. In our `uniqueElements()` example, we infer requirements from the function's return type.

2. Types appearing in inheritance clauses of generic parameter declarations. The only interesting case is a generic superclass bound. Here, the generic signature of `Foo` has the explicit superclass requirement [`τ_0_0: Base<τ_0_1>`] and an inferred requirement [`τ_0_1: Equatable`], the latter written in the commented-out `where` clause:

```
class Base<T: Equatable> {...}
struct Foo<T: Base<U>, U>
    /* where U: Equatable */ {...}
```

3. Types appearing within the requirements of a `where` clause; for example, the right-hand side of a same-type requirement. Here, we get the inferred requirement [`τ_0_1: Hashable`]:

```
struct Foo<T: Sequence, U>
    where T.Element == Set<U> /*, U: Hashable */ {...}
```

4. The typed throws feature, introduced with Swift 6 [116], allows specifying the type of error thrown by a function. For function, subscript and constructor declarations, we infer a conformance requirement to `Error` with the declaration's thrown error type as the subject type:

```
func f<E>(_: Int) throws(E) /* where E: Error */ {...}
```

   If requirement inference encounters a function type in any other position, it also considers the thrown error type there:

```
func f<E>(_: () throws(E) -> ()) /* where E: Error */ {...}
```

(From this list, we see that types appearing in the *body* of a declaration do not participate in requirement inference; it would be quite unfortunate if the interface type of a function, for example, could not be determined without type checking the function body.)

We resolve each of the enumerated type representations in the structural resolution stage, because the generic signature of the current declaration isn't known; we're building it right now. Let's say that H is the generic signature being built; we can still talk about H in the abstract, we just can't define anything concrete to depend on generic signature queries against H. The resolved type possibly contains type parameters of H, so it is element of $\textsc{Type}(H)$. From the resolved type, we obtain a set of inferred requirements by walking its child types recursively. We look for children that are generic nominal types or generic type alias types, and extract a substitution map from each one:

- For a generic nominal type, we take the context substitution map (Section 6.1).
- For a generic type alias type, we take the substitution map stored inside the type.

Let's call this substitution map Σ, and let G be the input generic signature of Σ. This is the generic signature of the referenced nominal type or type alias declaration. The output generic signature of Σ is H, so $\Sigma \in \textsc{Sub}(G, H)$. We proceed as if we were checking generic arguments, applying Σ to each requirement of G. This gives us a substituted requirement in $\textsc{Req}(H)$. (Recall that when checking generic arguments in the interface stage, we worked with archetypes, so we'd get something in $\textsc{Req}(\llbracket H \rrbracket)$; here, we instead want the substituted requirement to talk about type parameters).

In all examples we've seen so far, the inferred requirement's subject type was a type parameter of H, and the inferred requirement was just added to H unchanged. To preview the next section, the requirement's subject type might be fully concrete after substitution; for example, here we get the useless inferred requirement [`Int: Hashable`]:

```
func f<T>(_: T, _: Set<Int>) {}
```

This requirement is not a statement about the type parameters of `f()` at all; it is just "always true," so it does not affect the generic signature of `f()`. A more complex scenario is possible with conditional conformances. In the next example, the inferred requirement is [`τ_0_0: Hashable`]:

```
func f<T>(_: Set<Array<T>>) /* where T: Hashable */ {}
```

From the generic nominal type `Set<Array<τ_0_0>>`, we obtain the inferred conformance requirement [`Array<τ_0_0>: Hashable`]. The standard library declares a conditional conformance of `Array` to `Hashable` when the element type is `Hashable`. We replace our original requirement with the conditional requirements of this conformance, which gives us [`τ_0_0: Hashable`].

In general, when a requirement's subject type is not a type parameter, we repeatedly rewrite it into a (possibly empty) set of simpler requirements, until only requirements about type parameters remain; this *requirement desugaring* procedure is described in the next section.

After desugaring, inferred requirements are passed on to requirement minimization, where they might become redundant. For example, the user might re-state all inferred requirements explicitly, as in our original version of `uniqueElements()`; these duplicate requirements are eliminated by requirement minimization. More elaborate examples of redundant inferred requirements can also be constructed. In the below, we infer the requirement [`τ_0_0.Iterator: IteratorProtocol`], but it's not an explicit requirement of the generic signature of `f()` because it can be derived from [`τ_0_0: Sequence`]:

```
struct G<T: IteratorProtocol> {}

func f<T: Sequence>(_: T) -> G<T.Iterator> {...}
```

Of course, the user can explicitly re-state the redundant requirement in the `where` clause just for good luck, which would make it twice redundant, in a way.

Outer generic parameters. The outer generic parameters of a declaration can be subject to inferred requirements. Consider a generic struct with three methods:

```
struct G<T, U> {
  func f<V>(_: Set<U>, _: V) /* where U: Hashable */ {}
  func f(_: Set<U>) where T: Equatable /*, U: Hashable */ {}
  func f(_: Set<U>) {}  // error!
}
```

We infer requirements if the declaration has generic parameters *or* a `where` clause, so in the first two, we infer [`τ_0_1: Hashable`]. The third method inherits the generic signature of the struct, so the requirement is not satisfied and we diagnose an error.

Generic type aliases. A reference to a generic type alias resolves to a sugared type alias type (Section 3.2). This sugared type prints as written when it appears in diagnostic messages, but it is canonically equal to its substituted underlying type, behaving like it otherwise. So here, the interface type of "`x`" prints as `OptionalElement<Array<Int>>`, but it is canonically equal to its substituted underlying type, `Optional<Int>`:

```
typealias OptionalElement<T: Sequence> = Optional<T.Element>

let x: OptionalElement<Array<Int>> = ...
```

In the above, we form the substituted underlying type `Optional<Int>` by applying a substitution map that replaces `τ_0_0` with `Array<Int>` to the underlying type of the type alias declaration, `Optional<T.Element>`. This gives us the structural components of a type alias type: a reference to a type alias declaration, a substituted underlying type, and a substitution map. The substitution map is used when printing the sugared type's generic arguments. Crucially to our topic at hand, this substitution map is also considered by requirement inference, making this one of a handful of language features where the appearance of a sugared type *does* have a semantic effect. In this example, we infer the requirement [`τ_0_0: Sequence`] from considering `OptionalElement<τ_0_0>`:

```
func maybePickElement<T>(_ sequence: T) -> OptionalElement<T>
```

Mostly of theoretical interest is the fact that the underlying type of a type alias need not mention the generic parameter types of the type alias at all. Before parameterized protocol types were added to the language, a few people discovered a funny trick to simulate something similar:

```
typealias SequenceOf<T, E> = Any
    where T: Sequence, T.Element == E
```

The underlying type of `SequenceOf` is just `Any`, and it's generic signature has the two requirements [`τ_0_0: Sequence`] and [`τ_0_1 == τ_0_0.Element`]. Now, we can state this type alias in the inheritance clause of a generic parameter declaration:

```
func sum<S: SequenceOf<S, Int>>(_: T) {...}
```

The inheritance clause entry introduces the requirement [`τ_0_0: Any`]; this has no effect, because `Any` is the empty protocol composition. However, requirement inference also visits the type alias type `SequenceOf<S, Int>`, which is the whole trick. This type alias type has the following substitution map:

$$\Sigma := \{\texttt{τ_0_0} \mapsto \texttt{τ_0_0},\ \texttt{τ_0_1} \mapsto \texttt{Int};\ [\texttt{τ_0_0: Sequence}] \mapsto [\texttt{τ_0_0: Sequence}]\}$$

Applying Σ to the generic signature of `SequenceOf` gives us our two inferred requirements:

$$[\texttt{τ_0_0: Sequence}] \otimes \Sigma = [\texttt{τ_0_0: Sequence}]$$
$$[\texttt{τ_0_1 == τ_0_0.Element}] \otimes \Sigma = [\texttt{τ_0_1 == Int}]$$

These requirements survive minimization and appear in the generic signature of `sum()`. We get the same generic signature as both of the following:

```
func sum<S: Sequence<Int>>(_: T) {...}
func sum<S: Sequence>(_: T) where S.Element == Int {...}
```

Protocols. Unlike the inferred generic signature request, the requirement signature request does not perform requirement inference; all associated requirements imposed on the concrete conforming type must be explicitly stated in source. For example, we must state the [`Self.Particle: Hashable`] requirement below, because otherwise the type `Set<Self.Particle>` appearing in the same-type requirement would not satisfy `Set`'s [`τ_0_0: Hashable`] requirement:

```
protocol Cloud {
  associatedtype Particle /* must be : Hashable */
  associatedtype Particles: Sequence
      where Particles.Element == Set<Particle>  // error
}
```

The technical reason for this restriction is the following. Before we can build the requirement signature of a protocol, we must construct the *protocol dependency graph*. This graph, which we will meet in Section 12.3, encodes the relation between each protocol, and any protocols that appear on the right-hand sides of its associated conformance requirements. A key property is that this graph can be recovered by performing name lookup operations alone, without queries against other requirement signatures or generic signatures. If we could infer requirements in protocols, this would no longer hold; we could define `Cloud` as above without explicit mention of `Hashable`, so our protocol dependency graph would have a "non-obvious" edge relation.

11.2. Decomposition and Desugaring

Before proceeding to requirement minimization, we eliminate some unneeded generality in the specification of requirements. To backtrack a little, here is where we are:

- If we're evaluating the **inferred generic signature request**, we're about half-way through Figure 11.1; we've gathered a set of user-written and inferred requirements that together describe the type parameters of a generic declaration.
- If we're evaluating the **abstract generic signature request**, we *start* here; as shown in Figure 11.2, the caller gives us a set of requirements as input.

In the derived requirements formalism, the right-hand side of a conformance requirement is always a protocol type, however in the syntax we can also write a protocol composition type, or a parameterized protocol type. Such conformance requirements must be split up—this is *requirement decomposition*. We also recall that the left-hand side of a derived requirement is always a type parameter, whereas requirement inference (or even the user) can write down a requirement with any subject type. Such requirements are also split up into zero or more simpler requirements, which gives us *requirement desugaring*.

Decomposition. This formalizes the syntax sugar from Section 4.2 and Section 4.3. For example, the standard library defines the `Codable` type alias, whose underlying type is a composition of two protocols, `Decodable` and `Encodable`:

```
typealias Codable = Decodable & Encodable
```

We can state conformance to `Codable` in the inheritance clause of a generic parameter:

```
func ride<Horse: Codable>(_: Horse) {}
```

We can understand the above by decomposing [`τ_0_0: Decodable & Encodable`] into the two conformance requirements [`τ_0_0: Decodable`] and [`τ_0_0: Encodable`]. The right-hand side of a conformance requirement might also be parameterized protocol type, which is sugar for a series of same-type requirements, constraining each of the protocol's primary associated types to the corresponding generic argument:

```
func search<E, S: Sequence<E>>(...) {}
```

The conformance requirement [`τ_0_1: Sequence<τ_0_0>`] written above decomposes into two requirements, [`τ_0_1: Sequence`] and [`τ_0_1.[Sequence]Element == τ_0_0`].

We now present requirement decomposition in the form of an algorithm. Mostly this is a restatement of the above, with additional details and a couple of edge cases.

Algorithm 11A (Decompose conformance requirement)**.** Takes a list of conformance requirements as input. Outputs a new list of equivalent requirements where all conformance requirements have a protocol type on the right hand side.

1. Initialize an empty list to collect the output requirements.
2. Add all input requirements to the worklist.
3. (Check) If the worklist is empty, return the output list.
4. (Loop) Remove a conformance requirement [`T: X`] from the worklist, where `X` is a "protocol-like" type, with one of the three kinds below.
5. (Base case) If `X` is a protocol type, output the conformance requirement [`T: X`].
6. (Composition) If `X` is a protocol composition type $\texttt{M}_1$ `&` `...` `&` $\texttt{M}_n$, visit each member $\texttt{M}_i \in \texttt{X}$. If $\texttt{M}_i$ is a class type, output the superclass requirement [`T:` $\texttt{M}_i$]. If $\texttt{M}_i$ is `AnyObject`, output the layout requirement [`T: AnyObject`]. Otherwise, $\texttt{M}_i$ is either a protocol type, or it can be decomposed further. Add the conformance requirement [`T:` $\texttt{M}_i$] to the worklist.

7. (Parameterized) If `X` is a parameterized protocol type `P<B`$_1$`, ..., B`$_n$`>` with base protocol type `P` and generic arguments `B`$_i$, decompose the requirement as follows:
   a) Output the conformance requirement [`T: P`].
   b) For each primary associated type `A`$_i$ of `P`, apply the protocol substitution map for `T` to the declared interface type of `A`$_i$:

$$\texttt{Self.[P]A}_i \otimes \{\texttt{Self} \mapsto \texttt{T};\ [\texttt{Self: P}] \mapsto [\texttt{T: P}]\} = \texttt{T.[P]A}_i$$

   Output the same-type requirement [`T.[P]A`$_i$ `== G`$_i$].

8. (Error) If `X` is anything else, we have an invalid requirement, for example [`T: Int`]. Diagnose an error.

9. (Next) Go back to Step 3.

Usually `T` will be a type parameter in Step 7, and the substituted type becomes the dependent member type `T.[P]A`$_i$. Our algorithm relies on type substitution to also cope with `T` being a concrete type. When `T` is a concrete type, the subject type of each introduced same-type requirement has to be the type witness for `A`$_i$ in the concrete conformance [`T: P`]. For example, given [`Array<τ_0_0>: Sequence<Int>`], we output [`Array<τ_0_0>: Sequence`] and [`τ_0_0 == Int`]. Admittedly, this is a rather a silly edge case. One might think that requirement inference would exercise this scenario:

```
typealias G<T: Sequence<Int>> = T
func f<E>(_: G<Array<E>>) {}
```

However, we build the generic signature of `G` first, so we've already decomposed the requirement [`τ_0_0: Sequence<Int>`] by the time we get around to building the generic signature of `f()`; when we visit `G<Array<τ_0_0>` in requirement inference, we substitute the decomposed and desugared (and indeed, the minimal) requirements of `G`. In fact, the only way to exercise this is to write it down directly:

```
func f<E>(_: Array<E>) where Array<E>: Sequence<Int> {}
```

A more useful application of decomposition is the following. The **abstract generic signature request** performs decomposition to deal with the opaque result types of Chapter 13 and existential types of Chapter 14. We reason about these kinds of types by using this request to build an "auxiliary" generic signature describing the type. The constraint type after the `some` or `any` keyword defines a conformance requirement; this requirement must be decomposed by the same algorithm, so that we can interpret something like "`any Sequence<Int>`" or "`some Equatable & AnyObject`".

Desugaring. Requirement inference is one of several instances of a useful pattern: we apply a substitution map to the minimal requirements of a generic signature, and build a new generic signature from these substituted requirements. We use the **abstract generic signature request** in this way to check class method overrides, for example. When the original requirements are minimal, the substituted requirements are already decomposed, but it is possible for such a requirement to have a left-hand side that is not a type parameter, but a concrete type, introduced by the substitution.

A thought experiment helps us understand the meaning of a requirement whose left-hand side is not a type parameter. If the subject type of some requirement R is an interface type, not necessarily a type parameter, we cannot describe it with the derived requirements formalism. However, we can still apply a substitution map Σ to R, and check if the substituted requirement $R \otimes \Sigma$ is satisfied with Algorithm 9A; nothing in the algorithm depends on the *original* requirement's subject type being a type parameter.

We will transform R into a set of simpler requirements $\{R_1, \ldots, R_n\}$, such that for *every* substitution map Σ, $R \otimes \Sigma$ is satisfied if and only if for all $i \leq n$, $R_i \otimes \Sigma$ is satisfied. Of course we're not iterating over all possible substitution maps in the implementation, this is just a thought experiment. If we believe that the below transformation upholds the desired property, we can conclude that R can be replaced with $\{R_1, \ldots, R_n\}$ without changing the meaning of our generic signature.

This rule essentially determines the implementation of requirement desugaring. Let's first consider a requirement that does not contain type parameters at all, such as [`Int: Hashable`] or [`Int == String`]. Applying a substitution map cannot ever change our requirement, so it is always true or always false; we can check it with Algorithm 9A:

- If the requirement is satisfied, we can delete it, that is, replace it with the empty set of requirements, without violating our invariant. The requirement doesn't contribute anything new.
- On the other hand, if the requirement is unsatisfied, the only way to proceed is to replace this requirement with something else unsatisfiable, so we must diagnose an error and give up.

The second case merits further explanation. A generic declaration whose requirements cannot be satisfied by any substitution map is essentially useless, and we will see in Section 11.4 that we try to uncover situations where two requirements are in conflict with each other and cannot be simultaneously satisfied. Here though, requirement desugaring detects the trivial case where a requirement "conflicts" with itself.

Next, we consider a conformance requirement with a left-hand side that contains type parameters, like [`Array<τ_0_0>: Sequence`] or [`Array<τ_0_0>: Hashable`]. With global conformance lookup, we can find the concrete conformance of the subject type to this protocol. If the conformance is invalid, we have a trivial conflict; if unconditional, the requirement is trivially satisfied.

If we get a conditional conformance here, our invariant *leaves us no choice* but to replace the original requirement with the conditional requirements of this conformance (an unconditional conformance has an empty set of conditional requirements, so there's really just one rule here). For example, because of the conditional conformance of `Array` to `Hashable`, we know that `Array<τ_0_0>` $\otimes \Sigma$ conforms to `Hashable` if and only if `τ_0_0` $\otimes \Sigma$ conforms to `Hashable`, for any substitution map Σ. Therefore, [`Array<τ_0_0>: Hashable`] must desugar to [`τ_0_0: Hashable`], which explains how we infer [`τ_0_0: Hashable`] from `Set<Array<τ_0_0>>`, an example we saw earlier:

```
func f<T>(_: Set<Array<T>>) /* where T: Hashable */ {}
```

Finally, to desugar a same-type requirement, we consider four possibilities:

1. Both sides are type parameters.
2. Left-hand side is a type parameter, right-hand side is a concrete type.
3. Left-hand side is a concrete type, right-hand side is a type parameter.
4. Both sides are concrete types.

The first two cases already have the correct form, so we're done. The third case reduces to the second, because we can swap the two sides. Indeed, a same-type requirement is also a statement that both sides have the same reduced type, and this relation is symmetric. In the fourth case, we have something like this,

[`Dictionary<τ_0_0, Bool> == Dictionary<Int, τ_0_1>`]

which we would like to desugar into two requirements:

[`τ_0_0 == Int`]
[`Bool == τ_0_1`]

The first output is already desugared; the second one must be flipped, and then we're done. On the other hand, say we're given [`Array<τ_0_0> == Set<τ_0_0>`] instead. No substitution map can transform the two sides into the same type, so this requirement can never be satisfied. This gives us the general rule.

For two concrete types to be equivalent under all substitutions, they can only differ in certain ways. Two types *match* if they have the same kind, same number of structural component types, and exactly equal non-type information (examples of this include the declaration of a nominal type, the labels of a tuple, value ownership kinds of function parameters, and so on). Our definition of matching is not recursive, so `Array<Array<Int>>` and `Array<Set<Int>>` still match because everything lines up at the outermost level, but their two children `Array<Int>` and `Set<Int>` do not match.

If the two types in a same-type requirement do not match, we have a conflict, so we diagnose an error and give up. Otherwise, the types have an equal number of children; we walk the children in parallel and construct a simpler set of same-type requirements equivalent to the original. This is a recursive process; some of these requirements might need further desugaring, or lead to conflicts, and so on.

Algorithm 11B (Desugar same-type requirement)**.** Takes an arbitrary same-type requirement as input. Outputs a list of desugared requirements and a list of conflicting requirements.

1. Initialize empty output and conflict lists.
2. Add the input requirement to the worklist.
3. (Next) Take the next requirement [`T == U`] from the worklist.
4. (Abstract) If `T` and `U` are both type parameters, add [`T == U`] to the output list.
5. (Concrete) If `T` is a type parameter and `U` is concrete, output [`T == U`].
6. (Flipped) If `T` is concrete and `U` is a type parameter, output [`U == T`].
7. (Redundant) If `T` and `U` are canonically equal, nothing non-trivial will be generated below, so we immediately go to Step 10.
8. (Recurse) If `T` and `U` match, let $\texttt{T}_1 \ldots \texttt{T}_n$ and $\texttt{U}_1 \ldots \texttt{U}_n$ be the children of `T` and `U`. For each $1 \leq i \leq n$, add [$\texttt{T}_i$ `==` $\texttt{U}_i$] to the worklist.
9. (Conflict) If `T` and `U` do not match, add [`T == U`] to the conflict list and diagnose.
10. (Loop) If the worklist is empty, return. Otherwise, go back to Step 3.

Next we have the algorithm for desugaring an arbitrary requirement. This runs after Algorithm 11A, so we assume conformance requirements have been decomposed. Notice how this algorithm generalizes the "requirement is satisfied" check of Algorithm 9A: if we desugar a requirement that does not contain any type parameters, the conflict list will be empty if and only if the requirement is satisfied.

Algorithm 11C (Desugar requirement)**.** Takes an arbitrary requirement as input. Outputs a list of desugared requirements and a list of conflicting requirements.

1. Initialize empty output and conflict lists.
2. Add the input requirement to the worklist.
3. (Next) Take the next requirement from the worklist. If the requirement's subject type is a type parameter, add it to the output list and go to Step 5.

4. (Desugar) Otherwise, the subject type is a concrete type. Handle each requirement kind as follows:
   a) For a **conformance requirement** [`T: P`], perform the global conformance lookup $\langle \texttt{P} \rangle \otimes \texttt{T}$:
      i. If we get a concrete conformance, add the conditional requirements, if any, to the worklist.
      ii. If we get an invalid conformance, add [`T: P`] to the conflict list.
   b) For a **superclass requirement** [`T: C`]:
      i. If `T` and `C` are two specializations of the same class declaration, add the same-type requirement [`T == C`] to the worklist.
      ii. If `T` does not have a superclass type (Section 6.4), then `T` cannot be a subclass of `C`; add [`T: C`] to the conflict list.
      iii. Otherwise, let $\texttt{T}'$ be the superclass type of `T`. Add the superclass requirement [$\texttt{T}'$`: C`] to the worklist.
   c) For a **layout requirement** [`T: AnyObject`], any type parameters contained in the concrete type `T` have no bearing on the outcome. It suffices to apply Algorithm 9A. If unsatisfied, add [`T: AnyObject`] to the conflict list.
   d) For a **same-type requirement** [`T == U`], apply Algorithm 11B and add the results to the output and conflict lists.
5. (Loop) If the worklist is empty, return. Otherwise, go back to Step 3.

Requirements on the conflict list are diagnosed as errors. Requirements on the output list have the correct desugared form for minimization. To re-state:

Definition 11.1. A *desugared requirement* is a requirement where:

1. The left-hand side is a type parameter.
2. For a conformance requirement, the right-hand side is a protocol type.

11.3. Well-Formed Requirements

After desugaring and decomposition, our user-written requirements are now in a form where we can reason about them with the derived requirements formalism. So far, we have not attempted to impose any restrictions on the explicit requirements from which all else is derived, other than those requirements being *syntactically* well-formed. In the next section, we will precisely state the invariants of a generic signature, but first, we need to extend our theory with a notion of *semantically* well-formed requirements. It turns out we need this to diagnose certain malformed generic signatures.

To motivate our notion of well-formedness, let's return to the idea of a *valid type parameter* from Section 5.2. Specifically, we're going to look at prefixes of valid type parameters. Consider the type parameter `τ_0_0.Element.Element` below:

```
struct Concat<C: Collection> where C.Element: Collection {
  let x: C.Element.Element = ...
}
```

A type parameter is a *prefix* of another type parameter if it is equal to the other's base type, or the base type of the base type, at any level of nesting. So `τ_0_0.Element` and `τ_0_0` are the two prefixes of `τ_0_0.Element.Element`. From the user's point of view, if "`C.Element.Element`" is a meaningful utterance to write down in the program, "`C.Element`" certainly should be as well! This suggests a reasonable property that any "good" generic signature ought to have: *every prefix of a valid type parameter is itself a valid type parameter*. This is not our final condition though, so we generalize further.

When we say that a type parameter `T` is valid in a generic signature G, we mean that we have a derivation $G \vdash$ `T`. Only a few derivation steps derive type parameters, and we can also characterize them as follows.

Proposition 11.2. Let G be a generic signature, and let `T` be a type parameter.

- If `T` is a generic parameter, then `T` is valid if and only if it appears in G.
- If `T` is an unbound dependent member type `U.A`, then `T` is valid if and only if some protocol `P` declares an associated type named `A`, and $G \vdash$ [`U: P`].
- If `T` is a bound dependent member type `U.[P]A`, then `T` is valid if and only if $G \vdash$ [`U: P`], where `P` is the parent protocol of the associated type declaration denoted by `[P]A`.

If we have one of those generic signatures where every prefix of `T` is also valid, then in the case where `T` is a dependent member type above, the subject type `U` of [`U: P`] is a prefix of `T`, so `U` must be valid. We say that [`U: P`] is a *well-formed* requirement if its subject type `U` is valid. This idea of well-formedness allows us to subsume "prefix validity" with a more general idea: *every derived conformance requirement must be well-formed*. In other words, we want $G \vdash$ [`U: P`] to imply $G \vdash$ `U` for all `U` and `P`.

To generalize this further, consider type substitution. Let G be a generic signature where some derived requirement of G, not necessarily a conformance requirement, contains an invalid type parameter. By Algorithm 9A, no substitution map Σ can satisfy this requirement, because after we apply Σ, the invalid type parameter becomes an error type. Recalling our earlier notion of a well-formed *substitution map* from Definition 10.7, we conclude that G cannot have any well-formed substitution maps at all! To rule this out, we generalize our condition to all requirement kinds.

Definition 11.3. A requirement is *well-formed* with respect to a generic signature G if all type parameters contained in the requirement are valid type parameters of G:

- A **conformance requirement** [`T: P`] is well-formed if $G \vdash$ `T`.
- A **superclass requirement** [`T: C`], with $\{\texttt{C}_1, \ldots, \texttt{C}_n\}$ denoting the set of type parameters contained in `C`, is well-formed if $G \vdash$ `T` and $G \vdash \texttt{C}_i$ for all $1 \leq i \leq n$.
- A **layout requirement** [`T: AnyObject`] is well-formed if $G \vdash$ `T`.
- A **same-type requirement** [`T == U`], with $\{\texttt{U}_1, \ldots, \texttt{U}_n\}$ denoting the set of type parameters contained in `U`, is well-formed if $G \vdash$ `T` and $G \vdash \texttt{U}_i$ for all $1 \leq i \leq n$. (If the right-hand side `U` is a type parameter, the set of type parameters here is trivially {`U`}.)

We will be talking about both "derived" and "well-formed" requirements below, so before we completely lose the reader, let's make the distinction totally clear. Recall the generic signature of `Concat`:

```
<τ_0_0 where τ_0_0: Collection, τ_0_0.Element: Collection>
```

We cannot derive [`τ_0_0.Element.Element: Hashable`] from this generic signature; nothing in our signature is `Hashable`. However, this requirement is certainly well-formed, because `τ_0_0.Element.Element` is a valid type parameter in our generic signature. Thus, a derived requirement is provably *true*; a well-formed requirement makes sense as a *question*. So starting from prefix validity, we've arrived at our final statement: we want our generic signatures to only prove things that make sense!

Definition 11.4. A generic signature G is *well-formed* if all derived requirements of G are well-formed.

This definition doesn't immediately give us an algorithm for checking well-formedness. The derived requirements of a generic signature are an infinite set in general, so we cannot enumerate them all. We will resolve this dilemma shortly.

Notice how if $G \vdash$ [`T: P`] for some protocol `P`, the well-formedness of G implicitly depends on the well-formedness of the associated requirements of `P`; we interpret G with respect to its *protocol dependency set*, containing those protocols that can appear on the right-hand side of a derived conformance requirement. We will return to this topic in Section 15.1; for now, we can take this set to contain *all* protocols as a conservative approximation.

One more thing. The result we proved while motivating Definition 11.4 will be useful later, so let's re-state it for posterity:

Proposition 11.5. Let G be a well-formed generic signature, and let `T` be a valid type parameter of G. Then every prefix of `T` is also a valid type parameter of G.

Diagnostics. We now give an example of a generic signature that is not well-formed. We return to our `Concat` type, except here we "forgot" to state that `C` must conform to `Collection`:

```
struct Bad<C /* : Collection */> where C.Element: Collection {}
```

Nevertheless, starting with the explicit requirement [`τ_0_0.Element: Collection`], we can derive other requirements and valid type parameters; to pick a few at random:

1. [`τ_0_0.Element: Collection`] (Conf)
2. [`τ_0_0.Element: Sequence`] (AssocConf 1)
3. [`τ_0_0.Element.Iterator: IteratorProtocol`] (AssocConf 2)
4. `τ_0_0.Element.Iterator.Element` (AssocName 3)

The derived requirements (1) and (2) are not well-formed because their subject types are not valid type parameters, and the valid type parameter (4) has an invalid prefix `τ_0_0.Element`. Clearly, `Bad` ought to be rejected by the compiler. What actually happens when we type check `Bad`? Recall the type resolution stage from Chapter 9. We first resolve the requirement [`τ_0_0.Element: Collection`] in structural resolution stage, and we get a requirement whose subject type is an unbound dependent member type. We don't know that this requirement is not well-formed, yet.

After we build the generic signature for `Bad`, we revisit the `where` clause again, and resolve the requirement in the interface resolution stage. As the subject type is not a valid type parameter, type resolution diagnoses an error and returns an error type:

```
bad.swift:1:23: error: 'Element' is not a member type of type 'C'
struct Bad<C> where C.Element: Sequence {}
                      ^
```

Next, we define what it means for an associated requirement to be well-formed:

Definition 11.6. An associated requirement of a protocol `P` is *well-formed* if it is a well-formed requirement for the protocol generic signature $G_{\texttt{P}}$; that is, all type parameters it contains are valid type parameters in $G_{\texttt{P}}$.

A generic signature may depend on protocols written in source, or protocols from serialized modules. For protocols written in source, type resolution checks that their associated requirements are well-formed by visiting them again in the interface resolution stage, once the protocol's requirement signature has been built. Protocols from serialized modules already have well-formed associated requirements, because they were checked before serialization. Thus, if type resolution does not diagnose any errors, all user-written requirements are well-formed. The next theorem says this is a sufficient condition for all generic signatures in the main module to be well-formed.

Theorem 11.7. Suppose that a generic signature G satisfies these conditions:

- Every explicit requirement of G is well-formed.
- For every protocol `P` such that $G \vdash$ [`T: P`] for some `T`, every associated requirement of `P` is well-formed (in the protocol generic signature $G_{\texttt{P}}$).

Then every *derived* requirement of G is well-formed; in other words, G is well-formed.

To prove this theorem, we must expand our repertoire for reasoning about derivations. First, we recall the protocol generic signature from Section 5.1. If `P` is any protocol, then its generic signature, which we denote by $G_{\texttt{P}}$, has the single requirement [`Self: P`]. As always, the protocol `Self` type is sugar for `τ_0_0`.

The protocol generic signature describes the structure generated by the protocol's requirement signature. These are the valid type parameters and derived requirements inside the declaration of the protocol and its unconstrained extensions. These type parameters are all rooted in the protocol `Self` type, and the derived requirements talk about these `Self`-rooted type parameters. Informally, anything we can say about the protocol `Self` type in $G_{\texttt{P}}$, should also be true of an arbitrary type parameter `T` in some other generic signature G where $G \vdash$ [`T: P`]. We will now make this precise.

For example, we might first define an algorithm in a protocol extension of `Collection`, and then call our algorithm from another generic function:

```
extension Collection {
  func myComplicatedAlgorithm() {...}
}

func anotherAlgorithm<C: Collection>(_ c: C, _ index: C.Element.Index)
    where C.Element: Collection {
  c[index].myComplicatedAlgorithm()
}
```

The generic signature of `anotherAlgorithm()` is the same as `Concat` from earlier. Let's call it G. The reference to `myComplicatedAlgorithm()` has this substitution map:

$$\{\texttt{Self} \mapsto \texttt{τ_0_0.Element};\ [\texttt{Self: Collection}] \mapsto [\texttt{τ_0_0.Element: Collection}]\}$$

The input generic signature is $G_{\texttt{Collection}}$, and output generic signature is G, so for generics to "work" we would expect that applying this substitution map to a valid type parameter or derived requirement of $G_{\texttt{Collection}}$ should give us a valid type parameter or derived requirement in G.

Suppose that `myComplicatedAlgorithm()` makes use of `Self.SubSequence.Index` conforming to `Comparable` inside the body. We can derive this requirement, together with the validity of `Self.SubSequence.Index` (so our requirement is well-formed):

1. [`Self: Collection`] (Conf)
2. [`Self.SubSequence: Collection`] (AssocConf 1)
3. `Self.SubSequence.Index` (AssocName 2)
4. [`Self.SubSequence.Index: Comparable`] (AssocConf 2)

We apply our substitution map to (3) and (4). Of course, we have yet to explain how to apply a substitution map to a dependent member type! We will get to that in the next chapter, but for now, let's make the simplifying assumption that we're performing a syntactic replacement of "`Self`" with "`τ_0_0.Element`". This gives us:

`τ_0_0.Element.SubSequence.Index`

[`τ_0_0.Element.Self.SubSequence.Index: Comparable`]

We can show that the first is a valid type parameter of G, and the second a derived requirement of G, by taking our original derivation in $G_{\texttt{Collection}}$, and replacing `Self` with `τ_0_0.Element` throughout:

1. [`τ_0_0.Element: Collection`] (Conf)
2. [`τ_0_0.Element.SubSequence: Collection`] (AssocConf 1)
3. [`τ_0_0.Element.SubSequence.Index: Comparable`] (AssocConf 2)

This works because [`τ_0_0.Element: Collection`] was an *explicit* requirement of G, so it's not as general as we'd like. Suppose we started with a more complicated *derived* conformance requirement, like $G \vdash$ [`τ_0_0.Indices: Collection`] in the same signature:

1. [`τ_0_0: Collection`] (Conf)
2. [`τ_0_0.Indices: Collection`] (AssocConf 1)

To build a derivation for $G \vdash$ [`τ_0_0.Indices.SubSequence.Index: Comparable`], we first replace the elementary step [`Self: Collection`] with the entire *derivation* of $G \vdash$ [`τ_0_0.Indices: Collection`]; then we substitute `Self` with `τ_0_0.Indices` in all remaining steps:

1. [`τ_0_0: Collection`] (Conf)
2. [`τ_0_0.Indices: Collection`] (AssocConf 1)
3. [`τ_0_0.Indices.SubSequence: Collection`] (AssocConf 2)
4. [`τ_0_0.Indices.SubSequence.Index: Comparable`] (AssocConf 3)

A protocol generic signature has two elementary derivation steps, so we might instead have a derivation that starts with the GENERIC elementary step for `Self`. For example, we can derive the requirement [`Self == Self`] in $G_{\texttt{Collection}}$:

1. `Self` (GENERIC)
2. [`Self == Self`] (REFLEX 1)

To get a derivation $G \vdash$ [`τ_0_0.Indices == τ_0_0.Indices`], we make use of the fact that `τ_0_0.Indices` is a valid type parameter of G. We replace the elementary derivation step for `Self` with the entire derivation of $G \vdash$ `τ_0_0.Indices`:

1. [`τ_0_0: Collection`] (CONF)
2. `τ_0_0.Indices` (ASSOCNAME 1)
3. [`τ_0_0.Indices == τ_0_0.Indices`] (REFLEX 2)

We now state the general result. We need this for the proof of Theorem 11.7, and also later in Section 12.1, when we show that every derived conformance requirement has a derivation of a certain form.

Lemma 11.8 (Formal substitution)**.** Let G be an arbitrary generic signature. Suppose that $G \vdash$ `T` and $G \vdash$ [`T: P`] for some type parameter `T` and protocol `P`. Then if we take a valid type parameter or derived requirement of $G_{\texttt{P}}$ and replace `Self` with `T` throughout, we get a valid type parameter or derived requirement of G, just rooted in `T`. That is:

- If $G_{\texttt{P}} \vdash$ `Self.U`, then $G \vdash$ `T.U`.
- If $G_{\texttt{P}} \vdash R$, then $G \vdash R'$, where R' is the substituted requirement obtained by replacing `Self` with `T` in R.

Proof. Suppose we wanted to write down an *algorithm* for the above. As input, we're given a generic signature G, a pair of derivations $G \vdash$ `T` and $G \vdash$ [`T: P`], and some derivation in $G_{\texttt{P}}$. We rewrite the derivation in $G_{\texttt{P}}$ into a derivation in G by visiting each consecutive step. First, consider the elementary derivation steps of $G_{\texttt{P}}$:

`Self` (GENERIC)
[`Self: P`] (CONF)

Each one is replaced with the *entire derivation* of `T` or [`T: P`], respectively:

`T` (...)
[`T: P`] (...)

For all other derivation steps, we substitute `T` in place of `Self` anywhere it appears in the derivation step's statement. This gives us the desired derivation in G. □

A few words about the above proof. To be completely rigorous, we should go through each inference rule and argue that the substitution outputs a meaningful derivation step in each case. We're not going to do that, because we will demonstrate this form of exhaustive case analysis in the proof of Theorem 11.7 instead. Finally, note that our lemma makes the assumption that [`T: P`] is well-formed, because we need $G \vdash$ `T`. However, we are specifically not assuming that G is well-formed. In fact, we will use this lemma in the proof of Theorem 11.7, so such an assumption would be circular.

Structural induction. Suppose that $P(n)$ is some property of the natural numbers, and we wish to show that it is always true of all $n \in \mathbb{N}$. We can argue by *induction*, writing down a proof in two parts:

- The *base case*, that $P(0)$ is true.
- The *inductive step*, where we *assume* that $P(n)$ is true for a fixed but arbitrary n, and then prove that $P(n+1)$ is true as a consequence.

For example, let's say $P(n)$ is the statement that $0 + 1 + 2 + \cdots + n = n(n+1)/2$ for some given $n \in \mathbb{N}$. We can show this statement is always true it by induction:

- To show that $P(0)$, we substitute $n = 0$ into our formula. The left-hand side is zero, and the right-hand side is $n(n+1)/2 = 0(0+1)/2 = 0$.
- To show the inductive step, we assume $P(n)$, so, $0 + 1 + 2 + \cdots + n = n(n+1)/2$. We add $n+1$ to both sides. The right-hand side simplifies as follows, establishing $P(n+1)$:

$$\frac{n(n+1)}{2} + (n+1) = \frac{n^2 + n + 2n + 2}{2} = \frac{n^2 + 3n + 2}{2} = \frac{(n+1)(n+2)}{2}$$

A proof by induction is sort of like a "recipe" that tells us how to derive a proof for some specific $n \in \mathbb{N}$. For example, suppose we wish to show $P(2)$. We start by writing down the base case $P(0)$. Then, the inductive step with $n = 0$ shows that $P(0)$ implies $P(1)$, but we know $P(0)$ is true, thus $P(1)$ is true. Next, we apply the inductive step with $n = 1$, and conclude that $P(2)$ is true. We can use our "recipe" to construct a proof for any n this way.

Proof by induction is an axiom of *Peano arithmetic*, a formal system for the natural numbers. One way to think of it, is that induction says we can reach any natural number by repeated application of the *successor function* "1+" to "0". Notice how this encoding of the natural numbers resembles a Swift enum:

```
enum Nat {
  case zero
  indirect case successor(Nat)
}
```

For example, $3 = 1 + (1 + (1 + 0))$, or indeed:

```
let three: Nat = .successor(.successor(.successor(.zero)))
```

Now, we return to our derived requirements formalism. Our statements are finite data structures, and while they are more complex than instances of `Nat`, they have the same essential quality:

- The elementary statements—the generic parameters and explicit requirements—play the role of the number 0.
- Every other statement is generated by applying a fixed inference rule to one or more previous statements. The inference rules are analogous to the successor function that increments a number by 1.

If P is a property of requirements and type parameters and G is some generic signature, we can establish that P is true for all derived requirements and valid type parameters of G by *structural induction*. A proof by structural induction has two parts:

- The *base case* establishes that $P(D)$ is true for every elementary statement D. Recall that these are the Generic steps for each generic parameter of G, and the steps for each explicit requirement of G.
- The *inductive step*, where we have a derivation step with assumptions $D_1, \ldots, D_n$ and conclusion D. We assume $P(D_1), \ldots, P(D_n)$ hold, then argue that $P(D)$ must hold as a consequence. We can perform a case analysis over the various inference rules to handle each kind of inference rule.

If we wish to prove a statement about derived requirements only, we slightly modify the above scheme. The base case no longer needs to consider Generic steps, but also, the Reflex step *becomes* a base case, because none of its assumptions are requirements. The following proof, which we are now in a position to state, uses the modified scheme.

Proof of Theorem 11.7. We're given a requirement R such that $G \vdash R$, and a type parameter contained in R. We must construct a derivation of this type parameter.

Base case. The elementary statements have no assumptions at all:

[`T: P`]	(Conf)
[`T == U`]	(Same)
[`T == X`]	(Concrete)
[`T: C`]	(Super)
[`T: AnyObject`]	(Layout)

A requirement proved by an elementary statement is an explicit requirement of G. The first assumption in our theorem was that all explicit requirements of G are well-formed, so we're done. The Reflex inference rule is the other base case because it's assumption is a type parameter and not a requirement:

$$[\texttt{T == T}] \qquad \text{(Reflex T)}$$

The assumption $G \vdash \texttt{T}$ implies that the conclusion [T == T] is well-formed, by definition.

Inductive step. We must handle each kind of inference rule in turn. Consider an AssocBind step, for some type parameter T, protocol P and associated type A:

$$[\texttt{T.[P]A == T.A}] \qquad \text{(AssocBind [T: P])}$$

The conclusion is a derived same-type requirement that contains two type parameters, T.[P]A and T.A. We can derive both from the conformance requirement [T: P] using AssocDecl and AssocName:

1. [T: P] (...)
2. T.[P]A (AssocDecl 1)
3. T.A (AssocName 2)

Next, consider the SameName and SameDecl derivation steps:

$$[\texttt{T.A == U.A}] \qquad \text{(SameName [U: P] [T == U])}$$
$$[\texttt{T.[P]A == U.[P]A}] \qquad \text{(SameDecl [U: P] [T == U])}$$

We have four type parameters to derive: T.A, T.[P]A, U.A and U.[P]A. We first derive [T: P] from [U: P], and then apply AssocDecl and AssocName to [T: P] and [U: P]:

1. [U: P] (...)
2. [T == U] (...)
3. [T: P] (SameConf 1 2)
4. T.A (AssocName 3)
5. T.[P]A (AssocDecl 3)
6. U.A (AssocName 1)
7. U.[P]A (AssocDecl 1)

To handle the derivation steps generated by the associated requirements of a protocol P, we must make use of the second assumption of our theorem, together with Lemma 11.8. The simplest case is an AssocConf or AssocLayout step:

$$[\texttt{T.U: Q}] \qquad \text{(AssocConf } [\texttt{Self.U: Q}]_{\texttt{P}}\ [\texttt{T: P}]\text{)}$$
$$[\texttt{T.U: AnyObject}] \qquad \text{(AssocLayout } [\texttt{Self.U: AnyObject}]_{\texttt{P}}\ [\texttt{T: P}]\text{)}$$

By the induction hypothesis, $G \vdash \texttt{T}$. We must show that $G \vdash \texttt{T.U}$. The requirement on the right-hand side is obtained by replacing `Self` with `T` in some associated conformance requirement $[\texttt{Self.U: Q}]_\texttt{P}$. By assumption, this associated requirement is well-formed with respect to $G_\texttt{P}$, so we have a derivation $G_\texttt{P} \vdash \texttt{Self.U}$. All the conditions of Lemma 11.8 are satisfied, thus we can construct a derivation $G \vdash \texttt{T.U}$.

For an associated same-type requirement $[\texttt{Self.U == Self.V}]_\texttt{P}$ we repeat the same construction to derive $G \vdash \texttt{T.U}$ and $G \vdash \texttt{T.V}$:

$$[\texttt{T.U == T.V}] \qquad (\textsc{AssocSame}\ [\texttt{Self.U == Self.V}]_\texttt{P}\ [\texttt{T: P}])$$

If we are looking at a concrete same-type requirement $[\texttt{Self.U == X}]_\texttt{P}$, or a superclass requirement $[\texttt{Self.U: C}]_\texttt{P}$, we again use Lemma 11.8 to derive each type parameter that appears in $\texttt{X}'$ and $\texttt{C}'$, which we use to denote the types obtained from `X` and `C`, respectively, by structural replacement of `Self` with `T`:

$$[\texttt{T.U == X}'] \qquad (\textsc{AssocConcrete}\ [\texttt{Self.U == X}]_\texttt{P}\ [\texttt{T: P}])$$
$$[\texttt{T.U: C}'] \qquad (\textsc{AssocSuper}\ [\texttt{Self.U: C}]_\texttt{P}\ [\texttt{T: P}])$$

Finally, all other derivation steps have the property that the type parameters appearing in their conclusion already appear in their assumptions, so by the induction hypothesis, the derived requirement is well-formed:

$$[\texttt{U == T}] \qquad (\textsc{Sym}\ [\texttt{T == U}])$$
$$[\texttt{T == V}] \qquad (\textsc{Trans}\ [\texttt{T == U}]\ [\texttt{U == V}])$$
$$[\texttt{T: P}] \qquad (\textsc{SameConf}\ [\texttt{U: P}]\ [\texttt{T == U}])$$
$$[\texttt{T == X}] \qquad (\textsc{SameConcrete}\ [\texttt{U == X}]\ [\texttt{T == U}])$$
$$[\texttt{T: C}] \qquad (\textsc{SameSuper}\ [\texttt{U: C}]\ [\texttt{T == U}])$$
$$[\texttt{T: AnyObject}] \qquad (\textsc{SameLayout}\ [\texttt{U: AnyObject}]\ [\texttt{T == U}])$$

This completes the induction. □

Formally, structural induction depends on a well-founded order (Section 5.5), so we would use the "containment" order on derivations. However, the "recursive algorithm" viewpoint is good enough for us. Induction over the natural numbers is covered in introductory books such as [81]; for structural induction in formal logic, see something like [117]. We will use structural induction over derivations again to study conformance paths in Section 12.1, encode finitely-presented monoids as protocols in Section 16.3, and finally present a correctness proof for the Requirement Machine in Chapter 17.

Before we end this section, we revisit bound and unbound type parameters from Section 5.4, and prove one final result. We previously claimed that every equivalence class of type parameters contains both a bound and unbound type parameter representative. This actually requires the assumption that our generic signature is well-formed.

Theorem 11.9. Let G be a well-formed generic signature, and suppose $\texttt{T}$ is a valid type parameter of G. Then, the equivalence class of $\texttt{T}$ also contains two type parameters (not necessarily unique), which we denote by $\texttt{T}^*$ and $\texttt{T}_*$, such that:

1. $\texttt{T}^*$ is a bound type parameter,
2. $\texttt{T}_*$ is an unbound type parameter,
3. $\texttt{T}^*$ and $\texttt{T}_*$ have the same length as $\texttt{T}$,
4. $\texttt{T}^* \leq \texttt{T} \leq \texttt{T}_*$ under the type parameter order.

Furthermore, if $\texttt{T}$ is a reduced type parameter, then $\texttt{T}^*$ is canonically equal to $\texttt{T}$, and thus every reduced type parameter of G is a bound type parameter.

Proof. We proceed by induction on the length of the type parameter $\texttt{T}$. In the base case, we prove that the property holds for all generic parameters; in the inductive step, we prove it holds for any dependent member type whose base type again has the property.

Base case. If $\texttt{T}$ is a generic parameter type $\texttt{τ_d_i}$, we can set both $\texttt{T}^*$ and $\texttt{T}_*$ to $\texttt{T}$, so $\texttt{T}^* \leq \texttt{T} \leq \texttt{T}_*$ holds. Then, we derive a trivial same-type requirement twice:

1. $\texttt{τ_d_i}$ (Generic)
2. $[\texttt{τ_d_i == τ_d_i}]$ (Reflex 1)
3. $[\texttt{τ_d_i == τ_d_i}]$ (Reflex 1)

Inductive step. Assume that $\texttt{T}$ is a dependent member type, either $\texttt{U.[P]A}$ (bound) or $\texttt{U.A}$ (unbound), for some $\texttt{U}$ and $\texttt{A}$ of $\texttt{P}$. We have $G \vdash [\texttt{U: P}]$ because $\texttt{T}$ is valid, and $G \vdash \texttt{U}$ because $[\texttt{U: P}]$ is well-formed. The length of $\texttt{U}$ is one less than the length of $\texttt{T}$, so the induction hypothesis then gives us a pair of same-type requirements $[\texttt{U == U}^*]$ and $[\texttt{U == U}_*]$, such that $\texttt{U}^* \leq \texttt{U} \leq \texttt{U}_*$. We set $\texttt{T}^* := \texttt{U}^*\texttt{.[P]A}$ and $\texttt{T}_* := \texttt{U}_*\texttt{.A}$, and derive three new same-type requirements via SameDecl, AssocBind and SameName:

1. $[\texttt{U.[P]A == T}^*]$ (SameDecl $[\texttt{U == U}^*]$ $[\texttt{U: P}]$)
2. $[\texttt{U.[P]A == U.A}]$ (AssocBind $[\texttt{U: P}]$)
3. $[\texttt{U.A == T}_*]$ (SameName $[\texttt{U == U}_*]$ $[\texttt{U: P}]$)

From the definition of the type parameter order in Section 5.5, we also see that:

$$\texttt{T}^* \leq \texttt{U.[P]A} < \texttt{U.A} \leq \texttt{T}_*$$

If $\texttt{T}$ is the bound dependent member type $\texttt{U.[P]A}$, we already have $[\texttt{T == T}^*]$ as (1), but we must derive $[\texttt{T == T}_*]$:

4. $[\texttt{T == T}_*]$ (Trans 2 3)

If `T` is the unbound dependent member type `U.A`, we find ourselves in the opposite situation. We already have [`T == T`$_*$] as (3), but we derive [`T == T`*] with Sym and Trans:

5. [`U.[P]A == T`] (Sym 2)
6. [`T == T`*] (Trans 5 1)

This completes the induction. To prove the second part of the theorem, we further assume that `T` is reduced. By the preceding argument, we get a bound type parameter `T`* such that `T`$^* \leq$ `T`. On the other hand, a reduced type parameter is the smallest element in its equivalence class, so `T` $\leq$ `T`*. We conclude that `T`* is canonically equal to `T` if `T` is reduced. □

11.4. Requirement Minimization

The last step shown in Figure 11.1 and 11.2 is called *requirement minimization*. To finish building our generic signature, we must transform the list of desugared requirements into a list of *minimal* requirements. These minimal requirements are then given to the primitive constructor, together with the list of generic parameter types collected at the start of the process, and we have our generic signature!

Because of the central role that generic signatures play in the Swift ABI, it is worth describing the requirement minimization problem in the abstract, which is our goal in the present section. The implementation itself will be revealed in Part IV, when we build a convergent rewriting system from desugared requirements (Section 17.4) and then perform rewrite rule minimization (Chapter 20). The material in this section was based on [118], an earlier write-up about the `GenericSignatureBuilder`.

We can summarize the key behaviors of requirement minimization:

1. Type substitution only accepts bound dependent member types. To ensure that we can apply a substitution map to the requirements of a generic signature, as we do in Algorithm 9B for example, each requirement is rewritten to use bound dependent member types.

2. Generic signatures describe the calling convention of generic functions, the layout of nominal type metadata, the mangling of symbol names, and so on. To ensure that trivial syntactic changes do not affect ABI, each requirement in a generic signature is *reduced* into the simplest possible form, redundant requirements are dropped to produce a *minimal* list, and this list is sorted in canonical order.

3. We need to detect and diagnose generic signatures with *conflicting requirements* that cannot be satisfied by any well-formed substitution map. This allows us to assume that all generic signatures in the main module are satisfiable as long as no diagnostics are emitted during type checking.

Equivalence of requirements. In our derived requirements formalism, a generic signature is just a list of requirements. We saw in the previous section that the only real assumption we need from those requirements is that they are desugared requirements. Thus, when taken with the list of generic parameter types, the desugared requirements given to minimization already form a generic signature in our theory.

We will define a *minimal* generic signature as one satisfying the additional conditions we just described. We will understand requirement minimization as a mathematical function that takes a generic signature and outputs a minimal generic signature. In the implementation, all generic signatures come from requirement minimization, and so all generic signatures are actually minimal. To justify this transformation, we introduce the notion of *equivalent* generic signatures. Two generic signatures are equivalent if they generate the same *theory*; that is, they have the same set of valid type parameters and derived requirements. Note that in particular, if two generic signatures G_1 and G_2 are equivalent, then G_1 is well-formed if and only if G_2 is well-formed.

Proposition 11.10. Any two generic signatures G_1 and G_2 satisfying the below conditions are *equivalent*, meaning they generate the same theory:

1. G_1 and G_2 have the same list of generic parameter types.
2. Every explicit requirement of G_1 can be derived in G_2.
3. Every explicit requirement of G_2 can be derived in G_1.

Proof. We will show that G_1 and G_2 generate the same theory by first arguing that the theory of G_1 is a subset of the theory of G_2, and vice versa.

Suppose we are given a derivation $G_1 \vdash D$, where D is either a valid type parameter or derived requirement of G_1. We must show that D is a valid type parameter or derived requirement of G_2. The elementary derivation steps that can appear in $G_1 \vdash D$ are those defined by the generic parameters and explicit requirements of G_1. We can mechanically construct a derivation of $G_2 \vdash D$ from $G_1 \vdash D$ to arrive at our conclusion:

1. By the first assumption, every Generic step that derives a generic parameter of G_1 is already a valid Generic step for G_2, so we leave it unchanged.
2. By the second assumption, every elementary step for an explicit requirement of G_1 has a corresponding *derivation* of the same requirement in G_2. We replace each elementary step with its derivation.
3. All other derivation steps remain unchanged.

This is a derivation in G_2, and so the theory of G_1 is a subset of the theory of G_2. To get the other inclusion, we make the same argument but with G_1 and G_2 swapped, and use the third assumption in step 2 instead. □

Note that we do *not* attempt to delete "redundant generic parameters." For example, `<T, U where T: Sequence, U == T.Element>` and `<T where T: Sequence>` are also "equivalent" in some broader sense, because any derivation in the former theory can be transformed into one for the latter by replacing `U` with `T.Element`. The latter is also more "minimal" because the calling convention for a function with this signature only needs to pass type metadata for `T`, rather than both `T` and `U`. Since we do not pursue this direction, our notion of equivalence bakes in the idea that both the original and minimal generic signatures have the same identical list of generic parameter types.

We want requirement minimization to output a minimal generic signature that is equivalent, but there are two important exceptions to keep in mind:

1. If any of the original requirements are not well-formed, we cannot rewrite them to use bound dependent member types, so we drop them instead, and the minimal generic signature describes a smaller theory. This is fine; type resolution will have diagnosed errors already, and we will not proceed to code generation.

2. Our derived requirements formalism does not explain all implemented behaviors of superclass, layout and concrete same-type requirements. If these requirement kinds are among the explicit requirements of the generic signature, or the associated requirements of some protocol, minimization may output a generic signature with a different theory. We will see examples of this after.

As an aside, Proposition 11.10 and Theorem 11.9 allow us to finally explain why we can take the explicit requirements in a derivation to contain bound *or* unbound dependent member types, depending on occasion. Indeed, sometimes we would omit the `[P]`'s to save space, other times we left them in to be explicit. It turns out that it doesn't matter; as long as the explicit requirements we start with are well-formed, we can derive either list from the other and arrive at the same theory.

Reduced requirements. We're now going to work towards a definition of a minimal generic signature. Consider this function:

```
func uniqueElements1<T: Sequence>(_: T) -> Int
    where T.Element: Hashable {}
```

When building the generic signature of `uniqueElements1()`, we get two user-written requirements from type resolution in the structural resolution stage:

{[`T: Sequence`], [`T.Element: Hashable`]}

The subject type of the second requirement is an unbound dependent member type formed from the generic parameter type `T` and identifier `Element`. We can replace the

second requirement with [`T.[Sequence]Element: Hashable`]; the latter's subject type is a bound dependent member type, formed from `T` and the associated type declaration `Element` of `Sequence`. To justify this, we can derive [`T.[Sequence]Element: Hashable`] from [`T.Element: Hashable`]:

1. [`T.Element: Hashable`] (Conf)
2. [`T: Sequence`] (Conf)
3. [`T.[Sequence]Element == T.Element`] (AssocBind 2)
4. [`T.[Sequence]Element: Hashable`] (SameConf 1 3)

And vice versa:

1. [`T.[Sequence]Element: Hashable`] (Conf)
2. [`T: Sequence`] (Conf)
3. [`T.[Sequence]Element == T.Element`] (AssocBind 2)
4. [`T.Element == T.[Sequence]Element`] (Sym 3)
5. [`T.Element: Hashable`] (SameConf 1 4)

This is how we arrive at the generic signature that we see if we test this example with the `-debug-generic-signatures` flag:

```
<T where T: Sequence, T.[Sequence]Element: Hashable>
```

Now, notice that the second conformance requirement's subject type is not just bound, it is *reduced*. This suggests a stronger condition. We could instead spell the conformance to `Hashable` with the subject type `T.Iterator.Element`:

```
func uniqueElements2<T: Sequence>(_: T) -> Int
    where T.Iterator.Element: Hashable {}
```

If we test this with `-debug-generic-signatures`, we see that `uniqueElements2()` has the same generic signature as `uniqueElements1()`. We start from these requirements:

{[`T: Sequence`], [`T.Iterator.Element: Hashable`]}

We can similarly show that this is an equivalent list:

{[`T: Sequence`],
 [`T.[Sequence]Iterator.[IteratorProtocol]Element: Hashable`]}

However, the reduced type of `T.Iterator.Element` is `T.[Sequence]Element`, because of the associated same-type requirement in `Sequence`, so in fact we can simplify the second conformance requirement further without changing the theory:

{[`T: Sequence`], [`T.[Sequence]Element: Hashable`]}

This is a list of *reduced* requirements, because the subject type of each is a reduced type parameter in our generic signature. We will define reduced requirements more generally now. In what follows, there is an important distinction between same-type requirements between type parameters ([`T.A == T.B`]), and same-type requirements where the right hand side is a concrete type ([`T.A == Array<T.B>`]). We will treat them as essentially two different kinds of requirements. The below definition is straightforward for every requirement kind, except for a same-type requirement between type parameters.

Definition 11.11. Let G be a generic signature, and let R be an explicit requirement of G. We say R is a *reduced requirement* if the following holds:

- For a **conformance requirement** [`T: P`]: `T` is a reduced type parameter.
- For a **layout requirement** [`T: AnyObject`]: `T` is a reduced type parameter.
- For a **superclass requirement** [`T: C`]: both `T` and `C` are reduced types.
- For a **same-type requirement** [`T == X`] where `X` is a **concrete type**: `T` is a reduced type parameter and `X` is a reduced type.
- For a **same-type requirement** [`T == U`] where `U` is a **type parameter**:
  1. `T` $<$ `U` under the type parameter order.
  2. `T` is either a reduced type parameter, or identical to the right-hand side of some other (explicit) same-type requirement in G.
  3. `T` is not identical to the left-hand side of any same-type requirement of G.
  4. `U` has the property that the only derivations $G \vdash$ [`U′ == U`] where `U′` $<$ `U` must involve the explicit requirement [`T == U`] itself; that is, `U` cannot be further reduced by any other requirement of G.

For a same-type requirement between type parameters, we cannot simply say that both sides are reduced type parameters, because the only way this can happen is if we have a trivial same-type requirement [`T == T`] for some reduced type parameter `T`. An example will clarify the situation. Recall this funny protocol from Section 5.3:

```
protocol N {
  associatedtype A: N
}
```

Now, consider these two generic structs:

```
struct Hook1<T, U> where U: N, T == T.A, T.A == U.A {}
struct Hook2<T, U> where U: N, T == T.A, U.A == T {}
```

We will look at `Hook1` first, but we will see that both in fact have the same generic signature. To build the generic signature of `Hook1`, we start with these user-written requirements:

{[`U: N`], [`T == T.A`], [`T.A == U.A`]}

As before, we derive an equivalent list involving only bound dependent member types:

{[`T: N`], [`T == T.[N]A`], [`T.[N]A == U.[N]A`]}

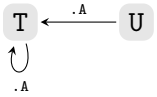
This list of requirements is already reduced and minimal, so we get this generic signature:

```
<T, U where U: N, T == T.[N]A, T.[N]A == U.[N]A>
```

To better understand this generic signature, note that we can derive [`T: N`]:

1. [`U: N`] (Conf)
2. [`U.[N]A: N`] (AssocConf 1)
3. [`T.[N]A == U.[N]A`] (Same)
4. [`T.[N]A: N`] (SameConf 1 3)
5. [`T == T.[N]A`] (Same)
6. [`T: N`] (SameConf 4 5)

Both `T` and `U` conform to `N`, so each has a member type named `A`, but because of our same-type requirements, these member types are equivalent to `T`. We get this type parameter graph:



Next, we look at `Hook2`, which only differs in how the second same-type requirement is specified. We write the requirements of `Hook2` with bound dependent member types:

{[`U: N`], [`T == T.[N]A`], [`U.[N]A == T`]}

The last requirement is not reduced, because `U.[N]A` > `T`. However, we can derive [`T.[N]A == U.[N]A`], which is reduced, from [`U.[N]A == T`]:

1. [`U.[N]A == T`] (Same)
2. [`T == T.[N]A`] (Same)
3. [`U.[N]A == T.[N]A`] (Trans 1 2)
4. [`T.[N]A == U.[N]A`] (Sym 3)

And vice versa:

1. [`T.[N]A == U.[N]A`] (Same)
2. [`T == T.[N]A`] (Same)
3. [`T == U.[N]A`] (Trans 1 2)
4. [`U.[N]A == T`] (Sym 3)

This shows that `Hook1` and `Hook2` actually have the same minimal generic signature, which `-debug-generic-signatures` will confirm.

There's a bit of history behind this test case. Our derivation of [`T: N`] involved the same-type requirement [`T == T.A`], which contains a member type of `T`. This was too difficult for the `GenericSignatureBuilder` to reason about, so we only accepted `Hook2` but not `Hook1`, despite `Hook1` being the one whose requirements are written in reduced form according to the rules of the Swift ABI. This bug was fixed with the Requirement Machine.

Minimal requirements. Here is another variant of a function we saw earlier:

```
func uniqueElements3<T: Sequence>(_: T) -> Int
    where T.Element: Hashable,
          T.Iterator: IteratorProtocol,
          T.Element: Equatable {}
```

We get this list of reduced requirements:

{[`T: Sequence`], [`T.[Sequence]Element: Hashable`],
 [`T.[Sequence]Iterator: IteratorProtocol`],
 [`T.[Sequence]Element: Equatable`]}

The third and fourth requirements do not give us anything new, because they can be derived from the other two. Thus, `uniqueElements3()` has the same minimal generic signature as `uniqueElements1()` and `uniqueElements2()`. (In the past, we diagnosed redundant requirements as a warning, but this was removed in Swift 5.7, so now they're just dropped.)

Definition 11.12. Let G be a generic signature.

- An explicit requirement R of G is a *redundant requirement* if we can write down a derivation $G \vdash R$ using only the remaining requirements, without invoking R as an elementary statement.
- We say G is a *minimal* generic signature if every explicit requirement of G is reduced, and no explicit requirement of G is redundant.

The next example shows that a list of reduced requirements may have more than one *distinct* minimal subset. We declare three generic structs that only differ in conformance requirements; the reader may again want to try this with `-debug-generic-signatures`:

```
struct Knot1<T, U> where T: N, T == U.A,       U == T.A {}
struct Knot2<T, U> where       T == U.A, U: N, U == T.A {}
struct Knot3<T, U> where T: N, T == U.A, U: N, U == T.A {}
```

We're going to look at the generic signature of `Knot1` first. Type resolution produces the following list of user-written requirements for `Knot1`:

{[T: N], [T == U.A], [U == T.A]}

Here is the equivalent list of reduced requirements:

{[T: N], [T == U.[N]A], [U == T.[N]A]}

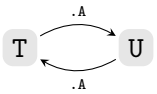
This list is minimal, so we get the following generic signature for `Knot1`:

<T, U where T: N, T == U.[N]A, U == T.[N]A>

Notice that we have an explicit requirement [T: N], and we can also derive [U: N]:

1. [T: N] (Conf)
2. [T.[N]A: N] (AssocConf 1)
3. [U == T.[N]A] (Same)
4. [U: N] (SameConf 2 3)

We will now show the type parameter graph for `Knot1`. The requirements look similar to `Hook1` because we have two equivalence classes that have a member type named `A`, but the same-type requirements act in a different way. Each member type `A` now takes us to *opposite* equivalence class:



Now, consider `Knot2`. Here is our list of reduced requirements:

{[T == U.[N]A], [U: N], [U == T.[N]A]}

This list is again minimal, so we get the following generic signature for `Knot2`:

<T, U where T == U.[N]A, U: N, U == T.[N]A>

In `Knot2`, we have an explicit requirement `[U: N]`, and we can derive `[T: N]`:

1. `[U: N]` (Conf)
2. `[U.[N]A: N]` (AssocConf 1)
3. `[T == U.[N]A]` (Same)
4. `[T: N]` (SameConf 2 3)

We've shown that each one of `[T: N]` and `[U: N]` can be derived in both `Knot1` and `Knot2`. Also, the remaining explicit requirements are identical, so we now see that we have two distinct minimal generic signatures that generate the same theory.

Finally, we look at `Knot3`. Here are our reduced requirements:

```
{[T: N], [T == U.[N]A], [U: N], [U == T.[N]A]}
```

Unlike `Knot1` and `Knot2`, the reduced requirements of `Knot3` are not minimal, because each one of `[T: N]` and `[U: N]` can be derived from the other. In a situation like this, requirement minimization prefers to delete the redundant requirement with the larger subject type under the type parameter order.

In this case, that means we delete `[U: N]` first. Our derivation of `[T: N]` involves `[U: N]`, so by Proposition 11.10, we can replace the explicit requirement `[U: N]` with its derivation to obtain a new derivation of `[T: N]`:

1. `[T: N]` (Conf)
2. `[T.[N]A: N]` (AssocConf 1)
3. `[U == T.[N]A]` (Same)
4. `[U: N]` (SameConf 2 3)
5. `[U.[N]A: N]` (AssocConf 4)
6. `[T == U.[N]A]` (Same)
7. `[T: N]` (SameConf 5 6)

We are no longer able to derive `[T: N]` from the *remaining* requirements, because the above derivation of `[T: N]` starts with the elementary derivation step for `[T: N]`. Thus, having deleted `[U: N]`, the explicit requirement `[T: N]` is no longer redundant, and the generic signature of `Knot3` is the same as `Knot1`:

```
<T, U where T: N, T == U.[N]A, U == T.[N]A>
```

The fact that `Knot2` has a distinct generic signature from the other two was actually due to a quirk of the `GenericSignatureBuilder`, and this behavior is now part of the Swift ABI. The stronger form of requirement minimization that guarantees uniqueness would actually be *simpler* to implement. A minor complication caused by the legacy behavior will be explained in Section 20.3.

There is another downside, from a theoretical standpoint. With type parameters, we are able to check equivalence by comparing their reduced types. However, we cannot check two lists of requirements for "theory equivalence" by comparing minimal generic signatures, because they are not unique. In practice though, nothing seems to call for this equivalence check; this is unlike type parameters, which are checked for reduced type equality all over the place.

The important invariant that we do maintain, to make textual interfaces work for example, is *idempotence*: meaning that when we pass from user-written requirements to minimal requirements for the first time, we are allowed to make choices amongst multiple minimal subsets, but if we take this output and build a new generic signature from those minimal requirements *again*, we must arrive at the *same* minimal generic signature. This also justifies the optimization to "install" requirement machines in Chapter 15.

Conflicting requirements. In the previous section, we related the idea of the well-formed substitution map (Definition 10.7) and the well-formed generic signature (Definition 11.4); a necessary condition for a generic signature to have any well-formed substitution maps at all, is for it to be a well-formed generic signature.

If we limit ourselves to the subset of the language with only conformance requirements and same-type requirements between type parameters, this condition is also sufficient, meaning we can mechanically construct a well-formed substitution map for a well-formed generic signature. We first declare a single concrete nominal type, call it struct `S`, and conform `S` to each protocol $\texttt{P}_i$ that our generic signature G depends on:

- Any method, variable and subscript requirements of $\texttt{P}_i$ can be witnessed by stub implementations that call `fatalError()`.
- Any associated types of $\texttt{P}_i$ can be witnessed by type alias members of `S` declared to have underlying type `S`.

We then build a substitution map for G by replacing each generic parameter type with `S` and each conformance requirement with the corresponding normal conformance $[\texttt{S: P}_i]$. Every derived requirement of G has the form $[\texttt{T: P}_i]$ for some type parameter `T` and protocol $\texttt{T}_i$, or `[T == U]` for a pair of type parameters `T` and `U`. Applying our substitution map, we always get either $[\texttt{S: P}_i]$ or `[S == S]`. Both requirements are satisfied no matter what, so our substitution map is well-formed.

With superclass, layout, and concrete same-type requirements, the picture is more complicated. When type parameters are required to have specific concrete types, we cannot "collapse" the entire generic signature down to a single point. Indeed, as we said, *conflicting requirements* might prevent our generic signature from being satisfied by *any* substitution map. There is another complication. While we can write derivations involving these "exotic" requirement kinds, certain inference rules are missing. The type substitution algebra will fill in some of these gaps.

Suppose that Σ is a substitution map where all replacement types are fully concrete, so we can apply Σ to a requirement and then apply Algorithm 9A to check if its satisfied. Further, let's say that we can derive two concrete same-type requirements, where both have the same subject type parameter `T` of G:

1. $[\texttt{T} \texttt{ == } \texttt{X}_1]$ $(\ldots)$
2. $[\texttt{T} \texttt{ == } \texttt{X}_2]$ $(\ldots)$

(Note that with our existing inference rules, each requirement is either explicit, or an associated requirement of a protocol after substitution of `Self`; there are no other ways to "compose" concrete same-type requirements right now.) We can apply Σ to both requirements to get a pair of substituted requirements:

$$[\texttt{T} \otimes \Sigma \texttt{ == } \texttt{X}_1 \otimes \Sigma] \qquad \text{and} \qquad [\texttt{T} \otimes \Sigma \texttt{ == } \texttt{X}_2 \otimes \Sigma].$$

If Σ is a well-formed substitution map, it must satisfy both substituted requirements; that is, $\texttt{T} \otimes \Sigma$ must be canonically equal to $\texttt{X}_1 \otimes \Sigma$, and also $\texttt{T} \otimes \Sigma$ must be canonically equal to $\texttt{X}_2 \otimes \Sigma$. But canonical type equality is transitive, so it follows that $\texttt{X}_1 \otimes \Sigma$ must be canonically equal to $\texttt{X}_2 \otimes \Sigma$.

In other words, a necessary condition for Σ to be well-formed is that it must satisfy the requirement $[\texttt{X}_1 \texttt{ == } \texttt{X}_2] \otimes \Sigma$. Now, $[\texttt{X}_1 \texttt{ == } \texttt{X}_2]$ is not a derived requirement of G, because it has a concrete type on both sides. However, we know what to do if the user writes such a requirement directly; we apply Algorithm 11B. If we do the same here, each of the possible outcomes gives us further information about G:

- If $\texttt{X}_1$ does not match $\texttt{X}_2$ in the sense used by the desugaring algorithm, then no substitution map Σ can simultaneously satisfy both $[\texttt{T} \texttt{ == } \texttt{X}_1]$ and $[\texttt{T} \texttt{ == } \texttt{X}_2]$, so these two requirements are in conflict with each other, and G must be rejected.
- Otherwise, we always obtain a simpler list of requirements $\{R_1, \ldots, R_n\}$ with the property that Σ satisfies $[\texttt{X}_1 \texttt{ == } \texttt{X}_2]$ if and only if it satisfies R_i for all $1 \leq i \leq n$. If one of the original derived requirements was actually an explicit requirement of G, we can replace it with $\{R_1, \ldots, R_n\}$ without changing the "intended meaning" of the generic signature. (This might change the theory though, but only because our theory is missing some inference rules, as we already said.)

Let's revisit Example 5.27. We declare a protocol `Foo` with two associated types, together with an associated same-type requirement $[\texttt{Self.A == Array<Self.B>}]_{\texttt{Foo}}$:

```
protocol Foo {
  associatedtype A where A == Array<B>
  associatedtype B
}
```

Now, consider this function:

```
func f1<T: Foo>(_: T) where T.A == Array<Int> {}
```

We can derive two concrete same-type requirements involving `T`; the first one is explicit, the second is a consequence of the associated same-type requirement in `Foo`:

1. [`T.A == Array<Int>`] (Concrete)
2. [`T: Foo`] (Conf)
3. [`T.A == Array<T.B>`] (AssocConcrete 2)

As per our discussion above, a substitution map satisfying these requirements must also satisfy [`Array<Int> == Array<T.B>`], which desugars to [`T.B == Int`]. We can, in fact, replace our explicit same-type requirement with this desugared requirement, so the final generic signature is this:

```
<T where T: Foo, T.[Foo]B == Int>
```

Notice how these two lists of requirements are *not* equivalent by Proposition 11.10, because we cannot derive [`T.B == Int`] from the first list of requirements; once the derived requirements formalism has been fully fleshed out, these ought to become equivalent:

{[`T: Foo`], [`T.A == Array<Int>`]}
{[`T: Foo`], [`T.B == Int`]}

Next, we're going to change our example slightly to get a pair of conflicting requirements, and we see that we diagnose an error:

```
func f2<T: Foo>(_: T) where T.A == Set<Int> {}

// error: no type for 'T.A' can satisfy both 'T.A == Set<Int>' and
// 'T.A == Array<T.C>'
```

What if *both* of our same-type requirements are explicit? We now take this protocol:

```
protocol Foe {
  associatedtype X
  associatedtype Y
}
```

So that we can define this function:

```
func f3<T: Foe>(_: T) where T.X == Set<T.Y>, T.A == Set<Int> {}
```

In `f3()`, we can replace *either* requirement with [`T.X == Int`] without changing the "meaning" of our generic signature, but since the first requirement's subject type is not reduced, we replace it first. Thus, we get the following generic signature:

```
<T where T: Foe, T.[Foe]X == Set<Int>, T.[Foe]Y == Int>
```

Definition 11.13. Let G be a well-formed generic signature. If G has a pair of derived requirements R_1 and R_2 such that for any substitution map Σ, at least one of $R_1 \otimes \Sigma$ or $R_2 \otimes \Sigma$ is always unsatisfied, then R_1 and R_2 are *conflicting requirements*. We can also characterize conflicting requirements in terms of requirement desugaring:

1. For two concrete same-type requirements [`T == X`$_1$] and [`T == X`$_2$], we desugar the "combined" requirement [`X`$_1$ `==` `X`$_2$] using Algorithm 11B. Desugaring will either detect a conflict, or produce a simpler list of requirements to replace one of the two original requirements, in which case we can look for conflicts again.

2. For a concrete same-type requirement [`T == X`] and superclass requirement [`T: C`], we desugar [`X: C`]. Desugaring will succeed if and only if `X` is a class type that is also a subclass of `C`, otherwise it will detect a conflict.

3. For a same-type requirement [`T == X`] and a layout requirement [`T: AnyObject`], we desugar [`X: AnyObject`]. This will succeed if and only if `X` is any class type.

4. For a same-type requirement [`T == X`] and a conformance requirement [`T: P`], we desugar [`X: P`]. This will succeed if and only if `X` conforms to `P`.

5. For two superclass requirements [`T: C`$_1$] and [`T: C`$_2$], we must consider the superclass relationship between the declarations of `C`$_1$ and `C`$_2$:
   a) If the class declaration of `C`$_1$ is a subclass of the declaration of `C`$_2$, we desugar [`C`$_1$`: C`$_2$], and [`T: C`$_2$] becomes redundant.
   b) If the class declaration of `C`$_2$ is a subclass of the declaration of `C`$_1$, we desugar [`C`$_2$`: C`$_1$], and [`T: C`$_1$] becomes redundant.
   c) If the two declarations are unrelated, we have a conflict.

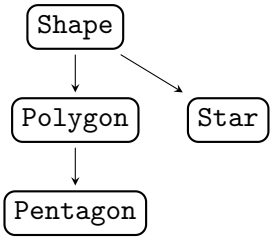
6. For a superclass requirement [`T: C`] and a layout requirement [`T: AnyObject`], we desugar [`C: AnyObject`], which is always satisfied and cannot conflict.

7. For a superclass requirement [`T: C`] and a conformance requirement [`T: P`], we desugar [`C: P`]. If `C` conforms to `P`, the conformance requirement [`T: P`] becomes redundant. However, if `C` does not conform to `P`, there is no conflict; the generic signature just requires that `T` is a subclass of `C` that *also* conforms to `P`.

We say G is *conflict-free* if it does not have any pairs of conflicting requirements.

We will investigate the compiler's implementation of superclass, layout, and concrete same-type requirements in Chapter 19, but in this section, we're just going to look at a couple of examples.

Our next example involves superclass requirements. While a description of generic classes awaits us in Chapter 6.4, we only need non-generic classes to demonstrate the key ideas in requirement minimization. Since it is such a cliché at this point, we're going to follow the trend and go with the classic object oriented "shape hierarchy", shown on the right. We also introduce a `Canvas` protocol, which subjects it associated type to an associated superclass requirement $[\texttt{Self.Boundary: Polygon}]_{\texttt{Canvas}}$:




```
class Shape {}
class Polygon: Shape {}
class Pentagon: Polygon {}
class Star: Shape {}

protocol Canvas {
  associatedtype Boundary: Polygon
}
```

Our first function imposes a more general superclass bound on `C.Boundary` than what is implied by the conformance requirement `[C: Canvas]`:

```
func h1<C: Canvas>(_: C) where C.Boundary: Shape {}
```

Because every `Polygon` is also a `Shape`, the requirement `[C.Boundary: Shape]` is redundant, so we're just left with `[C: Canvas]`.

The second function tightens the superclass bound on `C.Boundary`:

```
func h2<C: Canvas>(_: C) where C.Boundary: Pentagon {}
```

Not every `Polygon` is a `Pentagon`, so the minimal generic signature of `h2()` includes the requirement `[C.[Canvas]Boundary: Pentagon]` in addition to `[C: Canvas]`.

Finally, if we attempt to impose an unrelated superclass bound, we get an error diagnosing the conflict:

```
func h3<C: Canvas>(_: C) where C.Boundary: Star {}

// error: no type for 'C.Boundary' can satisfy both 'C.Boundary : Star'
// and 'C.Boundary : Polygon'
```

Our final example looks at the interaction between conformance and concrete same-type requirements. Consider the generic signature of the extension of `Box`:

```
struct Box<Contents: Sequence> {}
extension Box where Contents == Array<Int> {}
```

To build the generic signature of the extension, we take the generic signature of `Box`, and add the requirement [`Contents == Array<Int>`], so requirement minimization starts with this list of requirements:

{[`Contents: Sequence`], [`Contents == Array<Int>`]}

By Definition 11.13, we can understand the interaction between the two requirements by desugaring the requirement [`Array<Int>: Sequence`]. We look up the conformance and see that this requirement is satisfied, so our original requirement [`Contents: Sequence`] is redundant. We're left with the same-type requirement [`Contents == Array<Int>`], so here is our extension's generic signature:

`<Contents where Contents == Array<Int>>`

This generic signature certainly generates a different, much smaller theory than the original list of requirements that includes [`Contents: Sequence`]. For example, the dependent member type `Contents.Element` is not a valid type parameter in the new generic signature, precisely because we cannot derive [`Contents: Sequence`]. This gives us another example where requirement minimization does not preserve equivalence of generic signatures, because of gaps in our theoretical understanding.

This is one of those `GenericSignatureBuilder` behaviors that proved to be slightly obnoxious in hindsight; it would have been nicer if the conformance requirement was not actually considered to be redundant here. As a practical matter, it means that a valid type parameter of the extended type's generic signature might no longer be a valid type parameter in the generic signature of the extension. The implementation of the `getReducedType`() generic signature query makes a special exception to allow such type parameters anyway, by attempting to resolve the concrete conformance (Section 19.5).

Requirement order. Once we have our minimal requirements, the last step is to sort them in a canonical way. As defined, the below algorithm is a partial order because it can return "⊥", but this can only happen if both requirements have the same subject type and same kind, and they're not conformance requirements. Consulting Definition 11.13, we see this if two minimal requirements have this property, they must conflict. Thus, the requirements of a minimal conflict-free generic signature can be linearly ordered without ambiguity.

Algorithm 11D (Requirement order)**.** Takes two requirements as input, and returns one of "<", ">", "=" or "⊥" as output.

1. (Subject) Compare the subject types of the two requirements with Algorithm 5D. Return the result if it is "<" or ">". Otherwise, both requirements have the same subject type.

2. (Kind) Compare their kinds. If they have different kinds, return "<" or ">" based on the relative order below:

   $$\text{superclass} < \text{layout} < \text{conformance} < \text{same-type}$$

   Otherwise, both requirements have the same subject type and the same kind.

3. (Protocol) If both are conformance requirements, compare their protocols with Algorithm 5B, and return one of "<", "=", or ">".

4. (Incomparable) Otherwise, both requirements have the same subject type and kind, and they're not conformance requirements. Return "⊥".

Requirement signatures. The associated requirements in a protocol's requirement signature are minimized just like the explicit requirements of a generic signature. The proof of Proposition 11.10 can be tweaked slightly to instead use a equivalence relation on requirement signatures. (The key idea is that for every associated requirement of a protocol `P`, we can derive the corresponding requirement in the protocol generic signature $G_{\texttt{P}}$.) Minimal and reduced associated requirements are defined in the same way, and the **requirement signature request** always outputs a minimal requirement signature. We can describe the conflicts among the associated requirements using Definition 11.13, and finally, sort the minimal associated requirements in a requirement signature using Algorithm 11D.

The major difference is that we must minimize all requirement signatures of a set of mutually-dependent protocols, or a *protocol component*, simultaneously. We will discuss this again in Section 15.1 and see an example in Section 20.1.

11.5. Source Code Reference

Requests

Key source files:

- `include/swift/AST/TypeCheckRequests.h`
- `lib/AST/RequirementMachine/RequirementMachineRequests.cpp`

The header file declares the requests; the evaluation functions are implemented by the Requirement Machine (Section 15.3).

`GenericSignature` *class*

See also Section 5.7.

- `get()` is the primitive constructor, which builds a generic signature directly from a list of generic parameters and minimal requirements.

`GenericSignatureRequest` *class*

The `GenericContext::getGenericSignature()` method (Section 5.7) evaluates this request, which either returns the parent declaration's generic signature, or evaluates `InferredGenericSignatureRequest` with the appropriate arguments.

`InferredGenericSignatureRequest` *class*

The request evaluator request for building a generic signature from requirements written in source. The arguments were previously discussed at the beginning of this chapter:

1. The `GenericSignature` of the parent context, if any.
2. The current generic context's `GenericParamList`, if any.
3. The current generic context's `WhereClauseOwner`, if any.
4. A vector of `Requirement` storing any additional requirements to add.
5. A vector of `TypeLoc` eligible for requirement inference.
6. A flag indicating whether generic parameters can be subject to concrete same-type requirements.

`WhereClauseOwner` *class*

A reference to a `where` clause attached to a declaration, This is a wrapper type used by requirement resolution, which is the first step in building the generic signature in `InferredGenericSignatureRequest`. Constructed with one of the following:

- A `GenericContext` representing a generic declaration.
- An `AssociatedTypeDecl`, which can also have a `where` clause despite not being a `GenericContext`.
- A `GenericParamList`, which only has a `where` clause in textual SIL.

- A `SpecializeAttr` representing a `@_specialize` attribute.
- An instance of a `TrailingWhereClause`, which is the primitive form for several of the above.

A pair of methods are defined for working with the requirements in the `where` clause:

- The `getRequirements()` method returns an array of `RequirementRepr`, which is the parsed representation of a requirement.
- The `visitRequirements()` method takes a callback and a `TypeResolutionStage`. It resolves each `RequirementRepr` to a `Requirement`, and passes both to the callback.

The `visitRequirements()` method is called with `TypeResolutionStage::Structural` by the `InferredGenericSignatureRequest` and `RequirementSignatureRequest`.

The `TypeCheckPrimaryFileRequest` then visits all `where` clauses in every primary file again, this time with `TypeResolutionStage::Interface`. This is how we diagnose any invalid dependent member types in `where` clauses. Recall that the structural resolution stage builds unbound dependent member types, without any knowledge of what associated type declarations are visible.

`AbstractGenericSignatureRequest` *class*

The request evaluator request for building a generic signature from a list of generic parameters and requirements. The result is a pair consisting of a generic signature and some error flags. Most callers don't care about the error flags, so they use the following function instead.

`buildGenericSignature()` *function*

A utility function wrapping the `AbstractGenericSignatureRequest`. It checks and discards the returned error flags. If the `CompletionFailed` error flag is set, this aborts the compiler. The other two flags are ignored.

`GenericSignatureErrorFlags` *enum class*

Error flags returned by `AbstractGenericSignatureRequest`. We will see these conditions again in Chapter 15; they prevent the requirement machine for this signature from being *installed*.

- `HasInvalidRequirements`: the original requirements were not well-formed, or were in conflict with each other. Any errors in the requirements handed to this request usually mean there was another error diagnosed elsewhere, like an invalid conformance, so this flag being set is not really actionable to the rest of the compiler. Without source location information, this error cannot be diagnosed in a friendly manner.

- `HasConcreteConformances`: the generic signature had non-redundant concrete conformance requirements, which is an internal flag used to prevent the requirement machine from being installed. It does not indicate an error condition to the caller. See Section 20.5 for discussion.
- `CompletionFailed`: the completion procedure could not construct a convergent rewriting system within the maximum number of steps (see the discussion of termination that immediately follows Algorithm 18E). This is actually fatal, so the `buildGenericSignature()` wrapper function aborts the compiler in this case.

`RequirementSignature` *class*

See also Section 5.7.

- `get()` is the primitive constructor, which builds a requirement signature directly from a list of minimal requirements and protocol type aliases.

`RequirementSignatureRequest` *class*

The `ProtocolDecl::getRequirementSignature()` method (Section 5.7) evaluates this request, which computes the protocol's requirement signature if the protocol is in the main module, or deserializes it if the protocol is from a serialized module.

`StructuralRequirementsRequest` *class*

The `ProtocolDecl::getStructuralRequirements()` method evaluates this request to resolve the user-written requirements from which the protocol's requirement signature is formed.

`TypeAliasRequirementsRequest` *class*

The `ProtocolDecl::getTypeAliasRequirements()` method evaluates this request to collect the type alias declarations in a protocol and converts them into requirements from which the protocol's requirement signature is formed.

Requirement Resolution

Key source files:

- `include/swift/AST/Requirement.h`
- `lib/AST/RequirementMachine/RequirementLowering.cpp`

User-written requirements are wrapped in the `StructuralRequirement` type, which stores a `Requirement` together with a source location used for diagnostics. A couple of functions defined in `RequirementLowering.cpp` construct `StructuralRequirement`

instances. The `InferredGenericSignatureRequest` calls these functions directly. The `RequirementSignatureRequest` delegates to `StructuralRequirementsRequest`, which uses them to resolve requirements written in a protocol declaration.

`rewriting::realizeRequirement()` *function*

Calls the `WhereClauseOwner::visitRequirements()` method to resolve requirements written in `where` clauses, and wraps the results in `StructuralRequirement` instances.

`rewriting::realizeInheritedRequirements()` *function*

Resolves the inheritance clause entries of a type declaration, then builds conformance, superclass and layout requirements with the appropriate subject type, and wraps them in `StructuralRequirement` instances.

Requirement Inference

Key source files:

- `lib/AST/RequirementMachine/RequirementLowering.h`
- `lib/AST/RequirementMachine/RequirementLowering.cpp`

The `realizeRequirement()` and `realizeInheritedRequirements()` functions take a flag indicating if requirement inference should be performed; recall that requirement inference is not performed in protocols, so `StructuralRequirementsRequest` does not pass this flag.

`rewriting::inferRequirements()` *function*

Recursively walks a `Type` and constructs `StructuralRequirement` instances from all generic nominal and generic type alias types contained therein.

Requirement Desugaring

Key source files:

- `lib/AST/RequirementMachine/Diagnostics.h`
- `lib/AST/RequirementMachine/RequirementLowering.h`
- `lib/AST/RequirementMachine/RequirementLowering.cpp`

The `realizeRequirement()` and `realizeInheritedRequirements()` functions also perform requirement desugaring. For `AbstractGenericSignatureRequest`, requirement desugaring is the entry point where the fun begins; it starts from a list of requirements instead of resolving user-written requirement representations.

`rewriting::desugarRequirement()` *function*

Establishes the invariants in Definition 11.1, splitting up conformance requirements and simplifying requirements where the subject type is a concrete type.

`RequirementError` *class*

Represents a redundant or conflicting requirement detected by requirement desugaring or minimization.

Requirement Minimization

Key source files:

- `lib/AST/GenericSignature.cpp`

Just as Section 11.4 only describes the invariants around minimization, here we only call out the code related to checking those invariants. For the actual implementation of minimization, see Section 20.6.

`Requirement` *class*

See also Section 5.7.

- `compare()` implements the requirement order (Algorithm 11D), returning one of the following:
  - -1 if this requirement precedes the given requirement,
  - 0 if the two requirements are equal,
  - 1 if this requirement follows the right hand side.

  This method will assert if the two requirements are incomparable; an example is when two superclass requirements have the same subject type. Incomparable requirements should not appear in a generic signature.

`GenericSignatureImpl` *class*

See also Section 5.7.

- `verify()` ensures that all explicit requirements in this signature are desugared (Definition 11.1), reduced (Definition 11.11), minimal (Definition 11.12), and ordered (Algorithm 11D). Any violations report a fatal error that crashes the compiler even in no-assert builds, since such generic signatures should not be built at all.

12. Conformance Paths

We now return to tie up a loose end in type substitution. Let's start by reviewing what we know so far. A substitution map is a finite data structure—it stores a series of replacement types and conformances, corresponding to each generic parameter and conformance requirement of its generic signature. However, the semantic point of view is that a substitution map assigns a replacement type to each valid type parameter of its generic signature, of which there might be infinitely many. When the signature has no conformance requirements, the only valid type parameters *are* the generic parameters, so type substitution is easy to describe. In the general case, the set of valid type parameters also includes dependent member types derived from these conformance requirements.

In Section 7.4, we introduced the local conformance lookup operation, so the problem of applying a substitution map to a dependent member type became the projection of a type witness from a conformance. However, we punted on explaining how local conformance lookup works, except when the dependent member type can be derived in one step from an explicit conformance requirement. This corresponds to a *conformance path* of length 1. In this chapter, we will study more general conformance paths.

Example 12.1. To motivate the concept, let G be this generic signature:

```
<τ_0_0, τ_0_1 where τ_0_0: Collection,
                    τ_0_1 == τ_0_0.[Collection]SubSequence>
```

We will also be working with the `String`, `Substring`, and `Character` types from the standard library, and their conformances to `Sequence` and `Collection`. We already studied these protocols in Example 5.13 and Example 8.10. Listing 12.1 recalls the protocol declarations, and lists the relevant facts about the following normal conformances:

[`String: Collection`]

[`Substring: Collection`]

[`String: Sequence`]

[`Substring: Sequence`]

We're going to apply this substitution map to various dependent member types:

$\Sigma :=$ {`τ_0_0` $\mapsto$ `String`,
`τ_0_1` $\mapsto$ `Substring`;
[`τ_0_0: Collection`] $\mapsto$ [`String: Collection`]}

Listing 12.1.: Some standard library protocols and their conformances

```
protocol IteratorProtocol {
  associatedtype Element
}

protocol Sequence {
  associatedtype Element
  associatedtype Iterator: IteratorProtocol
    where Element == Iterator.Element
}

protocol Collection: Sequence {
  associatedtype SubSequence: Collection
    where Element == SubSequence.Element,
          SubSequence == SubSequence.SubSequence
}
```

♣ [String: Collection]		
⟨Collection|SubSequence	↦	Substring
⟨Self: Sequence]	↦	[String: Sequence]
⟨Self.SubSequence: Collection]	↦	[Substring: Collection]

♣ [Substring: Collection]		
⟨Collection|SubSequence	↦	Substring
⟨Self: Sequence]	↦	[Substring: Sequence]
⟨Self.SubSequence: Collection]	↦	[Substring: Collection]

♣ [String: Sequence]		
⟨Sequence|Element	↦	Character
⟨Sequence|Iterator	↦	IndexingIterator<String>
⟨Self.Iterator: IteratorProtocol]	↦	[IndexingIterator<String>: IP]

♣ [Substring: Sequence]		
⟨Sequence|Element	↦	Character
⟨Sequence|Iterator	↦	IndexingIterator<Substring>
⟨Self.Iterator: IteratorProtocol]	↦	[IndexingIterator<Substring>: IP]

("IP" should say "IteratorProtocol" above, but of course it doesn't fit.)

We're going to jump back and forth between the derived requirements formalism and the type substitution algebra, so we recall the terminology that linked them from Section 7.4:

- A *conformance requirement* is an *abstract conformance*.
- A *derived* conformance requirement is a *valid* abstract conformance.
- An *explicit* conformance requirement is a *root* abstract conformance.

To warm up, let's use Algorithm 9B to show that Σ satisfies the explicit requirements of G. It is clear that it satisfies the conformance requirement. To see that it satisfies the same-type requirement, we apply Σ to both sides. The right-hand side is a dependent member type, so we write it as the type witness of a root abstract conformance:

$$\begin{aligned}
&\texttt{τ_0_1} \otimes \Sigma = \texttt{Substring} \\
&\texttt{τ_0_0.[Collection]SubSequence} \otimes \Sigma \\
&\qquad = \langle \texttt{Collection} | \texttt{SubSequence} \otimes [\texttt{τ_0_0: Collection}] \otimes \Sigma \\
&\qquad = \langle \texttt{Collection} | \texttt{SubSequence} \otimes [\texttt{String: Collection}] \\
&\qquad = \texttt{Substring}
\end{aligned}$$

We see that Σ maps both sides of our same-type requirement to `Substring`.

At this point, we haven't encountered anything new. Now, we will attempt to compute $\texttt{τ_0_1.[Sequence]Iterator} \otimes \Sigma$. Again, we can express the dependent member type as the type witness of an abstract conformance:

$$\begin{aligned}
&\texttt{τ_0_1.[Sequence]Iterator} \otimes \Sigma \\
&\qquad = \langle \texttt{Sequence} | \texttt{Iterator} \otimes [\texttt{τ_0_1: Sequence}] \otimes \Sigma
\end{aligned}$$

This time, $[\texttt{τ_0_1: Sequence}]$ is not an explicit conformance requirement of G. Using the fact that $[\texttt{τ_0_1: Sequence}] = \langle \texttt{Sequence} \rangle \otimes \texttt{τ_0_1}$ and $\texttt{τ_0_1} \otimes \Sigma = \texttt{Substring}$, we still can figure out the answer:

$$\begin{aligned}
&[\texttt{τ_0_1: Sequence}] \otimes \Sigma \\
&\qquad = \langle \texttt{Sequence} \rangle \otimes \texttt{τ_0_1} \otimes \Sigma \\
&\qquad = \langle \texttt{Sequence} \rangle \otimes \texttt{Substring} \\
&\qquad = [\texttt{Substring: Sequence}]
\end{aligned}$$

However, we wish to recover $[\texttt{Substring: Sequence}]$ from Σ alone, without resorting to a *global* conformance lookup, as we did in the above calculation. We don't have a lot of options. In fact, pretty much the *only* thing local conformance lookup can do is

take one of the root conformances stored in the substitution map, and proceed to apply associated conformance projection operations.

Our generic signature G only has one explicit conformance requirement, so we start with the lone root conformance of Σ:

$$[\texttt{τ_0_0: Collection}] \otimes \Sigma = [\texttt{String: Collection}]$$

The associated conformance for $\langle$`Self.SubSequence: Collection`] looks interesting, so we project it out:

$$\langle\texttt{Self.SubSequence: Collection}] \otimes [\texttt{String: Collection}]$$
$$= [\texttt{Substring: Collection}]$$

We're looking for [`Substring: Sequence`], not [`Substring: Collection`], but we can get there with one more associated conformance projection:

$$\langle\texttt{Self: Sequence}] \otimes [\texttt{Substring: Collection}]$$
$$= [\texttt{Substring: Sequence}]$$

Once we have the [`Substring: Sequence`] conformance, projecting the type witness for `Iterator` gives us the final substituted type:

$$\langle\texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Substring: Sequence}]$$
$$= \texttt{IndexingIterator<Substring>}$$

If we set aside for now the question of *why* this appears to work—a very important question we will study in detail—we see that $\texttt{τ_0_1.[Sequence]Iterator} \otimes \Sigma$ can be recovered from Σ in four steps: we project the root conformance from Σ, then we project an associated conformance, then another, and finally we project a type witness:

$$\texttt{τ_0_1.[Sequence]Iterator} \otimes \Sigma$$
$$= \langle\texttt{Sequence}|\texttt{Iterator} \otimes \underbrace{[\texttt{τ_0_1: Sequence}] \otimes \Sigma}_{\text{local conformance lookup}}$$
$$= \langle\texttt{Sequence}|\texttt{Iterator}$$
$$\otimes \Big(\langle\texttt{Self: Sequence}]$$
$$\otimes \big(\langle\texttt{Self.SubSequence: Collection}]$$
$$\otimes \big([\texttt{τ_0_0: Collection}]$$
$$\otimes \Sigma)\big)\Big) \Big\} \text{ local conformance lookup}$$
$$= \langle\texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Substring: Sequence}]$$
$$= \texttt{IndexingIterator<Substring>}$$

If we peel off the type witness projection and substitution map at either end of the long expression in the middle, we're left with a conformance path for the abstract conformance $[\texttt{τ_0_1: Sequence}]$ of G:

$$\langle \texttt{Self: Sequence} \rangle \otimes \langle \texttt{Self.SubSequence: Collection} \rangle \otimes [\texttt{τ_0_0: Collection}]$$

A conformance path only depends on the generic signature, and not on any particular substitution map. We can use the above conformance path to perform a local conformance lookup of $[\texttt{τ_0_1: Sequence}]$ within *any* substitution map with the same input generic signature G.

Definition 12.2. Let G be a generic signature. Formally, a *conformance path* is an ordered tuple $(s_1, \ldots, s_n)$ of length $n \geq 1$, such that the first step s_1 is a root abstract conformance of G, and each subsequent step s_i is an associated conformance requirement stated by the protocol that appears on the right-hand side of the previous step. However, instead of using tuple notation, we write conformance paths from *right to left*:

$$s_n \otimes \cdots \otimes s_1$$

Alternatively, we can write something like this:

$$\langle \texttt{Self.U}_n \texttt{: P}_n \rangle \otimes \cdots \otimes \langle \texttt{Self.U}_2 \texttt{: P}_2 \rangle \otimes [\texttt{T}_1 \texttt{: P}_1]$$

A conformance path of length 1 is just single root abstract conformance $[\texttt{T}_1 \texttt{: P}_1]$.

We can now reveal the missing subroutine of Algorithm 7B. We still don't know how to actually find a conformance path for a given abstract conformance, but if we delegate that to yet another subroutine, we see that local conformance lookup must *evaluate* this conformance path with the given substitution map Σ:

$$(s_n \otimes (\cdots \otimes (s_2 \otimes (s_1 \otimes \Sigma)) \cdots))$$

Algorithm 12A (Local conformance lookup)**.** As input, takes a substitution map Σ, and an abstract conformance $[\texttt{T: P}]$. Returns $[\texttt{T: P}] \otimes \Sigma$.

1. Find a conformance path $s_n \otimes \cdots \otimes s_1$ for $[\texttt{T: P}]$ using Algorithm 12D.
2. (Root) Suppose that $s_1 = [\texttt{T}_1 \texttt{: P}_1]$. Set $C \leftarrow [\texttt{T}_1 \texttt{: P}_1] \otimes \Sigma$. (This is not a recursive call; we just load the root conformance from Σ.) Set $i \leftarrow 2$.
3. (Check) If $i = n + 1$, return C.
4. (Project) Since $i > 1$, we know that $s_i = \langle \texttt{Self.U}_i \texttt{: P}_i \rangle$ for some type parameter $\texttt{Self.U}_i$ in the protocol generic signature of $\texttt{P}_i$. Set $C \leftarrow \langle \texttt{Self.U}_i \texttt{: P}_i \rangle \otimes C$.
5. (Next) Set $i \leftarrow i + 1$ and go back to Step 3.

In Step 4, the projection operation only makes sense if C is a conformance to $\texttt{P}_{i-1}$; this is guaranteed by the condition in Definition 12.2. Furthermore, the protocol $\texttt{P}_n$ that appears on the right-hand side of the final step s_n must be identical to the given $\texttt{P}$.

12.1. Validity and Existence

Instead of evaluating a conformance path with a substitution map, we can also view it as a standalone expression in the type substitution algebra:

$$\begin{aligned}&\langle \texttt{Self: Sequence}] \otimes \langle \texttt{Self.SubSequence: Collection}] \otimes [\texttt{τ_0_0: Collection}] \\ &\quad = \langle \texttt{Self: Sequence}] \otimes [\texttt{τ_0_0.[Collection]SubSequence: Collection}] \\ &\quad = [\texttt{τ_0_0.[Collection]SubSequence: Sequence}]\end{aligned}$$

(We write associated conformance projections with unbound type parameters to save space, but really all type parameters in a requirement signature are bound.) We say that a conformance path *represents* the abstract conformance obtained in this manner, and an abstract conformance is *principal* if it can be represented by a conformance path.

Algorithm 12B (Simplify conformance path)**.** Takes a conformance path $s_n \otimes \cdots \otimes s_1$ and outputs the principal abstract conformance represented by this conformance path.

1. (Root) Suppose that $s_1 = [\texttt{T}_1\texttt{: P}_1]$. Set $\texttt{P} \leftarrow \texttt{P}_1$, $\texttt{T} \leftarrow \texttt{T}_1$, and $i \leftarrow 2$.
2. (Check) If $i = n + 1$, return $[\texttt{T: P}]$.
3. (Step) Suppose that $s_i = \langle \texttt{Self.U}_i\texttt{: P}_i]$. Set $\texttt{P} \leftarrow \texttt{P}_i$ and $\texttt{T} \leftarrow \texttt{T.U}_i$, where $\texttt{T.U}_i$ is formed by replacing `Self` with `T` in $\texttt{Self.U}_i$.
4. (Next) Set $i \leftarrow i + 1$ and go back to Step 2.

The above is actually just a special case of Algorithm 12A with the identity substitution map 1_G, but it is convenient for what follows to implement it directly.

Equivalence. In Example 12.1, our conformance path for $[\texttt{τ_0_1: Sequence}]$ actually represents $[\texttt{τ_0_0.[Collection]SubSequence: Sequence}]$. While not identical, their subject types $\texttt{τ_0_1}$ and $\texttt{τ_0_0.[Collection]SubSequence}$ are equivalent, because of the same-type requirement of G. This suggests the following equivalence relation.

Definition 12.3. Let G be a generic signature. We say that two abstract conformances $[\texttt{T: P}]$ and $[\texttt{T}'\texttt{: P}]$ are *equivalent* with respect to G if we can derive $G \vdash [\texttt{T == T}']$, or in other words, if `T` and $\texttt{T}'$ are equivalent type parameters of G. Under this relation, every equivalence class contains exactly one *reduced abstract conformance*, which is the one whose subject type is a reduced type parameter of G.

So, $[\texttt{τ_0_1: Sequence}]$ and $[\texttt{τ_0_0.[Collection]SubSequence: Sequence}]$ belong to the same equivalence class in our example; the former is reduced, while the latter is principal. As far as type substitution is concerned, both abstract conformances are interchangeable. In general, if $\Sigma \in \textsc{Sub}(G, H)$ is well-formed (Definition 10.7), then applying Σ to the subject types of two equivalent abstract conformances in G will always output two equivalent interface types of H.

Validity. So far, we've only established the legitimacy of one single conformance path. To generalize this further, we must give an affirmative answer to three questions:

1. Is every principal abstract conformance also a valid abstract conformance?
2. Is every valid abstract conformance equivalent to a principal abstract conformance?
3. Given a valid abstract conformance, can we find a conformance path representing an equivalent principal abstract conformance?

We will answer (1) and (2) first, before we look at Algorithm 12D to solve (3) in the next section. Recall that $[\texttt{T: P}]$ is a valid abstract conformance if $G \vdash [\texttt{T: P}]$. To establish (1), we can show that a conformance path can be translated into a special kind of derivation.

Definition 12.4. Let G be a generic signature. A *principal derivation* is a derivation of a conformance requirement that consists of a single Conf elementary statement, followed by zero or more steps that apply AssocConf to the conclusion of the previous step, with no unused conclusions or other kinds of steps:

1. $[\texttt{T}_1\texttt{: P}_1]$ (Conf)
2. $[\texttt{T}_1\texttt{.U}_2\texttt{: P}_2]$ (AssocConf $[\texttt{Self.U}_2\texttt{: P}_2]_{\texttt{P}_1}$ $[\texttt{T}_1\texttt{: P}_1]$)

$\vdots$

n. $[\texttt{T}_1\texttt{.U}_2\texttt{...U}_n\texttt{: P}_n]$ (AssocConf $[\texttt{Self.U}_n\texttt{: P}_n]_{\texttt{P}_{n-1}}$ $[\texttt{T}_1\texttt{.U}_2\texttt{...U}_{n-1}\texttt{: P}_{n-1}]$)

We can always write down a principal derivation for a conformance path, and likewise, "read off" a conformance path if we are given a principal derivation. The final conclusion of a principal derivation is identical to the principal abstract conformance we get if we apply Algorithm 12B to the conformance path.

Example 12.5. Once again, here is our favorite conformance path:

$$\langle\texttt{Self: Sequence}] \otimes \langle\texttt{Self.SubSequence: Collection}] \otimes [\texttt{τ_0_0: Collection}]$$

Here is the principal derivation of $G \vdash [\texttt{T.[Collection]SubSequence: Sequence}]$:

1. $[\texttt{τ_0_0: Collection}]$ (Conf)
2. $[\texttt{τ_0_0.[Collection]SubSequence: Collection}]$ (AssocConf 1)
3. $[\texttt{τ_0_0.[Collection]SubSequence: Sequence}]$ (AssocConf 2)

We can extend the above principal derivation with a SameConf step to get a derivation for $[\texttt{τ_0_1: Sequence}]$. This is no longer a principal derivation:

4. $[\texttt{τ_0_1 == τ_0_0.[Collection]SubSequence}]$ (Same)
5. $[\texttt{τ_0_1: Sequence}]$ (SameConf 3 4)

Existence. We will now establish that every equivalence class of abstract conformances contains at least one principal abstract conformance. To do this, we will show that a derivation of a conformance requirement always splits into a principal derivation, followed by a same-type requirement. The principal derivation defines a conformance path, and this conformance path represents a principal abstract conformance, equivalent to the original abstract conformance, via the same-type requirement.

First, we need a preliminary result, which will also be useful in Section 17.1. Recall from Section 5.3 and Section 5.4, that for each associated type `A` of each protocol `P` we defined a pair of inference rules, SameName and SameDecl:

$$[\texttt{T.A == U.A}] \qquad (\textsc{SameName}\ [\texttt{U: P}]\ [\texttt{T == U}])$$

$$[\texttt{T.[P]A == U.[P]A}] \qquad (\textsc{SameDecl}\ [\texttt{U: P}]\ [\texttt{T == U}])$$

These rules wrap both sides of a same-type requirement in a dependent member type, increasing their lengths by one. The below lemma shows that we can also iterate these steps and produce "longer" same-type requirements. This requires proof by induction, together with some facts about well-formed generic signatures (Section 11.3).

Lemma 12.6. Let G be a well-formed generic signature, with $G \vdash [\texttt{T == U}]$ for some type parameters `T` and `U`. Suppose that `T.V` and `U.V` are type parameters formed by wrapping `T` and `U` in dependent member types with pairwise identical identifiers or associated type declarations, so for some $n \geq 0$:

$$\texttt{T.V} := \texttt{T.A}_1\texttt{...A}_n \quad \texttt{and} \quad \texttt{U.V} := \texttt{U.A}_1\texttt{...A}_n$$

Also, suppose that at least one of `T.U` or `U.V` is valid, so $G \vdash \texttt{T.V}$ or $G \vdash \texttt{U.V}$. Then:

$$G \vdash [\texttt{T.V == U.V}]$$

Proof. We consider the case where $G \vdash \texttt{U.V}$ first. Note that $|\texttt{T.V}| = |\texttt{T}| + n$ and $|\texttt{U.V}| = |\texttt{U}| + n$. We proceed by induction on n.

Base case. If $n = 0$, then `T.V` is just `T`, and `U.V` is also `U`. We have $G \vdash [\texttt{T == U}]$ by assumption, so we already have the desired conclusion $G \vdash [\texttt{T.V == U.V}]$.

Inductive step. Suppose that $n > 0$, and consider the outermost dependent member type of `U.V`, which is either bound $\texttt{U}'\texttt{.[P]A}$ or unbound $\texttt{U}'\texttt{.A}$, for some type parameter $\texttt{U}'$ and associated type `A` of a protocol `P`. By assumption, `T.V` has the same form $\texttt{T}'\texttt{.[P]A}$ or $\texttt{T}'\texttt{.A}$, for some type parameter $\texttt{T}'$. Note that $|\texttt{T}'| = |\texttt{T}| - 1$ and $|\texttt{U}'| = |\texttt{U}| - 1$.

Since $G \vdash \texttt{U.V}$, we have $G \vdash [\texttt{U}'\texttt{: P}]$ by Proposition 11.2:

1. $[\texttt{U}'\texttt{: P}]$ (...)

Also, G is well-formed, so $G \vdash \texttt{U}'$. Applying the induction hypothesis, we get:

2. $[\texttt{T}' \texttt{ == } \texttt{U}']$ (...)

If `T.V` is $\texttt{T}'\texttt{.[P]A}$ and `U.V` is $\texttt{U}'\texttt{.[P]A}$, we apply SameDecl to (1) and (2):

3. $[\texttt{T}'\texttt{.[P]A == U}'\texttt{.[P]A}]$ (SameDecl 1 2)

If `U.V` is $\texttt{T}'\texttt{.A}$ and `U.V` is $\texttt{U}'\texttt{.A}$, we apply SameName to (1) and (2) instead:

3. $[\texttt{T}'\texttt{.A == U}'\texttt{.A}]$ (SameName 1 2)

In both cases, we conclude that $G \vdash [\texttt{T.V == U.V}]$, closing the induction.

To prove the same result from $G \vdash \texttt{T.V}$ instead of $G \vdash \texttt{U.V}$, we flip $[\texttt{T == U}]$ using Sym, and repeat the above construction with $[\texttt{U == T}]$ to get $[\texttt{U.V == T.V}]$. A second application of Sym yields $G \vdash [\texttt{T.V == U.V}]$. □

Example 12.7. If G is the generic signature of Example 12.1, we can use our lemma to derive $G \vdash [\texttt{τ_0_1.Iterator.Element == τ_0_0.SubSequence.Iterator.Element}]$ from $G \vdash [\texttt{τ_0_1 == τ_0_0.SubSequence}]$ and $G \vdash \texttt{τ_0_1.Iterator.Element}$.

Theorem 12.8. Let G be a well-formed generic signature. If $G \vdash [\texttt{T: P}]$ for some type parameter `T` and protocol `P`, there exists a valid type parameter $\texttt{T}'$ such that $G \vdash [\texttt{T}'\texttt{: P}]$ via a principal derivation, and $G \vdash [\texttt{T == T}']$.

Before we look at the proof, the reader may wish to review Appendix B, and recall that conformance requirements are always derived by one of three kinds of steps:

$[\texttt{T: P}]$ (Conf)

$[\texttt{T: P}]$ (SameConf $[\texttt{U: P}]$ $[\texttt{T == U}]$)

$[\texttt{U.V: P}]$ (AssocConf $[\texttt{Self.V: P}]_{\texttt{Q}}$ $[\texttt{U: Q}]$)

If $G \vdash [\texttt{T: P}]$ does not involve SameConf, it is already principal, and likewise if the final step applies SameConf to some principal derivation, we're done as well. The main difficulty is that AssocConf might be applied to the conclusion of SameConf, which requires us to "untangle" the two derivations.

Proof. We proceed by structural induction on the derivation of $G \vdash [\texttt{T: P}]$. We will construct the two desired derivations at each step.

Base case. A Conf elementary statement is the base case of our structural induction, because it has no assumptions. A derivation of an explicit conformance requirement is already principal by our definition, so we set $\texttt{T}' := \texttt{T}$. Our first derivation is just:

$[\texttt{T: P}]$ (Conf)

To get the second derivation, we note that G is well-formed, and `T` appears in $[\texttt{T: P}]$, so $G \vdash \texttt{T}$. We then derive $G \vdash [\texttt{T == T}]$ from $G \vdash \texttt{T}$ using the Reflex inference rule:

1. `T` (...)
2. $[\texttt{T == T}]$ (Reflex 1)

Inductive step. Otherwise, we have a derivation whose final step is an application of SameConf or AssocConf. We consider SameConf first:

$$[\texttt{T: P}] \qquad (\textsc{SameConf}\ [\texttt{U: P}]\ [\texttt{T == U}])$$

By the induction hypothesis, $G \vdash [\texttt{U: P}]$ splits up into our desired principal derivation $G \vdash [\texttt{T'': P}]$ for some $\texttt{T}''$, and a derivation of a same-type requirement $G \vdash [\texttt{U == T}'']$. We then apply Trans to $G \vdash [\texttt{T == U}]$ and $G \vdash [\texttt{U == T}'']$ to derive $G \vdash [\texttt{T == T}'']$.

The remaining case is when our derivation ends with AssocConf, so $\texttt{T}$ has the form $\texttt{U.V}$, meaning it is the result of replacing $\texttt{Self}$ with $\texttt{U}$ in some type parameter $\texttt{Self.V}$ of some protocol generic signature $G_{\texttt{Q}}$:

$$[\texttt{U.V: P}] \qquad (\textsc{AssocConf}\ [\texttt{Self.V: P}]_{\texttt{Q}}\ [\texttt{U: Q}])$$

This is the "untangling" part. While $G \vdash [\texttt{U: Q}]$ might not be principal, by induction, it splits up into a principal derivation $G \vdash [\texttt{U'': Q}]$ for some $\texttt{U}''$, and a same-type requirement $G \vdash [\texttt{U == U}'']$. We apply AssocConf to the first derivation to get a conformance to $\texttt{P}$, but with some other subject type $\texttt{U}''\texttt{.V}$:

1. $[\texttt{U}'': \texttt{Q}]$ $\qquad (\ldots)$
2. $[\texttt{U}''\texttt{.V: P}]$ $\qquad (\textsc{AssocConf}\ 1)$

This is our desired principal derivation, so we set $\texttt{T}'' := \texttt{U}''\texttt{.V}$. Since G is well-formed, we also have $G \vdash \texttt{U}''\texttt{.V}$. The conditions of Lemma 12.6 are satisfied, so we can derive the same-type requirement $G \vdash [\texttt{U.V == U}''\texttt{.V}]$, or in other words, $G \vdash [\texttt{T == T}'']$. □

12.2. The Conformance Path Graph

We embarked on our tour of type substitution in Chapter 6, with the following three algorithms marking key waypoints along the journey:

1. Algorithm 6A implements type parameter substitution, handling generic parameter types directly while delegating the dependent member type case.
2. Algorithm 7B implements dependent member type substitution in terms of local conformance lookup.
3. Algorithm 12A implements local conformance lookup, while delegating the finding of the conformance path.

With Theorem 12.8, we've established that, up to equivalence, every dependent member type can be written as the type witness of *some* conformance path. This provides the theoretical basis for all of the above, but it doesn't immediately lead to an effective procedure for finding a conformance path. However, the implementation will become apparent once we reformulate our problem in the language of graph theory.

Definition 12.9. Let G be a generic signature. The *conformance path graph* of G is the directed graph defined as follows:

- The vertices are the equivalence classes of abstract conformances of G. Each vertex is labeled with the reduced abstract conformance of its equivalence class.
- For each pair of abstract conformances $[\texttt{T: P}]$ and $[\texttt{T}'\texttt{: Q}]$ such that $[\texttt{T}'\texttt{: Q}]$ is equivalent to $[\texttt{T: P}] \otimes \langle\texttt{Self.U: Q}]$ for some associated conformance projection $\langle\texttt{Self.U: Q}]$, we add an edge with source vertex $[\texttt{T: P}]$ and destination vertex $[\texttt{T}'\texttt{: Q}]$. The edge is labeled $\langle\texttt{Self.U: Q}]$.

A conformance path is just a path in the conformance path graph. The path starts from a root abstract conformance, and ends at the principal abstract conformance that the conformance path represents; the vertices visited along the way are the intermediate values of $[\texttt{T: P}]$ in Algorithm 12B. We previously studied directed graphs in Section 8.3 when discussing the type parameter graph of a generic signature. The conformance path graph construction is similar, and it is instructive to compare the two:

	Type parameter graph	**Conformance path graph**
Roots	Generic parameter type	Root abstract conformance
Vertices	Reduced type parameter	Reduced abstract conformance
Edges	Associated type declaration	Associated conformance requirement
Paths	Type parameter	Conformance path

Theorem 12.8 is actually a statement about the conformance path graph, because it tells us that every vertex is reachable from at least one root vertex:

Definition 12.10. Let (V, E) be a directed graph. If $u, v \in V$, we say that v is *reachable* from u if there exists a path p with $\mathrm{src}(p) = u$ and $\mathrm{dst}(p) = v$. Note that every $v \in V$ is always reachable from itself via the empty path 1_v.

Example 12.11. Let's construct the conformance path graph from Example 12.1. The root abstract conformance gives us a single conformance path of length 1:

$$p_{11} := [\texttt{τ_0_0: Collection}]$$

By projecting each associated conformance requirement of `Collection`, we get the two conformance paths of length 2:

$$p_{21} := \langle\texttt{Self: Sequence}] \otimes p_{11}$$
$$p_{22} := \langle\texttt{Self.SubSequence: Collection}] \otimes p_{11}$$

From p_{21}, we get a conformance path of length 3:

$$p_{31} := \langle\texttt{Self.Iterator: IteratorProtocol}] \otimes p_{21}$$

The path p_{22} is another conformance to `Collection`, and the two projections give us two more conformance paths of length 3. Our favorite conformance path is p_{32} below:

$$p_{32} := \langle \texttt{Self: Sequence} \rangle \otimes p_{22}$$
$$p_{33} := \langle \texttt{Self.SubSequence: Collection} \rangle \otimes p_{22}$$

Finally, the path p_{32} can be extended to a conformance path of length 4:

$$p_{41} := \langle \texttt{Self.Iterator: IteratorProtocol} \rangle \otimes p_{32}$$

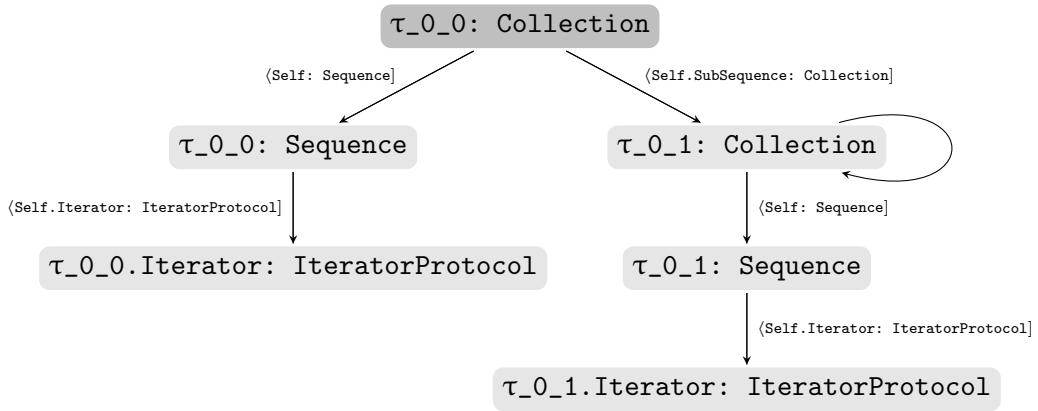
At this point, only the paths that start from p_{33} remain unexplored, but we observe that p_{22} and p_{33} describe equivalent abstract conformances:

$$[\texttt{τ_0_0.[Collection]SubSequence: Collection}]$$
$$[\texttt{τ_0_0.[Collection]SubSequence.[Collection]SubSequence: Collection}]$$

Thus, exploring p_{33} won't reveal anything new, so we're done. To get the vertex set of our graph, we simplify each conformance path to a principal abstract conformance, and compute the reduced type of its subject type to get a reduced abstract conformance:

$$p_{11} \to [\texttt{τ_0_0: Collection}]$$
$$p_{21} \to [\texttt{τ_0_0: Sequence}]$$
$$p_{22} \to [\texttt{τ_0_1: Collection}]$$
$$p_{31} \to [\texttt{τ_0_0.[Sequence]Iterator: IteratorProtocol}]$$
$$p_{32} \to [\texttt{τ_0_1: Sequence}]$$
$$p_{41} \to [\texttt{τ_0_1.[Sequence]Iterator: IteratorProtocol}]$$

Our generic signature therefore has the conformance path graph below; the root abstract conformance is distinguished by a darker shade of gray. Note that `[τ_0_1: Collection]` has a *loop*—an edge with the same source and destination:



The above conformance path graph is finite, but this will not be true in general, and we will investigate infinite conformance path graphs in the next section. What *is* true in general, is that each vertex has a finite set of immediate successors, or in other words, there are only finitely many edges sharing a given source vertex. Thus, just like the type parameter graph, the conformance path graph is *locally finite* (Definition 8.12). An important property of locally finite graphs, which can be shown by induction on path length, is that if we fix $v \in V$ and $n \in \mathbb{N}$, the set of all distinct paths with source vertex v and length n is necessarily finite.

To find a conformance path for an abstract conformance, we perform a breadth-first search of the conformance path graph. We start from our finite set of root vertices, and visit all paths of length n before proceeding to paths of length $n + 1$. We stop when we find a path which represents the equivalence class of the given abstract conformance. In fact, this will always find a path of minimum length. However, there may be more than one path of minimum length. Conformance paths appear in the ABI, as part of the symbol mangling scheme, so we need to make a deterministic choice.

We say that two conformance paths are *equivalent* if they represent equivalent principal abstract conformances—or in other words, if both paths end at the same vertex in the graph. Next, we impose a linear order on conformance paths, similar to the type parameter order of Section 5.5. Shorter paths precede longer paths, while two paths with the same length are ordered by comparing their elements. (Just like the type parameter order, the conformance path order is an instance of a shortlex order, which we define in its full generality in Section 16.5.)

Algorithm 12C (Conformance path order)**.** Takes two conformance paths as input, denoted $x := x_m \otimes \cdots \otimes x_1$ and $y := y_n \otimes \cdots \otimes y_1$, and returns one of "<", ">", or "=".

1. (Shorter) If $n < m$, then $x < y$, so return "<".
2. (Longer) If $n > m$, then $x > y$, so return ">".
3. (Initialize) Otherwise, $n = m$, so we must compare elements. Set $i \leftarrow 1$.
4. (Equal) If $i = n + 1$, our conformance paths are identical. Return "=".
5. (Compare) Viewing x_i and y_i as conformance requirements, compare their subject types and protocols with Algorithm 11D. Return the result if it is not "=".
6. (Next) Otherwise, x_i and y_i are identical. Set $i \leftarrow i + 1$ and go back to Step 4.

While the requirement order of Algorithm 11D is a partial order in general, it is linear on conformance requirements, so Step 5 cannot return "⊥", and the conformance path order is linear as well. The argument used in Proposition 5.25 can also show that the conformance path order is well-founded. Therefore, each equivalence class of conformance paths contains a unique minimum element, or a *reduced conformance path*.

While Algorithm 12C is part of the formal model, the compiler does not implement it directly. Instead, we ensure that our breadth-first search always visits requirements in increasing order at each step, which is enough to guarantee that we always find the reduced conformance path in each equivalence class first. The breadth-first search associates some persistent state with each generic signature G:

- A hash table, mapping reduced abstract conformances to conformance paths.
- An integer N, initially 0.
- A growable array B, of all conformance paths of length N, initially empty.
- A growable array B_1, to temporarily hold all conformance paths of length $N + 1$.

Here is the actual algorithm at last. Each invocation will either immediately return an existing conformance path from the table, or proceed to enumerate conformance paths in increasing order, until the one we are looking for appears in the table.

Algorithm 12D (Find conformance path)**.** Receives a generic signature G and a valid abstract conformance $[\texttt{T: P}]$ as input. Outputs the reduced conformance path for $[\texttt{T: P}]$.

1. (Assert) If $\texttt{requiresProtocol}(G, \texttt{T}, \texttt{P})$ is false, signal an error.
2. (Reduce) Set $\texttt{T} \leftarrow \texttt{getReducedType}(G, \texttt{T})$.
3. (Check) If the table already has a conformance path for $[\texttt{T: P}]$, return it.
4. (Initialize) If $N > 0$, skip this step. Otherwise, add each root abstract conformance of G to B in requirement order (Algorithm 11D), and set $N \leftarrow 1$.
5. (Fill B_1) For each conformance path $p \in B$:
   a) Apply Algorithm 12B to p to get a principal abstract conformance $[\texttt{T}_p\texttt{: P}_p]$.
   b) Set $\texttt{T}_p \leftarrow \texttt{getReducedType}(G, \texttt{T}_p)$ to get a reduced abstract conformance.
   c) If the table already has a conformance path for $[\texttt{T}_p\texttt{: P}_p]$, skip d) and e).
   d) Add the key $[\texttt{T}_p\texttt{: P}_p]$ with value p to the table.
   e) For each associated conformance requirement $\langle\texttt{Self.U: Q}]$ of protocol $\texttt{P}_p$ in requirement order: add $\langle\texttt{Self.U: Q}] \otimes p$ to B_1.
6. (Repeat) Swap $B \leftrightarrow B_1$. Clear B_1. Set $N \leftarrow N + 1$. Go back to Step 3.

In Step 1, we bail out if the given type parameter $\texttt{T}$ does not actually conform to $\texttt{P}$, because then we will never find a conformance path for $[\texttt{T: P}]$. Once we establish the precondition, we can guarantee termination, by Theorem 12.8 together with our choice

of conformance path order. In particular, this means that the B array will never be empty at the start of Step 5, because then we would enter an infinite loop, contradicting our theorem. We don't *have to* show why it cannot be empty! (Let's do it anyway. If B is empty but $N > 0$, our generic signature G has a finite set of conformance paths, and we've already enumerated all of them on a previous invocation. In this case though, Step 3 will always return a value before we get to Step 5.)

Example 12.12. Termination is great, but how large is "finite"? In the worst case, the running time of Algorithm 12D is exponential in the length of the subject type parameter. Consider the protocol generic signature $G_{\texttt{M}}$, with `M` as below:

```
protocol M {
  associatedtype A: M
  associatedtype B: M
}
```

A conformance path in $G_{\texttt{M}}$ consists of $[\texttt{τ_0_0: M}]$ followed by an arbitrary combination of $\langle\texttt{Self.A: M}\rangle$ and $\langle\texttt{Self.B: M}\rangle$. For example, the following conformance path represents $[\texttt{τ_0_0.A.A.B.B: M}]$:

$$\langle\texttt{Self.B: M}\rangle \otimes \langle\texttt{Self.B: M}\rangle \otimes \langle\texttt{Self.A: M}\rangle \otimes \langle\texttt{Self.A: M}\rangle \otimes [\texttt{τ_0_0: M}]$$

Every valid type parameter of $G_{\texttt{M}}$ conforms to `M`, and if `T` is some arbitrary valid type parameter of length n, a moment's thought reveals we can easily construct a conformance path, also of length n, that represents `[T: M]`. However, Algorithm 12D is not as clever, and it just enumerates *all* conformance paths up to some length. Our generic signature has 1 conformance path of length 1, 2 conformance paths of length 2, 4 conformance paths of length 3, and more generally, 2^{n-1} unique conformance paths of length n. Thus, if $|\texttt{T}| = n$, we perform $1 + 2 + \cdots + 2^{n-1} = 2^n - 1$ iterations of Step 5:

```
func callee<T: M>(_: T.Type) {}

func caller<T: M>(_: T.Type) {
  callee(T.A.B.A.B.A.B.A.B.A.B.A.B.A.B.A.B.A.B.A.B.A.B.self)  // slow
}
```

For now at least, we have yet to observe this form of combinatorial explosion in practice. In realistic programs, conformance path graphs tend to be relatively simple, and type parameters tend not to be overly long.

Now, we pause to take a breather. Our implementation of Algorithm 6A is almost completely filled in, except for the `getReducedType`() and `requiresProtocol`() generic signature queries, whose implementation is explained in Part IV. It is worth noting that the pathological protocol `M` shown above is quite interesting for other reasons, and it will reappear in Section 16.3.

12.3. Recursive Conformances

In Section 8.4, we classified generic signatures into three families of increasing complexity. To review, a generic signature may have a finite set of type parameters in all, an infinite set of type parameters that collapse down to a finite set of equivalence classes, or most generally, an infinite set of equivalence classes.

We can restate this in terms of the type parameter graph, as follows. In the first two cases, the type parameter graph is finite. Furthermore, in the first case, the type parameter graph is a directed acyclic graph (Definition 8.3). Indeed, as Example 8.10 shows, we can exhibit an infinite sequence of equivalent type parameters by following a cycle an arbitrary number of times.

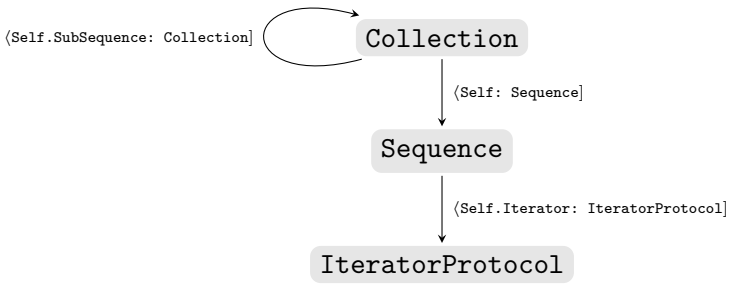
We will now see that we obtain a similar classification of generic signatures if we instead consider the conformance path graph. Our conformance path graph might be finite and acyclic, so we only have finitely many conformance paths, or it might be finite but contain a cycle, so we have a finite set of equivalence classes where at least one contains infinitely many paths, or we might have infinitely many equivalence classes of conformance paths.

It turns out that the two classifications overlap, but do not completely coincide. To understand why, we introduce another directed graph.

Definition 12.13. The *protocol dependency graph* is the directed graph whose vertices are protocol declarations, or more precisely, its vertex set is the universe of all protocol declarations visible to name lookup, which we denoted by Proto in Section 7.1. The edge set consists of all associated conformance requirements from all protocols, such that the source vertex of $[\texttt{Self.U: Q}]_{\texttt{P}}$ is `P` and the destination vertex is `Q`.

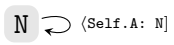
We will encounter the protocol dependency graph again in Section 15.1, when we describe the construction of the requirement machine for a generic signature.

Example 12.14. The universe of protocols Proto is finite, but potentially quite large. However, the edge set is rather sparse, and this graph contains many disjoint *connected components* in practice. If we say that `Collection`, `Sequence`, and `IteratorProtocol` from Listing 12.1 exist in their own little universe, we can draw the protocol dependency graph below. Notice how there's a loop at `Collection`:



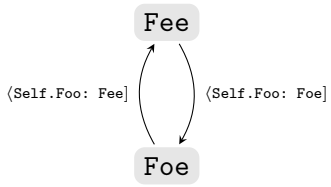
Example 12.15. The protocol dependency graph for protocol `N` from Example 5.12 is just a single vertex with loop:

```
protocol N {
  associatedtype A: N
}
```



More generally, a cycle may involve multiple associated conformance requirements:

```
protocol Fee {
  associatedtype Foo: Foe
}

protocol Foe {
  associatedtype Foo: Fee
}
```



We give a name to such associated conformance requirements:

Definition 12.16. An associated conformance requirement is *recursive* if its edge is part of a cycle in the protocol dependency graph.

In the three preceding examples, all of $[\texttt{Self.SubSequence: Collection}]_{\texttt{Collection}}$, $[\texttt{Self.A: N}]_{\texttt{N}}$, $[\texttt{Self.Foo: Foe}]_{\texttt{Fee}}$, and $[\texttt{Self.Foo: Fee}]_{\texttt{Foe}}$ are all recursive. We recall from Section 8.4 that recursive conformance requirements appeared in Swift 4.1 [105]; prior to this, the protocol dependency graph was always acyclic.

To relate the conformance path graph with the protocol dependency graph, consider the function $\pi\colon \textsc{Conf}(G) \to \textsc{Proto}$:

$$\pi([\texttt{X: P}]) := \texttt{P}$$

This operation is defined on both abstract and concrete conformances. If we consider its restriction to abstract conformances, we see that it maps the vertices of the conformance path graph to the vertices of the protocol dependency graph. Furthermore, if two vertices in the conformance path graph are joined by an edge, the corresponding vertices in the protocol dependency graph are also joined by an edge with the same label.

Definition 12.17. Let $X := (V_X, E_X)$ and $Y := (V_Y, E_Y)$ be directed graphs. A *graph homomorphism* $f\colon X \to Y$ is a pair of functions, $f_v\colon V_X \to V_Y$ for mapping vertices to vertices, and $f_e\colon E_X \to E_Y$ for mapping edges to edges, such that for all $e \in E_X$,

$$f_v(\mathrm{src}(e)) = \mathrm{src}(f_e(e))$$
$$f_v(\mathrm{dst}(e)) = \mathrm{dst}(f_e(e))$$

Note that f_e is permitted to map $\mathrm{src}(e)$ and $\mathrm{dst}(e)$ to the same vertex as long as Y has a loop—an edge with the same source and destination—at this vertex.

We can also apply our graph homomorphism to a path in G_1, by first applying it to the source vertex, and then to each successive edge. By induction, the result is a valid path in G_2. In particular, if we apply π to a path in the conformance path graph, we get a path in the protocol dependency graph. A path in the protocol dependency graph is a sequence of associated conformance projections, so it's like a conformance path but without the first step. The general form of a path from $\texttt{P}_1$ to $\texttt{P}_n$ can be denoted by:

$$\langle \texttt{Self.U}_n\texttt{: P}_n] \otimes \cdots \otimes \langle \texttt{Self.U}_1\texttt{: P}_1]$$

We can now state this interesting fact.

Proposition 12.18. Let G be a generic signature. The following are equivalent:

1. G has an infinite set of conformance paths.
2. There exists a pair of protocols `P` and `Q` (possibly identical), such that the protocol dependency graph has a path from `P` to `Q`, and a cycle of recursive conformance requirements at `Q`. Also, G states an explicit conformance requirement [`T: P`].

Proof. ($1 \Rightarrow 2$) Suppose we are given an infinite set of conformance paths for G. There are only finitely many paths of any fixed length, so our set must contain conformance paths of arbitrary length. If we then choose a conformance path $s_n \otimes \cdots \otimes s_1$ such that $n > |\textsc{Proto}|$, the corresponding path $\pi(s_n \otimes \cdots \otimes s_1)$ in the protocol dependency graph must visit some vertex `Q` more than once. This exhibits a cycle at protocol `Q`. If we then let $\texttt{P} := \pi(s_1)$, we see that $s_n \otimes \cdots s_2$ is the desired path from `P` to `Q` in the protocol dependency graph, and s_1 is the desired explicit conformance requirement [`T: P`].

($2 \Rightarrow 1$) We form a distinct conformance path p_n for each $n \in \mathbb{N}$ as follows. We take the root abstract conformance [`T: P`], and append the associated conformance projections visited along the path from `P` to `Q` in the protocol dependency graph. This gives us a conformance path representing an abstract conformance to `Q`. Then, we append the edges visited by the cycle at `Q`, repeated n times. □

Swift does not allow circular protocol inheritance—which means that every cycle in the protocol dependency graph contains at least one associated conformance requirement whose subject type is not `Self`. In the above proof, every "trip" around the cycle at `Q` therefore increases the length of the subject type parameter by at least one, so our conformance path p_n represents some [`T: Q`] with $|\texttt{T}| \geq n$. If G is well-formed, then this `T` must be a valid type parameter of G. It immediately follows that:

Proposition 12.19. Let G be a well-formed generic signature. If G has an infinite set of conformance paths, then G has an infinite set of valid type parameters.

Therefore, if G has a finite set of valid type parameters, that is, if the type parameter graph of G is finite and acyclic, then the conformance path graph of G must be finite and acyclic as well. However, we cannot strengthen this to an "if and only if," as the following example shows.

Example 12.20. Let G be the generic signature of the protocol extension:

```
protocol Tip { associatedtype Top }
extension Tip where Self == Self.Top {}
```

The type parameter graph of G is finite but not acyclic, because it is a single vertex with a loop labeled "`.Top`", while the conformance path graph of G is finite and acyclic, because it is a single vertex with no edges. We can derive an infinite set of equivalent type parameters: `τ_0_0`, `τ_0_0.Top`, `τ_0_0.Top.Top`, and so on. All valid type parameters of G also conform to `Tip`, but there is only one root abstract conformance [`τ_0_0: Tip`], and no associated conformance requirements, so in fact we only have *one* trivial conformance path, [`τ_0_0: Tip`].

However, if we instead count equivalence classes, we get a stronger result.

Proposition 12.21. Let G be a generic signature. The following are equivalent:

1. G has an infinite set of equivalence classes of type parameters.
2. G has an infinite set of equivalence classes of conformance paths.

Proof. (1 ⇒ 2) We take the reduced type parameter from each equivalence class, and discard the finite subset of generic parameter types, so we're left with an infinite set of reduced dependent member types. Consider the function f that maps a dependent member type `T.[P]A` to the abstract conformance [`T: P`]. Every conformance only has a finite set of type witnesses, so applying f to an infinite set of reduced bound dependent member types must output an infinite set of reduced abstract conformances. We can then find a conformance path for each reduced abstract conformance, to obtain an infinite set of inequivalent conformance paths.

(2 ⇒ 1) We choose an conformance path from each equivalence class, and find the reduced abstract conformance for each one. This gives us an infinite set of reduced abstract conformances. Consider a function g that maps an abstract conformance to its subject type. There can only be finitely many distinct reduced abstract conformances with the same subject type, because there are only finitely many protocols in Proto. Thus, g maps the infinite set of reduced abstract conformances to an infinite set of reduced type parameters. □

To summarize, we can refine our classification of generic signatures into *four* families, by considering the type parameter graph and conformance path graph together:

1. Both graphs are finite and acyclic.
2. Both graphs are finite, but only the conformance path graph is acyclic.
3. Both graphs are finite, and neither is acyclic.
4. Both graphs are infinite.

Here is a classic result by Dénes Kőnig ([119], or Section 2.3.4.3 of [102]):

Theorem 12.22 (Kőnig's infinity lemma)**.** Let (V, E) be a directed graph, with a fixed finite subset $R \subseteq V$ of root vertices. Suppose that the following holds:

1. The vertex set V is infinite.
2. The graph is locally finite, so for each $v \in V$, there are only finitely many $e \in E$ with $\mathrm{src}(e) = v$.
3. Every vertex is reachable from at least one root, so for each $v \in V$ there exists a path p with $\mathrm{src}(p) \in R$ and $\mathrm{dst}(p) = v$.

Under these assumptions, V contains an infinite sequence of distinct vertices v_1, v_2, ... such that each consecutive pair of vertices v_j and v_{j+1} is joined by an edge in E.

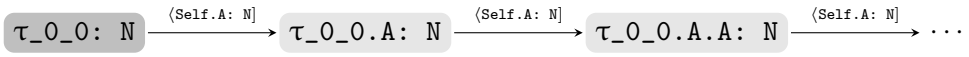
Proof. We introduce some notation first. A *simple* path is a path that does not visit the same vertex more than once. A path q is said to *extend* a path p if $\mathrm{src}(p) = \mathrm{src}(q)$, and the sequence of edges visited by p is a prefix of the edges visited by q. We then define $S(p)$ to be the set of all simple paths that extend a given simple path p. Also, recall our notation for an empty path.

If $r \in R$, then $S(1_r)$ is just the set of all vertices reachable from r. By assumption, every $v \in V$ is reachable from some $r \in R$, but R is finite while V is infinite, so for at least one $r \in R$, the set $S(1_r)$ is infinite. We set $p_1 := 1_r$ and $v_1 := r$.

Now, suppose we have a simple path p_i which visits vertices v_1, ..., v_i, such that $S(p_i)$ is infinite. We consider the set of edges $e \in E$ with $\mathrm{src}(e) = \mathrm{dst}(p_i)$. This set is finite because our graph is locally finite, so at least one such edge e has the property that $S(p_i \circ e)$ is infinite. We set $p_{i+1} := p_i \circ e$ and $v_{i+1} := \mathrm{dst}(e)$.

We can repeat this to generate paths of any length, so the conclusion follows. □

So if we have an infinite generic signature, we can find a *ray*, or an infinite path, in each of our two graphs associated with this generic signature. Let's look at protocol `N`, where the ray is the entire graph. We already saw the type parameter graph of $G_{\texttt{N}}$ in Example 8.8. The conformance path graph of $G_{\texttt{N}}$ is identical, except for labels:



Now, suppose we have a protocol substitution map $\Sigma_{[\texttt{X: N}]}$ for some conformance `[X: N]`. We can imagine applying this substitution map to each vertex in the conformance path graph of $G_{\texttt{N}}$, to produce an infinite sequence of conformances to `N`:

$$[\texttt{τ_0_0: N}] \otimes \Sigma_{[\texttt{X: N}]} = [\texttt{X: N}]$$
$$[\texttt{τ_0_0.A: N}] \otimes \Sigma_{[\texttt{X: N}]} = \langle\texttt{Self.A: N}] \otimes [\texttt{X: N}]$$
$$[\texttt{τ_0_0.A.A: N}] \otimes \Sigma_{[\texttt{X: N}]} = \langle\texttt{Self.A: N}] \otimes \langle\texttt{Self.A: N}] \otimes [\texttt{X: N}]$$
...

Just like abstract conformances, concrete conformances are also related by associated conformance projection. In fact, the set of substituted conformances that we obtain by applying a substitution map to each vertex in the conformance path graph, can also be equipped with the structure of a directed graph.

Conformance substitution graph. As always, G and H are generic signatures in what follows, and $\Sigma \in \textsc{Sub}(G, H)$ is a substitution map. First, we define an equivalence relation on $\textsc{Conf}(H)$; two conformances are equivalent if they name the same protocol, and their conforming types are equivalent under the reduced type equality relation on $\textsc{Type}(H)$. Now, if we apply Σ to each vertex in the conformance path graph of G, we see that each vertex maps to an equivalence class of conformances in $\textsc{Conf}(H)$. This mapping respects associated conformance projection, as we saw in Section 7.5.

Definition 12.23. The *conformance substitution graph* of Σ is the directed graph defined as follows:

- The vertices are the equivalence classes of $[\texttt{T: P}] \otimes \Sigma$ under the above relation, where $[\texttt{T: P}]$ ranges over all abstract conformances of G.
- The edge relation is associated conformance projection.

With this construction, local conformance lookup becomes a graph homomorphism from the conformance path graph of G to the conformance substitution graph of Σ. (The conformance substitution graph also explains the instantiation of witness tables in the Swift runtime. A generic function receives a witness table for each explicit conformance requirement of its generic signature to start. Witness tables then refer to the witness tables for their associated conformances; this is lazily populated with a runtime entry point, so we explore some finite subgraph of the conformance substitution graph.)

We now ask, if $[\texttt{X: N}]$ is a conformance to `N`, what does the conformance substitution graph of $\Sigma_{[\texttt{X: N}]}$ look like? Another classic result from graph theory tells us that the image of a ray under graph homomorphism must be one the following:

1. A cycle.
2. Another infinite ray.
3. A finite path leading to a cycle.

Therefore, if start with $[\texttt{X: N}]$ and repeatedly apply $\langle \texttt{Self.A: N}]$, one of three things will eventually happen:

1. We end up back at $[\texttt{X: N}]$.
2. We produce an infinite sequence of distinct conformances to `N`.
3. We end up at a conformance we've already seen, but not $[\texttt{X: N}]$.

We will now construct an example of each behavior, followed by a surprise.

Recursive normal conformances. We can form a cycle out of normal conformances:

```
struct Toe: N { typealias A = Tie }
struct Tie: N { typealias A = Pie }
struct Pie: N { typealias A = Poe }
struct Poe: N { typealias A = Toe }
```

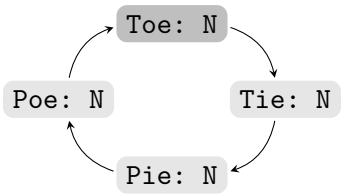
We note that:

$$\langle \texttt{Self.A: N} \rangle \otimes [\texttt{Toe: N}] = [\texttt{Tie: N}] \qquad \langle \texttt{Self.A: N} \rangle \otimes [\texttt{Pie: N}] = [\texttt{Poe: N}]$$
$$\langle \texttt{Self.A: N} \rangle \otimes [\texttt{Tie: N}] = [\texttt{Pie: N}] \qquad \langle \texttt{Self.A: N} \rangle \otimes [\texttt{Poe: N}] = [\texttt{Toe: N}]$$

In particular:

$$\langle \texttt{Self.A: N} \rangle \otimes \langle \texttt{Self.A: N} \rangle \otimes \langle \texttt{Self.A: N} \rangle \otimes \langle \texttt{Self.A: N} \rangle \otimes [\texttt{Toe: N}] = [\texttt{Toe: N}]$$

Here is the conformance substitution graph for $\Sigma_{[\texttt{Toe: N}]}$:



Recursive specialized conformances. To get an infinite conformance substitution graph, we start with a generic nominal type conforming to `N`:

```
struct G<T>: N {
  typealias A = G<G<T>>
}
```

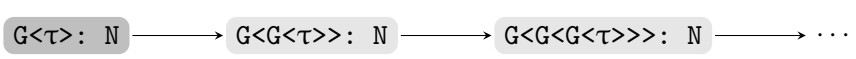
Here, the associated conformance requirement is witnessed by a specialized conformance. Let's denote `τ_0_0` by just τ, so if we let $\Sigma := \{\tau \mapsto \texttt{G<τ>}\}$, we see that:

$$\langle \texttt{Self.A: N} \rangle \otimes [\texttt{G<}\tau\texttt{>: N}] = [\texttt{G<G<}\tau\texttt{>>: N}] = [\texttt{G<}\tau\texttt{>: N}] \otimes \Sigma$$

More generally, we get an infinite sequence of distinct specialized conformances:

$$\underbrace{\langle \texttt{Self.A: N} \rangle \otimes \cdots \otimes \langle \texttt{Self.A: N} \rangle}_{n \text{ times}} \otimes [\texttt{G<}\tau\texttt{>: N}] = [\texttt{G<}\tau\texttt{>: N}] \otimes \underbrace{\Sigma \otimes \cdots \otimes \Sigma}_{n \text{ times}}$$

Here is the conformance substitution graph for $\Sigma_{[\texttt{G<}\tau\texttt{>: N}]}$:



Recursive abstract conformances. Here we have a generic nominal type where the associated conformance is witnessed by an abstract conformance:

```
struct H<T: N>: N {
  typealias A = T
}
```

Once again, it is convenient to write τ instead of τ_0_0. Notice that:

$$\langle \texttt{Self.A: N} \rangle \otimes [\texttt{H<}\tau\texttt{>: N}] = [\tau\texttt{: N}]$$

Now, we take the nominal types `Toe`, `Tie`, `Pie`, and `Poe` from before, and consider the specialized conformance `[H<H<Toe>>: N]`. We give a name to two substitution maps, since they will appear multiple times below:

$$\Sigma_{\texttt{H}} := \{\tau \mapsto \texttt{H<}\tau\texttt{>}; [\tau\texttt{: N}] \mapsto [\texttt{H<}\tau\texttt{>: N}]\}$$
$$\Sigma_{\texttt{Toe}} := \{\tau \mapsto \texttt{Toe}; [\tau\texttt{: N}] \mapsto [\texttt{Toe: N}]\}$$

Using the above notation, we have:

$$[\texttt{H<H<Toe>>: N}] = [\texttt{H<}\tau\texttt{>: N}] \otimes \Sigma_{\texttt{H}} \otimes \Sigma_{\texttt{Toe}}$$

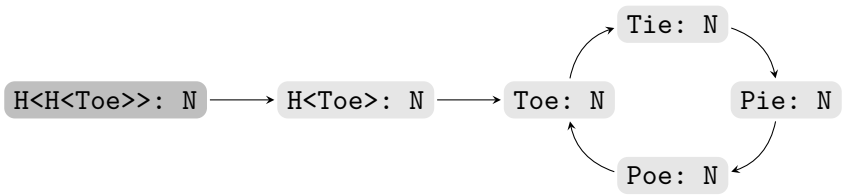
We can calculate $\langle \texttt{Self.A: N} \rangle \otimes [\texttt{H<H<Toe>>: N}]$:

$$\begin{aligned}
&\langle \texttt{Self.A: N} \rangle \otimes [\texttt{H<H<Toe>>: N}] \\
&\quad = \langle \texttt{Self.A: N} \rangle \otimes [\texttt{H<}\tau\texttt{>: N}] \otimes \Sigma_{\texttt{H}} \otimes \Sigma_{\texttt{Toe}} \\
&\quad = [\tau\texttt{: N}] \otimes \Sigma_{\texttt{H}} \otimes \Sigma_{\texttt{Toe}} \\
&\quad = [\texttt{H<}\tau\texttt{>: N}] \otimes \Sigma_{\texttt{Toe}} \\
&\quad = [\texttt{H<Toe>: N}]
\end{aligned}$$

If we project the associated conformance again, we get:

$$\begin{aligned}
&\langle \texttt{Self.A: N} \rangle \otimes [\texttt{H<Toe>: N}] \\
&\quad = \langle \texttt{Self.A: N} \rangle \otimes [\texttt{H<}\tau\texttt{>: N}] \otimes \Sigma_{\texttt{Toe}} \\
&\quad = [\tau\texttt{: N}] \otimes \Sigma_{\texttt{Toe}} \\
&\quad = [\texttt{Toe: N}]
\end{aligned}$$

Here is the conformance substitution graph for $\Sigma_{[\texttt{H<H<Toe>>: N}]}$:



Non-terminating substitutions. This example demonstrates a new behavior:

```
struct S: N {
  typealias A = F<S>
}

struct F<T: N>: N {
  typealias A = T.A.A
}
```

We define these two substitution maps:

$$\Sigma_{\texttt{S}} := \{\tau \mapsto \texttt{S};\ [\tau\texttt{: N}] \mapsto [\texttt{S: N}]\}$$
$$\Sigma_{\texttt{F}} := \{\tau \mapsto \texttt{F<}\tau\texttt{>};\ [\tau\texttt{: N}] \mapsto [\texttt{F<}\tau\texttt{>: N}]\}$$

We can express the associated conformance of $[\texttt{S: N}]$ in terms of $\Sigma_{\texttt{S}}$:

$$\langle\texttt{Self.A: N}] \otimes [\texttt{S: N}] = [\texttt{F<}\tau\texttt{>: N}] \otimes \Sigma_{\texttt{S}} \tag{1}$$

We also write the type witness of $[\texttt{F<}\tau\texttt{>: N}]$ using a conformance path:

$$\langle\texttt{N|A} \otimes [\texttt{F<}\tau\texttt{>: N}] = \tau\texttt{.A.A} = \langle\texttt{N|A} \otimes \langle\texttt{Self.A: N}] \otimes [\tau\texttt{: N}] \tag{2}$$

We will now attempt to calculate the substituted return type of the below call to `f()`:

```
func f<T: N>(_: T) -> T.A {...}

let value = f(F<S>())  // What is the type of ‘value’?
```

The substitution map for the call is $\Sigma_{\texttt{F}} \otimes \Sigma_{\texttt{S}} = \{\tau \mapsto \texttt{F<S>};\ [\tau\texttt{: N}] \mapsto [\texttt{F<S>: N}]\}$, and the dependent member type being substituted is $\tau\texttt{.A}$. We begin by expanding the dependent member type, and applying the identity $[\tau\texttt{: N}] \otimes \Sigma_{\texttt{F}} = [\texttt{F<}\tau\texttt{>: N}]$:

$$\begin{aligned}\tau\texttt{.A} \otimes \Sigma_{\texttt{F}} \otimes \Sigma_{\texttt{S}} &= \langle\texttt{N|A} \otimes [\tau\texttt{: N}] \otimes \Sigma_{\texttt{F}} \otimes \Sigma_{\texttt{S}} \\ &= \langle\texttt{N|A} \otimes [\texttt{F<}\tau\texttt{>: N}] \otimes \Sigma_{\texttt{S}}\end{aligned} \tag{3}$$

We can go further by applying (2), followed by the identity $[\tau\texttt{: N}] \otimes \Sigma_{\texttt{S}} = [\texttt{S: N}]$:

$$\begin{aligned}&= \langle\texttt{N|A} \otimes \langle\texttt{Self.A: N}] \otimes [\tau\texttt{: N}] \otimes \Sigma_{\texttt{S}} \\ &= \langle\texttt{N|A} \otimes \langle\texttt{Self.A: N}] \otimes [\texttt{S: N}]\end{aligned}$$

Unfortunately, our only remaining move is to apply (1), but that brings us back to (3):

$$= \langle\texttt{N|A} \otimes [\texttt{F<}\tau\texttt{>: N}] \otimes \Sigma_{\texttt{S}}$$

Normal forms. We've been making the tacit assumption that every valid expression written using the $\otimes$ "turntable" operator can be reduced to a *normal form*—a single type, substitution map, or conformance—by the finite application of algebraic identities, until no occurrences of $\otimes$ remain. This turns out to be a false assumption. In the prior example, the record keeps spinning and the party doesn't stop: $\otimes \quad \oplus \quad \otimes \quad \oplus \quad \otimes \quad \cdots$

We showed that Algorithm 12D for *finding* a conformance path always terminates, but we clearly cannot say the same about Algorithm 12A for *evaluating* a conformance path! Local conformance lookup—or more generally, the $\otimes$ operator—is actually a *partial* function, which does not always produce a result. Similarly, the substitution map Σ_S described above does not *have* a conformance substitution graph.

At the present time, the compiler terminates with a stack overflow if a non-terminating type substitution is attempted. While Algorithm 12A itself is a simple counted loop, the recursion is coming from Step 4. If C is a specialized conformance, projecting an associated conformance from C applies a substitution map to some other conformance; if this conformance is abstract, we recursively call into local conformance lookup again. Our example happens to arrange everything just right so that we end up repeating the *same* local conformance lookup.

12.4. The Halting Problem

In the future, the compiler should detect non-terminating type substitution and diagnose the problem, instead of crashing. Two approaches spring to mind:

1. We could impose a maximum operation count on every single type substitution, terminating with an error if this count is exceeded.
2. We could try to detect such behavior ahead of time, by carefully analyzing all substitution maps and conformances in the program, and rejecting those that encode non-terminating computation.

It turns out that we cannot hope to do better than (1) above in general. Before we can understand why, we first recall some computability theory. A classic text for the below is [120], while [95] has a gentler introduction.

Computable numbers. In 1937, Alan Turing set out to formalize what is meant by a "computable number," in the sense that $1/3$, $(1 + \sqrt{5})/2$, and $\pi := 3.14159\ldots$ are computable [121]. If we start from the observation that every integer is "inherently" computable by virtue of being representable by a finite sequence of binary digits, then it remains to define the computability of real numbers in the interval $(0, 1)$. The binary representation of such a number consists of the radix point, followed by an infinite sequence of "0" and "1" digits; if we can write down an *effective procedure* for generating this sequence of digits to any degree of precision, the number is computable.

To describe such an effective procedure, Turing introduced what we now call the *Turing machine* formalism. The precise definition can be found in the literature, but for our purposes, a Turing machine is the following:

1. The fixed *machine description* consists of a finite set of *symbols* ("0" and "1" will do), a finite set of *states* one of which is the *initial state*, and a table mapping (state, symbol) keys to (state, symbol, direction) values.

2. The Turing machine's *head* reads and writes symbols on a *tape* of consecutive storage locations, where each location either contains a symbol, or a blank space. The machine has a register to hold its *current state*, which will be the initial state at the start of execution.

3. At each step, the machine reads the symbol at the position of the head, and looks up a key, formed from this symbol together with the current state, in the transition table. The value found in the transition table determines the next state, a new symbol to write at the head, and the direction to move the head, which can only move by one position to the left or right.

To establish that π is computable, for example, we could exhibit a Turing machine that starts executing with a blank tape and proceeds to write consecutive binary digits of π to the tape forever. If we are willing to wait long enough, we can use this Turing machine to generate an approximation of π to any level of precision we desire.

An important point is that even if our number has a terminating binary expansion, such as 1/2 for example, we still want a Turing machine that will continue to output digits forever, in this case by producing an infinite sequence of "0" at the end. We must not represent a computable number by a Turing machine that eventually stops generating digits, because then we could not determine if the binary expansion has truly ended, or if we just need to let the machine run for a while longer, so that it can output another digit. Turing said that a machine is "circle-free" if it continues to output binary digits forever. A "circular" machine is one that is not circle-free, so it gets "stuck" after it outputs some finite number of binary digits.

A natural question arises: is there an effective procedure to determine if a given Turing machine is circle-free? We make this precise as follows. Instead of starting with a blank tape, a Turing machine can more generally receive an input string, which we imagine is already on the tape when execution begins. To pass a Turing machine to another Turing machine as input, we define the *standard description* of a Turing machine, which encodes its complete state as a finite string of symbols. We say that a string is *satisfactory* if it represents the standard description of some circle-free machine, so we're looking for a circle-free Turing machine which outputs "1", "1", ... if the input string is satisfactory, and "0", "0", ... otherwise. Turing showed that this is impossible.

Theorem 12.24. There is no circle-free Turing machine that can decide if a given input string is satisfactory.

The Church-Turing thesis. Today, we formulate Turing's result in a different but equivalent way. Instead of circle-free machines, we talk about the *halting problem*, which asks if a Turing machine eventually reaches some fixed final "halting state":

Theorem 12.25. There is no effective procedure to decide if a given Turing machine halts in a finite number of steps.

Turing's result also applies to any formalism that can *simulate* a Turing machine, of which there are many. Concurrently with Turing's work, Alonzo Church introduced the *lambda calculus*, and was able to prove a similar undecidability result [122]. Lambda calculus is *Turing-complete*, meaning that every lambda-computable function can be computed by Turing machine, and vice versa. In fact, this happens to be the case with every formalization of "effective procedure" or "computable function" discovered so far. The *Church-Turing thesis* is the observation that every sufficiently general form of computation is equal in expressive power to the Turing machine.

The Swift programming language is Turing-complete, because we can simulate the execution of a Turing machine with a Swift program, and at least in theory, we can simulate the execution of any Swift program with a Turing machine. We will now give an informal proof of Theorem 12.25, but using Swift, instead of Turing machines.

Proof. We imagine a Swift dialect without side effects or non-determinism of any kind, but with a new introspection facility, that converts an arbitrary function value into its complete source code, or syntax tree, or whatever. Using this facility, we might try to implement a `halts()` function, that takes another function as input, performs an arbitrary but finite amount of computation, and returns a boolean that tells us if the input function would halt or not if called:

```
// Return true if f() would halt, or false if f() would run forever.
func halts(_ f: () -> ()) -> Bool {...}
```

Now, consider what happens when we run this function:

```
func naughty() {
  if halts(naughty) { while true {} }  // I won't do what you tell me!
}
```

If `halts(naughty)` were to return true, then `naughty()` would enter an infinite loop and not halt, thus `halts(naughty)` cannot be true. If `halts(naughty)` were to return false, then `naughty()` actually does halt, so `halts(naughty)` seemingly cannot be false, either. The only remaining possibility is that `halts(naughty)` runs forever and does not return anything, but this contradicts our assumption that `halts()` itself can make a decision in a finite number of steps. Every outcome leads to a contradiction, so a correct implementation of `halts()` cannot exist—the halting problem is *undecidable*. □

While this is not a rigorous proof, the key intuition is that once our formal system is sufficiently expressive, we can always "outwit" any purported termination checking algorithm, no matter how sophisticated, by encoding the termination checker itself within our formalism, and doing the opposite of what it tells us to do.

Tag systems. We will now demonstrate that the Swift type substitution algebra is also Turing-complete, so a Swift program can ask the compiler to perform arbitrary computation at *compile time*. Rather than attempting a direct encoding of a Turing machine, we consider *tag systems*, a computational formalism first introduced in 1943 by Emil L. Post [123].

Definition 12.26. A tag system consists of the following:

- A finite alphabet of symbols.
- A fixed $n \in \mathbb{N}$, called the *deletion number*. (We can assume that $n > 1$, because otherwise the tag system is pretty trivial.)
- A table mapping each symbol to a finite string of symbols. This string is referred to as the *production* for this symbol.

Like a Turing machine, a tag system operates on a tape of symbols, but with a simpler evaluation rule. We imagine there are *two* heads, both moving in the same direction. The first head reads one symbol at a time and communicates this symbol to the second head, which then writes zero or more symbols to the tape by following the corresponding production rule. The deletion number then tells the read head how many positions to skip. In its initial state, the machine positions its read head at the start of the input string, and the write head at the end. Evaluation stops if the read head ever catches up to the write head, otherwise the chase continues forever.

Algorithm 12E (Evaluate tag system)**.** Takes a table of production rules, a deletion number n, and a string as input. Repeatedly modifies the input string.

1. If the length of the string is less than the deletion number n, halt.
2. Read a symbol from the start of the string.
3. Write the production for this symbol at the end of the string.
4. Delete n symbols from the start of the string.
5. Go back to Step 1.

The above is easy to implement in any Turing-complete programming language. What is far less obvious is that we can go in the other direction, and translate an arbitrary Turing machine into a tag system; this was only established in 1961 (see [124], or [125]). This means that tag systems are Turing-complete, thus their halting problem is also undecidable. Now, let's look at a non-trivial tag system.

The Collatz conjecture. The *Collatz conjecture*, named after Lothar Collatz, is a famous unsolved problem in number theory, regarding this function $f\colon \mathbb{N} \to \mathbb{N}$:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

Starting with any $n \in \mathbb{N}$, we can construct an infinite sequence of natural numbers, by repeated application of f to n. This is called the *Collatz sequence* for n:

$$n,\ f(n),\ f(f(n)),\ f(f(f(n))),\ \ldots$$

For example, if we take $n = 3$, then we get the following Collatz sequence:

$$3,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1,\ 4,\ 2,\ 1,\ \ldots$$

If the sequence reaches 1, the rest of the sequence cycles through 1, 4, 2, 1 forever. Except for the trivial case where $n = 0$, every known Collatz sequence eventually reaches 1, but the question of whether this holds in general remains open. A remarkable fact is that the Collatz sequence can be computed by this tag system from [126]:

Theorem 12.27. Consider the tag system with alphabet $\{a, b, c\}$, deletion number 2, and the following production rules:

$$a \mapsto bc$$
$$b \mapsto a$$
$$c \mapsto aaa$$

If $n \geq 1$, then evaluating the above tag system with the input string consisting of "a" repeated n times, will compute the Collatz sequence for n, the intermediate states where all symbols are "a" being the elements. Evaluation halts if the sequence reaches 1.

Example 12.28. Let's evaluate our tag system with "$aaaa$" to compute the (very short) Collatz sequence for 4. We read a symbol $\Rightarrow$ write the production $\Rightarrow$ delete 2 symbols:

$$\underline{a}aaa \Rightarrow aaaa\underline{bc} \Rightarrow aabc \tag{4}$$
$$\underline{a}abc \Rightarrow aabc\underline{bc} \Rightarrow bcbc$$
$$\underline{b}cbc \Rightarrow bcbc\underline{a} \Rightarrow bca$$
$$\underline{b}ca \Rightarrow bca\underline{a} \Rightarrow aa$$
$$\underline{a}a \Rightarrow aa\underline{bc} \Rightarrow bc \tag{2}$$
$$\underline{b}c \Rightarrow bc\underline{a} \Rightarrow a$$
$$a \tag{1}$$

The string is all a in the steps labeled (4), (2), and (1); these are the elements of the Collatz sequence for 4. Once we reach 1, evaluation stops, because the length of "a" is smaller than the deletion number of 2. If we start with "aaa" instead, it takes many more steps to halt, and one of the intermediate steps is "$aaaaaaaaaaaaaaaa$".

Swift type substitution. We will now encode the Collatz tag system as a series of protocol conformances, and then write down a substitution map, that when applied to a certain dependent member type, will evaluate the Collatz sequence for a given $n \geq 1$. The same encoding generalizes easily to encode any tag system.

We begin with a protocol named `Tag`. We add one associated type for each symbol in our alphabet, and state a recursive conformance requirement to `Tag` on each:

```
protocol Tag {
  associatedtype A: Tag
  associatedtype B: Tag
  associatedtype C: Tag
```

We also add two more associated types, which we will describe shortly; the first one again states a recursive conformance requirement:

```
  associatedtype Del: Tag
  associatedtype Next
}
```

Finally, we have the nominal types `AA`, `BB`, `CC`, `End`, and `Halt`, shown in Listing 12.2, which all conform to `Tag`. The `Tag` protocol is straightforward, so all the action happens in these concrete conformances to `Tag`. In the following, we will be working with the protocol generic signature $G_{\texttt{Tag}}$, and we denote its generic parameter `τ_0_0` by τ.

Definition 12.29. We encode strings over the alphabet $\{a, b, c\}$ as compositions of the below substitution maps:

$$\Sigma_{\texttt{AA}} := \{\tau \mapsto \texttt{AA<}\tau\texttt{>};\ [\tau\texttt{: Tag}] \mapsto [\texttt{AA<}\tau\texttt{>: Tag}]\}$$
$$\Sigma_{\texttt{BB}} := \{\tau \mapsto \texttt{BB<}\tau\texttt{>};\ [\tau\texttt{: Tag}] \mapsto [\texttt{BB<}\tau\texttt{>: Tag}]\}$$
$$\Sigma_{\texttt{CC}} := \{\tau \mapsto \texttt{CC<}\tau\texttt{>};\ [\tau\texttt{: Tag}] \mapsto [\texttt{CC<}\tau\texttt{>: Tag}]\}$$
$$\Sigma_{\texttt{End}} := \{\tau \mapsto \texttt{End};\ [\tau\texttt{: Tag}] \mapsto [\texttt{End: Tag}]\}$$

We map "a" to $\Sigma_{\texttt{AA}}$, "b" to $\Sigma_{\texttt{BB}}$, "c" to $\Sigma_{\texttt{CC}}$, and then append $\Sigma_{\texttt{End}}$ at the end; finally, we insert "$\otimes$" in between each pair of substitution maps.

Example 12.30. The empty string "" becomes $\Sigma_{\texttt{End}}$, the string "b" becomes $\Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{End}}$, while "$abac$" becomes $\Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{End}}$. If we apply each composition to the generic parameter type τ, we get a concrete type that encodes the string, built up from `End`, `AA<>`, `BB<>`, and `CC<>`:

$$\tau \otimes \Sigma_{\texttt{End}} = \texttt{End}$$
$$\tau \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{End}} = \texttt{BB<End>}$$
$$\tau \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{End}} = \texttt{AA<BB<AA<CC<End>>>>}$$

Listing 12.2.: Concrete types for the Collatz tag system

```
struct AA<T: Tag>: Tag {
  typealias A = AA<T.A>
  typealias B = AA<T.B>
  typealias C = AA<T.C>
  typealias Del = T
  typealias Next = Del.Del.B.C.Next
}

struct BB<T: Tag>: Tag {
  typealias A = BB<T.A>
  typealias B = BB<T.B>
  typealias C = BB<T.C>
  typealias Del = T
  typealias Next = Del.Del.A.Next
}

struct CC<T: Tag>: Tag {
  typealias A = CC<T.A>
  typealias B = CC<T.B>
  typealias C = CC<T.C>
  typealias Del = T
  typealias Next = Del.Del.A.A.A.Next
}

struct End: Tag {
  typealias A = AA<End>
  typealias B = BB<End>
  typealias C = CC<End>
  typealias Del = Halt
  typealias Next = Halt
}

struct Halt: Tag {
  typealias A = Halt
  typealias B = Halt
  typealias C = Halt
  typealias Del = Halt
  typealias Next = Halt
}
```

We will now prove a series of lemmas, to explain how to delete symbols from the start of the string, how to detect the halting state, how to append symbols at the end of the string, and finally, how to encode the production rules. Each lemma has the form of an identity involving substitution map composition.

We start with deletion of symbols from the start of the string. We denote the identity substitution map of $G_{\texttt{Tag}}$ by 1, and define $\Sigma_{\texttt{Del}}$ as follows:

$$1 := \{\tau \mapsto \tau;\ [\tau\texttt{: Tag}] \mapsto [\tau\texttt{: Tag}]\}$$
$$\Sigma_{\texttt{Del}} := \{\tau \mapsto \tau\texttt{.Del};\ [\tau\texttt{: Tag}] \mapsto [\tau\texttt{.Del: Tag}]\}$$

Lemma 12.31. If Σ represents a non-empty string, then $\Sigma_{\texttt{Del}} \otimes \Sigma$ represents the string obtained by deleting the first symbol of Σ.

Proof. Since substitution map composition is associative, it suffices to show that:

$$\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{AA}} = 1$$
$$\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{BB}} = 1$$
$$\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{CC}} = 1$$

From Listing 12.2, we see that applying $\langle\texttt{Tag|Del}$ and $\langle\texttt{Self.Del: Tag}]$ to $[\texttt{AA<}\tau\texttt{>: Tag}]$ produces the following:

$$\langle\texttt{Tag|Del} \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = \tau$$
$$\langle\texttt{Self.Del: Tag}] \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = [\tau\texttt{: Tag}]$$

Using the above, we apply $\Sigma_{\texttt{AA}}$ to the constituents of $\Sigma_{\texttt{Del}}$:

$$\tau\texttt{.Del} \otimes \Sigma_{\texttt{AA}} = \langle\texttt{Tag|Del} \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = \tau$$
$$[\tau\texttt{.Del: Tag}] \otimes \Sigma_{\texttt{AA}} = \langle\texttt{Self.Del: Tag}] \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = [\tau\texttt{: Tag}]$$

Notice that we get τ and $[\tau\texttt{: Tag}]$, which are the constituents of the identity substitution map of $G_{\texttt{Tag}}$, so $\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{AA}} = 1$. The other two identities are similar. □

Example 12.32. Deleting the first symbol from "*abac*" gives us "*bac*":

$$\begin{aligned}\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{End}} \\ = 1 \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{End}} \\ = \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{End}}\end{aligned}$$

Attempting to delete a symbol from the empty string takes us to the halting state.

Lemma 12.33. Let $\Sigma_{\texttt{Halt}} := \{\tau \mapsto \texttt{Halt};\ [\tau\texttt{: Tag}] \mapsto [\texttt{Halt: Tag}]\}$. Then:

$$\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{End}} = \Sigma_{\texttt{Halt}}$$
$$\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{Halt}} = \Sigma_{\texttt{Halt}}$$

Proof. Looking at the declaration of `End` in Listing 12.2, we see that:

$$\langle \texttt{Tag}|\texttt{Del} \otimes [\texttt{End: Tag}] = \texttt{Halt}$$
$$\langle \texttt{Self.Del: Tag}\rangle \otimes [\texttt{End: Tag}] = [\texttt{Halt: Tag}]$$

Using the above, we apply $\Sigma_{\texttt{Del}}$ to each constituent of $\Sigma_{\texttt{End}}$:

$$\tau\texttt{.Del} \otimes \Sigma_{\texttt{End}} = \langle \texttt{Tag}|\texttt{Del} \otimes [\texttt{End: Tag}] = \texttt{Halt}$$
$$[\tau\texttt{.Del: Tag}] \otimes \Sigma_{\texttt{End}} = \langle \texttt{Self.Del: Tag}\rangle \otimes [\texttt{End: Tag}] = [\texttt{Halt: Tag}]$$

These are the constituents of $\Sigma_{\texttt{Halt}}$, so $\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{End}} = \Sigma_{\texttt{Halt}}$. To get the second identity, note that:

$$\langle \texttt{Tag}|\texttt{Del} \otimes [\texttt{Halt: Tag}] = \texttt{Halt}$$
$$\langle \texttt{Self.Del: Tag}\rangle \otimes [\texttt{Halt: Tag}] = [\texttt{Halt: Tag}]$$

It follows that $\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{Halt}} = \Sigma_{\texttt{Halt}}$. □

We also need to append symbols at the end of a string.

Lemma 12.34. Given the following substitution maps:

$$\Sigma_{\texttt{A}} := \{\tau \mapsto \tau\texttt{.A}; [\texttt{Tag: Tag}] \mapsto [\tau\texttt{.A: Tag}]\}$$
$$\Sigma_{\texttt{B}} := \{\tau \mapsto \tau\texttt{.B}; [\texttt{Tag: Tag}] \mapsto [\tau\texttt{.B: Tag}]\}$$
$$\Sigma_{\texttt{C}} := \{\tau \mapsto \tau\texttt{.C}; [\texttt{Tag: Tag}] \mapsto [\tau\texttt{.C: Tag}]\}$$

We have:

$$\Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{End}} = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{End}}$$
$$\Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{End}} = \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{End}}$$
$$\Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{End}} = \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{End}}$$

Proof. First, consider $\Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{End}}$; we're adding the symbol "a" to the empty string "". Looking at the declaration of `End` in Listing 12.2, we see that we can express the result of applying $\langle \texttt{Tag}|\texttt{A}$ and $\langle \texttt{Self.A: Tag}\rangle$ to $[\texttt{End: Tag}]$ in terms of $\Sigma_{\texttt{End}}$:

$$\langle \texttt{Tag}|\texttt{A} \otimes [\texttt{End: Tag}] = \texttt{AA<}\tau\texttt{>} \otimes \Sigma_{\texttt{End}}$$
$$\langle \texttt{Self.A: Tag}\rangle \otimes [\texttt{End: Tag}] = [\texttt{AA<}\tau\texttt{>: Tag}] \otimes \Sigma_{\texttt{End}}$$

Now, we apply $\Sigma_{\texttt{End}}$ to the constituents of $\Sigma_{\texttt{A}}$:

$$\tau\texttt{.A} \otimes \Sigma_{\texttt{End}} = \langle \texttt{Tag}|\texttt{A} \otimes [\texttt{End: Tag}] = \texttt{AA<}\tau\texttt{>} \otimes \Sigma_{\texttt{End}}$$
$$[\tau\texttt{.A: Tag}] \otimes \Sigma_{\texttt{End}} = \langle \texttt{Self.A: Tag}\rangle \otimes [\texttt{End: Tag}] = [\texttt{AA<}\tau\texttt{>: Tag}] \otimes \Sigma_{\texttt{End}}$$

We see that $\Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{End}} = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{End}}$, and the other cases are similar. □

Appending symbols to a non-empty string is more interesting. Composing our string on the *left* with $\Sigma_{\texttt{A}}$, $\Sigma_{\texttt{B}}$, or $\Sigma_{\texttt{C}}$ has the effect of inserting a $\Sigma_{\texttt{AA}}$, $\Sigma_{\texttt{BB}}$, or $\Sigma_{\texttt{CC}}$ on the *right*. This is a consequence of the fact that our "single letter" substitution maps *commute* with the "double letter" ones.

Lemma 12.35. Let $\Sigma_{\texttt{A}}$, $\Sigma_{\texttt{B}}$, and $\Sigma_{\texttt{C}}$ be as in Lemma 12.34. We have:

$$\begin{aligned}
&\Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{AA}} = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{A}} \qquad \Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{AA}} = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{B}} \qquad \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{AA}} = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{C}} \\
&\Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{BB}} = \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{A}} \qquad \Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{BB}} = \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{B}} \qquad \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{BB}} = \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{C}} \\
&\Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{CC}} = \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{A}} \qquad \Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{CC}} = \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{B}} \qquad \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{CC}} = \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{C}}
\end{aligned}$$

Proof. From looking the declaration of `AA` in Listing 12.2, we can write the type witness for $\langle\texttt{Tag}|\texttt{B}$ and associated conformance for $\langle\texttt{Self.B: Tag}]$ in terms of $\Sigma_{\texttt{B}}$:

$$\begin{aligned}
&\langle\texttt{Tag}|\texttt{B} \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = \texttt{AA<}\tau\texttt{.B>} = \texttt{AA<}\tau\texttt{>} \otimes \Sigma_{\texttt{B}} \\
&\langle\texttt{Self.B: Tag}] \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = [\texttt{AA<}\tau\texttt{.B>: Tag}] = [\texttt{AA<}\tau\texttt{>: Tag}] \otimes \Sigma_{\texttt{B}}
\end{aligned}$$

Now, we can show that $\Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{AA}} = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{B}}$ by applying $\Sigma_{\texttt{AA}}$ to the constituents of $\Sigma_{\texttt{B}}$:

$$\begin{aligned}
&\tau\texttt{.B} \otimes \Sigma_{\texttt{AA}} = \langle\texttt{Tag}|\texttt{B} \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = \texttt{AA<}\tau\texttt{>} \otimes \Sigma_{\texttt{B}} \\
&[\tau\texttt{.B: Tag}] \otimes \Sigma_{\texttt{AA}} = \langle\texttt{Self.B: Tag}] \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = [\texttt{AA<}\tau\texttt{>: Tag}] \otimes \Sigma_{\texttt{B}}
\end{aligned}$$

The remaining identities are similar. □

Example 12.36. To append "*c*" at the end of "*ab*", we compose on the *left* with $\Sigma_{\texttt{C}}$; notice how it trades places with each neighbor until it reaches $\Sigma_{\texttt{End}}$, at which point it morphs into $\Sigma_{\texttt{CC}}$:

$$\begin{aligned}
&\Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{End}} \\
&\quad = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{End}} \\
&\quad = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{End}} \\
&\quad = \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{BB}} \otimes \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{End}}
\end{aligned}$$

However, once we reach the halting state, we can't append any more symbols.

Lemma 12.37. Let $\Sigma_{\texttt{Halt}}$ be as in Lemma 12.33. We have:

$$\begin{aligned}
&\Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{Halt}} = \Sigma_{\texttt{Halt}} \\
&\Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{Halt}} = \Sigma_{\texttt{Halt}} \\
&\Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{Halt}} = \Sigma_{\texttt{Halt}}
\end{aligned}$$

Proof. In Listing 12.2, the type witnesses for `A`, `B` and `C` in declaration of `Halt` are all `Halt` again, and each associated conformance is $[\texttt{Halt: Tag}]$, so the result follows. □

Finally, we encode the deletion number and production rules in the type witnesses for the `Next` associated type in each conformance to `Tag`:

$$\langle \texttt{Tag}|\texttt{Next} \otimes [\texttt{AA<}\tau\texttt{>: Tag}] = \tau\texttt{.Del.B.C.Next}$$
$$\langle \texttt{Tag}|\texttt{Next} \otimes [\texttt{BB<}\tau\texttt{>: Tag}] = \tau\texttt{.Del.A.Next}$$
$$\langle \texttt{Tag}|\texttt{Next} \otimes [\texttt{CC<}\tau\texttt{>: Tag}] = \tau\texttt{.Del.A.A.A.Next}$$

Note that if the string is empty, we immediately enter the halting state; and if we're already in the halting state, we remain there.

$$\langle \texttt{Tag}|\texttt{Next} \otimes [\texttt{End: Tag}] = \texttt{Halt}$$
$$\langle \texttt{Tag}|\texttt{Next} \otimes [\texttt{Halt: Tag}] = \texttt{Halt}$$

The following result ties everything together.

Lemma 12.38. We can express the above in terms of substitution map composition:

$$\tau\texttt{.Next} \otimes \Sigma_{\texttt{AA}} = \tau\texttt{.Next} \otimes \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{Del}}$$
$$\tau\texttt{.Next} \otimes \Sigma_{\texttt{BB}} = \tau\texttt{.Next} \otimes \Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{Del}}$$
$$\tau\texttt{.Next} \otimes \Sigma_{\texttt{CC}} = \tau\texttt{.Next} \otimes \Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{Del}}$$

$$\tau\texttt{.Next} \otimes \Sigma_{\texttt{End}} = \texttt{Halt}$$
$$\tau\texttt{.Next} \otimes \Sigma_{\texttt{Halt}} = \texttt{Halt}$$

Proof. Consider the first identity. On the left-hand side, we have:

$$\tau\texttt{.Next} \otimes \Sigma_{\texttt{AA}} = \langle \texttt{Tag}|\texttt{Next} \otimes [\tau\texttt{: Tag}] \otimes \Sigma_{\texttt{AA}} = \langle \texttt{Tag}|\texttt{Next} \otimes [\texttt{AA<}\tau\texttt{>: Tag}]$$

To get the right-hand side, we note that:

$$\langle \texttt{Self.A: Tag}] \otimes [\tau\texttt{: Tag}] = [\tau\texttt{: Tag}] \otimes \Sigma_{\texttt{A}}$$
$$\langle \texttt{Self.B: Tag}] \otimes [\tau\texttt{: Tag}] = [\tau\texttt{: Tag}] \otimes \Sigma_{\texttt{B}}$$
$$\langle \texttt{Self.C: Tag}] \otimes [\tau\texttt{: Tag}] = [\tau\texttt{: Tag}] \otimes \Sigma_{\texttt{C}}$$
$$\langle \texttt{Self.Del: Tag}] \otimes [\tau\texttt{: Tag}] = [\tau\texttt{: Tag}] \otimes \Sigma_{\texttt{Del}}$$

Therefore, if we expand $\tau\texttt{.Del.B.C.Next}$ using a conformance path, we see that:

$$\begin{aligned}
&\tau\texttt{.Del.B.C.Next} \\
&= \langle \texttt{Tag}|\texttt{Next} \otimes \langle \texttt{Self.C: Tag}] \otimes \langle \texttt{Self.B: Tag}] \otimes \langle \texttt{Self.Del: Tag}] \otimes [\tau\texttt{: Tag}] \\
&= \langle \texttt{Tag}|\texttt{Next} \otimes \langle \texttt{Self.C: Tag}] \otimes \langle \texttt{Self.B: Tag}] \otimes [\tau\texttt{: Tag}] \otimes \Sigma_{\texttt{Del}} \\
&= \langle \texttt{Tag}|\texttt{Next} \otimes \langle \texttt{Self.C: Tag}] \otimes [\tau\texttt{: Tag}] \otimes \Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{Del}} \\
&= \langle \texttt{Tag}|\texttt{Next} \otimes [\tau\texttt{: Tag}] \otimes \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{Del}} \\
&= \tau\texttt{.Next} \otimes \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{Del}}
\end{aligned}$$

A similar calculation establishes the rest. □

Example 12.39. The following program calculates the Collatz sequence for 3 at compile time. (The `fatalError()` call traps at run time, but that's irrelevant here.)

```
func collatz<T: Tag>(_: T) -> T.Next {
  fatalError()
}

let x = collatz(AA<AA<AA<End>>>())  // what is the type of 'x'?
```

To get the substituted return type for the call expression, we apply the substitution map $\Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{End}}$ to the original return type $\tau.\texttt{Next}$. The substituted type will happen to be `Halt`, because the Collatz sequence for 3 reaches 1.

For a worked example, we instead consider what happens when the substitution map is $\Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{End}}$, which just evaluates the Collatz sequence for 2. We make use of the previous lemmas. Each pair of parentheses indicates the next reduction step:

$$
\begin{aligned}
&(\tau.\texttt{Next} \otimes \Sigma_{\texttt{AA}}) \otimes \Sigma_{\texttt{AA}} \otimes \Sigma_{\texttt{End}} && \text{(Lemma 12.38)}\\
&\quad = \tau.\texttt{Next} \otimes \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{B}} \otimes (\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{AA}}) \otimes \Sigma_{\texttt{End}} && \text{(Lemma 12.31)}\\
&\quad = \tau.\texttt{Next} \otimes \Sigma_{\texttt{C}} \otimes (\Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{End}}) && \text{(Lemma 12.34)}\\
&\quad = \tau.\texttt{Next} \otimes (\Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{BB}}) \otimes \Sigma_{\texttt{End}} && \text{(Lemma 12.35)}\\
&\quad = \tau.\texttt{Next} \otimes \Sigma_{\texttt{BB}} \otimes (\Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{End}}) && \text{(Lemma 12.34)}\\
&\quad = (\tau.\texttt{Next} \otimes \Sigma_{\texttt{BB}}) \otimes \Sigma_{\texttt{CC}} \otimes \Sigma_{\texttt{End}} && \text{(Lemma 12.38)}\\
&\quad = \tau.\texttt{Next} \otimes \Sigma_{\texttt{A}} \otimes (\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{CC}}) \otimes \Sigma_{\texttt{End}} && \text{(Lemma 12.31)}\\
&\quad = \tau.\texttt{Next} \otimes (\Sigma_{\texttt{A}} \otimes \Sigma_{\texttt{End}}) && \text{(Lemma 12.34)}\\
&\quad = (\tau.\texttt{Next} \otimes \Sigma_{\texttt{AA}}) \otimes \Sigma_{\texttt{End}} && \text{(Lemma 12.38)}\\
&\quad = \tau.\texttt{Next} \otimes \Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{B}} \otimes (\Sigma_{\texttt{Del}} \otimes \Sigma_{\texttt{End}}) && \text{(Lemma 12.33)}\\
&\quad = \tau.\texttt{Next} \otimes \Sigma_{\texttt{C}} \otimes (\Sigma_{\texttt{B}} \otimes \Sigma_{\texttt{Halt}}) && \text{(Lemma 12.37)}\\
&\quad = \tau.\texttt{Next} \otimes (\Sigma_{\texttt{C}} \otimes \Sigma_{\texttt{Halt}}) && \text{(Lemma 12.37)}\\
&\quad = (\tau.\texttt{Next} \otimes \Sigma_{\texttt{Halt}}) && \text{(Lemma 12.38)}\\
&\quad = \texttt{Halt}
\end{aligned}
$$

The Collatz sequence for 2 is just 2 followed by 1, so clearly, our encoding is not very efficient; it takes a great deal of work just to compute $2 \div 2 = 1$! Evaluating the Collatz tag system with $n = 19$ takes approximately 1/10th of a second on the author's machine, while attempting $n = 27$ traps with a stack overflow, because the intermediate strings are too long. Practical concerns aside, it should be clear that our scheme can encode any tag system. We can add new symbols, change the deletion number and production rules as needed, and kick it off with an arbitrary input string. Without a doubt:

Theorem 12.40. Swift type substitution is Turing-complete.

Further discussion. In Section 10.4, we looked at an example of a non-terminating conditional conformance check in Swift, and cited a similar example in Rust. In fact, the construction in this section can be tweaked slightly to encode the tag system in terms of conditional conformances instead, so Swift conditional conformance checking is again Turing-complete, beyond simply encoding infinite loops. Looking ahead, Section 16.4 will show that Swift reduced type equality is also undecidable in the general case, but we will get a termination guarantee by restricting the problem.

A contributor to the Swift forums discovered another Turing machine in the Swift type system [127], by constructing an interpreter for the "Brainfuck" programming language in terms of the *key path member lookup* feature [128]. Countless other examples of undecidable type checking problems are described in the literature. For example, Java generics are known to be Turing-complete [129, 130].

Finally, an encoding of the Collatz sequence in the TypeScript language is described in [131]. For further discussion of the Collatz conjecture, see [132] and [133].

12.5. Source Code Reference

Key source files:

- `include/swift/AST/GenericSignature.h`
- `include/swift/AST/SubstitutionMap.h`
- `lib/AST/GenericSignature.cpp`
- `lib/AST/SubstitutionMap.cpp`
- `lib/AST/TypeSubstitution.cpp`

`ConformancePath::Entry` *type alias*

A conformance path element, represented as a `std::pair<CanType, ProtocolDecl *>`. The pair either encodes a root abstract conformance, or an associated conformance requirement.

`ConformancePath` *class*

A wrapper around `ArrayRef<ConformancePath::Entry>` which represents a conformance path.

`SubstitutionMap` *class*

The `lookupConformance()` method implements Algorithm 12A for performing a local conformance lookup. For other methods, see Section 6.6.

Finding Conformance Paths

Key source files:

- `lib/AST/RequirementMachine/GenericSignatureQueries.cpp`

`GenericSignatureImpl` *class*

The `getConformancePath()` method returns the reduced conformance path for a given type parameter and protocol declaration. For other methods, see Section 5.7.

`rewriting::RequirementMachine` *class*

The `getConformancePath()` method on `GenericSignature` calls the method with the same name on the `RequirementMachine` class. The latter implements Algorithm 12D. The `RequirementMachine` class has a pair of instance variables to store the algorithm's persistent state:

- `ConformancePaths` is the table of known conformance paths.
- `CurrentConformancePaths` is the buffer of conformance paths at the currently enumerated length, which we denoted by B.

Algorithm 12D traffics in reduced type parameters, while the actual implementation deals with instances of `Term`. A term is the internal Requirement Machine representation of a type parameter, as we will learn in Chapter 17. This avoids round-trip conversions between `Term` and `Type` when computing reduced types, but does not fundamentally alter the algorithm.

13. Opaque Result Types

Opaque result types introduce a new form of abstraction that cannot be expressed with the other language features we've described so far. Despite this conceptual leap, we will see that opaque result types are assembled from existing building blocks: generic signatures (Chapter 5), substitution maps (Chapter 6), and abstract conformances (Section 7.4). We will begin with an intuitive overview, look at the concrete syntax, and finally, detail the semantics.

The fundamental idea behind opaque result types is that they "reverse" the usual relationship between the caller and callee of a generic declaration. While an ordinary "input" generic parameter abstracts over a generic argument provided by the caller, an opaque result type is like an "output" generic parameter in that it abstracts over a fixed concrete type that is only known to the *callee*. We refer to this concrete type as the *underlying type* of the opaque result type.

Like ordinary generic parameters, opaque result types can be subject to requirements. When type checking the body of a declaration with an opaque result type, we infer its underlying type from the statements in the body, and we ensure this underlying type satisfies the requirements imposed upon it. If we then encounter a call to a declaration with an opaque result type, we say that result type of the call is an *opaque archetype* subject to these requirements; the caller does not know the underlying type.

Opaque result declarations. Recall how in parameter position, the `some` keyword denotes an opaque *parameter* declaration, which was simply syntax sugar for an unnamed generic parameter together with a single conformance requirement (Section 4.1). Now, when `some` appears in the *return type* of a function or subscript declaration, or as the type of a variable declaration, we get an opaque *result* declaration instead:

```
func foo() -> some Sequence {...}
var bar: some Sequence {...}
struct S {
  subscript(_: Int) -> some Sequence {...}
}
```

Opaque result declarations come into existence when we evaluate the **opaque result type request**. This request receives a value declaration as input. In the evaluation function, we check if the declaration's return type contains any occurrences of `some`. If not, we're done; otherwise, we construct an opaque result declaration.

Opaque result generic signatures. An opaque result declaration points back to its *owner declaration*, which is the value declaration that declares this opaque result type. There is a one-to-one correspondence between opaque result declarations, and their owner declarations. Since the `some` keyword may appear any number of times in the owner declaration's return type, an opaque result declaration will in general declare one or more *opaque result types*.

We describe these opaque result types using the generic signature of the opaque result declaration, which we build within the **opaque result type request**. We will call this generic signature the *opaque result generic signature*, while the *outer generic signature* is what we call the generic signature of the owner declaration.

The first step is to collect all occurrences of `some`. We're working with syntactic type representations here (Chapter 9), so we walk the type representation of the owner declaration's return type, and invoke type resolution to resolve each constraint type that follows each `some`. Once we have this list of constraint types, we kick off the **abstract generic signature request** (Chapter 11) with the following parameters:

1. We pass the outer generic signature as the parent signature.
2. We add a generic parameter for each occurrence of the `some` keyword in the owner declaration's return type.

   If the owner declaration is not actually generic, all new generic parameters have depth 0; otherwise, we set their depth to be one more than the maximum depth of the outer generic signature's parameters. The index of each new generic parameter is determined by the order in which the `some` keyword appears.
3. For each new generic parameter, we also add a requirement with this generic parameter on the left-hand side, and the constraint type on the right-hand side.

   When the constraint type is a protocol type, a protocol composition type, or a parameterized protocol type, this will be a conformance requirement. When we have a protocol composition or parameterized protocol type, the request will decompose the requirement into simpler requirements automatically (Section 11.2).

   Otherwise, the constraint type must be a class type, and we add a superclass requirement instead.

The opaque result generic signature does not depend on the underlying type of its opaque result types, so we can compute it without looking at the function body. This is important, because we omit function bodies in textual interfaces, except for `@inlinable` functions (Section 2.5). Similarly, the parser does not parse bodies of declarations in the secondary files of a frontend job. The only time we attempt to compute the underlying type is when the owner declaration appears in a primary source file for this frontend job. For this reason, we can omit function bodies in the examples on the following page.

Example 13.1. The below is a list of the opaque result generic signatures that we obtain in a few simple instances. We will revisit some of these examples later.

1. Here we get a completely unconstrained opaque result type, because `Any` is the empty protocol composition; a conformance requirement to `Any` is a no-op:

   ```
   func fullyOpaque() -> some Any {...}
   ```

   Opaque result generic signature: `<τ_0_0>`

2. The following opaque result declaration has two unconstrained opaque result types:

   ```
   func fullyOpaquePair() -> (some Any, some Any) {...}
   ```

   Opaque result generic signature: `<τ_0_0, τ_0_1>`

3. If the constraint type is a single protocol, we get an opaque result generic signature with a conformance requirement to this protocol:

   ```
   func someEquatable(_ b: Bool) -> some Equatable {...}
   ```

   Opaque result generic signature:
   `<τ_0_0 where τ_0_0: Equatable>`

4. If the owner declaration is generic, we incorporate its generic signature into the opaque result generic signature. We will see why in the next section:

   ```
   func someEquatable2<T>(_: T) -> some Equatable {...}
   ```

   Opaque result generic signature:
   `<τ_0_0, τ_1_0 where τ_1_0: Equatable>`

5. A parameterized protocol type decomposes into a conformance requirement and one or more same-type requirements:

   ```
   func someSequenceOfInt() -> some Sequence<Int> {...}
   ```

   Opaque result generic signature:
   `<τ_0_0 where τ_0_0: Sequence, τ_0_0.Element == Int>`

6. Finally, a parameterized protocol type can introduce a same-type requirement between an opaque result type, and a type parameter of the owner declaration:

   ```
   func someSequenceOfT<T>(_: T) -> some Sequence<T> {...}
   ```

   Opaque result generic signature:
   `<τ_0_0, τ_1_0 where τ_0_0 == τ_1_0.Element, τ_1_0: Sequence>`

Inferring the underlying type. When type checking a function body in a primary file, we infer the underlying types of its opaque result types after we assign types to expressions. We collect all `return` statements that appear in the function body, and consider the type of each returned expression.

Example 13.2. The `hungryHorses` computed property demonstrates a typical use case for opaque result types, which is to "hide" a non-trivial generic return type:

```
struct Horse {
  var isHungry: Bool
}

struct Farm {
  var horses: [Horse] = []
  var hungryHorses: some Collection<Horse> {
    return horses.lazy.filter(\.isHungry)
  }
}
```

The `return` statement's type is `LazyFilterSequence<Array<Horse>>`, but the caller only sees that it is a `Collection` with an `Element` type of `Horse`.

The fundamental invariant we must maintain is that the underlying type of an opaque result type cannot change while a program is running. In general, if the body of a function with an opaque result type contains multiple `return` statements, they must all return the same exact type. (This is also true of an ordinary function, as well.)

Example 13.3. The following is thus not allowed; to model this kind of dynamism, the function could return an existential `any Sequence` instead (Chapter 14):

```
func twoSequences(_ b: Bool) -> some Sequence {
  if b {
    return [1, 2, 3]  // Array<Int>
  } else {
    return ["a", "ab", "ba"]  // Array<String>
  }
}
```

There is one exception to this rule. We allow the underlying type of an opaque result type to depend on *availability*. We will not discuss availability checking in the this book, except to say that the outcome of an availability check, which is written with the special "`if #available(...)`" syntax, does not change during program execution. Thus, we allow a function with an opaque result type to contain `return` statements with mismatched types, as long as they're in different branches of an availability check.

Example 13.4. On a macOS host, the `bestWidget()` function will return one of two underlying types, depending on the operating system version:

```
protocol Widget {}
struct OldWidget: Widget {}

@available(macOS 11, *)
struct NewWidget: Widget {}

func bestWidget() -> some Widget {
  if #available(macOS 11, *) {
    return NewWidget()
  } else {
    return OldWidget()
  }
}
```

We record the underlying types of an opaque result declaration in a series of *underlying type substitution maps*, where each substitution map corresponds to a disjoint *availability range*. Henceforth, we will assume that each opaque result declaration only has one availability range, and thus one underlying type substitution map.

The underlying type substitution map's input generic signature is the opaque result generic signature, while its output generic signature is, by construction, the outer generic signature. This substitution map sends the generic parameters of the outer generic signature to themselves, and the generic parameters that represent opaque result types to their corresponding underlying types. We check that this substitution map satisfies the requirements of the opaque result generic signature using Algorithm 9B.

Example 13.5. We reject the following program:

```
func invalidUnderlyingType() -> some Sequence<Int> {
  return ["ab", "ba", "aab"]  // error
}
```

The opaque result generic signature is

$$\texttt{<}\tau\texttt{_0_0 where }\tau\texttt{_0_0: Sequence, }\tau\texttt{_0_0.Element == Int>}$$

We form the following underlying type substitution map for this generic signature:

$$\{\tau\texttt{_0_0} \mapsto \texttt{Array<String>};$$
$$\ [\tau\texttt{_0_0: Array<String>}] \mapsto [\texttt{Sequence: Array<String>}]\}$$

Applying this substitution map to the same-type requirement [`τ_0_0.Element == Int`] produces [`String == Int`], which is unsatisfied, so we diagnose an error.

History. Opaque result types were first introduced in Swift 5.1 [134], so they actually predate opaque parameter declarations [76]. The initial implementation only allowed `some` to appear once at the outermost level of a type representation. Swift 5.7 generalized this to allow `some` to appear one or more times, nested in arbitrary types [135]. Swift 5.7 also introduced opaque result types that depend on availability [136].

13.1. Opaque Archetypes

Once we have the opaque result generic signature, we can proceed to type resolution, where we build the interface type of the owner declaration. To get the interface type for an occurrence of `some` in return position, we take the corresponding generic parameter in the opaque result generic signature, and form the *opaque archetype* representing this generic parameter. This opaque archetype then appears in the interface type of the owner declaration.

We met archetypes and the generic environments that spawn them in Chapter 8. Our focus there was on *primary archetypes*, and we recall some facts about primary archetypes first. Formally, a primary archetype is a pair $(G, \texttt{T})$ for some generic signature G and (reduced) type parameter `T`. We denoted a primary archetype by $[\![\texttt{T}]\!]_G$, or just $[\![\texttt{T}]\!]$ when G is clear from context. Primary archetypes only appear in the types of expressions within function bodies, and subsequently in SIL instructions.

An opaque archetype similarly packages up a type parameter with a generic signature:

Definition 13.6. If d is an opaque result declaration and `T` is a reduced type parameter for the opaque result generic signature of d, we will denote the opaque archetype for `T` by $[\![\circlearrowleft \texttt{T}]\!]_d$, or $[\![\circlearrowleft \texttt{T}]\!]$ when d is implied from context. (The symbol "$\circlearrowleft$" is, of course, an amalgamation of "o" for opaque and "`->`" for result type.)

Example 13.7. Recall that when the owner declaration is not generic, the opaque result generic signature's generic parameters have depth 0. Consider this function:

```
func someEquatable(_ b: Bool) -> some Equatable {
  return b ? 1 : 2
}
```

The opaque result generic signature is `<τ_0_0 where τ_0_0: Equatable>` in the above, and we resolve this function's return type to the opaque archetype $[\![\circlearrowleft \texttt{τ_0_0}]\!]$. The interface type of this function declaration is the following function type:

$$\texttt{(Bool) -> } [\![\circlearrowleft \texttt{τ_0_0}]\!]$$

Note that opaque archetypes from distinct opaque result declarations are themselves distinct, even if their opaque result generic signatures are equal; in that case, they just represent unrelated types that happen to satisfy the same requirements.

Opaque archetypes interact with the type substitution algebra in two ways. First, an opaque archetype may appear in a substitution map as a replacement type. Second, we can apply a substitution map to an opaque archetype to get a new opaque archetype. We will now consider each possibility in turn.

Example 13.8. In the previous example, the underlying type of the opaque result type of `someEquatable()` is `Int`, but the caller cannot observe this fact. Instead, we get the opaque archetype $[\![\circlearrowleft \tau_0_0]\!]$, and only the operations provided by the `Equatable` protocol are available to us. We also know that every call returns the same type:

```
print(someEquatable(false) == someEquatable(true))   // prints false
print(someEquatable(false) == someEquatable(false))  // prints true
```

Recall that both arguments to "`==`" must have the same type, and this type must conform to `Equatable`. The generic signature of "`==`" is `<τ_0_0 where τ_0_0: Equatable>`, so we form this substitution map that contains the replacement type $[\![\circlearrowleft \tau_0_0]\!]$:

$$\{\tau_0_0 \mapsto [\![\circlearrowleft \tau_0_0]\!];$$
$$\ [\tau_0_0\texttt{: Equatable}] \mapsto [[\![\circlearrowleft \tau_0_0]\!]\texttt{: Equatable}]\}$$

Our input generic signature states a conformance requirement, so we need to look up the conformance of $[\![\circlearrowleft \tau_0_0]\!]$ to `Equatable` when we build our substitution map. We extended global conformance lookup to archetypes in Section 8.1. It behaves identically with primary and opaque archetypes. When given an opaque archetype whose opaque result generic signature states a conformance requirement, global conformance lookup will output an abstract conformance whose subject type is this opaque archetype:

$$\langle\texttt{Equatable}] \otimes [\![\circlearrowleft \tau_0_0]\!] = [[\![\circlearrowleft \tau_0_0]\!]\texttt{: Equatable}]$$

We shorten this by saying we get an *opaque abstract conformance*. Next, we will demonstrate type witness projection from an opaque abstract conformance. To do that, we constrain our opaque result type to a protocol with associated types.

Example 13.9. In the final line of the below listing, the call expression `someSequence()` has the type $[\![\circlearrowleft \tau_0_0]\!]$. We will deduce the type of `pick(someSequence())`:

```
func someSequence() -> some Sequence {
  return [1, 2, 3]
}

func pick<S: Sequence>(_ s: S) -> S.Element {...}

print(pick(someSequence()))
```

The substitution map for the call to `pick()` is:

$$\Sigma := \{\texttt{τ_0_0} \mapsto [\![\circlearrowleft \texttt{τ_0_0}]\!];$$
$$[\texttt{τ_0_0: Sequence}] \mapsto [[\![\circlearrowleft \texttt{τ_0_0}]\!]\texttt{: Sequence}]\}$$

We apply Σ to `pick()`'s original return type, `τ_0_0.Element`, to get the answer. If we factor this dependent member type and apply Σ to its abstract conformance, we get the opaque abstract conformance from our substitution map:

$$\texttt{τ_0_0.Element} \otimes \Sigma$$
$$= \langle \texttt{Sequence}|\texttt{Element} \otimes [\texttt{τ_0_0: Sequence}] \otimes \Sigma$$
$$= \langle \texttt{Sequence}|\texttt{Element} \otimes [[\![\circlearrowleft \texttt{τ_0_0}]\!]\texttt{: Sequence}]$$

To get the final result, we project the type witness. This outputs an opaque archetype for the same opaque result declaration, but representing our dependent member type:

$$\langle \texttt{Sequence}|\texttt{Element} \otimes [[\![\circlearrowleft \texttt{τ_0_0}]\!]\texttt{: Sequence}] = [\![\circlearrowleft \texttt{τ_0_0.Element}]\!]$$

Note that `someSequence()` has the following underlying type substitution map:

$$\{\texttt{τ_0_0} \mapsto \texttt{Array<Int>};$$
$$[\texttt{τ_0_0: Sequence}] \mapsto [\texttt{Array<Int>: Sequence}]\}$$

Therefore, at run time, a value of type $[\![\circlearrowleft \texttt{τ_0_0.Element}]\!]$ is actually an `Int`.

Declared interface type. Recall our discussion of type declarations from Chapter 4. We didn't mention it at the time, but an opaque result declaration is actually a special kind of type declaration, so we must also give it a declared interface type. We say that the declared interface type of an opaque result declaration is the owner declaration's return type. Note that this type will *contain* at least one opaque archetype, but it may not necessarily *be* an opaque archetype. Thus, an opaque result declaration can be said to "declare" all opaque archetypes that appear in its declared interface type, which can otherwise contain arbitrary structure. In the implementation, the declared interface type of a type declaration must be set to *something*, and setting it to *this* turns out to be convenient for the implementation.

Type substitution. Under Algorithm 6A, a primary archetype $[\![\texttt{T}]\!]$ behaves like the type parameter `T` that it represents, in the sense that if we have a substitution map Σ, then $[\![\texttt{T}]\!] \otimes \Sigma = \texttt{T} \otimes \Sigma$. Opaque archetypes are rather different: applying a substitution map to an opaque archetype produces a new opaque archetype.

In full generality, an opaque archetype is really a *triple* $(\texttt{T}, d, \Sigma)$ where `T` is a reduced type parameter in the opaque result generic signature of d, and Σ is a substitution map for the outer generic signature of d. (If the owner declaration of d is not generic, Σ is always the empty substitution map, and all of the below trivializes.)

Definition 13.10. We will continue to use the notation $[\![\circlearrowleft \texttt{T}]\!]_d$ to denote an opaque archetype for the identity substitution map of its outer generic signature. More generally, $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ will denote a *substituted opaque archetype* for a substitution map Σ.

Type resolution always resolves each occurrence of `some` to an opaque archetype for the identity substitution map of its outer generic signature. Of course, as the notation suggests, if we apply a substitution map Σ to $[\![\circlearrowleft \texttt{T}]\!]$, we get $[\![\circlearrowleft \texttt{T}]\!] \otimes \Sigma$. We remark that this behavior is analogous to the context substitution map of a generic nominal type.

Example 13.11. Here is a generic function that states an opaque result type:

```
func someEquatable2<T: Equatable>(_ t: T) -> some Equatable {
  return [t]
}
```

The opaque result generic signature includes the outer generic signature, and also adds a generic parameter with depth 1, for the opaque result type:

```
<τ_0_0, τ_1_0 where τ_0_0: Equatable, τ_1_0: Equatable>
```

The interface type of `someEquatable2()` is the following generic function type:

```
<τ_0_0 where τ_0_0: Equatable> (τ_0_0) -> [⥁ τ_1_0]
```

Now, we consider type checking the following two statements:

```
print(someEquatable2(1) == someEquatable2(2))  // prints false
print(someEquatable2(1) == someEquatable2("hello"))  // type error
```

In the first statement, both calls to `someEquatable2()` have the same substitution map:

$$\Sigma_1 := \{\texttt{τ_0_0} \mapsto \texttt{Int};\\ \quad [\texttt{τ_0_0: Equatable}] \mapsto [\texttt{Int: Equatable}]\}$$

It follows that the first statement is well-typed, because we call "`==`" with two arguments of type $[\![\circlearrowleft \texttt{τ_1_0}]\!] \otimes \Sigma_1$. Evaluating it would print `false`, because the two values are not actually equal. We don't get that far, however. In the second statement, the right-hand side is a call to `someEquatable2()` with a different substitution map:

$$\Sigma_2 := \{\texttt{τ_0_0} \mapsto \texttt{String};\\ \quad [\texttt{τ_0_0: Equatable}] \mapsto [\texttt{String: Equatable}]\}$$

Thus, the type checker rejects the second statement, because the type of the right-hand side is $[\![\circlearrowleft \texttt{τ_1_0}]\!] \otimes \Sigma_2$, which is distinct from the left-hand side $[\![\circlearrowleft \texttt{τ_1_0}]\!] \otimes \Sigma_1$.

We can justify this behavior if we note that the underlying type of an opaque result type similarly depends on its outer generic signature. Consider the underlying type substitution map of `someEquatable2()`:

$$\{\texttt{τ_0_0} \mapsto \texttt{τ_0_0},$$
$$\texttt{τ_1_0} \mapsto \texttt{Array<τ_0_0>};$$
$$[\texttt{τ_0_0: Equatable}] \mapsto [\texttt{τ_0_0: Equatable}]$$
$$[\texttt{τ_1_0: Equatable}] \mapsto [\texttt{Array<τ_0_0>: Equatable}]\}$$

The output generic signature of this substitution map is the generic signature of the owner declaration, `someEquatable2()`. We will elaborate this in Section 13.3.

We saw that if we take an opaque archetype for the identity substitution map, and apply a substitution map Σ, we get a new opaque archetype for this substitution map Σ. More generally, if an opaque archetype carries an arbitrary substitution map, applying another substitution map to this archetype will compose the two substitution maps.

Algorithm 13A (Apply substitution map to opaque archetype)**.** Receives an opaque archetype $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ and a substitution map Σ' as input. Outputs $([\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma) \otimes \Sigma'$.

1. Decompose $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ into a triple $(\texttt{T}, d, \Sigma)$ for some type parameter `T`, opaque result declaration d, and substitution map Σ.
2. Construct $\Sigma \otimes \Sigma'$ (Definition 6.6).
3. Construct an opaque archetype from the triple $(\texttt{T}, d, \Sigma \otimes \Sigma')$, and return it. (This opaque archetype is written as $[\![\circlearrowleft \texttt{T}]\!]_d \otimes (\Sigma \otimes \Sigma')$ in our notation, for reasons that are now hopefully clear.)

Notice how if we start with $[\![\circlearrowleft \texttt{T}]\!]_d = [\![\circlearrowleft \texttt{T}]\!]_d \otimes 1_G$ and apply a substitution map Σ, the above algorithm reduces to directly constructing a new opaque archetype with this substitution map, because $1_G \otimes \Sigma = \Sigma$:

$$([\![\circlearrowleft \texttt{T}]\!]_d \otimes 1_G) \otimes \Sigma = [\![\circlearrowleft \texttt{T}]\!]_d \otimes (1_G \otimes \Sigma) = [\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$$

Interface types. The fact that opaque archetypes can appear in the interface types of declarations is important. Certainly, when a declaration states an opaque result type, its interface type will contain its own opaque archetype. However, the interface type of a declaration may involve opaque archetypes declared elsewhere. The most obvious case is a variable declaration whose initial value expression calls a function with an opaque result type, for example:

```
let result = someEquatable(false)
```

However, this is not always permitted, depending on the variable's declaration context. We allow this for local variables, as well as any global variables in the main source file of a module (if it has one). Global variables in library source files, and stored properties of nominal type declarations, are prohibited from referencing opaque archetypes from other owner declarations, and we diagnose an error in this case.

This distinction between these two kinds of variables is artificial though, and we will see an even more general way to reference the opaque archetypes of another declaration in Section 13.2. Thus, we extend our definition of an interface type to encompass types that contain certain opaque archetypes, without any restriction on the opaque archetype's owner declaration.

Definition 13.12. We amend Definition 6.4 as follows. Suppose that $[\![\circlearrowleft \mathtt{T}]\!]_d \otimes \Sigma$ is an opaque archetype. If G is the outer generic signature of d, and H is some other generic signature such that $\Sigma \in \textsc{Sub}(G, H)$, then we say that:

$$[\![\circlearrowleft \mathtt{T}]\!]_d \otimes \Sigma \in \textsc{Type}(H)$$

Thus, an opaque archetype is an interface type for the output generic signature of its substitution map. In particular, if Σ is the identity substitution map 1_G, then:

$$[\![\circlearrowleft \mathtt{T}]\!] \otimes 1_G = [\![\circlearrowleft \mathtt{T}]\!] \in \textsc{Type}(G)$$

That is, when we resolve an occurrence of `some` to an opaque archetype in type resolution, we get an interface type for its outer generic signature, as we expect.

Conformance substitution. We now ponder what it means to apply a substitution map to an opaque abstract conformance. As originally defined, an abstract conformance $[\mathtt{T}\colon\ \mathtt{P}]$ has a type parameter $\mathtt{T}$ as its subject type, and we saw that applying a substitution map Σ to $[\mathtt{T}\colon\ \mathtt{P}]$ reduces to a local conformance lookup into Σ. We related this to global conformance lookup on the type $\mathtt{T} \otimes \Sigma$ using the following identity from Section 7.4:

$$[\mathtt{T}\colon\ \mathtt{P}] \otimes \Sigma = (\langle \mathtt{P} \rangle \otimes \mathtt{T}) \otimes \Sigma = \langle \mathtt{P} \rangle \otimes (\mathtt{T} \otimes \Sigma)$$

In Section 8.2, we encountered abstract conformances of primary archetypes, and from the above, we deduced they act like type parameters under conformance substitution, in the sense that $[[\![\mathtt{T}]\!]\colon\ \mathtt{P}] \otimes \Sigma = [\mathtt{T}\colon\ \mathtt{P}] \otimes \Sigma$. Now, suppose we apply Σ to an opaque abstract conformance, say $[[\![\circlearrowleft \mathtt{T}]\!]\colon\ \mathtt{P}]$. Again, we have $[[\![\circlearrowleft \mathtt{T}]\!]\colon\ \mathtt{P}] = \langle \mathtt{P} \rangle \otimes [\![\circlearrowleft \mathtt{T}]\!]$, so:

$$[[\![\circlearrowleft \mathtt{T}]\!]\colon\ \mathtt{P}] \otimes \Sigma = (\langle \mathtt{P} \rangle \otimes [\![\circlearrowleft \mathtt{T}]\!]) \otimes \Sigma = \langle \mathtt{P} \rangle \otimes ([\![\circlearrowleft \mathtt{T}]\!] \otimes \Sigma)$$

However, this time, $[\![\circlearrowleft \mathtt{T}]\!] \otimes \Sigma$ is another opaque archetype, and $\langle \mathtt{P} \rangle \otimes ([\![\circlearrowleft \mathtt{T}]\!] \otimes \Sigma)$ reduces to another abstract conformance to $\mathtt{P}$, except now for the substituted archetype. We define conformance substitution with an opaque abstract conformance thusly:

$$[[\![\circlearrowleft \mathtt{T}]\!]\colon\ \mathtt{P}] \otimes \Sigma := [([\![\circlearrowleft \mathtt{T}]\!] \otimes \Sigma)\colon\ \mathtt{P}]$$

More generally, the subject type of an opaque abstract conformance might have a non-identity substitution map, at which point the notation becomes slightly awkward:

$$[(\llbracket \circlearrowleft \mathtt{T} \rrbracket \otimes \Sigma)\colon\ \mathtt{P}] \otimes \Sigma' := [(\llbracket \circlearrowleft \mathtt{T} \rrbracket \otimes (\Sigma \otimes \Sigma'))\colon\ \mathtt{P}]$$

The key fact is applying a substitution map to an opaque abstract conformance always outputs another opaque abstract conformance. This is unlike the situation with abstract conformances whose subject types are type parameters or primary archetypes.

Interface types and contextual types. Recall that a contextual type is one that contains primary archetypes. In Section 8.2, we introduced the in_G and out_G operations for mapping between the interface types and contextual types of a generic signature G, two sets which we denoted by $\textsc{Type}(G)$ and $\textsc{Type}(\llbracket G \rrbracket)$.

We also characterized in_G and out_G in terms of type substitution, as the application of the forwarding substitution map $1_{\llbracket G \rrbracket}$ or the identity substitution map 1_G to a type, respectively. This gave us analogous mappings between $\textsc{Sub}(G, H)$ and $\textsc{Sub}(G, \llbracket H \rrbracket)$; we compose a substitution map on the right with $1_{\llbracket H \rrbracket}$ or 1_H to transform its replacement types into contextual or interface types, as desired.

Naturally, we extend "in" and "out" to opaque archetypes by composing the opaque archetype's substitution map with the forwarding or identity substitution map for the appropriate generic signature on the right. Let's work out the details.

First, suppose that $\llbracket \circlearrowleft \mathtt{T} \rrbracket_d$ is an opaque archetype for the identity substitution map 1_G, where G is the outer generic signature of d. When we map $\llbracket \circlearrowleft \mathtt{T} \rrbracket_d$ into the primary generic environment of G, we get the following opaque archetype:

$$\mathsf{in}_G(\llbracket \circlearrowleft \mathtt{T} \rrbracket_d) = \llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes 1_{\llbracket G \rrbracket} \in \textsc{Type}(\llbracket G \rrbracket)$$

In the general case, we have an opaque archetype $\llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes \Sigma$ for some $\Sigma \in \textsc{Sub}(G, H)$. This opaque archetype is an element of $\textsc{Type}(H)$, and we can map it into the primary generic environment of H to get a new opaque archetype whose substitution map is the corresponding element of $\textsc{Sub}(G, \llbracket H \rrbracket)$:

$$\mathsf{in}_H(\llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes \Sigma) = \llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes (\Sigma \otimes 1_{\llbracket H \rrbracket}) \in \textsc{Type}(\llbracket H \rrbracket)$$

In the other direction, we can apply out_G to the contextual type $\llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes 1_{\llbracket G \rrbracket}$. Since $1_{\llbracket G \rrbracket} \otimes 1_G = 1_G$, we receive the original opaque archetype $\llbracket \circlearrowleft \mathtt{T} \rrbracket_d$:

$$\mathsf{out}_G(\llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes 1_{\llbracket G \rrbracket}) = \llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes (1_{\llbracket G \rrbracket} \otimes 1_G) = \llbracket \circlearrowleft \mathtt{T} \rrbracket_d \in \textsc{Type}(G)$$

Finally, in the general case where $\Sigma \in \textsc{Sub}(G, \llbracket H \rrbracket)$, we get:

$$\mathsf{out}_H(\llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes \Sigma) = \llbracket \circlearrowleft \mathtt{T} \rrbracket_d \otimes (\Sigma \otimes 1_H) \in \textsc{Type}(H)$$

To summarize, an opaque archetype can either play the role of an interface type or a contextual type, depending on whether its substitution map stores interface or contextual replacement types.

Example 13.13. A detail we left out of Example 13.11 is that the replacement types of substitution maps appearing in expressions are contextual types. We use an opaque archetype as a contextual type below:

```
func someEquatable3<T: Equatable>(_ t: T) -> some Equatable {
  return [someEquatable2(t)]
}
```

Let G denote the generic signature of `someEquatable3()`. Note that `someEquatable3()` calls `someEquatable2()` with the following substitution map, whose replacement type is a primary archetype of G:

$$\{\texttt{τ_0_0} \mapsto [\![\texttt{τ_0_0}]\!];$$
$$\ [\texttt{τ_0_0: Equatable}] \mapsto [[\![\texttt{τ_0_0}]\!]\texttt{: Equatable}]\}$$

In fact, the above is just the forwarding substitution map for G. If d is the opaque result declaration of `someEquatable2()`, then the original return type of `someEquatable2()` is the opaque archetype $[\![\circlearrowleft\ \texttt{τ_1_0}]\!]_d$, and the type of the call `someEquatable2(t)` is $[\![\circlearrowleft\ \texttt{τ_1_0}]\!]_d \otimes 1_{[\![G]\!]}$. This call expression is nested inside an array literal, so the return expression has the type $\texttt{Array<}[\![\circlearrowleft\ \texttt{τ_1_0}]\!]_d \otimes 1_{[\![G]\!]}\texttt{>}$.

To build the underlying type substitution map, we map the return expression's type out of the primary environment of G, to get the interface type $\texttt{Array<}[\![\circlearrowleft\ \texttt{τ_1_0}]\!]_d\texttt{>}$. We then look up this type's conformance to `Equatable`, which gives us a conditional specialized conformance. We obtain this substitution map:

$$\{\texttt{τ_0_0} \mapsto \texttt{τ_0_0}$$
$$\ \texttt{τ_1_0} \mapsto \texttt{Array<}[\![\circlearrowleft\ \texttt{τ_1_0}]\!]_d\texttt{>};$$
$$\ [\texttt{τ_0_0: Equatable}] \mapsto [\texttt{τ_0_0: Equatable}]$$
$$\ [\texttt{τ_1_0: Equatable}] \mapsto [\texttt{Array<}[\![\circlearrowleft\ \texttt{τ_1_0}]\!]_d\texttt{>: Equatable}]\}$$

We observe that the opaque result type of `someEquatable3()` is defined in terms of the opaque result type of `someEquatable2()`. This is a common occurrence in frameworks that make use of opaque result types.

Opaque generic environments. Every opaque archetype is instantiated from an *opaque generic environment*, and every opaque generic environment is uniquely identified by a pair (d, Σ), where d is an opaque result declaration, and Σ is a substitution map for the outer generic signature of d. Recall the discussion of reduced types from Section 5.5. Within an opaque generic environment, we lazily populate a lookup table where each key is a reduced type parameter `T`, storing the opaque archetype $[\![\circlearrowleft\ \texttt{T}]\!]_d \otimes \Sigma$. Time has come to take a closer look at this mapping.

While every opaque archetype represents a reduced type parameter, it is not the case that every reduced type parameter in the opaque result generic signature is represented by an opaque archetype. We recall that a generic parameter in the opaque result generic signature is either part of the outer generic signature, or it represents an opaque result type. Thus, if we consider the *root* generic parameter that remains after we peel off any dependent member types, we see the opaque result generic signature has two varieties of reduced type parameter:

1. If a type parameter is rooted in a generic parameter of the owner declaration, we say it is an *outer* type parameter.
2. All other valid type parameters are rooted in one of the new generic parameters added by the opaque result declaration, so they represent opaque result types.

Only reduced type parameters of the second kind are represented by opaque archetypes; a type parameter of the first kind is handled differently below.

Algorithm 13B (Map type parameter into opaque generic environment)**.** As input, takes an opaque generic environment (d, Σ) for some opaque result declaration d and substitution map Σ, and a type parameter `T`. Outputs $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ if the archetype exists, otherwise an interface or contextual type for the output generic signature of Σ.

1. Let O be the opaque result generic signature of d.
2. (Reduce) Set `X` $\leftarrow$ `getReducedType`$(O,$ `T`$)$.
3. (Concrete) If `X` is a concrete type, it may still contain type parameters. Recursively apply this algorithm to each type parameter of `X`, and return the result.
4. (Abstract) Otherwise, `X` is a reduced type parameter. Set `T` $\leftarrow$ `X`.
5. (Outer) If `T` is an outer type parameter, return `T` $\otimes \Sigma$.
6. (Archetype) If not, return the opaque archetype $(\texttt{T}, d, \Sigma)$, denoted $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$.

To actually observe a concrete type in Step 3 or an outer type parameter in Step 5, we can consider an opaque result type constrained to a parameterized protocol type.

Example 13.14. The following is a fancier variant of Example 13.9:

```
func someSequenceOfInt() -> some Sequence<Int> {
  return [n]
}

func pick<S: Sequence>(_ s: S) -> S.Element {...}

print(pick(someSequenceOfInt()))
```

We make a note of the opaque result generic signature of `someSequenceOfInt()`, which has a same-type requirement from the parameterized protocol type:

$$\texttt{<}\tau\texttt{_0_0 where }\tau\texttt{_0_0: Sequence, }\tau\texttt{_0_0.Element == Int>}$$

The return type of the call to `someSequenceOfInt()` is the opaque archetype $[\![\circlearrowleft \tau\texttt{_0_0}]\!]$ for this signature. To get the return type of `pick(someSequenceOfInt())`, we proceed as in Example 13.9, but when we project the type witness from the opaque abstract conformance in the last step, we see that we do not get another opaque archetype back. Instead, when type witness projection maps `τ_0_0.Element` into the opaque generic environment, we end up in Step 3 of Algorithm 13B, and we get back `Int`:

$$\langle\texttt{Sequence}|\texttt{Element} \otimes [[\![\circlearrowleft \tau\texttt{_0_0}]\!]\texttt{: Sequence}] = \texttt{Int}$$

Therefore, the call expression `pick(someSequenceOfInt())` has type `Int`.

Example 13.15. If the owner declaration is generic, the parameterized protocol type can constrain its primary associated types to the type parameters of the outer generic signature. We show that in the below, the type of the argument to `print()` is `Bool`:

```
func someSequenceOfT<T>(_ t: T) -> some Sequence<T> {
  return [t]
}

func pick<S: Sequence>(_ s: S) -> S.Element {...}

print(pick(sequenceOfT(true)))
```

The generic signature of `sequenceOfT()` is `<τ_0_0>`, and we call it with the substitution map $\Sigma := \{\tau\texttt{_0_0} \mapsto \texttt{Bool}\}$ on the final line of the above. Also, let d be the opaque result declaration of `someSequenceOfT()`, and let O be the opaque result generic signature:

$$\texttt{<}\tau\texttt{_0_0, }\tau\texttt{_1_0 where }\tau\texttt{_0_0 == }\tau\texttt{_1_0.Element, }\tau\texttt{_1_0: Sequence>}$$

The return type of `sequenceOfT(true)`, and hence the argument type of `pick()`, is the opaque archetype $[\![\circlearrowleft \tau\texttt{_1_0}]\!] \otimes \Sigma$. Hence, we can obtain the return type of `pick()` by projecting the type witness for $\langle\texttt{Sequence}|\texttt{Element}$ from the conformance $[([\![\circlearrowleft \tau\texttt{_1_0}]\!] \otimes \Sigma)\texttt{: Sequence}]$.

To get this type witness, we map `τ_1_0.Element` into the opaque generic environment (d, Σ), and to do that, we must first compute $\texttt{getReducedType}(O, \tau\texttt{_1_0.Element})$. The result is `τ_0_0`, which is an outer type parameter of d, so we end up in Step 5 of Algorithm 13B. We apply Σ to `τ_0_0` and arrive at our answer:

$$\begin{aligned}&\langle\texttt{Sequence}|\texttt{Element} \otimes [([\![\circlearrowleft \tau\texttt{_1_0}]\!] \otimes \Sigma)\texttt{: Sequence}]\\ &\quad= \tau\texttt{_0_0} \otimes \Sigma\\ &\quad= \texttt{Bool}\end{aligned}$$

Algorithm 13B assumes that if a type parameter `T` is equivalent to some outer type parameter, then `getReducedType`$(O,$ `T`$)$ will always be an outer type parameter. For this to hold, outer type parameters must come first in the type parameter order. However, Algorithm 5D does not have this property, because we compare type parameters by length first; we can have $|\texttt{T}| > |\texttt{U}|$ with `T` being outer and `U` not. This was the subject of a bug report, now fixed [137]. Before we describe the fix, let's look at an example.

Example 13.16. We use protocol `N` from Example 5.12 in the below, but in fact, any protocol that allows us to state a reduced type parameter of length > 2 will do. (Also, we take `pick()` from the previous two examples.)

```
func someSequenceOfLong<T: N>() -> some Sequence<T.A.A> {
  return Array<T.A.A>
}

print(pick(someSequenceOfLong(...)))
```

We call `someSequenceOfLong()` with some substitution map Σ, so we get the opaque archetype $[\![\circlearrowleft\ \texttt{τ_1_0}]\!] \otimes \Sigma$. As in Example 13.9, we determine the substituted type of the call to `pick()` by projecting the type witness for $\langle$`Sequence`$|$`Element` from a conformance. Notice how the generic argument of the opaque result type is `τ_0_0.A.A`, a type parameter of length 3. Here is the opaque result generic signature:

```
<τ_0_0, τ_1_0 where τ_0_0: N, τ_1_0: Sequence,
                    τ_1_0.Element == τ_0_0.A.A>
```

In the above generic signature, `τ_1_0.Element` is equivalent to the outer type parameter `τ_0_0.A.A`. Thus, to project the type witness, we apply Σ to `τ_0_0.A.A`:

$$\langle\texttt{Sequence}|\texttt{Element} \otimes [[\![\circlearrowleft\ \texttt{τ_1_0}]\!] \otimes \Sigma\texttt{: Sequence}] = \texttt{τ_0_0.A.A} \otimes \Sigma$$

Since $|\texttt{τ_1_0.Element}| = 2$ and $|\texttt{τ_0_0.A.A}| = 3$, Algorithm 5D tells us the reduced type must have length 2. If that were the case, however, type witness projection would output an opaque archetype $[\![\circlearrowleft\ \texttt{τ_1_0.Element}]\!] \otimes \Sigma$, which is incorrect. Thus, we must modify the type parameter order used in the opaque result generic signature.

Definition 13.17. In addition to storing its depth and index, a generic parameter type stores a *weight*, which is either 0 or 1. The generic parameters of an ordinary generic signature have weight 0, so in particular, the outer generic parameters of the opaque result generic signature have weight 0. Generic parameters that represent opaque result types in the opaque result generic signature have weight 1.

The *weighted type parameter order* compares weight before comparing length, so that outer type parameters (weight 0) always precede opaque result types (weight 1).

Algorithm 13C (Weighted type parameter order)**.** Takes two type parameters `T` and `U` as input. Returns one of "<", ">", or "=" as output.

1. Let $\texttt{T}'$ and $\texttt{U}'$ denote the root generic parameters of `T` and `U`, respectively.
2. If $\texttt{T}'$ has weight 0 and $\texttt{U}'$ has weight 1, return "<".
3. If $\texttt{T}'$ has weight 1 and $\texttt{U}'$ has weight 0, return ">".
4. Otherwise both have equal weight. Compare `T` and `U` with Algorithm 5D.

We claimed that the type parameter order was well-founded in Proposition 5.25. This remains true with the weighted order above. (Suppose we are given a non-empty set of type parameters; we can produce a minimum element as follows. If at least one type parameter in this set has weight 0, we can discard any elements of weight 1, because they cannot be smaller than this element. Now, since all remaining elements have equal weight, the minimum element of the remaining set under the original type parameter order, is in fact the minimum element of the original set under the modified order.)

We close this section with two minor implementation limitations.

Example 13.18. An opaque result type cannot be nested inside the constraint type of another, so today, one cannot write this:

```
func unsupportedNesting() -> some Sequence<some Equatable> {...}
```

In fact, the above has a clear interpretation in terms of the following opaque result generic signature:

```
<τ_0_0, τ_1_0 where τ_0_0: Sequence, τ_0_1: Equatable,
                    τ_0_1 == τ_0_0.Element>
```

This one could be resolved with relatively little effort.

Example 13.19. Unlike ordinary "input" type parameters, opaque result types are not considered by requirement inference (Section 11.1):

```
func goodInference(_: Set<some Any>) {}  // actually Hashable
func badInference() -> Set<some Any> {}  // error
```

We infer the requirement [`τ_0_0: Hashable`] when we build the generic signature of the first declaration. There is no equivalent behavior in the second declaration, so we do not infer the requirement [`τ_0_0: Hashable`] in the opaque result generic signature. The opaque archetype ⟦⟳ `τ_0_0`⟧ for the "`some Any`" does not conform to `Hashable`, so it cannot be a generic argument type for `Set`, and we diagnose an error. While extending requirement inference to the opaque result generic signature would certainly be possible, it may be undesirable from a code legibility standpoint.

13.2. Opaque Type Witnesses

Recall the discussion of type witnesses from Section 7.3, and associated type inference from Section 7.6. We will now see that if a value requirement in a protocol returns an associated type, and a candidate value witness returns an opaque archetype, associated type inference will deduce that the type witness is this opaque archetype. Therefore, an opaque archetype can witness an associated type requirement in a conformance.

Example 13.20. The normal conformance [`Horses: Sequence`] shown below witnesses the `Element` and `Iterator` associated types with a pair of opaque archetypes:

```
struct Horses: Sequence {
  func makeIterator() -> some IteratorProtocol {
    return ["Noby", "Neo"].makeIterator()
  }
}
```

Our `makeIterator()` method returns an opaque archetype $[\![\circlearrowleft \tau_0_0]\!]$. We deduce that the `Iterator` type witness is the opaque archetype it returns:

$$\langle \texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Horses: Sequence}] = [\![\circlearrowleft \tau_0_0]\!]$$

This opaque archetype conforms to `IteratorProtocol`, so it satisfies the `Sequence` protocol's associated conformance requirement with an opaque abstract conformance:

$$\begin{aligned}&\langle \texttt{Self.Iterator: IteratorProtocol}] \otimes [\texttt{Horses: Sequence}] \\ &\quad = [[\![\circlearrowleft \tau_0_0]\!]\texttt{: IteratorProtocol}]\end{aligned}$$

To get `Element`, we proceed as in Example 7.13, and consider the `Sequence` protocol's associated same-type requirement. This produces another opaque archetype:

$$\begin{aligned}&\langle \texttt{Sequence}|\texttt{Element} \otimes [\texttt{Horses: Sequence}] \\ &\quad = \langle \texttt{Sequence}|\texttt{Element} \otimes [[\![\circlearrowleft \tau_0_0]\!]\texttt{: IteratorProtocol}] \\ &\quad = [\![\circlearrowleft \tau_0_0.\texttt{Element}]\!]\end{aligned}$$

Associated type inference also synthesizes two type alias declarations, named `Iterator` and `Element`. These allow us to refer to these opaque archetypes by name. For example, `ride()` receives the opaque archetype $[\![\circlearrowleft \tau_0_0.\texttt{Element}]\!]$ as a *parameter*:

```
func ride(_: Horses.Element) {...}  // how?

for horse in Horses() {
  ride(horse)
}
```

When the candidate value witness is defined in a superclass of the conforming type, or a protocol extension of some protocol it conforms to, we must apply a substitution map to the candidate type witness, to get an interface type for the right generic signature. This is analogous to type resolution of member types in Section 9.2. For the next two examples, we will declare conformances to this protocol:

```
protocol P {
  associatedtype A
  func f() -> A
}
```

Example 13.21. Here, we have a default witness in a protocol extension:

```
extension P {
  func f() -> some Any {...}  // underlying type can depend on 'Self'
}

struct S: P {}
```

The `P.f()` default witness returns an opaque archetype $[\![\circlearrowleft \tau_1_0]\!]$ whose outer generic signature is the protocol generic signature $G_{\texttt{P}}$. We deduce that the type witness for `A` is $[\![\circlearrowleft \tau_1_0]\!]$ with the protocol substitution map $\Sigma_{[\texttt{S: P}]}$ applied:

$$\langle \texttt{P}|\texttt{A} \otimes [\texttt{S: P}] = [\![\circlearrowleft \tau_1_0]\!] \otimes \{\tau_0_0 \mapsto \texttt{S};\ [\tau_0_0\texttt{: P}] \mapsto [\texttt{S: P}]\}$$

Notice how the substitution map refers back to the normal conformance `[S: P]`.

Example 13.22. The final possibility is that the witness is in a superclass:

```
class Base<T> {
  func f() -> some Any {...}  // underlying type can depend on 'T'
}

class Derived: Base<Int>, P {}
```

The `Base.f()` class method returns an opaque archetype $[\![\circlearrowleft \tau_1_0]\!]$ whose outer generic signature is the generic signature of `Base`. We apply the superclass substitution map for `Derived` to get the type witness in the conformance of `Derived` to `P`:

$$\langle \texttt{P}|\texttt{A} \otimes [\texttt{Derived: P}] = [\![\circlearrowleft \tau_1_0]\!] \otimes \{\tau_0_0 \mapsto \texttt{Int}\}$$

By modifying the declarations of `Base` and `Derived` as appropriate, we can set the type witness to an opaque archetype with an arbitrary substitution map.

Example 13.23. If the conforming type is generic, we can substitute the type witness. Consider the [`Barn<τ_0_0>: Sequence`] normal conformance:

```
struct Barn<T>: Sequence {
  func makeIterator() -> some IteratorProtocol<T> {...}
}
```

In this conformance, the type witness for `Iterator` is the opaque archetype $[\![\circlearrowleft \texttt{τ_1_0}]\!]$ of `makeIterator()`, while the type witness for `Element` is the outer generic parameter `τ_0_0`, as in Example 13.15:

$$\langle \texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Barn<τ_0_0>: Sequence}] = [\![\circlearrowleft \texttt{τ_1_0}]\!]$$

$$\langle \texttt{Sequence}|\texttt{Element} \otimes [\texttt{Barn<τ_0_0>: Sequence}] = \texttt{τ_0_0}$$

The type witnesses of the specialized conformance [`Barn<Int>: Sequence`] are thus:

$$\langle \texttt{Sequence}|\texttt{Iterator} \otimes [\texttt{Barn<Int>: Sequence}] = [\![\circlearrowleft \texttt{τ_1_0}]\!] \otimes \{\texttt{τ_0_0} \mapsto \texttt{Int}\}$$

$$\langle \texttt{Sequence}|\texttt{Element} \otimes [\texttt{Barn<Int>: Sequence}] = \texttt{Int}$$

Example 13.24. An opaque archetype cannot witness an associated type requirement if its owner declaration has a generic parameter list:

```
protocol GenericP {
  associatedtype A
  func f<T>(_: T) -> A
}

struct Bad: GenericP {  // error
  func f<T>(_: T) -> some Any {...}
}
```

This is not an implementation limitation, but rather a consequence of the language semantics. A type witness must be an interface type for the generic signature of its conformance. However, an opaque result type is parameterized by the generic signature of its owner declaration. The two no longer coincide if the owner declaration introduces generic parameters. There is an analogous situation with nested nominal types:

```
protocol P {
  associatedtype A
}

struct S: P {  // error
  struct A<T> {}
}
```

Textual interfaces. When building a shared library for distribution, we generate a textual interface file, by printing each declaration in the module with the AST printer (Section 2.5). The interface file includes all synthesized declarations, and in particular, type aliases synthesized by associated type inference. We are now faced with a new dilemma. The underlying type of a synthesized type alias declaration may reference an existing opaque archetype. However, the Swift language does not have a syntax for referencing an opaque archetype; the `some` keyword *declares* an opaque archetype.

We solve this problem with a special syntax, permitted only in textual interface files. This syntax encodes a reference to an opaque archetype $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ in terms of the `mangling` of the owner declaration of d, and the `index` of the root generic parameter of `T`. If Σ is empty, and `T` is just a generic parameter type, this looks like so:

```
@_opaqueReturnTypeOf("mangling", index) __
```

If `T` is a dependent member type, we wrap it in member type representation syntax:

```
(@_opaqueReturnTypeOf("mangling", index) __).Element
```

If the owner declaration of d is generic, then Σ is non-empty. We print the replacement types of Σ (say `X`, `Y`, `Z`) in a generic argument list:

```
@_opaqueReturnTypeOf("mangling", index) __<X, Y, Z>
(@_opaqueReturnTypeOf("mangling", index) __<X, Y, Z>).Element
```

As this syntax is not meant for human consumption, it is buried within the "`@foo`" type attribute grammar, which also simplifies the implementation. Usually, a type attribute modifies the immediately following type representation, like `@escaping` in front of a function type for example, but in the case of `@_opaqueReturnTypeOf`, the attribute itself entirely specifies the type, and the type representation that follows is not used, except for its generic arguments. Indeed, "`__`" can be any valid identifier, and the AST printer actually used an emoji prior to Swift 5.5, however this innovation was removed [138].

Example 13.25. We can generate a textual interface for Example 13.20 by making the declarations `public` and invoking `swiftc` with the `-enable-library-evolution` and `-emit-module-interface` flags. Here is part of the output, with line breaks added:

```
public struct Horses : Swift.Sequence {
  public func makeIterator() -> some Swift.IteratorProtocol

  public typealias Element =
    (@_opaqueReturnTypeOf("$s5horse6HorsesV12makeIteratorQryF", 0) __)
      .Element
  public typealias Iterator =
    @_opaqueReturnTypeOf("$s5horse6HorsesV12makeIteratorQryF", 0) __
}
```

We use the "some" syntax when we print out the return type of `Horse.makeIterator()`, because this denotes the declaration of a new opaque result type. The underlying types of our two type aliases, on the other hand, use the `@_opaqueReturnTypeOf` syntax to refer to existing opaque archetypes. Note that "`$s5horse6HorsesV12makeIteratorQryF`" is the mangled name of our `makeIterator()` method.

Example 13.26. The textual interface for Example 13.21 looks like this, if we make the declarations public, and set the module name to "p":

```
public protocol P {
  associatedtype A
  func f() -> Self.A
}
extension p.P {
  public func f() -> some Any

}
public struct S : p.P {
  public typealias A =
    @_opaqueReturnTypeOf("$s1p1PPAAE1fQryF", 0) __<p.S>
}
```

Notice how we refer to the opaque archetype with a substitution map. Let's use the `swift-demangle` tool to print out the mangled name of the owner declaration:

```
$ swift-demangle s1p1PPAAE1fQryF
s1p1PPAAE1fQryF ---> (extension in p):p.P.f() -> some
```

The `-expand` flag prints the mangled name's structure in more detail. Try it.

Example 13.27. Finally, here is the textual interface for Example 13.22:

```
public class Base<T> {
  public init()
  public func f() -> some Any
}
@_inheritsConvenienceInitializers
public class Derived : p.Base<Swift.Int>, p.P {
  override public init()
  public typealias A =
    @_opaqueReturnTypeOf("$s1p4BaseC1fQryF", 0) __<Swift.Int>
}
```

To support the `@_opaqueReturnTypeOf` syntax in type resolution, we introduce some bookkeeping in the parser when parsing a textual interface file, to collect all opaque result declarations in a per-source file list. We also maintain a lookup table, which is initially empty, to map the mangled names of owner declarations back to themselves. This table is populated on the first invocation of the below algorithm.

Algorithm 13D (Resolve opaque archetype)**.** As input, takes a mangled name s, an integer `i`, and an optional list of generic argument types. Returns the corresponding opaque archetype.

1. If the list of parsed opaque result declarations is empty, go to Step 3.
2. Otherwise, remove the next opaque result declaration from this list. Invoke the mangler to construct the mangled name of its owner declaration, which will trigger various requests, such as the **interface type request**. Add an entry to the lookup table to associate the mangled name with the owner declaration. Go back to Step 1.
3. Look up s in the table to get the opaque result declaration d.
4. Form a generic parameter type `τ_d_i`, where `d` is the maximum depth of the opaque result generic signature of d, and `i` is the index input to the algorithm.
5. Let G be the outer generic signature of d. If G is non-empty, we must have a list of generic arguments. Form a substitution map Σ for G, from these generic arguments, using global conformance lookup to populate the substitution map's conformances. Otherwise, if G is empty, let Σ be the empty substitution map.
6. Call Algorithm 13B to map `τ_d_i` into the opaque generic environment (d, Σ).
7. Return this opaque archetype $[\![\circlearrowleft \texttt{τ_d_i}]\!]_d \otimes \Sigma$.

13.3. Runtime Representation

Back in Chapter 1, we learned that Swift implements separate compilation of generic functions by encoding the function's generic signature in the calling convention. The caller constructs a runtime representation of each replacement type and conformance in the substitution map, and the callee manipulates generic values abstractly, using the metadata and witness tables provided by the caller. The implementation of opaque result types is analogous, but "backwards."

The caller of a function with an opaque result type must manipulate the resulting value abstractly, using runtime type metadata. To this end, the compiler emits an *opaque type descriptor* when compiling the callee. The opaque type descriptor references the runtime type metadata for the opaque archetype's underlying type.

We will first define the underlying type of an opaque archetype in a precise way. To simplify the discussion, we only consider the case where the underlying type of an opaque result type does not depend on availability, so there is only one underlying type substitution map. Say we have an opaque result declaration d with opaque result generic signature O, underlying type substitution map Σ', and outer generic signature G.

Definition 13.28. Let $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ be an opaque archetype. The *underlying type* of $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ is the following substituted type:

$$\texttt{T} \otimes \Sigma' \otimes \Sigma$$

If $\Sigma \in \textsc{Sub}(G, H)$, then $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma \in \textsc{Type}(H)$, by definition. As we expect, the underlying type of $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ is also an element of $\textsc{Type}(H)$. Indeed:

$$\texttt{T} \otimes \Sigma' \in \textsc{Type}(O)$$

$$\texttt{T} \otimes \Sigma' \otimes \Sigma \in \textsc{Type}(H)$$

Analogously, if $\Sigma \in \textsc{Sub}(G, [\![H]\!])$, then the underlying type of $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ is an element of $\textsc{Type}([\![H]\!])$.

Definition 13.29. We also define the *underlying conformance* of an opaque abstract conformance $[([\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma)\texttt{: P}]$ as follows:

$$[\texttt{T: P}] \otimes \Sigma' \otimes \Sigma$$

That is, we apply the underlying type substitution map to the abstract conformance for the type parameter `T` of our opaque archetype. Once again, by the definition of substitution map composition, if $[([\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma)\texttt{: P}] \in \textsc{Conf}(H)$, then its underlying conformance is also an element of $\textsc{Conf}(H)$. (Likewise for $\textsc{Conf}([\![H]\!])$.)

Example 13.30. Consider this function:

```
func someSequence2<T>(_ t: T) -> some Sequence {
  return [t]
}
```

Here is the underlying type substitution map:

$$\Sigma' := \{\texttt{τ_0_0} \mapsto \texttt{τ_0_0}$$

$$\texttt{τ_1_0} \mapsto \texttt{Array<τ_0_0>};$$

$$[\texttt{τ_1_0: Sequence}] \mapsto [\texttt{Array<τ_0_0>: Sequence}]\}$$

Now, let $\Sigma := \{\texttt{τ_0_0} \mapsto \texttt{Int}\}$ be a substitution map for the outer generic signature `<τ_0_0>`. The underlying type of $[\![\circlearrowleft \texttt{τ_1_0}]\!] \otimes \Sigma$ is the following:

$$\texttt{τ_1_0} \otimes \Sigma' \otimes \Sigma = \texttt{Array<τ_0_0>} \otimes \Sigma = \texttt{Array<Int>}$$

The underlying type of $[\![\circlearrowleft \texttt{τ_1_0.Element}]\!] \otimes \Sigma$ is:

$$\texttt{τ_1_0.Element} \otimes \Sigma' \otimes \Sigma = \texttt{τ_0_0} \otimes \Sigma = \texttt{Int}$$

Runtime entry points. The Swift runtime exports a pair of entry points that receive an opaque type descriptor, and project the type metadata and witness tables for the replacement types and conformances of its underlying type substitution map:

- `swift_getOpaqueTypeMetadata2()` takes an opaque type descriptor and the index of an opaque result type in the opaque result generic signature, and returns type metadata for the corresponding underlying type.
- `swift_getOpaqueTypeConformance2()` takes an opaque type descriptor and the index of a conformance requirement in the opaque result generic signature, and returns the witness table for the corresponding underlying conformance.

Since the underlying type substitution map depends on the outer generic signature, each runtime entry point also takes the full set of type metadata and witness tables that would be passed to a generic function with the same signature. Thus, the caller of a function with an opaque result type must invoke these entry points with the same substitution map that was used for the call. This will yield type metadata and witness tables describing the opaque archetype, and from that point on, instances of this opaque archetype can be manipulated like instances of ordinary type parameters.

Example 13.31. We continue with Example 13.30. Let's call our `someSequence2()` function as follows:

```
func pick<S: Sequence>(_ s: S) -> S.Element {...}

print(pick(someSequence2(123)))
```

The entry point for `someSequence2()` takes three parameters: a buffer to hold the return value of type $[\![\circlearrowleft \tau_1_0]\!]$, the type metadata for τ_0_0 which here is `Int`, and a pointer to a value of type τ_0_0.

To determine the size of the return value buffer, we generate a call to the appropriate Swift runtime entry point, handing it the opaque type descriptor for `someSequence2()`, and the type metadata for τ_0_0. At run time, this outputs type metadata for the underlying type, which is `Array<Int>`. We recover the type's size from its metadata, and generate a dynamic stack allocation with the result.

We then hand the result of `someSequence2()` to `pick()`. The entry point for `pick()` takes four parameters: a buffer for the return value of type τ_0_0`.Element`, the type metadata for τ_0_0, and a witness table for $[\tau_0_0$`: Sequence`$]$. For the latter, we must pass in the witness table for $[[\![\circlearrowleft \tau_1_0]\!] \otimes \Sigma$`: Sequence`$]$. To obtain this witness table, we generate a call to the other Swift runtime entry point, again handing it the opaque type descriptor, and the type metadata for `Int`. This runtime entry point outputs the witness table for the underlying conformance, [`Array<Int>: Sequence`].

Finally, to abstractly manipulate the return value of `pick()`, and in particular, to recover the size of a `τ_0_0.Element`, we generate a call to the metadata access function for the `Element` associated type in the witness table for $[[\circlearrowleft \texttt{τ_1_0}] \otimes \Sigma\texttt{: Sequence}]$.

Specialization. This implementation strategy is resilient to library evolution. If the callee is in a shared library and the callee links against this library, we can freely change the callee's underlying type. As long as the opaque result generic signature does not change, the layout of the generated opaque type descriptor remains the same, and binary compatibility with the caller is maintained.

On the other hand, there are many situations where the caller and callee are always compiled together—for example, if they're declared in the same source file. In this case, the type checker must continue to maintain the illusion that the underlying type of an opaque archetype is hidden from the caller. However, in code generation, we can avoid the abstraction penalty imposed by runtime type metadata, instead manipulating values of the opaque archetype as if they were the underlying type.

Recall that we use a similar implementation strategy for ordinary generic functions. Separate compilation is the default, but in the case where the function's body is visible to the caller, the SIL optimizer can generate a specialization of the function from the caller's substitution map.

In fact, the situation with opaque result types is somewhat simpler, because an opaque archetype only has one underlying type. Instead of a separate optimizer pass, we perform this replacement in SILGen while lowering a type checked AST to SIL instructions. To be precise, this replacement happens as part of SIL type lowering (Section 6.5).

Definition 13.32. Suppose that f is the function declaration currently being lowered, and $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$ is some opaque archetype referenced from f. Formally, the underlying type substitution map of d is *visible* from f if either of the following holds:

1. The owner declaration of d is in another module, and either:
   a) the owner declaration is `@inlinable`, or
   b) this other module was built without library evolution.
2. The owner declaration of d is in a primary file of the main module, and either:
   a) f itself is *not* `@inlinable`, or
   b) the owner declaration of d is also `@inlinable`.

In whole module mode, every source file is primary, so the optimization is most effective in that case. The `@inlinable` restriction ensures we avoid leaking implementation details when we serialize the SIL representation of an `@inlinable` function as part of a binary module (Section 2.5). This serialized representation cannot depend on the underlying types of any non-`@inlinable` functions, even within the same module.

When the above visibility condition holds, we can obtain the underlying type substitution map Σ' for d, and safely assume that the underlying type of $[\![\circlearrowleft \texttt{T}]\!]$ will always be equal to $\texttt{T} \otimes \Sigma'$ inside f. However, there is one more thing to check.

The *access control* keywords (`fileprivate`, `internal`, `public`) determine not only the visibility of declarations from name lookup at compile time, but also the visibility of symbols in the generated object file. In particular, we can only replace an opaque archetype with its underlying type if every nominal type appearing in the underlying type is visible. For example, if our opaque result type d is declared in another source file and its underlying type involves a `private struct`, we cannot directly reference the underlying type, even if we are allowed to know what this type is.

A *type expansion context* collects the input data for the visibility check. It consists of the parent declaration context of f, paired with a flag indicating if f is `@inlinable`.

The three mutually-recursive algorithms below are used by SIL type lowering to replace opaque archetypes appearing within types, conformances, and substitution maps.

Algorithm 13E (Specialize opaque archetypes within a type)**.** Receives a type expansion context and a type.

- For an **opaque archetype** $[\![\circlearrowleft \texttt{T}]\!]_d \otimes \Sigma$:
  1. If its underlying type is visible, substitute in underlying type, and recurse if this underlying type further contains opaque archetypes.
  2. Otherwise, recursively transform Σ and form a new opaque archetype.
- For an **existential archetype** $[\![\exists \texttt{T}]\!] \otimes \Sigma$: recursively transform Σ. (We will meet these in Section 14.1.)
- For **every other type** `X`: recursively transform the child types of `X` if any, and construct a new type with the transformed children.

Algorithm 13F (Specialize opaque archetypes within a conformance)**.** Receives a type expansion context and a conformance.

- For a **specialized conformance** $[\texttt{X}\colon\ \texttt{P}] \otimes \Sigma$: recursively transform Σ and return a new specialized conformance.
- For an **opaque abstract conformance** $[([\![\circlearrowleft \texttt{T}]\!] \otimes \Sigma)\colon\ \texttt{P}]$:
  1. If its underlying conformance is visible, substitute in the underlying conformance. Recurse if the conforming type again contains opaque archetypes.
  2. Otherwise, recursively transform Σ.
- For an **existential abstract conformance** $[([\![\exists \texttt{T}]\!] \otimes \Sigma)\colon\ \texttt{P}]$: recursively transform Σ and form a new existential abstract conformance.

Algorithm 13G (Specialize opaque archetypes within a substitution map)**.** Receives a type expansion context and substitution map.

- Recursively transform each replacement type and conformance with Algorithm 13E and Algorithm 13F, and form a new substitution map.

Circularity. We encountered non-terminating compile-time computation in our prior discussion of conditional and recursive conformances (Section 10.4, Section 12.3). This issue can also arise with opaque result types, because nothing in our discussion so far precludes the possibility of an opaque result type whose underlying type is defined in terms of itself. If this were to happen, our algorithm for replacing an opaque archetype with its underlying type would run forever, if implemented as described. However, in reality, we attempt to detect this and diagnose an error, as we will see now.

Example 13.33. The simplest example is when a function with an opaque result type calls itself on all control flow paths. Here, the opaque result types of `f1a()` and `f1b()` are defined in terms of each other:

```
func f1a() -> some Any {
  return f1b()
  // error: function opaque return type was inferred as ‘some Any’,
  // which defines the opaque type in terms of itself
}

func f1b() -> some Any {
  return f1a()
  // error: function opaque return type was inferred as ‘some Any’,
  // which defines the opaque type in terms of itself
}
```

Example 13.34. More complex recursion is possible, where the underlying type contains the opaque result type in structural position. Here, the underlying type of our opaque result type ⟦⟲ T⟧ is `Array<`⟦⟲ T⟧`>`:

```
func f2() -> some Any {
  return [f2()]
  // error: function opaque return type was inferred as ‘[some Any]’,
  // which defines the opaque type in terms of itself
}
```

Beyond these simple examples, the technique in Section 12.4 of encoding a tag system in terms of type substitution can be modified slightly to use opaque result types instead,

so in fact, opaque archetype specialization is Turing-complete, and it an undecidable in general if this process terminates. Thus, the only way to prevent non-terminating computation is to impose an upper bound on the total amount of work we can do, and signal an error if this limit is breached:

1. To catch most cases, we eagerly attempt to fully specialize every declared opaque result type from the **type-check primary file request**. If this happens, we emit a diagnostic, as shown with the two examples above.

2. In more complex scenarios, a circular opaque result type only becomes apparent in the SIL optimizer, after some number of inlining and specialization passes have run. In this case, we print a fatal error message and stop compilation.

In the future, it would be nice to diagnose an error with a useful source location in the second scenario as well. However, even this cannot catch everything. With separate compilation, one can always define mutually-recursive opaque result types which do not become apparent until run time. In this case, the program will crash with infinite recursion inside the runtime type metadata instantiation logic.

Finally, while the default limits should be sufficient for all reasonable programs, it is possible to change them with a pair of frontend flags:

- `-max-substitution-depth` is the maximum number of recursive calls, in the sense where the underlying type of an opaque archetype involves another opaque archetype, which involves another, and so on. The default value is 500.

- `-max-substitution-count` is the total number of opaque archetypes that can be visited within a single type before we give up. The default value is 120,000.

13.4. Source Code Reference

Key source files:

- `lib/Sema/MiscDiagnostics.cpp`

- `lib/Sema/TypeCheckGeneric.cpp`

`OpaqueTypeDecl`

An opaque result declaration.

- `getNamingDecl()` returns this declaration's owner declaration.

- `getOpaqueInterfaceGenericSignature()` returns this declaration's opaque result generic signature.

- `getUniqueUnderlyingTypeSubstitutions()` returns this declaration's underlying type substitution map, if it does not depend on availability.

  Returns `nullopt` if the substitution map not been computed yet, or if it cannot be computed because the owner declaration does not have a body.

- `getConditionallyAvailableSubstitutions()` is the general form for when we have more than one availability range and underlying type substitution map.

`ValueDecl::getOpaqueTypeDecl()` *method*

Returns this declaration's opaque result declaration if it has one, or `nullptr` otherwise. See also Section 4.6.

`OpaqueResultTypeRequest` *class*

A request for constructing opaque result type declarations. This request is evaluated by calling `ValueDecl::getOpaqueTypeDecl()` above.

`OpaqueUnderlyingTypeChecker` *class*

An AST walker to collect the `return` statements inside a function body and fill in its opaque result declaration's underlying type substitution map. Also diagnoses errors, such as multiple `return` statements with mismatched return types. The type checker performs this walk after assigning types to expressions.

Opaque Archetypes

Key source files:

- `include/swift/AST/GenericEnvironment.h`
- `lib/AST/GenericEnvironment.cpp`

`TypeBase::hasOpaqueArchetype()` *method*

Returns true if this type contains an opaque archetype. See also Section 3.4.

`OpaqueTypeArchetypeType` *class*

Subclass of `ArchetypeType` representing an opaque archetype.

Recall from Section 8.5 that every archetype has a `getInterfaceType()` method that returns its type parameter, and a `getGenericEnvironment()` method that returns its parent generic environment. Opaque archetypes have two more accessor methods:

- `getOpaqueDecl()` returns the opaque result declaration for this opaque archetype.
- `getSubstitutions()` returns this opaque archetype's substitution map.

`GenericEnvironment` *class*

See also Section 8.5.

- `forOpaqueType()` is a static factory method that returns the unique opaque generic environment for the given opaque result declaration and substitution map.
- `getKind()` returns `GenericEnvironment::Kind::Opaque` for an opaque generic environment. The generic environment of an opaque archetype is always one.
- `getOpaqueTypeDecl()` returns the opaque result declaration for this environment.
- `getOuterSubstitutions()` returns this generic environment's substitution map.

Opaque Type Witnesses

Key source files:

- `lib/AST/SourceFile.cpp`
- `lib/Sema/TypeCheckType.cpp`

`TypeResolver::resolveOpaqueReturnType()` *method*

This is Algorithm 13D, to resolve the special `@_opaqueReturnTypeOf` syntax.

`SourceFile` *class*

See also Section 2.6.

- `addUnvalidatedDeclWithOpaqueResultType()` records a declaration as having an opaque result type in the per-source file list. Invoked by the parser.
- `getOpaqueReturnTypeDecls()` walks this above list, constructs each opaque result declaration, and populates the per-source file lookup table with the mangled name of each owner declaration. This is Step 1 and 2 of Algorithm 13D.
- `lookupOpaqueResultType()` returns an opaque result declaration for the given mangled name. This is Step 3 of Algorithm 13D.

AST Demangler

Key source files:

- `include/swift/AST/ASTDemangler.h`
- `lib/AST/ASTDemangler.cpp`

We didn't say it in Section 13.2, but the mangled name in a `@_opaqueReturnTypeOf` may refer to an opaque archetype from another module. In this case, we do not consult the per-source file list. Instead, we query the "AST demangler."

`ASTBuilder::resolveOpaqueType()` *method*

Calls `lookupOpaqueResultType()` on the appropriate `SourceFile` if the mangled name refers to the main module, otherwise it takes a different path.

Runtime Representation

Key source files:

- `lib/IRGen/GenMeta.cpp`
- `stdlib/public/runtime/MetadataLookup.cpp`

`IRGenModule::emitOpaqueTypeDecl()` *method*

Emits an opaque type descriptor.

`swift_getOpaqueTypeMetadata2()` *function*

Runtime entry point to construct runtime type metadata for a replacement type in an opaque type descriptor's underlying type substitution map.

`swift_getOpaqueTypeConformance2()` *function*

Runtime entry point to construct a witness table for a conformance in an opaque type descriptor's underlying type substitution map.

Specialization

Key source files:

- `include/swift/AST/Type.h`
- `lib/AST/TypeSubstitution.cpp`

`TypeExpansionContext` *class*

A data type to encode a declaration context together with the `@inlinable` flag.

`swift::substOpaqueTypesWithUnderlyingTypes()` *function*

The three overloads of this function, for types, conformances, and substitution maps, implement Algorithm 13E, Algorithm 13F, and Algorithm 13G. Each overload also takes a `TypeExpansionContext`.

14. Existential Types

14.1. Existential Archetypes

Algorithm 14A (Apply substitution map to existential archetype)**.** Hello.

14.2. Runtime Representation

14.3. Self-Conforming Protocols

14.4. Source Code Reference

Part IV.

The Requirement Machine

15. Basic Operation

The final part of this book is devoted to the implementation of generic signature queries (Section 5.6) and requirement minimization (Section 11.4). Our goal will be to understand a *decision procedure* for the derived requirements formalism. We will learn how to translate the requirements of each generic signature and protocol into *rewrite rules*, and then we will analyze a certain relation generated by these rules.

We use the proper noun "the Requirement Machine" to mean the compiler component responsible for this reasoning, while *a* requirement machine—a common noun—is an *instance* of a data structure containing those rewrite rules that describe a specific generic signature or protocol.

In this chapter, we will gain a high-level overview of how requirement machines operate, without saying anything about what goes on inside. Understanding their inner workings will take two more chapters. Chapter 16 introduces the theory of finitely-presented monoids and string rewriting, and Chapter 17 details the translation of requirements into rewrite rules.

Let's go. There are two principal entry points into the Requirement Machine:

- To **answer a generic signature query**, we first construct a requirement machine from the requirements of the given generic signature; we call this a *query machine*. We then consult the query machine's *property map*: a description of all conformance, superclass, layout, and concrete type requirements imposed on each type parameter.
- To **build a new generic signature**, we construct a requirement machine from a list of user-written requirements; we call this a *minimization machine*. We get the new signature's minimal requirements as a side-effect of the construction process.

In addition, we have two more entry points related to protocol declarations:

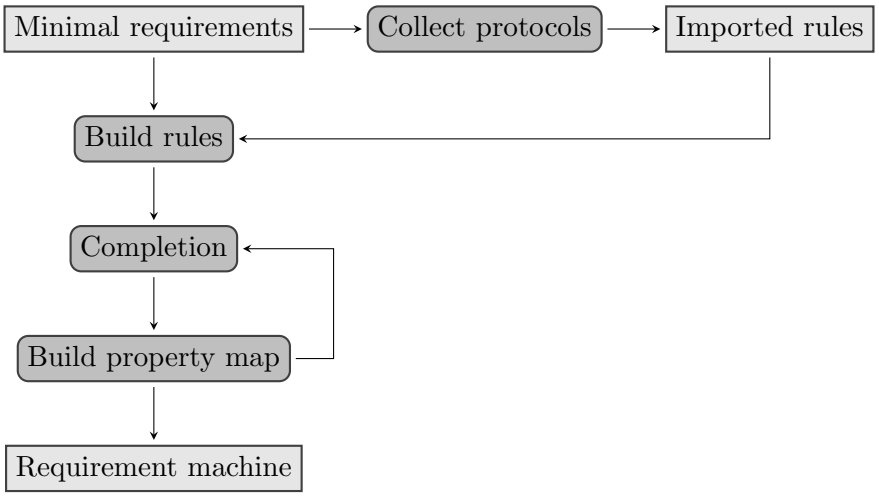
- To construct a query or minimization machine for a generic signature that states a conformance requirement, we first build a requirement machine for the protocol's requirement signature; we call this a *protocol machine*.
- To **build a new requirement signature** for a protocol written in source, we construct a requirement machine from the list of user-written requirements; we call this a *protocol minimization machine*. We compute the minimal associated requirements as a side-effect of the construction process.

A requirement machine's rewrite rules are divided into *local rules*, corresponding to the initial requirements the machine was built from, and *imported rules*, which as we see shortly, describe associated requirements of protocols referenced by the local rules. We now look at each of the four requirement machine kinds in detail.

Query machines. We maintain a table of *query machine* instances inside a singleton object, called the *rewrite context*. We can use *canonical* generic signatures as keys, because the translation of requirements into rewrite rules ignores type sugar. Once built, a query machine remains live for the duration of the compilation session. We build a query machine from a generic signature like so:

1. We translate the generic signature's explicit requirements into rewrite rules using Algorithm 17J to get the list of local rules for our query machine.
2. We lazily construct a *protocol machine* for each protocol appearing on the right-hand side of a conformance requirement in our signature.
3. We use Algorithm 15B to collect local rules from each of these protocol machines as well as all protocol machines they transitively reference, to get the list of imported rules for our query machine.
4. We run the *completion procedure* (Chapter 18) to add new local rules which are "consequences" of other rules. This gives us a *convergent rewriting system*.
5. We build the property map data structure (Chapter 19).
6. If property map construction added new local rules, we return to Step 4 and repeat completion and property map construction again.

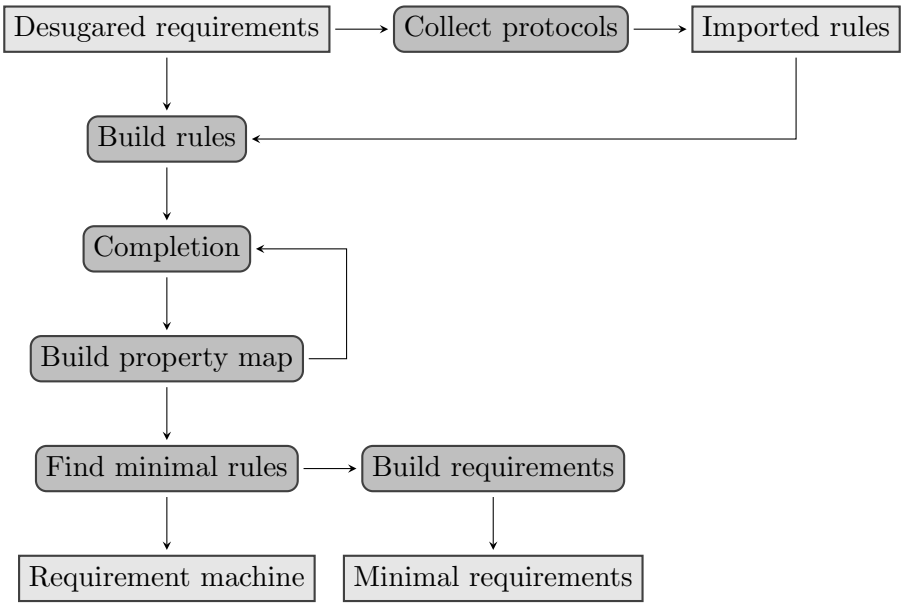
Here is the flowchart:



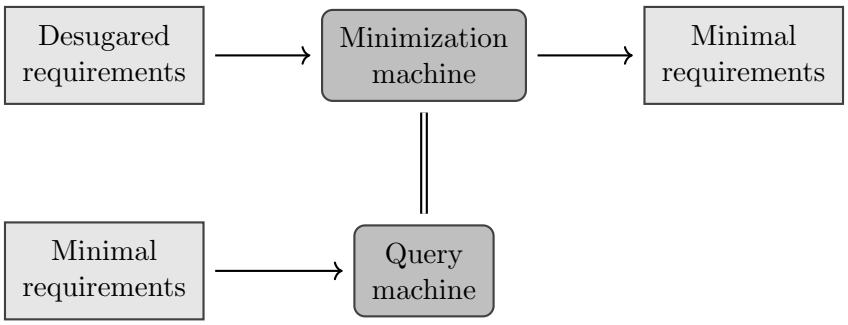
Minimization machines. To compute the list of minimal requirements for a new generic signature, we build a *minimization machine*. This is the final step in Figure 11.1 and Figure 11.2 of Chapter 11. Minimization machines have a temporary lifetime owned by the **inferred generic signature request** or **abstract generic signature request**. The flow is similar to building a query machine, except now the inputs are user-written requirements after desugaring (Section 11.2). The other main differences are:

1. Completion must do additional work if the input requirements contain unbound dependent member types, as they do when they come from the structural resolution stage.
2. Completion also records *rewrite loops*, which describe relations between rewrite rules—in particular, this describes which rules are redundant because they are a consequence of existing rules.
3. When constructing the property map, we note any conflicting requirements, to be diagnosed if this generic signature has a source location.
4. After completion and property map construction, we process the rewrite loops to find a minimal subset of local rules which imply the rest. This is the topic of Chapter 20.
5. Finally, we translate the minimal rules into requirements to get the list of minimal requirements that we output. This is explained in Section 20.4.

Here is the flowchart:



An optimization. When building a new generic signature, we get the list of minimal requirements as a *side effect* of constructing the minimization machine. While we could just discard the minimization machine once we're done, we first check the rewrite context for the presence of a query machine for the same generic signature. If we have not built one yet, we *install* the new minimization machine, transferring ownership of it to the rewrite context. This saves effort in the case where we build a new generic signature for a declaration written in the main module and then proceed to issue queries against the new generic signature while type checking the declaration's body.



For this optimization to work, the requirements as the user wrote them must generate the same theory, in the sense of Proposition 11.10, as the minimal requirements *output* by the minimization machine. This equivalence almost always holds true, except for four situations we describe below. The first three only occur with invalid code, and are accompanied by diagnostics. The fourth is not an error, just a rare edge case where the optimization cannot be performed:

1. Minimization always outputs a list of well-formed requirements, meaning they only refer to valid type parameters (Section 11.3). If any of the user-written requirements are not well-formed, we drop them, so we output a list of minimal requirements that generates a different theory.

2. When two requirements are in conflict such that both cannot be simultaneously satisfied (Definition 11.13), we sometimes drop one or the other. A new query machine will, again, not include the conflicting requirements.

3. As we will see in Section 16.4, completion may fail if the user-written requirements are too complex to reason about. In this case the minimization machine outputs an empty list of minimal requirements.

4. If a conformance requirement is made redundant by a same-type requirement that fixes a type parameter to a concrete type (such as [`T: P`] and [`T == X`] where `X` is a concrete type conforming to `P`), we drop the conformance requirement, but this changes the theory; we will talk about this in Section 19.4.

If any of these conditions hold, we record this fact while building the minimization machine. This prevents the minimization machine from being installed into the rewrite context, forcing us to discard it instead. The `-disable-requirement-machine-reuse` frontend flag is intended for debugging. It disables this optimization entirely, forcing us to discard all minimization machines immediately without attempting to install them.

Protocol machines. A *protocol machine* collects the associated requirements of a *protocol component*, or a set of mutually-dependent protocol declarations. For now, we proceed as if each component consists of a single protocol, which is the typical scenario; the next section describes the general case.

Protocol machines have a global lifetime and are owned by the rewrite context. The procedure for building a protocol machine is similar to building a query machine:

1. We translate the associated requirements of each protocol into rewrite rules, using Algorithm 17K. These become the protocol machine's local rules.

2. We lazily construct a protocol machine for each protocol that appears on the right-hand side of an associated conformance requirement.

3. We use Algorithm 15B to collect local rules from each of these protocol machines as well as all protocol machines they transitively reference, to get our imported rules. Thus, protocol machines recursively import rules from other protocol machines.

4. Completion and property map construction proceed as in the query machine case.

A typical scenario is that the user's program declares an assortment of protocols, and then proceeds to reference those protocols multiple times, by stating conformance requirements on various generic types and functions. For this reason, the requirement machines for two different generic signatures may have many rewrite rules in common, if they depend on the same protocols.

Protocol machines serve as containers for these shared rules. This eliminates the overhead of repeatedly translating the associated requirements of a protocol into rewrite rules, and then running the completion procedure on these rules. Instead, when a query or minimization machine depends on a protocol, we lazily create the protocol machine for this protocol once, and *import* its rewrite rules whenever they are needed to build a requirement machine.

Protocol minimization machines. To actually build the requirement signature of a protocol written in source, we construct a *protocol minimization machine*. A protocol minimization machine has a temporary lifetime, scoped to the **requirement signature request**. The minimal requirements of the requirement signature are computed as a side-effect of constructing the protocol minimization machine.

In the absence of error conditions, we install the protocol minimization machine in the rewrite context, so that it becomes a long-lived protocol machine for the same protocol component. In fact, we only ever directly construct a protocol machine for a protocol written in source if we failed to install the protocol minimization machine. Otherwise, protocol machines are usually only built when the user's program references a protocol from a serialized module, such as the standard library.

Example 15.1. We talked about these declarations from the start of Chapter 5 most recently in Example 5.11:

```
func sameElt<S1: Sequence, S2: Sequence>(_ s1: S1, _ s2: S2)
    where S1.Element == S2.Element {...}

func sameIter<S1: Sequence, S2: Sequence>(_ s1: S1, _ s2: S2)
    where S1.Iterator == S2.Iterator {...}

func sameEltAndIter<S1: Sequence, S2: Sequence>(_ s1: S1, _ s2: S2)
    where S1.Element == S2.Element,
          S1.Iterator == S2.Iterator {...}
```

We remarked that `sameElt()` and `sameEltAndIter()` have the same generic signature, while `sameIter()` is distinct from the other two. Suppose we type check the declarations in order as written above:

1. We construct a minimization machine from the requirements of `sameElt()`. We have two conformance requirements to `Sequence`, so we must first construct the protocol machine for `Sequence`. Since `Sequence` states an associated conformance requirement to `IteratorProtocol`, we also construct the protocol machine for `IteratorProtocol`.

2. Once we get the generic signature of `sameElt()`, we associate the generic signature with this minimization machine in the rewrite context. While type checking the body of `sameElt()`, we will issue queries against this generic signature, reusing the minimization machine we already built.

3. Next, we construct a minimization machine from the requirements of `sameIter()`. We import rules from the protocol machines for `Sequence` and `IteratorProtocol`, which we already built, saving some work. We get our new generic signature and install it in the rewrite context.

4. Finally, we construct a minimization machine to build the generic signature of `sameEltAndIter()`. We get the same generic signature as `sameIter()`, and we already have a query machine for it in the rewrite context. We discard the new minimization machine.

If we pass the `-debug-requirement-machine=timers` frontend flag, we can watch this happen in real time:

```
$ swiftc three.swift -Xfrontend -debug-requirement-machine=timers
+ started InferredGenericSignatureRequest @ three.swift:1:6
| + started getRequirementMachine() [ Sequence ]
| | + started getRequirementMachine() [ IteratorProtocol ]
| | + finished getRequirementMachine() in 21us: [ IteratorProtocol ]
| + finished getRequirementMachine() in 61us: [ Sequence ]
+ finished InferredGenericSignatureRequest in 204us:
<S1, S2 where S1 : Sequence, S2 : Sequence, S1.Element == S2.Element>
+ started InferredGenericSignatureRequest @ three.swift:4:6
+ finished InferredGenericSignatureRequest in 88us:
<S1, S2 where S1 : Sequence, S2 : Sequence, S1.Iterator == S2.Iterator>
+ started InferredGenericSignatureRequest @ three.swift:7:6
+ finished InferredGenericSignatureRequest in 78us:
<S1, S2 where S1 : Sequence, S2 : Sequence, S1.Iterator == S2.Iterator>
```

We will see more Requirement Machine debugging flags in Section 15.2.

Example 15.2. If `P` is any protocol, a query machine for the protocol generic signature `<Self where Self: P>` is particularly easy to build; we take the protocol machine for `P` and all protocols `P` depends on, collect all of their local rules, and then add a single local rule for [`Self: P`].

Circularity. If we encounter a type parameter subject to both a conformance and a concrete same-type requirement, say [`T: P`] and [`T == X`] where `X` is a concrete type conforming to `P`, we may look up one or more type witnesses in this conformance. Type witness projection calls into type substitution, which can perform generic signature queries. If we recursively call into the same query machine we are constructing now, we cannot answer the query, so we report a fatal error and give up.

Summary. The four machine kinds correspond to the four combinations of "domain" (generic signature or requirement signature) and "purpose" (querying an existing entity, or building a new entity of that kind):

Query machine	**Minimization machine**
Protocol machine	**Protocol minimization machine**

Notice that a protocol machine is the protocol version of a query machine, and a protocol minimization machine is the protocol version of a minimization machine. Machines in the first column are owned by the rewrite context; those in the second have a temporary lifetime, but can transition to the corresponding kind in the first column if installed in the rewrite context.

History. To continue the historical sketch of Section 8.4: the Requirement Machine shipped in Swift 5.6, where it was used for generic signature queries only. Requirement minimization was re-implemented in Swift 5.7, with the old logic available behind a flag. The `GenericSignatureBuilder` was then fully removed in Swift 5.8.

15.1. Protocol Components

In this section, we take a closer look at how generic signatures and protocols depend on other protocols, and fully explain how protocol machines work. We begin by revisiting the protocol dependency graph, from Definition 12.13. We recall that the vertices are protocol declarations, and the edges are associated conformance requirements. That is, if $[\texttt{Self.U: P}]_{\texttt{Q}}$ is some associated conformance requirement, we define:

$$\text{src}([\texttt{Self.U: Q}]_{\texttt{P}}) := \texttt{P},$$
$$\text{dst}([\texttt{Self.U: Q}]_{\texttt{P}}) := \texttt{Q}.$$

There is an alternative definition of this concept using derived requirements.

Definition 15.3. A protocol `P` *depends on* a protocol `Q` if we can derive a conformance to `Q` from the protocol generic signature $G_{\texttt{P}}$; that is, if $G_{\texttt{P}} \vdash [\texttt{Self.U: Q}]$ for some type parameter `Self.U`. We write $\texttt{P} \prec \texttt{Q}$ if this relationship holds. The set of *protocol dependencies* of `P` is the set of all protocols `Q` such that $\texttt{P} \prec \texttt{Q}$.

Proposition 15.4. The $\prec$ relation is reflexive and transitive.

Proof. For the first part, let `P` be any protocol. We can always derive $G_{\texttt{P}} \vdash [\texttt{Self: P}]$ via the explicit requirement $[\texttt{Self: P}]$ of $G_{\texttt{P}}$, therefore $\texttt{P} \prec \texttt{P}$, so $\prec$ is reflexive.

For the second part, let `P`, `Q`, and `R` be protocols such that $\texttt{P} \prec \texttt{Q}$ and $\texttt{Q} \prec \texttt{R}$. By definition, $G_{\texttt{P}} \vdash [\texttt{Self.U: Q}]$ and $G_{\texttt{Q}} \vdash [\texttt{Self.V: R}]$ for some type parameters `Self.U` and `Self.V`. By Lemma 11.8, $G_{\texttt{P}} \vdash [\texttt{Self.U.V: R}]$, where `Self.U.V` is the type parameter formed by replacing `Self` with `Self.U` in `Self.V`. Therefore, $\texttt{P} \prec \texttt{R}$. □

In fact, $\prec$ is exactly the reachability relation in the protocol dependency graph.

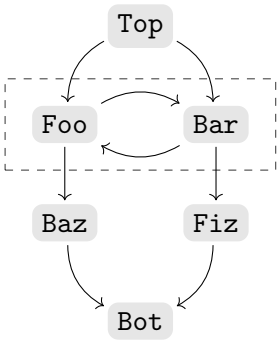
Proposition 15.5. Let `P` and `Q` be protocols. Then, $\texttt{P} \prec \texttt{Q}$ if and only if `Q` is reachable from `P` by a path in the protocol dependency graph.

Proof. A non-empty path in the protocol dependency graph is determined by the edges visited, that is, by a sequence of associated conformance requirements. This sequence of requirements has the property that each consecutive associated requirement has to be declared in the protocol named by the previous requirement. Also, when a derivation contains a sequence of consecutive AssocConf steps, the sequence of requirements referenced by each step obeys the same compatibility condition.

($\Leftarrow$) Suppose that `P` $\prec$ `Q`. By definition, there exists a type parameter `Self.U` such that $G_{\texttt{P}} \vdash$ [`Self.U: Q`]. By Theorem 12.8, we can find a type parameter `Self.U′` such that $G_{\texttt{P}} \vdash$ [`Self.U == Self.U′`], and the derivation of [`Self.U′: Q`] takes a particularly simple form: it begins with a Conf step for the explicit requirement [`Self: P`], followed by a series of AssocConf derivation steps. By the previous remark, this sequence of steps give us a path from `P` to `Q` in the protocol dependency graph.

($\Rightarrow$) Suppose there is a path from `P` to `Q` in the protocol dependency graph. Consider the sequence of visited edges. We construct a derivation of a conformance requirement in $G_{\texttt{P}}$ by starting with a Conf step for the explicit requirement [`Self: P`], and adding a series of AssocConf steps for each edge visited in our path. By the previous remark, this is a valid derivation. We see that $G_{\texttt{P}} \vdash$ [`Self.U: Q`] for some type parameter `Self.U`, showing that `P` $\prec$ `Q`. □

Recursive conformances. In Section 12.3, we said that a conformance requirement is *recursive* if it is part of a cycle in the protocol dependency graph. Listing 15.1 shows some protocol declarations. The protocol dependency graph for this example, drawn on the right, contains a cycle.



Each one of `Foo` and `Bar` points at the other via a pair of mutually-recursive associated conformance requirements. Based on our description so far, we cannot build one protocol machine without first building the other: a circular dependency.

We solve this by grouping protocols into *protocol components*, so in our example, `Foo` and `Bar` belong to the same component. A protocol machine describes an entire protocol component, and the local rules of a protocol machine include the associated requirements of all protocols in the component. If we consider dependencies between protocol *components* as opposed to *protocols*, we get a directed *acyclic* graph. To understand how these components are formed, we first consider directed graphs in the abstract.

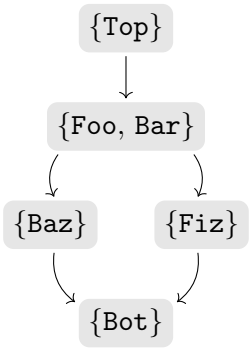
Strongly connected components. Suppose (V, E) is any directed graph, and $\prec$ is the reachability relation, so $x \prec y$ if there is a path with source x and destination y. We say that x and y are *strongly connected* if both $x \prec y$ and $y \prec x$ are true; we denote this relation by $x \equiv y$ in the following. This is an equivalence relation:

- (Reflexivity) If $x \in V$, then $x \prec x$ via the empty path at x, and thus $x \equiv x$.
- (Symmetry) If $x \equiv y$, then $y \equiv x$, by definition of $\equiv$.
- (Transitivity) If $x \equiv y$ and $y \equiv z$, then in particular $x \prec y$ and $y \prec z$, so $x \prec z$, because $\prec$ is transitive. By the same argument, we also have $y \prec x$ and $z \prec y$, hence $z \prec x$. Therefore, $x \equiv z$.

Listing 15.1.: Protocol component demonstration

```
protocol Top {
  associatedtype A: Foo
  associatedtype B: Bar
}

protocol Foo {
  associatedtype A: Bar
  associatedtype B: Baz
}

protocol Bar {
  associatedtype A: Foo
  associatedtype B: Fiz
}

protocol Baz {
  associatedtype A: Bot
}

protocol Fiz {
  associatedtype A: Bot
}

protocol Bot {}
```

The equivalence classes of $\equiv$ are called the *strongly connected components* of (V, E). We can then construct a graph whose *vertices* are the strongly connected components of the original graph; two components are joined by an edge if and only if some vertex in the first component is joined by a path with another vertex in the second component in the original graph. (This is well-defined, because if $x_1 \equiv x_2$ and $y_1 \equiv y_2$, then $x_1 \prec y_1$ if and only if $x_2 \prec y_2$.) The graph of strongly connected components is always acyclic. Furthermore, if our original graph is acyclic, we have $x \equiv y$ if and only if x and y are in fact the same vertex; in other words, the graph of strongly-connected components of a directed acyclic graph is isomorphic to the original graph itself.



Let's define the *protocol component graph* to be the graph of strongly connected components of the protocol dependency graph. The protocol component graph of Listing 15.1 is shown on the left. This looks almost the same as the original protocol dependency graph, except that we've collapsed `Foo` and `Bar` down to a single vertex. While the protocol component graph is always acyclic, it is not a tree or a forest in the general case; as we see in our example, we have two distinct paths with source `Top` and destination `Bot`, so components can share "children." To ensure we do not import duplicate rules, we must visit every downstream protocol machine once, and take only the *local* rules from each.

A debugging flag prints out each strongly connected component as it is formed. Let's try with our example:

```
$ swiftc protocols.swift -Xfrontend -debug-requirement-machine=protocol-dependencies
Connected component: [Bot]
Connected component: [Fiz]
Connected component: [Baz]
Connected component: [Bar, Foo]
Connected component: [Top]
```

The `-debug-requirement-machine=timers` frontend flag also logs the construction of protocol components. The indentation follows the recursive construction of protocol machines:

```
+ started RequirementSignatureRequest [ Top ]
| + started RequirementSignatureRequest [ Bar Foo ]
| | + started RequirementSignatureRequest [ Fiz ]
| | | + started RequirementSignatureRequest [ Bot ]
| | | + finished RequirementSignatureRequest in 46us: [ Bot ]
| | + finished RequirementSignatureRequest in 81us: [ Fiz ]
| | + started RequirementSignatureRequest [ Baz ]
| | + finished RequirementSignatureRequest in 21us: [ Baz ]
| + finished RequirementSignatureRequest in 179us: [ Bar Foo ]
+ finished RequirementSignatureRequest in 211us: [ Top ]
```

Tarjan's algorithm. We use an algorithm invented by Robert Tarjan [139] to discover the strongly connected components of the protocol dependency graph. Specifically, we number the strongly connected components, assign a component ID to each vertex, and build a table mapping each component ID to the list of vertices it contains.

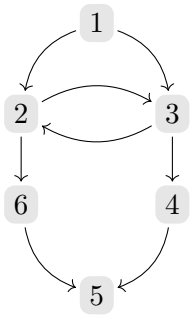
Tarjan's algorithm is incremental. We do not need to build the entire input graph upfront, which would force us to deserialize every protocol in every imported module first. When asked to produce the component ID for a given vertex, we check if the vertex has already been assigned a component ID, in which case we return it immediately. If one has not yet been assigned, the vertex must belong to a component distinct from all others discovered so far; we visit all vertices reachable from the original vertex and form connected components along the way.

Recall that if $v \in V$ is a vertex, then $w \in V$ is a *successor* of v if there exists an edge $e \in E$ with $\mathrm{src}(e) = v$ and $\mathrm{dst}(e) = w$. To find the strongly connected component of v, Tarjan's algorithm performs a *depth-first search*: given v, we recursively visit the first successor of v we haven't visited yet, and its successors, and so on, before moving on to the next successor of v. (Contrast this with the *breadth-first search* for finding a conformance path in Section 12.2, where we visit all successors of v before digging deeper into their successors.)

Tarjan's algorithm requires us to specify the input graph in the form of a callback mapping a given vertex to its list of successors. This allows the graph to be explored incrementally. In our case, the successors of a protocol are obtained by evaluating the **protocol dependencies request**, which is implemented as follows:

- If the protocol is declared inside the main module, we evaluate the **structural requirements request** and collect the protocols appearing on the right-hand side of the protocol's associated conformance requirements written in source.
- If the protocol is part of a serialized module, we evaluate the **requirement signature request** and collect the protocols appearing on the right-hand side of the protocol's associated requirements.

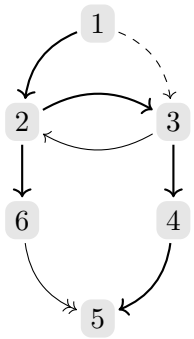
We number vertices in the order they are visited, by assigning the current value of a counter to each visited vertex v, prior to visiting the successors of v. We denote the value assigned to a vertex v by `NUMBER`(v). Our traversal can determine if a vertex has been previously visited by checking if `NUMBER`(v) is set.



We return to the protocol dependency graph from Listing 15.1. A depth-first search originating from `Top` produces the numbering of vertices shown on the right, where the first vertex is assigned the value 1. Here, the entire graph was ultimately reachable from 1; more generally, we get a numbering of the subgraph reachable from the initial vertex.

When we visit a vertex v, we look at each edge $e \in E$ with $\text{src}(e) = v$. Suppose that $\text{dst}(e)$ is some other vertex w; we consider the state of $\texttt{NUMBER}(w)$, and classify the edge e as a tree edge, ignored edge, frond, or cross-link.

If $\texttt{NUMBER}(w)$ has not yet been set, we say that e is a *tree edge*; the tree edges define a *spanning tree* of the subgraph explored by the search. Otherwise, we saw w earlier in our search, and the edge e has one of the three remaining kinds. If $\texttt{NUMBER}(w) \geq \texttt{NUMBER}(v)$, (with equality corresponding to an edge from v to itself, which we allow) any path from v to w must pass through a common ancestor of v and w in the spanning tree. We say that e is an *ignored edge*, because e cannot generate any new strongly connected components. The final two kinds arise when $\texttt{NUMBER}(w) < \texttt{NUMBER}(v)$. We say that e is a *frond* if w is also an ancestor of v in the spanning tree; otherwise, e is a *cross-link*. (While the distinction between the final two kinds plays a role in Tarjan's correctness proof, we'll see that algorithm handles fronds and cross-links uniformly.)



If the input graph is a forest, then every edge is a tree edge; we visit a subgraph that is a spanning tree of itself. If the input graph is a directed *acyclic* graph, then all edges are either tree edges or ignored edges, and once again, no two distinct vertices can be strongly connected. Tarjan's insight is that we can understand the non-trivial strongly connected components of our graph by looking at the fronds and cross-links.

The taxonomy of edges in our running example is illustrated on the left. The tree edges are drawn with a thick arrow ($\longrightarrow$), and we also have one edge of each of the remaining kinds: an ignored edge ($\dashrightarrow$), a frond ($\longrightarrow$), and a cross-link ($\twoheadrightarrow$).

The correctness of the algorithm hinges on the following property of depth-first search: any two strongly connected vertices have a common ancestor in the spanning tree, so each strongly connected component induces a *subtree* of the spanning tree. We define the *root* of a strongly connected component as the root of this subtree. To find the root of a strongly connected component, we associate another integer with each vertex v, denoted by $\texttt{LOWLINK}(v)$. This value is defined such that if $\texttt{LOWLINK}(v) = \texttt{NUMBER}(v)$, then v is the root of a strongly connected component. In our example, $\texttt{LOWLINK}(v) = \texttt{NUMBER}(v)$ for all v with the exception of "3"; we have $\texttt{LOWLINK}(3) = \texttt{NUMBER}(2)$, since "2" and "3" are in the same strongly connected component having "2" as the root.

We repeatedly update $\texttt{LOWLINK}(v)$ while visiting the successors of v, and check if $\texttt{LOWLINK}(v) = \texttt{NUMBER}(v)$ afterwards. If so, we've discovered a new strongly connected component! We maintain a pushdown stack in such a way that at this point, the strongly connected component consists of some number of vertices from the top of the stack.

We push each vertex onto the stack before visiting its successors, but we only pop vertices from the stack when forming a new strongly connected component; a vertex remains on the stack until we visit its component's root. Thus, while the pushes and pops appear unbalanced, every vertex that is pushed is eventually popped.

The algorithm also needs a cheap way to check if a vertex is currently on the stack. We associate one more piece of state with each vertex v, a single bit denoted $\texttt{ONSTACK}(v)$, to be set when v is pushed and cleared when v is popped.

Algorithm 15A (Tarjan's algorithm)**.** As input, takes a vertex v, and a callback to output the successor vertices of an arbitrary vertex reachable from v. Upon returning, v will have a component ID assigned. The algorithm updates two maps: to each vertex, we assign a component ID, and to each component ID, we assign a list of member vertices. Also uses a stack, and two global incrementing counters: the next unused vertex ID, and the next component ID.

1. (Memoize) If v already has a component ID assigned, return this component ID, which can be used to look up its members.
2. (Invariant) Otherwise, ensure that $\texttt{NUMBER}(v)$ has not been set. If it is set but v does not have a component ID, we performed an invalid re-entrant call (see below).
3. (Visit) Set $\texttt{NUMBER}(v)$ to the next vertex ID. Set $\texttt{LOWLINK}(v) \leftarrow \texttt{NUMBER}(v)$. Set $\texttt{ONSTACK}(v) \leftarrow$ true. Push $\texttt{NUMBER}(v)$ on the stack.
4. (Successors) For each successor w of v:
   a) If $\texttt{NUMBER}(w)$ is not set, we have a tree edge. Recursively invoke the algorithm on w, and set $\texttt{LOWLINK}(v) \leftarrow \min(\texttt{LOWLINK}(v), \texttt{LOWLINK}(w))$.
   b) If $\texttt{NUMBER}(w) \geq \texttt{NUMBER}(v)$, we have an ignored edge; do nothing.
   c) Otherwise, $\texttt{NUMBER}(w) < \texttt{NUMBER}(v)$, so we have a frond or cross-link. If $\texttt{ONSTACK}(w)$ is true, set $\texttt{LOWLINK}(v) \leftarrow \min(\texttt{LOWLINK}(v), \texttt{NUMBER}(w))$.
5. (Not root?) If $\texttt{LOWLINK}(v) \neq \texttt{NUMBER}(v)$, return.
6. (Root) Assign the next unused component ID to v, and add v to this component ID.
7. (Add vertex) Pop a vertex v' from the stack (which must be non-empty at this point) and clear $\texttt{ONSTACK}(v')$. Add v' to the component of v, and associate the component ID with v'.
8. (Repeat) If the stack is not empty, go back to Step 7. Otherwise, return.

After the outermost recursive call returns, the stack will always be empty. Note that while this algorithm is recursive, it is not re-entrant; in particular, the act of getting the successors of a vertex must not trigger the computation of the same strongly connected components. This is enforced in Step 2. In our case, this can happen because getting the successors of a protocol performs type resolution; in practice this should be extremely difficult to hit, so for simplicity we report a fatal error and exit the compiler instead of attempting to recover.

Protocol components. We maintain two tables in the rewrite context:

1. A map from protocol declarations to *protocol nodes*.

   The protocol node for `P` stores the state associated with `P` by Tarjan's algorithm: `NUMBER(P)`, `LOWLINK(P)`, `ONSTACK(P)` and a component ID for `P`.

2. A map from component IDs to *protocol components*.

   A protocol component is a list of protocol declarations together with a requirement machine. The latter is either a protocol minimization machine constructed from user-written requirements, or a protocol machine constructed from the requirement signatures of those protocols. This requirement machine is lazily constructed when first needed, and not when the component itself is formed by Algorithm 15A.

Generic signatures. The idea of a protocol having a dependency relationship on another protocol generalizes to a generic signature depending on a protocol. The protocol dependencies of a generic signature are those that appear on the right-hand side of a derived conformance requirement.

Definition 15.6. A generic signature G has a *protocol dependency* on a protocol `P`, denoted $G \prec \texttt{P}$, if we can derive a conformance to `P` from G; that is, $G \vdash [\texttt{T: P}]$ for some type parameter `T`. The *protocol dependency set* of a generic signature G is the set of all protocols `P` such that $G \prec \texttt{P}$.

This new form of $\prec$ is not a binary relation in the sense of Definition 5.3, because the operands are in different sets. However, it is still transitive in the sense that $G \prec \texttt{P}$ and $\texttt{P} \prec \texttt{Q}$ together imply $G \prec \texttt{Q}$. Note that the two definitions of $\prec$ are related via the protocol generic signature: $G_{\texttt{P}} \prec \texttt{Q}$ if and only if $\texttt{P} \prec \texttt{Q}$.

Now, consider the protocols appearing on the right-hand side of the *explicit* conformance requirements of our generic signature G; call these the *successors* of G, by analogy with the successors of a protocol in the protocol dependency graph. If we consider all protocols reachable via paths originating from the successors of G, we arrive at the complete set of protocol dependencies of G.

We end this section with the algorithm to collect imported rules for a new requirement machine. We find all protocols reachable from some initial set, compute their strongly connected components, lazily construct a protocol machine for each component, and finally, collect the local rules from each requirement machine.

Algorithm 15B (Import rules from protocol components)**.** Takes a list of protocols that are the immediate successors of a generic signature or protocol. Outputs a list of imported rules.

1. (Initialize) Initialize a worklist and add all input protocols to the worklist. Initialize an empty set S of visited protocols. Initialize an empty set M of requirement machines (uniqued by pointer equality).

2. (Check) If the worklist is empty, go to Step 8.
3. (Next) Otherwise, remove the next protocol `P` from the worklist. If `P` $\in S$, go back to Step 2, otherwise set $S \leftarrow S \cup \{\texttt{P}\}$.
4. (Component) Use Algorithm 15A to compute the component ID for `P`.
5. (Machine) Let m be the requirement machine for this protocol component, creating it first if necessary. If $m \notin M$, set $M \leftarrow M \cup \{m\}$.
6. (Successors) Add each protocol dependency of `P` to the worklist.
7. (Loop) Go back to Step 1.
8. (Collect) Collect the local rules from each $m \in M$ and return.

We will encounter the protocol dependency graph again when we talk about the Knuth-Bendix completion procedure in Chapter 18. Completion looks for *overlaps* between pairs of rules, and we will use the protocol dependency graph to cut down on the work performed by showing that pairs of rules where both are imported need not be considered.

A protocol component is always operated on as an indivisible unit; for example, in Chapter 20 we will see that requirement minimization must consider all protocols in a component simultaneously to get correct results.

15.2. Debugging Flags

We've already seen two examples of the `-debug-requirement-machine` flag:

```
-debug-requirement-machine=timers
-debug-requirement-machine=protocol-dependencies
```

More generally, the argument to this flag is a comma-separated list of options, where the possible options are the following:

- `timers`: Chapter 15.
- `protocol-dependencies`: Section 15.1.
- `simplify`: Section 17.6.
- `add`, `completion`: Chapter 18.
- `concrete-unification`, `conflicting-rules`, `property-map`: Chapter 19.
- `concretize-nested-types`, `conditional-requirements`: Section 19.4.

- `concrete-contraction`: Section 20.5.
- `homotopy-reduction`, `homotopy-reduction-detail`, `propagate-requirement-ids`: Section 20.1.
- `minimal-conformances`, `minimal-conformances-detail`: Section 20.3.
- `minimization`, `redundant-rules`, `redundant-rules-detail`, `split-concrete-equiv-class`: Chapter 20.4.

There are two final debugging flags. We maintain some histograms in the rewrite context. The `-analyze-requirement-machine` flag dumps them at the end of the compilation session. They can be helpful when working on performance optimizations:

- A count of the unique symbols allocated, by kind (Section 17.2).
- A count of the number of terms allocated, by length (Section 17.3).
- Statistics about the rule trie (Section 17.6) and property map trie (Chapter 19).
- Statistics about the minimal conformances algorithm (Section 20.3).

The `-dump-requirement-machine` flag prints each requirement machine before and after the completion procedure runs. The printed representation includes a list of rewrite rules, the property map, and all rewrite loops. The output will begin to make sense after Chapter 17.

15.3. Source Code Reference

Key source files:

- `lib/AST/RequirementMachine/`

The Requirement Machine implementation is private to `lib/AST/`. The remainder of the compiler interacts with it indirectly, through the generic signature query methods on `GenericSignature` (Section 5.7) and the various requests for building new generic signatures (Section 11.5).

The Rewrite Context

Key source files:

- `lib/AST/RequirementMachine/RewriteContext.h`
- `lib/AST/RequirementMachine/RewriteContext.cpp`

`ASTContext` *class*

The global singleton for a single frontend instance. See also Section 2.6.

- `getRewriteContext()` returns the global singleton `RewriteContext` for this frontend instance.

`rewriting::RewriteContext` *class*

A singleton object to manage construction of requirement machines, building of the protocol component graph, and unique allocation of symbols and terms. See also Section 17.7.

- `getRequirementMachine(CanGenericSignature)` returns a query machine for the given generic signature, creating one first if necessary.
- `getRequirementMachine(ProtocolDecl *)` returns a for the protocol component which contains the given protocol, creating one first if necessary.
- `getProtocolComponentImpl()` is a private helper which returns the protocol component containing the given protocol.
- `getProtocolComponentRec()` implements Tarjan's algorithm (Algorithm 15A) for computing strongly connected components.
- `isRecursivelyConstructingRequirementMachine(CanGenericSignature)` returns true if we are currently constructing a query machine for this generic signature.
- `isRecursivelyConstructingRequirementMachine(ProtocolDecl *)` returns true if we are currently constructing a protocol machine for the given protocol's component. These two methods are used to break cycles in associated type inference, since otherwise re-entrant construction triggers an assertion in the compiler.
- `installRequirementMachine(CanGenericSignature, std::unique_ptr<RequirementMachine>)` takes a minimization machine and associates it with the given signature.
- `installRequirementMachine(ProtocolDecl *, std::unique_ptr<RequirementMachine>)` takes a protocol minimization machine and associates it with the given protocol's component.

`GenericSignatureImpl` *class*

See also Section 5.7.

- `getRequirementMachine()` returns the query machine for this generic signature, by asking the rewrite context to produce one and then caching the result in an instance variable of the `GenericSignatureImpl` instance itself.

  This method is used by the implementation of generic signature queries; apart from that, there should be no reason to reach inside the requirement machine instance from elsewhere in the compiler.

`ProtocolDependenciesRequest` *class*

A request evaluator request which computes the all protocols referenced from a given protocol's associated conformance requirements. These are the successors of the protocol in the .

`ProtocolDecl` *class*

See also Section 4.6.

- `getProtocolDependencies()` evaluates the `ProtocolDependenciesRequest`.

`rewriting::RequirementMachine` *class*

A list of rewrite rules and a property map. See also Section 17.7 and Section 19.7. Entry points for initializing a requirement machine, called by the rewrite context and various requests:

- `initWithGenericSignature()` initializes a new query machine from the requirements of an existing generic signature.
- `initWithWrittenRequirements()` initializes a new minimization machine from user-written requirements when building a new generic signature.
- `initWithProtocolSignatureRequirements()` initializes a new protocol machine from the requirement signature of each protocol of a protocol component.
- `initWithProtocolWrittenRequirements()` initializes a new protocol minimization machine from the user-written requirements in each protocol of a protocol component.

Taking a requirement machine apart:

- `getRewriteSystem()` returns the `RewriteSystem` (Section 17.7).

- `getPropertyMap()` returns the `PropertyMap` (Section 19.7).

Miscellaneous:

- `verify()` checks various invariants. We call this after constructing this requirement machine.
- `dump()` dumps the `RewriteSystem` and `PropertyMap`.

Requests

Key source files:

- `lib/AST/RequirementMachine/RequirementMachineRequests.cpp`

The evaluation functions for the below requests are implemented in the Requirement Machine.

`InferredGenericSignatureRequest::evaluate` *method*

Evaluation function for building a new generic signature from requirements written in source. Constructs a minimization machine.

This ultimately implements the `GenericContext::getGenericSignature()` method; see Section 5.7.

`AbstractGenericSignatureRequest::evaluate` *method*

Evaluation function for building a new generic signature from a list of generic parameters and requirements. Constructs a minimization machine. This ultimately implements the `buildGenericSignature()` function; see Section 11.5.

`RequirementSignatureRequest::evaluate` *method*

Evaluation function for obtaining the requirement signature of a protocol. This either deserializes the requirement signature, or constructs a protocol minimization machine from user-written requirements and uses it to build a new requirement signature. This ultimately implements the `ProtocolDecl::getRequirementSignature()` method; see Section 5.7.

Debugging

Key source files:

- `lib/AST/RequirementMachine/Debug.h`
- `lib/AST/RequirementMachine/Histogram.h`

`rewriting::RewriteContext` *class*

- `RewriteContext()` the rewrite context constructor splits the string value of the `-debug-requirement-machine` flag into comma-separated tokens, and maps each token to an element of the `DebugFlags` enum.
- `getDebugFlags()` returns the `DebugFlags` enum.
- `beginTimer()` starts a timer identified by the given string, and logs a message.
- `endTimer()` ends a timer identified by the given string, and logs a message. Must be paired with the call to `beginTimer()`.

`rewriting::DebugFlags` *enum class*

An `OptionSet` of debugging flags to control various forms of debug output printed by the Requirement Machine.

`rewriting::Histogram` *class*

A utility class for collating points into a histogram. Each histogram has a fixed number of buckets, together with an initial value, and an "overflow" bucket. Points are mapped to buckets by subtracting the initial value and comparing against the total number of buckets. If the result exceeds the total number of buckets, it is recorded in an "overflow" bucket.

If the compiler is invoked with the `-analyze-requirement-machine` frontend flag, various histograms in the `RequirementMachine` instance are printed when the compiler exits.

- `Histogram(unsigned size, unsigned start)` creates a new histogram.
- `add(unsigned)` records a point in the histogram.
- `dump()` prints the histogram as ASCII art.

16. Monoids

MONOIDS are one of the fundamental objects that we study in abstract algebra. Their simplicity allows for a great deal of generality, so we will quickly narrow our focus to *finitely-presented* monoids. We will see that every finitely-presented monoid can be translated into a Swift generic signature, which establishes a connection between the derived requirements formalism of Section 5.2, and the *word problem*. Like the halting problem of Section 12.4, we will see that the word problem is undecidable in general. On the other hand, it can be solved in those finitely-presented monoids that admit a *convergent* presentation. This prepares us for the next chapter, where we proceed to "solve" the derived requirements formalism by encoding a generic signature as a finitely-presented monoid. We begin with standard definitions, found in [140] or [141].

Definition 16.1. A *monoid* $(M, \cdot, \varepsilon)$ is a structure consisting of a set M, a binary operation "$\cdot$", and an identity element $\varepsilon \in M$, which together satisfy the following three axioms:

- The set M is *closed* under the binary operation: for all $x, y \in M$, $x \cdot y \in M$.
- The binary operation is *associative*: for all $x, y, z \in M$, $x \cdot (y \cdot z) = (x \cdot y) \cdot z$.
- The element ε acts as the *identity*: for all $x \in M$, $x \cdot \varepsilon = \varepsilon \cdot x = x$.

When the binary operation and identity element are clear from context, we can write M instead of $(M, \cdot, \varepsilon)$. We also sometimes omit the symbol "$\cdot$", so if $x, y \in M$, then we can write xy instead of $x \cdot y$. Finally, associativity allows us to omit parentheses without ambiguity in an expression like $x \cdot y \cdot z$ or xyz.

Specific instances of monoids can be quite complicated and abstract, but our first example is easy to describe.

Example 16.2. The set of natural numbers $\mathbb{N}$ satisfies the monoid axioms if we take addition as our operation and zero as the identity element:

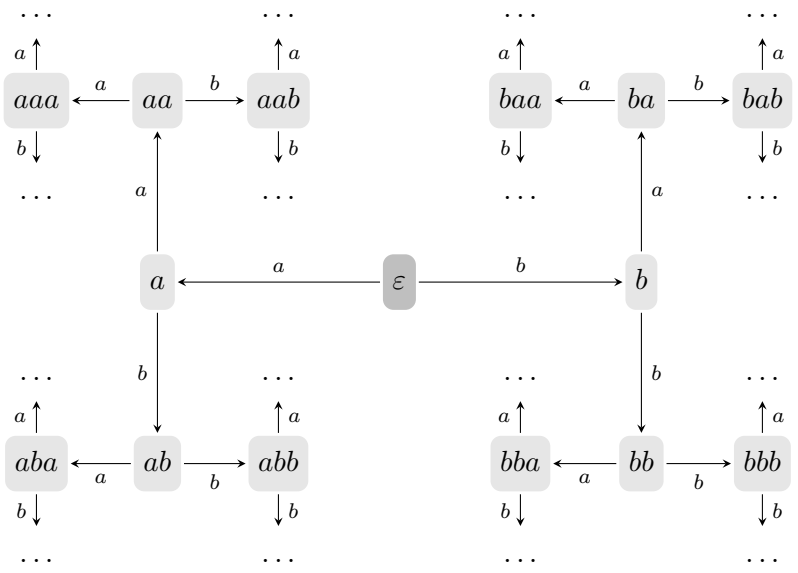
- The sum of two natural numbers is again a natural number.
- Addition is associative: for all $x, y, z \in \mathbb{N}$, $(x + y) + z = x + (y + z)$.
- Zero is the identity: for all $x \in \mathbb{N}$, $x + 0 = 0 + x = x$.

We will see shortly that $(\mathbb{N}, +, 0)$ is an instance of the following construction.

Definition 16.3. Let A be a set. The *free monoid* generated by A, denoted A^*, is the set of all finite strings of elements of A. (This is the same notation as the " $*$ " operator in a *regular expression*.) The set A is called the *alphabet* or *generating set* of A^*. The generating set may be finite or infinite, but we assume it is finite unless stated otherwise. The binary operation on A^* is *string concatenation*, and the identity element ε is the empty string. The elements of A^* are also called *terms*. We say that u is a *subterm* of t if $t = xuy$ for some $x, y \in A^*$. The *length* of a term $t \in A^*$ is denoted $|t| \in \mathbb{N}$.

The reader might wish to review the discussion of the type parameter graph from Section 8.3. In abstract algebra, the *Cayley graph* of a monoid plays a similar role, and we will discover many parallels between the two. We will describe the general construction in the next section, but for now we'll just look the Cayley graph of the free monoid A^*. The vertices in this graph are the terms of A^*, and the vertex for the identity element is the distinguished root vertex. Then, for each vertex t and each generator $g \in A$, we also add an edge with source t and destination tg, and label this edge "g". (This is sometimes called the *right* Cayley graph, and the left Cayley graph can then be defined in the opposite way by joining t with gt.)

Example 16.4. The free monoid with two generators $\{a, b\}^*$ consists of all finite strings made up of a and b. Two typical elements are $abba$ and bab, and their concatenation is $abba \cdot bab = abbabab$. Unlike $(\mathbb{N}, +, 0)$, this monoid operation is not *commutative*, so for example, $abba \cdot bab \neq bab \cdot abba$. The Cayley graph of $\{a, b\}^*$ is an infinite binary tree. Every vertex has two successors, corresponding to multiplication on the right by a and b, respectively:



Every term defines a path in the Cayley graph; we start from the root ε, and follow a series of successive "a" or "b" edges until we arrive at the vertex representing our term. In general, every vertex in the Cayley graph has the same number of successors, equal to the number of elements in our generating set A.

Before we continue, we need a shorthand where xxx becomes x^3, and so on:

Definition 16.5. Let M be a monoid. If $x \in M$ and $n \in \mathbb{N}$, we can define x^n as the "nth power" of x:

$$x^n = \begin{cases} \varepsilon & n = 0 \\ x & n = 1 \\ x^{n-1} \cdot x & n > 1 \end{cases}$$

The choice of notation is justified by the following "law of exponents."

Proposition 16.6. Let M be a monoid. If $x \in M$ is some element, then $x^m \cdot x^n = x^{m+n}$ for all m, $n \in \mathbb{N}$.

Proof. We start with a fixed but arbitrary $m \in \mathbb{N}$, and then prove the result by induction on $n \in \mathbb{N}$ (Section 11.3).

Base case. We need to show that $x^m \cdot x^0 = x^{m+0}$. We make use of the definition of x^0, the identity element axiom of M, and the fact that $m = m + 0$ in $\mathbb{N}$:

$$x^m \cdot x^0 = x^m \cdot \varepsilon = x^m = x^{m+0}$$

Inductive step. First, we assume the induction hypothesis:

$$x^m \cdot x^{n-1} = x^{m+n-1}$$

Then, we use the monoid operation of M to multiply both sides by x:

$$(x^m \cdot x^{n-1}) \cdot x = x^{m+n-1} \cdot x$$

We use the associativity of M and the definition of x^n to rewrite the left-hand side:

$$(x^m \cdot x^{n-1}) \cdot x = x^m \cdot (x^{n-1} \cdot x) = x^m \cdot x^{n-1+1} = x^m \cdot x^n$$

Finally, we use the definition of x^n to rewrite the right-hand side:

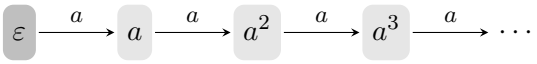
$$x^{m+n-1} \cdot x = x^{m+n-1+1} = x^{m+n}$$

Putting everything together, we get $x^m \cdot x^n = x^{m+n}$. By induction, this is actually true for all $n \in \mathbb{N}$, and since we also started with an arbitrary $m \in \mathbb{N}$, we're done. □

Example 16.7. The free monoid over one generator $\{a\}^*$ is *isomorphic* to $(\mathbb{N}, +, 0)$. We will talk about isomorphisms later, but essentially this means that both define the same object, up to a choice of notation.

Indeed, every non-zero element of $\mathbb{N}$ can be uniquely expressed as a sum of 1's. For example, $3 = 1 + 1 + 1$ and $5 = 1 + 1 + 1 + 1 + 1$. This means we can write ε instead of 0, a instead of 1, and $\cdot$ instead of $+$. Now, the equation $3 + 5 = 8$ translates into $aaa \cdot aaaaa = aaaaaaaa$, or using exponent notation, $a^3 \cdot a^5 = a^8$.

Next, consider the Cayley graph of $\{a\}^*$. Here, every vertex has one successor, because the generating set has a single element a. This looks like the type parameter graph of protocol `N` from Example 8.8, and we will see why in Section 16.3:



Example 16.8. The free monoid over an empty set $\varnothing^*$ is $\{\varepsilon\}$, the one-element set that consists of the empty string. This is called the *trivial monoid*. Its Cayley graph is a single vertex.

16.1. Finitely-Presented Monoids

Every element of a free monoid has a *unique* expression as a product of generators. Finitely-presented monoids are more general, because multiple distinct combinations can name the same element. To model this phenomenon, we add a finite set of *rewrite rules* (or *relations*); these then generate an equivalence relation on terms. We first turned to equivalence relations in Section 5.3, to understand the same-type requirements of a generic signature; the below construction is similar.

A *monoid presentation* is a list of generators and rewrite rules:

$$\langle \underbrace{a_1, \ldots, a_m}_{\text{generators}} \mid \underbrace{u_1 \sim v_1, \ldots, u_n \sim v_n}_{\text{rewrite rules}} \rangle$$

Also, if $A := \{a_1, \ldots, a_m\}$ and $R := \{(u_1, v_1), \ldots, (u_n, v_n)\} \subseteq A^* \times A^*$ are finite sets, we can form the monoid presentation $\langle A \mid R \rangle$. The equivalence relation that R generates on the terms of A^* is denoted by $x \sim_R y$, or $x \sim y$ when R is clear from context. Note that when we write $x = y$, we always mean that x and y are *identical* in A^*, not just equivalent in $\langle A \mid R \rangle$.

We will describe the intuitive model behind the term equivalence relation first, and then give a rigorous definition in the next section. Syntactically, a rewrite rule $(u, v) \in R$ is a pair of terms. Semantically, a rewrite rule tells us that if we find u anywhere within a term, we can replace this subterm with v, and obtain another equivalent term. Term equivalence is symmetric, so we can also replace v with u, too. Finally, it is transitive, so we can iterate these rewrite steps any number of times to prove an equivalence.

A monoid presentation $\langle A \,|\, R \rangle$ defines a *finitely-presented monoid* where the elements are the equivalence classes of $\sim_R$, the identity element is the equivalence class of ε, and the binary operation is concatenation of terms (we will show it is well-defined later).

Example 16.9. Finitely-presented monoids generalize the free monoids, because every free monoid (with a finite generating set) is also finitely presented if we add an empty set of rewrite rules. In this case $x \sim y$ if and only if $x = y$.

Example 16.10. Consider the finite set $\{0, 1, 2, 3\}$ with the binary operation $+$ given by the below table. This method of defining a finite monoid is called a *Cayley table*:

$+$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

The above operation is addition modulo 4, and we will denote this monoid by $\mathbb{Z}_4$. If we write a instead of 1, ε instead of 0, and $\cdot$ instead of $+$, we can present $\mathbb{Z}_4$ with one generator and one rewrite rule:

$$\mathbb{Z}_4 := \langle a \,|\, a^4 \sim \varepsilon \rangle$$

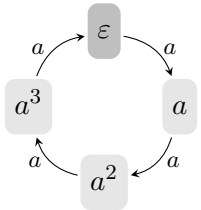
The rule $a^4 \sim \varepsilon$ allows us to insert or delete $aaaa$, which gives us this general principle:

$$a^m \sim a^n \qquad \text{if and only if} \qquad m \equiv n \pmod 4$$

Therefore, the rule partitions the terms of $\{a\}^*$ into four infinite equivalence classes:

$$\{\varepsilon, a^4, a^8, \ldots, a^{4k}, \ldots\}$$
$$\{a, a^5, a^9, \ldots, a^{4k+1}, \ldots\}$$
$$\{a^2, a^6, a^{10}, \ldots, a^{4k+2}, \ldots\}$$
$$\{a^3, a^7, a^{11}, \ldots, a^{4k+3}, \ldots\}$$

We construct the Cayley graph of a finitely-presented monoid by taking the vertices to be equivalence classes of terms. The edge relation is now defined on these equivalence classes, so for each equivalence class $[\![t]\!]$ and generator $g \in A$, we add an edge with source $[\![t]\!]$ and destination $[\![tg]\!]$. If we label each vertex with the shortest term in its equivalence class, we see that the Cayley graph for $\mathbb{Z}_4$ looks like the type parameter graph of the `Z4` protocol from Example 8.9:



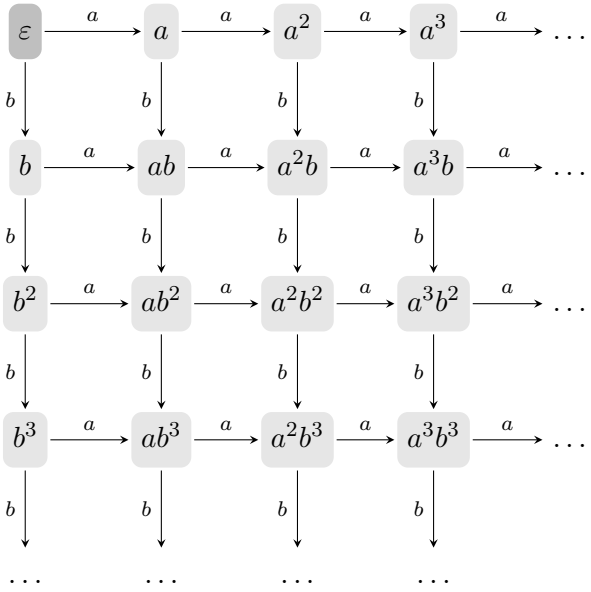
Example 16.11. We saw that the free monoid $\{a, b\}^*$ is not commutative. If we wish to make it commutative in the "most general" way, we can consider the *free commutative monoid* on two generators:

$$\langle a, b \,|\, ba \sim ab \rangle$$

The rule says we can permute the a's and b's within a term, but nothing else, so in particular the terms in an equivalence class must all have the same length. For this reason, every equivalence class is finite, and furthermore, we can transform an arbitrary term into a "canonical form" where the a's come first. This allows us to calculate by summing exponents (in fact, this monoid is isomorphic to the Cartesian product $\mathbb{N} \times \mathbb{N}$, whose elements are ordered pairs added component-wise). For example:

$$a^2b^3 \cdot a^7b = a^9b^4$$

The existence of this canonical form means that every equivalence class contains a unique term of the form $a^m b^n$ for m, $n \in \mathbb{N}$, so we will use these as the labels in our Cayley graph. Now, multiplication by a increments the first exponent, and multiplication by b increments the second. If we arrange the vertices on a grid, then we see that each vertex is joined with the vertex immediately below and to the right:



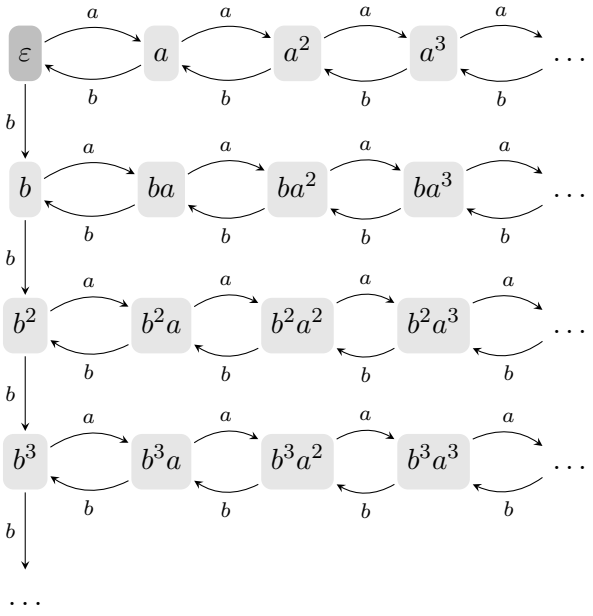
This Cayley graph (as well as the one for $\mathbb{Z}_4$) has something we didn't observe in free monoids: multiple paths with the same source and destination. For example, because $ababab \sim aaabbb$, we can start from ε and then follow the edges by reading either term from left to right. In either case, we end up at a^3b^3. The equivalence class of a^3b^3 describes the "walking routes" that take us to the bottom right corner of the grid.

Example 16.12. For the next example, we also start with two generators and a single rewrite rule, but the resulting monoid looks rather different.

If we're given a string of "`(`" and "`)`" and we wish to determine if the parentheses are balanced, we can repeatedly delete occurrences of the substring "`()`" until none remain. If we end up with the empty string, then we know that every opening parenthesis was matched by a closing parenthesis in the original string. Now, write a instead of "`(`" and b instead of "`)`". This is the *bicyclic monoid*, where balanced strings of parentheses are precisely those terms equivalent to ε:

$$\langle a, b \,|\, ab \sim \varepsilon \rangle$$

Consider the term $baaba$, which contains ab as a subterm. Our rule allows us to replace this subterm with ε, thus $baaba \sim baa$. We can then insert ab in a different position, so for example $baa \sim babaa$. Equivalence is transitive, so we also have $baaba \sim babaa$. Note that this monoid is not commutative, for instance $ab \not\sim ba$. In this monoid, every equivalence class has a unique term of minimum length, the one without any ab's. This term has the form $b^m a^n$ for $m, n \in \mathbb{N}$, so we will label our vertices with those. As in the previous example, we arrange the vertices of the Cayley graph on a grid:



To understand the edge relation better, observe that if we multiply any term by a on the right, we can then multiply by b to "cancel out" the a and end up back where we started. On the other hand, if there are no more a's to cancel on the right, adding a new b takes us into a new "state" from which we cannot return. There is a unique shortest path from ε to every other vertex; it is the one that does not contain ab.

Example 16.13. Not every "elementary" monoid is finitely presented, one example being the natural numbers under *multiplication*. This is a consequence of these four properties of the natural numbers:

1. Every non-zero natural number has a unique representation as a product of prime numbers (*The Fundamental Theorem of Arithmetic*).
2. There are infinitely many prime numbers (*Euclid's Theorem*).
3. Multiplication of natural numbers is commutative.
4. Multiplication by zero always yields zero.

These give us an *infinite* presentation of $(\mathbb{N}, \cdot, 1)$. This monoid is generated by zero and all the prime numbers: $\{0, 2, 3, 5, 7, 11, \ldots\}$. Then, we add the rule $0 \cdot 0 \sim 0$ because zero times zero is zero, followed by two infinite sets of rewrite rules:

- For every pair of distinct prime numbers p and q, we have $p \cdot q \sim q \cdot p$, so our binary operation is commutative.
- For every prime number p, we have $0 \cdot p \sim 0$ and $p \cdot 0 \sim 0$, so that zero behaves like the zero element.

The unique prime factorization of a natural number is really a term written using these generators, for example $84 = 2^2 \cdot 3 \cdot 7$. This monoid is a far more intricate object than $(\mathbb{N}, +, 0)$, which we recall was just the free monoid with one generator.

Mathematical aside. A typical course of abstract algebra often starts with the study of *groups*, which are monoids where each element x has an inverse element x^{-1}, such that $x \cdot x^{-1} = x^{-1} \cdot x = \varepsilon$. Groups have a lot more structure than monoids, all because of this one additional axiom. The natural numbers under addition are not a group, but if we instead consider the set of all integers $\mathbb{Z}$, then $\mathbb{Z}$ is a group under addition, because each $n \in \mathbb{Z}$ has an inverse element: $n + (-n) = (-n) + n = 0$. Addition is commutative, and a group with a commutative binary operation is also called an *Abelian group*.

The next level of structure is the *ring*. A ring is a set R with *two* binary operations, denoted $+$ and $\cdot$, and two special elements, 0 and 1. The ring axioms state that $(R, +, 0)$ is an Abelian group, $(R, \cdot, 1)$ is a monoid, and $+$ and $\cdot$ are related via the *distributive law*, so $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$. With the usual addition and multiplication operations, $\mathbb{Z}$ becomes a ring. If every non-zero element has a multiplicative inverse, we have a *division ring*. If we also say that multiplication is commutative, we have a *field*. In a field, the non-zero elements define an Abelian group under multiplication. The set of rational numbers $\mathbb{Q}$ is a field.

This tower of abstractions also describes the complex numbers, quaternions, matrices, polynomials, modular arithmetic, and other "number-like" objects which commonly arise in various applications of mathematics.

16.2. Equivalence of Terms

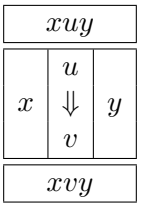
The Cayley graph of a monoid presentation helped us understand the relationships between equivalence classes. To better see the structure within a single equivalence class, we follow [142] and introduce another directed graph, called the *rewrite graph* of a monoid presentation $\langle A \,|\, R \rangle$. We begin by describing the edges of our graph.

Definition 16.14. A monoid presentation $\langle A \,|\, R \rangle$ generates a set of *rewrite steps*:

1. A rewrite step is an ordered tuple of terms, written $x(u \Rightarrow v)y$, where x, $y \in A^*$, and one of (u, v) or $(v, u) \in R$.
2. If $(u, v) \in R$, we say that $x(u \Rightarrow v)y$ is a *positive* rewrite step. If $(v, u) \in R$, then $x(u \Rightarrow v)y$ is a *negative* rewrite step.
3. The terms x and y are called the left and right *whiskers*, respectively. If x or y is identically ε, we omit them in the notation.
4. For rewrite steps to serve as edges in our graph, we define the *source* and *destination* of a rewrite step $s := x(u \Rightarrow v)y$ as the following terms of A^*:

$$\mathrm{src}(s) := xuy$$
$$\mathrm{dst}(s) := xvy$$

The rewrite step $x(u \Rightarrow v)y$ represents the transformation of xuy into xvy via the replacement of u with v, while leaving the whiskers x and y unchanged. (We can assume without loss of generality that $u \neq v$, ruling out redundant steps where $\mathrm{src}(s) = \mathrm{dst}(s)$.) We can draw a picture:



Definition 16.15. The *rewrite graph* of a monoid presentation $\langle A \,|\, R \rangle$ has the terms of A^* as vertices, and rewrite steps as edges. Except in the trivial case where $A = \varnothing$, this graph is infinite, but we will only look at small subgraphs at any one time. For example, every rewrite step determines a subgraph with two vertices and one edge:

$$xuy \xrightarrow{x(u \Rightarrow v)y} xvy$$

From the rewrite graph, we will then define $x \sim_R y$ to mean that this graph has a *path* with source x and destination y. The following definition of a rewrite path is identical to the general case of a path in a directed graph from Section 8.3.

Definition 16.16. A *rewrite path* consists of an initial term $t \in A^*$ together with a sequence of zero or more rewrite steps $(s_1, \ldots, s_n)$ which satisfy the following conditions:

1. If the rewrite path contains at least one rewrite step, the source of the first rewrite step must equal the initial term: $\mathrm{src}(s_1) = t$.

2. If the rewrite path contains at least two steps, the source of each subsequent step must equal the destination of the preceding step: $\mathrm{src}(s_{i+1}) = \mathrm{dst}(s_i)$ for $0 < i \leq n$.

Like a rewrite step, each rewrite path also has a source and destination. We define the source and destination of a rewrite path p as follows, where n is the length of p:

$$\mathrm{src}(p) := t$$
$$\mathrm{dst}(p) := \begin{cases} \mathrm{dst}(s_n) & \text{if } n > 0 \\ t & \text{if } n = 0 \end{cases}$$

To each term $t \in A^*$ we can associate a rewrite path with an empty sequence of rewrite steps, called the *empty rewrite path* and denoted 1_t. We have $\mathrm{src}(1_t) = \mathrm{dst}(1_t) = t$. (This is why a rewrite path must store its initial term, to specify the source and destination in the empty case.) A non-empty rewrite path p can be written as a composition of steps: $p = s_1 \circ \cdots \circ s_n$. We can also visualize a rewrite path as a path in the rewrite graph:

$$\mathrm{src}(p) \xRightarrow{s_1} \cdots \xRightarrow{\cdots} \cdots \xRightarrow{s_n} \mathrm{dst}(p)$$

Example 16.17. Recall the bicyclic monoid $\langle a, b \,|\, ab \sim \varepsilon \rangle$ from Example 16.12. We saw that $baaba \sim baa$ because we obtain baa by deleting the subterm ab from $baaba$. We can encode this with a rewrite step $ba(ab \Rightarrow \varepsilon)a$:

$$baaba \xRightarrow{ba(ab \Rightarrow \varepsilon)a} baa$$

We also saw that $baa \sim babaa$. We can encode the transformation of baa into $babaa$ with a rewrite step $b(\varepsilon \Rightarrow ab)aa$:

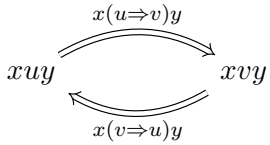
$$baa \xRightarrow{b(\varepsilon \Rightarrow ab)aa} babaa$$

The destination of the first rewrite step is also the source of the second rewrite step. We can compose them to form the rewrite path $ba(ab \Rightarrow \varepsilon)a \circ b(\varepsilon \Rightarrow ab)aa$. This rewrite path encodes the fact that $baaba \sim babaa$:

$$baaba \xRightarrow{ba(ab \Rightarrow \varepsilon)a} baa \xRightarrow{b(\varepsilon \Rightarrow ab)aa} babaa$$

The algebra of rewriting. To ensure that the term equivalence relation ultimately satisfies the necessary axioms, we introduce some algebraic operations on rewrite steps and paths, called inversion, composition, and whiskering.

Definition 16.18. The *inverse* of a rewrite step $s := x(u \Rightarrow v)y$, denoted s^{-1}, is defined as $x(v \Rightarrow u)y$, so we swap the u and the v. The inverse of a positive rewrite step is negative, and vice versa, which means that the edges in the rewrite graph come in complementary pairs:



We can also invert a rewrite *path*, to get a new path which undoes the transformation described by the original path. The empty rewrite path is its own inverse. For a non-empty rewrite path, we invert each rewrite step and perform them in reverse order:

$$p^{-1} := \begin{cases} 1_t & \text{if } p = 1_t \\ s_n^{-1} \circ \cdots \circ s_1^{-1} & \text{if } p = s_1 \circ \cdots \circ s_n \end{cases}$$

When shown in the rewrite graph, the inverse path p^{-1} is p but backwards:

$$\mathrm{src}(p) \xLeftarrow{s_1} \cdots \xLeftarrow{\cdots} \cdots \xLeftarrow{s_n} \mathrm{dst}(p)$$

We can think of a rewrite step as a rewrite path with length 1. Then, if p is *either* a rewrite step or a rewrite path, we see that taking the inverse of p interchanges its source and destination:

$$\mathrm{src}(p^{-1}) = \mathrm{dst}(p)$$
$$\mathrm{dst}(p^{-1}) = \mathrm{src}(p)$$

Example 16.19. We exhibited this rewrite path in Example 16.17, with source *baaba* and destination *babaa*:

$$baaba \xRightarrow{ba(ab \Rightarrow \varepsilon)a} baa \xRightarrow{b(\varepsilon \Rightarrow ab)aa} babaa$$

We can construct the inverse rewrite path from *babaa* to *baaba* by inverting both steps and swapping the order in which we apply them:

$$babaa \xRightarrow{b(ab \Rightarrow \varepsilon)aa} baa \xRightarrow{ba(\varepsilon \Rightarrow ab)a} baaba$$

Definition 16.20. If p_1 and p_2 are two rewrite paths with $\mathrm{dst}(p_1) = \mathrm{src}(p_2)$, we define their *composition* $p_1 \circ p_2$ as the rewrite path consisting of the initial term of p_1, followed by the rewrite steps of p_1, and finally the steps of p_2. This gives us a new rewrite path with the same source as p_1 and the same destination as p_2:

$$\mathrm{src}(p_1 \circ p_2) = \mathrm{src}(p_1)$$
$$\mathrm{dst}(p_1 \circ p_2) = \mathrm{dst}(p_2)$$

Shown as a diagram, composition joins two paths (the "=" in the middle is not an edge in the graph, but denotes we have two identical terms):

$$\mathrm{src}(p_1) \Longrightarrow \cdots \Longrightarrow (\mathrm{dst}(p_1) = \mathrm{src}(p_2)) \Longrightarrow \cdots \Longrightarrow \mathrm{dst}(p_2)$$

Empty rewrite paths act as the identity with respect to composition:

$$1_{\mathrm{src}(p)} \circ p = p$$
$$p \circ 1_{\mathrm{dst}(p)} = p$$

The inverse of the composition is the composition of the inverses, but flipped:

$$(p_1 \circ p_2)^{-1} = p_2^{-1} \circ p_1^{-1}$$

Finally, rewrite path composition is associative. If p_3 is another rewrite path such that $\mathrm{dst}(p_2) = \mathrm{src}(p_3)$:

$$p_1 \circ (p_2 \circ p_3) = (p_1 \circ p_2) \circ p_3$$

Example 16.21. We continue with Example 16.19. Consider these two rewrite paths in the bicyclic monoid, which we will call p_1 and p_2:

$$baabb \xRightarrow{ba(ab \Rightarrow \varepsilon)b} bab \xRightarrow{b(ab \Rightarrow \varepsilon)} b$$

$$aabbb \xRightarrow{a(ab \Rightarrow \varepsilon)bb} abb \xRightarrow{(ab \Rightarrow \varepsilon)b} b$$

The two paths have the same destination term b, but different source terms. If we invert the second path and compose it with the first, we get the rewrite path $p_1 \circ p_2^{-1}$, which transforms *baabb* into *aabbb*:

$$baabb \xRightarrow{ba(ab \Rightarrow \varepsilon)b} bab \xRightarrow{b(ab \Rightarrow \varepsilon)} b \xRightarrow{(\varepsilon \Rightarrow ab)b} abb \xRightarrow{a(\varepsilon \Rightarrow ab)bb} aabbb$$

Definition 16.22. If $s := x(u \Rightarrow v)y$ and $z \in A^*$, we define the left and right *whiskering* action of z on s. Whiskering extends a rewrite step by lengthening its whiskers:

$$z \triangleleft s := zx(u \Rightarrow v)y$$
$$s \triangleright z := x(u \Rightarrow v)yz$$

The whiskering operation generalizes to rewrite paths as follows. First, we apply the monoid operation to the initial term, and then we whisker each rewrite step. If the path is empty, whiskering produces another empty rewrite path on a new term:

$$z \triangleleft 1_t = 1_{z \cdot t}$$
$$1_t \triangleright z = 1_{t \cdot z}$$

If the path is non-empty, the distributive law relates the whiskering of paths and steps:

$$z \triangleleft (s_1 \circ \cdots \circ s_n) = (z \triangleleft s_1) \circ \cdots \circ (z \triangleleft s_n)$$
$$(s_1 \circ \cdots \circ s_n) \triangleright z = (s_1 \triangleright z) \circ \cdots \circ (s_n \triangleright z)$$

Here is a diagram for $z \triangleleft p$ and $p \triangleright z$:

$$z \cdot \mathrm{src}(p) \xRightarrow{z \triangleleft s_1} z \cdots \Longrightarrow z \cdots \xRightarrow{z \triangleleft s_n} z \cdot \mathrm{dst}(p)$$

$$\mathrm{src}(p) \cdot z \xRightarrow{s_1 \triangleright z} \cdots z \Longrightarrow \cdots z \xRightarrow{s_n \triangleright z} \mathrm{dst}(p) \cdot z$$

Once again, if we let p denote either a rewrite path or a single rewrite step, we see that whiskering is related to the monoid operation of A^* in two ways:

$$\mathrm{src}(z \triangleleft p) = z \cdot \mathrm{src}(p)$$
$$\mathrm{dst}(z \triangleleft p) = z \cdot \mathrm{dst}(p)$$

$$\mathrm{src}(p \triangleright z) = \mathrm{src}(p) \cdot z$$
$$\mathrm{dst}(p \triangleright z) = \mathrm{dst}(p) \cdot z$$

Also, if $z' \in A^*$ is some other term:

$$z \triangleleft (z' \triangleleft p) = (z \cdot z') \triangleleft p$$
$$(p \triangleright z) \triangleright z' = p \triangleright (z \cdot z')$$

The left and right whiskering actions are compatible with each other:

$$(z \triangleleft p) \triangleright z' = z \triangleleft (p \triangleright z')$$

Finally, the distributive law generalizes from rewrite steps to rewrite paths:

$$z \triangleleft (p_1 \circ p_2) = (z \triangleleft p_1) \circ (z \triangleleft p_2)$$
$$(p_1 \circ p_2) \triangleright z = (p_1 \triangleright z) \circ (p_2 \triangleright z)$$

Proposition 16.23. Let $\langle A \mid R \rangle$ be a monoid presentation. Let P be the set of rewrite paths of $\langle A \mid R \rangle$. The following axioms characterize P:

1. (Base case) For each $(u, v) \in R$, there is an "elementary" rewrite path $(u \Rightarrow v) \in P$.
2. (Closure under inversion) If $p \in P$, then $p^{-1} \in P$.
3. (Closure under composition) If $p_1, p_2 \in P$ with $\mathrm{dst}(p_1) = \mathrm{src}(p_2)$, then $p_1 \circ p_2 \in P$.
4. (Closure under whiskering) If $p \in P$ and $z \in A^*$, then $z \triangleleft p$, $p \triangleright z \in P$.

Proof. By Definition 16.14 there is a positive rewrite step $(u \Rightarrow v)$ for each $(u, v) \in R$. Since every rewrite step is also a rewrite path of length 1, this establishes (1). Also, rewrite steps are closed under inverses and whiskering, which establishes (2) and (4) for rewrite paths of length 1. The general case of (2), (3) and (4) can be shown by induction on the length of the rewrite path. □

Recall that $x \sim_R y$ means our rewrite graph has a path from x to y.

Proposition 16.24. Let $\langle A \mid R \rangle$ be a monoid presentation. Then $\sim_R$ is an equivalence relation on A^*.

Proof. Let P be the set of rewrite paths of $\langle A \mid R \rangle$. We check each axiom in turn.

1. (Reflexivity) For each $x \in A^*$, there is an empty rewrite path $1_x \in P$. Since $\mathrm{src}(1_x) = \mathrm{dst}(1_x) = x$, it follows that $x \sim x$.
2. (Symmetry) Assume $x \sim y$. Then there is a $p \in P$ such that $x = \mathrm{src}(p)$ and $y = \mathrm{dst}(p)$. Since P is closed under taking inverses, $p^{-1} \in P$. But $y = \mathrm{src}(p^{-1})$ and $x = \mathrm{dst}(p^{-1})$, so we have $y \sim x$.
3. (Transitivity) Assume $x \sim y$ and $y \sim z$. Then there is a $p_1 \in P$ with $x = \mathrm{src}(p_1)$ and $y = \mathrm{dst}(p_1)$, and a $p_2 \in P$ with $y = \mathrm{src}(p_2)$ and $z = \mathrm{dst}(p_2)$. We have $\mathrm{dst}(p_1) = \mathrm{src}(p_2)$, and P is closed under composition so $p_1 \circ p_2 \in P$. We also know that:

   $$\mathrm{src}(p_1 \circ p_2) = \mathrm{src}(p_1) = x$$
   $$\mathrm{dst}(p_1 \circ p_2) = \mathrm{dst}(p_2) = z$$

   Thus, $x \sim z$.

Notice how we used inversion and composition above, but not whiskering. □

We will use whiskering to show that $\sim$ is *compatible* with term concatenation:

Definition 16.25. Let M be a monoid. A relation $R \subseteq M \times M$ is *translation-invariant* if for all $(x, y) \in R$ and $z \in M$, we have both (zx, zy) and $(xz, yz) \in R$.

A translation-invariant equivalence relation is known as a *monoid congruence*.

Example 16.26. Consider the monoid $(\mathbb{N}, +, 0)$. The standard linear order $<$ on $\mathbb{N}$ is translation-invariant (but not an equivalence relation). For instance $5 < 7$ also implies that $5 + 2 < 7 + 2$. (Hence we're using "translation" in the geometric sense.)

Theorem 16.27. Let $\langle A \mid R \rangle$ be a monoid presentation. Then the term equivalence relation $\sim_R$ is a monoid congruence.

Proof. We've seen that $\sim$ is an equivalence relation, so it remains to establish translation invariance. Let P be the set of rewrite paths of $\langle A \mid R \rangle$. Suppose that x, y, $z \in A^*$, and also that $x \sim y$, so we have $p \in P$ such that $x = \mathrm{src}(p)$ and $y = \mathrm{dst}(p)$. We must show $zx \sim zy$ and $xz \sim yz$. For the first claim, we notice that $z \triangleleft p \in P$, so $zx \sim zy$:

$$\mathrm{src}(z \triangleleft p) = z \cdot \mathrm{src}(p) = zx$$
$$\mathrm{dst}(z \triangleleft p) = z \cdot \mathrm{dst}(p) = zy$$

To show that $xz \sim yz$, consider $p \triangleright z$ instead. □

Thus, *the set of equivalence classes* of $\sim_R$ has the structure of a monoid:

Theorem 16.28. Let $\langle A \mid R \rangle$ be a monoid presentation. Term concatenation is well-defined on the equivalence classes of $\sim_R$; that is, if we pick two representative terms and concatenate them together, the equivalence class of the result does not depend on the choice of representatives.

Proof. Suppose we're given x, x', y, $y' \in A^*$ such that $x \sim x'$ and $y \sim y'$. We must show that $xy \sim x'y'$. By assumption, there exist a pair of rewrite paths p_x, p_y such that:

$$\mathrm{src}(p_x) = x$$
$$\mathrm{dst}(p_x) = x'$$

$$\mathrm{src}(p_y) = y$$
$$\mathrm{dst}(p_y) = y'$$

Now, we form the rewrite path $p := (p_x \triangleright y) \circ (x' \triangleleft p_y)$ by whiskering and composing p_x and p_y. The composition is valid:

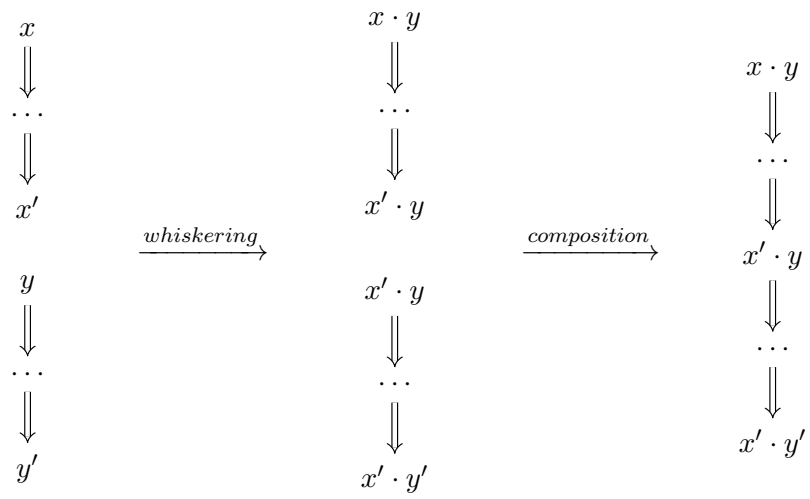
$$\mathrm{dst}(p_x \triangleright y) = \mathrm{dst}(p_x) \cdot y = x' \cdot y$$
$$\mathrm{src}(x' \triangleleft p_y) = x' \cdot \mathrm{src}(p_y) = x' \cdot y$$

We also have:

$$\mathrm{src}(p) = \mathrm{src}(p_x \triangleright y) = \mathrm{src}(p_x) \cdot y = x \cdot y$$
$$\mathrm{dst}(p) = \mathrm{dst}(x' \triangleleft p_y) = x' \cdot \mathrm{dst}(p_y) = x' \cdot y'$$

The existence of this rewrite path establishes that $xy \sim x'y'$. A pictorial proof is shown in Figure 16.1. □

Figure 16.1.: Term concatenation is well-defined on the equivalence classes of $\sim$



Example 16.29. We can apply this theorem to the bicyclic monoid $\langle a, b \mid ab \sim \varepsilon \rangle$ from Example 16.12. Consider the four terms $x := baba$, $x' := ba$, $y := a$, $y' := aba$. We have $x \sim x'$ and $y \sim y'$ via these two rewrite paths:

$$p_x := b(ab \Rightarrow \varepsilon)a$$
$$p_y := (\varepsilon \Rightarrow ab)a$$

Now, $(p_x \triangleright a) \circ (ba \triangleleft p_y)$ is a rewrite path from $babaa$ to $baaba$, showing that $xy \sim x'y'$:

$$b(ab \Rightarrow \varepsilon)aa \circ ba(\varepsilon \Rightarrow ab)a$$

Our original example showed that the bicyclic monoid can encode the problem of matching parentheses, by checking if a term is equivalent to ε. This was somewhat trivial, because we only looked within a single equivalence class, without making use of the monoid operation. A more impressive party trick is to use the bicyclic monoid to model the *stack effects* of programs in a *stack language*. This encoding relies on the monoid operation.

A stack language program is a sequence of tokens, where each token is either a *literal* or a *word*. To run the program, we evaluate each token sequentially from left to right:

1. A literal is immediately pushed on the stack.
2. A word, which is analogous to a function in other languages, pops some number of input values from the stack, processes those values, and then pushes zero or more results back on the stack.

To each token, we assign a *stack effect*, which is a pair of natural numbers (m, n) that encodes the number of inputs m and the number of outputs n that result from evaluating this token on the stack. It turns out if we encode each stack effect as an element $b^m a^n$ in the bicyclic monoid, then the monoid operation gives us a composition operation on stack effects.

Consider a stack language whose literals are integers and whose words are as below.

Token	**Description**	**Stack effect**
A literal	Any integer value.	a
`neg`	Pop the top of the stack and push its negation.	ba
`+`	Pop two values from the stack and push their sum.	b^2a
`*`	Pop two values from the stack and push their product.	b^2a
`dup`	Duplicate the top of the stack.	ba^2
`.`	Pop the top of the stack and print it out.	b

We can use this language to evaluate simple postfix arithmetic expressions:

```
5 dup * 3 neg + .
```

The above program has no overall effect on the stack; it will execute correctly when run on an empty stack, but if there were values on the stack, they remain unchanged. On the other hand, we can also have a program like "`dup * .`" which expects at least one input value to be present, otherwise the evaluation of "`dup`" will attempt to pop the top of an empty stack. Finally, we can write a program like "`5 7 + 1`" which expects no inputs, but leaves values behind.

We can determine the stack effect of each of these programs without evaluating them, as follows. We map each token to its stack effect using the above table, and then compose them together. The following closed form expression for the monoid operation is helpful here:

$$b^i a^j \cdot b^k a^l = \begin{cases} b^{i-j+k} a^l & \text{if } j < k \\ b^i a^{j-k+l} & \text{if } j \geq k \end{cases}$$

For example, we can calculate that the stack effect of "`5 dup * 3 neg + .`" as follows, confirming that this program has no effect on the stack:

$$a \cdot ba^2 \cdot b^2 a \cdot a \cdot ba \cdot b^2 a \cdot b = \varepsilon$$

Because the monoid operation is associative, it follows that the stack effect of the *concatenation* of two programs is equal to the composition of their stack effects. For this reason, stack-based languages are also known as *concatenative languages*. Examples of stack-based languages include Forth [143], Joy [144], and Factor [145]. The Factor compiler implements a static stack effect checker based on the idea we just described.

16.3. A Swift Connection

In this section, we will show that finitely-presented monoids map to Swift protocols in a very natural way. We make use of the derived requirements formalism we defined in Section 5.2, and some further results from Section 11.3. By now, it should be apparent that there are some similarities between the two theories, even if the concepts do not completely overlap:

Swift generics	**F-P monoids**
Type parameters	Terms
Explicit requirements	Rewrite rules
Derived requirements	Rewrite paths
Reduced type equality	Term equivalence
Type parameter graph	Cayley graph

Free monoids. The below protocol—or rather the protocol generic signature $G_{\texttt{M}}$—models the free monoid over the two generators $\{a, b\}^*$:

```
protocol M {
  associatedtype A: M
  associatedtype B: M
}
```

We define a function φ that sends each element of $\{a, b\}^*$ to a type parameter of $G_{\texttt{M}}$ by recursively constructing a dependent member type for each "a" and "b", until we finally get to the empty term, which becomes the protocol `Self` type:

$$\varphi(\varepsilon) := \texttt{Self}$$
$$\varphi(ua) := \varphi(u)\texttt{.A}$$
$$\varphi(ub) := \varphi(u)\texttt{.B}$$

For example, the four distinct terms of length 2 map to the following type parameters:

$$\varphi(aa) = \texttt{Self.A.A}$$
$$\varphi(ab) = \texttt{Self.A.B}$$
$$\varphi(ba) = \texttt{Self.B.A}$$
$$\varphi(bb) = \texttt{Self.B.B}$$

The requirement signature of `M` also declares two associated conformance requirements:

$$[\texttt{Self.A: M}]_{\texttt{M}}$$
$$[\texttt{Self.B: M}]_{\texttt{M}}$$

As a consequence of these requirements, every type parameter output by φ is a valid type parameter of $G_{\texttt{M}}$, and every such type parameter also conforms to `M`:

Proposition 16.30. If $t \in \{a, b\}^*$ and $\texttt{T} := \varphi(t)$, then $G_{\texttt{M}} \vdash \texttt{T}$ and $G_{\texttt{M}} \vdash [\texttt{T: M}]$.

Proof. We proceed by induction on the length of t.

Base case. If $|t| = 0$, then $t = \varepsilon$, and $\varphi(\varepsilon) = \texttt{Self}$. We can derive `Self` via a Generic elementary statement:

1. `Self` (Generic)

Likewise we get the conformance via a Conf elementary statement:

2. `[Self: M]` (Conf)

Inductive step. We know that $t = ua$ or $t = ub$ for some $u \in A^*$, so let $\texttt{U} := \varphi(u)$. By induction, $G_{\texttt{M}} \vdash \texttt{U}$ and $G_{\texttt{M}} \vdash [\texttt{U: M}]$. (We actually only need the latter.) First, we derive `U.A` or `U.B` by applying the AssocName inference rule to `[U: M]`:

1. `[U: M]` (...)
2. `U.A` (AssocName 1)
3. `U.B` (AssocName 1)

To derive `[U.A: M]` or `[U.B: M]` from `[U: M]`, we apply the AssocConf inference rule for the appropriate associated requirement:

1. `[U: M]` (...)
2. `[U.A: M]` (AssocConf 1)
3. `[U.B: M]` (AssocConf 1)

This completes the induction. □

We wish to equip the type parameters of $G_{\texttt{M}}$ with the structure of a monoid. The identity element we take to be `Self`, since this is $\varphi(\varepsilon)$.

The binary operation " $\cdot$ " is defined by *formal substitution*, replacing `Self` in the type parameter on the right-hand side with the entire type parameter on the left. Consider the two type parameters `Self.A.A` and `Self.B.B`:

$$(\texttt{Self.A.A}) \cdot (\texttt{Self.B.B}) = \texttt{Self.A.A.B.B}$$

To ensure this gives us a valid type parameter, we turn to Lemma 11.8. Suppose we are given two valid type parameters $G_{\texttt{M}} \vdash \texttt{U}$ and $G_{\texttt{M}} \vdash \texttt{V}$. By Proposition 16.30, we know that $G_{\texttt{M}} \vdash [\texttt{U: M}]$. The conditions of Lemma 11.8 are satisfied, thus $G_{\texttt{M}} \vdash \texttt{U} \cdot \texttt{V}$.

We now have a set of elements, a binary operation, and an identity element. The final question is, in what sense is this monoid "equivalent" to the free monoid $\{a, b\}^*$? The answer lies in the fact that φ is an *isomorphism* of monoids.

Definition 16.31. Let M and N be monoids.

1. A *monoid homomorphism* is a function $f\colon M \to N$ that sends the identity to the identity, and is compatible with the binary operation. That is, $f(\varepsilon_M) = \varepsilon_N$, and for all $x, y \in M$, $f(x \cdot_M y) = f(x) \cdot_N f(y)$.
2. A *monoid isomorphism* is a monoid homomorphism f which has an inverse f^{-1}, so $f^{-1}(f(x)) = x$ for all $x \in M$ and $f(f^{-1}(f(x)) = x$ for all $x \in N$. In this case we also say that M and N are *isomorphic*.

We already saw that φ maps the identity element ε of $\{a, b\}^*$ to the identity element `Self` of $G_{\texttt{M}}$. It also respects the monoid operation. For example,

$$\begin{aligned}
&\varphi(aa) \cdot \varphi(bb) \\
&\qquad = (\texttt{Self.A.A}) \cdot (\texttt{Self.B.B}) \\
&\qquad = \texttt{Self.A.A.B.B} \\
&\qquad = \varphi(aabb)
\end{aligned}$$

Finally, there is an inverse φ^{-1} mapping the type parameters of $G_{\texttt{M}}$ to elements of $\{a, b\}^*$:

$$\begin{aligned}
&\varphi^{-1}(\texttt{Self}) := \varepsilon \\
&\varphi^{-1}(\texttt{U.A}) := \varphi^{-1}(\texttt{U}) \cdot a \\
&\varphi^{-1}(\texttt{U.B}) := \varphi^{-1}(\texttt{U}) \cdot b
\end{aligned}$$

We've shown that the type parameters of $G_{\texttt{M}}$ have a monoid structure isomorphic to $\{a, b\}^*$. Furthermore, nothing in our construction relied on there being exactly two generators; we can add or remove associated type declarations in `M` as needed. For example, if we apply this construction to the free monoid with one generator, we get protocol `N` from Example 8.8. This monoid is isomorphic to $(\mathbb{N}, +, 0)$, which justifies the name `N`.

Finitely-presented monoids. We extend our encoding from a free monoid A^* to a finitely-presented monoid $\langle A \,|\, R\rangle$. We start with a protocol `M` that encodes A^*, then for each $(u, v) \in R$, we apply φ to u and v to obtain an associated same-type requirement $[\varphi(u)$ `==` $\varphi(v)]_{\texttt{M}}$. Finally, we attach a trailing `where` clause to `M` where we state these requirements. Let's try the bicyclic monoid $\langle a, b \,|\, ab \sim \varepsilon\rangle$ from Example 16.12:

```
protocol M {
  associatedtype A: M
  associatedtype B: M where Self.A.B == Self
}
```

The same-type requirements of `M` give the protocol generic signature $G_{\texttt{M}}$ a non-trivial equivalence class structure under the reduced type equality relation.

Example 16.32. Recalling Example 16.17, we saw that $baaba \sim babaa$ in the bicyclic monoid, with $ba(ab \Rightarrow \varepsilon)a \circ b(\varepsilon \Rightarrow ab)aa$ being one possible rewrite path from $baaba$ to $babaa$. Let `M` be the protocol that encodes this monoid. We can construct a derivation of $G_{\texttt{M}} \vdash [\texttt{Self.B.A.A.B.A == Self.B.A.B.A.A}]$ by translating each rewrite step:

$ba(ab \Rightarrow \varepsilon)a$ $\qquad$ $G_{\texttt{M}} \vdash [\texttt{Self.B.A.A.B.A == Self.B.A.A}]$

$b(\varepsilon \Rightarrow ab)aa$ $\qquad$ $G_{\texttt{M}} \vdash [\texttt{Self.B.A.A == Self.B.A.B.A.A}]$

The first rewrite step, $ba(ab \Rightarrow \varepsilon)a$, proves an equivalence between $baaba$ and baa. We will build a derivation piecemeal. We start with the left whisker ba:

1. [`Self: M`] (Conf)
2. [`Self.B: M`] (AssocConf 1)
3. [`Self.B.A: M`] (AssocConf 2)

We then apply the AssocSame inference rule to get $ba(ab \Rightarrow \varepsilon)$:

4. [`Self.B.A.A.B == Self.B.A`] (AssocSame 3)

Finally, we apply the SameName inference rule to get $ba(ab \Rightarrow \varepsilon)a$:

5. [`Self.B.A.A.B.A == Self.B.A.A`] (SameName 3 4)

The second rewrite step, $b(\varepsilon \Rightarrow ab)aa$, proves an equivalence between baa and $babaa$. We can reuse some of the above derivation steps to get $b(\varepsilon \Rightarrow ab)$:

6. [`Self.B.A.B == Self.B`] (AssocSame 2)
7. [`Self.B == Self.B.A.B`] (Sym 6)

We then build $b(\varepsilon \Rightarrow ab)a$:

8. [`Self.B.A.B: M`] (AssocConf 3)
9. [`Self.B.A == Self.B.A.B.A`] (SameName 8 7)

One more a, and we get $b(\varepsilon \Rightarrow ab)aa$:

10. [`Self.B.A.B.A: M`] (AssocConf 8)
11. [`Self.B.A.A == Self.B.A.B.A.A`] (SameName 10 9)

Finally, we compose the two rewriting steps with Trans, and we're done:

12. [`Self.B.A.A.B.A == Self.B.A.B.A.A`] (Trans 9 11)

We now prove that for any monoid presentation $\langle A \mid R \rangle$, the corresponding generic signature $G_{\texttt{M}}$ has the same equivalence class structure as $\sim_R$:

1. If $x \sim y$, then $\varphi(x)$ and $\varphi(y)$ are equivalent type parameters in $G_{\texttt{M}}$.
2. If $\varphi(x)$ and $\varphi(y)$ are equivalent type parameters in $G_{\texttt{M}}$, then $x \sim y$.

Theorem 16.33. Let $\langle A \mid R \rangle$ be a monoid presentation, and let `M` be the protocol that encodes $\langle A \mid R \rangle$. Then for all $x, y \in A^*$, $x \sim_R y$ implies $G_{\texttt{M}} \vdash [\varphi(x) \texttt{ == } \varphi(y)]$.

Proof. We are given a rewrite path p with $\mathrm{src}(p) = x$ and $\mathrm{dst}(p) = y$, and we must construct a derivation of $G_{\texttt{M}} \vdash [\varphi(x) \texttt{ == } \varphi(y)]$. We proceed by induction on the length of the rewrite path p.

Base case. We have an empty rewrite path, so $p = 1_t$ for some $t \in A^*$. We construct a derivation of $G_{\texttt{M}} \vdash \varphi(t)$ using Proposition 16.30, and then apply the Reflex inference rule to derive $[\varphi(t) \texttt{ == } \varphi(t)]$:

1. $\varphi(t)$ $(\ldots)$
2. $[\varphi(t) \texttt{ == } \varphi(t)]$ (Reflex 1)

Inductive step. Suppose that p has length at least 1, so that $p = p' \circ s$ for some shorter rewrite path p' and rewrite step s. Let's say that $s = t(u \Rightarrow v)w$ where $t, w \in A^*$, and (u, v) or $(v, u) \in R$. Let's look at the source and destination of p' and s:

$$\mathrm{src}(p') = x \qquad \mathrm{src}(s) = tuw$$
$$\mathrm{dst}(p') = tuw \qquad \mathrm{dst}(s) = tvw = y$$

By induction, we know that $G_{\texttt{M}} \vdash [\varphi(x) \texttt{ == } \varphi(tuw)]$. It remains to construct a derivation of $G_{\texttt{M}} \vdash [\varphi(tuw) \texttt{ == } \varphi(tvw)]$. We do this by first constructing $G_{\texttt{M}} \vdash [\varphi(tu) \texttt{ == } \varphi(tv)]$.

By Proposition 16.30, we know $G_{\texttt{M}} \vdash [\varphi(t)\texttt{: M}]$. Now, consider the direction of s:

1. If s is positive, then `M` states an associated same-type requirement $[\varphi(u) \texttt{ == } \varphi(v)]_{\texttt{M}}$. We apply AssocSame to $[\varphi(t)\texttt{: M}]$, so we get $G_{\texttt{M}} \vdash [\varphi(tu) \texttt{ == } \varphi(tv)]$.
2. If s is negative, AssocSame gives us $G_{\texttt{M}} \vdash [\varphi(tv) \texttt{ == } \varphi(tu)]$ instead, so we use the Sym inference rule to get $G_{\texttt{M}} \vdash [\varphi(tu) \texttt{ == } \varphi(tv)]$.

Next, Proposition 16.30 gives us $G_{\texttt{M}} \vdash \varphi(w)$. Since we have $G_{\texttt{M}} \vdash [\varphi(tu) \texttt{ == } \varphi(tv)]$ and $G_{\texttt{M}} \vdash \varphi(w)$, Lemma 11.8 implies that $G_{\texttt{M}} \vdash [\varphi(tuw) \texttt{ == } \varphi(tvw)]$.

Finally, we use Trans to derive $G_{\texttt{M}} \vdash [\varphi(x) \texttt{ == } \varphi(y)]$ (recall that $y = tvw$):

1. $[\varphi(x) \texttt{ == } \varphi(tuw)]$ $(\ldots)$
2. $[\varphi(tuw) \texttt{ == } \varphi(y)]$ $(\ldots)$
3. $[\varphi(x) \texttt{ == } \varphi(y)]$ (Trans 1 2)

Thus, every rewrite path p maps to a derived same-type requirement in $G_{\texttt{M}}$. □

Theorem 16.34. Let $\langle A \mid R \rangle$ be a monoid presentation, and let `M` be the protocol that encodes $\langle A \mid R \rangle$. Then for all x, $y \in A^*$, $G_{\texttt{M}} \vdash [\varphi(x) \texttt{ == } \varphi(y)]$ implies that $x \sim_R y$.

Proof. We are given a derivation of $[\varphi(x) \texttt{ == } \varphi(y)]$, and we must show that $x \sim y$ by constructing a rewrite path p with source x and destination y. We proceed by structural induction on derived requirements.

The following derivation steps produce same-type requirements (however, Same does not come up in our case, because $G_{\texttt{M}}$ does not have explicit same-type requirements; all same-type requirements are derived from the requirement signature of `M`):

Same	AssocSame	SameName
Reflex	Sym	Trans

In each case, we must construct a new rewrite path p whose source and destination match the same-type requirement in the conclusion of the step, from the rewrite paths corresponding to each assumption used by the step, if any.

Base case. The base case is when our derivation step does not have any same-type requirement assumptions, but the conclusion is some same-type requirement. There are two such kinds of steps, and in both we construct the rewrite path p directly.

An application of the AssocSame inference rule, when written in the form below, implies that we have a rewrite rule $(u, v) \in R$. We set $p := t(u \Rightarrow v)$:

$$[\varphi(tu) \texttt{ == } \varphi(tv)] \qquad (\text{AssocSame } [\varphi(t)\texttt{: M}])$$

The other base case is a Reflex step, where we set $p := 1_t$:

$$[\varphi(t) \texttt{ == } \varphi(t)] \qquad (\text{Reflex } \varphi(t))$$

Inductive step. We handle Sym by inverting our rewrite path. Induction gives us p' with $\mathrm{src}(p') = t$ and $\mathrm{dst}(p') = u$. We set $p := (p')^{-1}$, because $\mathrm{src}(p) = u$ and $\mathrm{dst}(p) = t$:

$$[\varphi(u) \texttt{ == } \varphi(t)] \qquad (\text{Sym } [\varphi(t) \texttt{ == } \varphi(u)])$$

We handle Trans by composition. Induction gives us p_1, p_2 with $\mathrm{src}(p_1) = t$, $\mathrm{dst}(p_1) = \mathrm{src}(p_2) = u$, and $\mathrm{dst}(p_2) = v$. We set $p := p_1 \circ p_2$, because $\mathrm{src}(p) = t$ and $\mathrm{dst}(p) = v$:

$$[\varphi(t) \texttt{ == } \varphi(v)] \qquad (\text{Trans } [\varphi(t) \texttt{ == } \varphi(u)]\ [\varphi(u) \texttt{ == } \varphi(v)])$$

Finally, we handle SameName by whiskering. Here, $g \in A$ is a generator, corresponding to an associated type declaration in `M`. Induction gives us p' with $\mathrm{src}(p') = t$ and $\mathrm{dst}(p') = u$. We set $p := p' \triangleright g$, because $\mathrm{src}(p) = tg$ and $\mathrm{dst}(p) = ug$:

$$[\varphi(tg) \texttt{ == } \varphi(ug)] \qquad (\text{SameName } [\varphi(u)\texttt{: M}]\ [\varphi(t) \texttt{ == } \varphi(u)])$$

Thus, every derived same-type requirement of $G_{\texttt{M}}$ maps to a rewrite path p. □

In general, the Cayley graph is not preserved by a monoid isomorphism, because it depends on the choice of generating set. However, in our Swift encoding, the Cayley graph manifests directly. We now know that `N` and `Z4` encode the monoids $\{a\}^*$ and $\langle a \,|\, a^4 \sim \varepsilon \rangle$ respectively. We also previously remarked that:

- The Cayley graph of $\{a\}^*$ (Example 16.7) looks like the type parameter graph of $G_{\texttt{N}}$ (Example 8.8).
- The Cayley graph of $\langle a \,|\, a^4 \sim \varepsilon \rangle$ (Example 16.10) looks like the type parameter graph of $G_{\texttt{Z4}}$ (Example 8.9).

This always works, and furthermore the conformance path graph of Section 12.2 coincides with the other two graphs in this case as well:

Theorem 16.35. Let $\langle A \,|\, R \rangle$ be a monoid presentation, and let `M` be the protocol that encodes $\langle A \,|\, R \rangle$. The following three graphs are isomorphic:

1. The Cayley graph of $\langle A \,|\, R \rangle$.
2. The type parameter graph of $G_{\texttt{M}}$.
3. The conformance path graph of $G_{\texttt{M}}$.

Proof. First, consider the vertex set of each graph. We've already established that terms and type parameters have the same equivalence class structure. From this it follows the only difference between the vertex sets is essentially our choice of labels:

1. A vertex in (1) is an equivalence class of terms: $[\![t]\!]$ for some $t \in A^*$.
2. A vertex in (2) is an equivalence class of type parameters: $[\![\varphi(t)]\!]$ for some $t \in A^*$.
3. A vertex in (3) is an equivalence class of abstract conformances. Since every type parameter of $G_{\texttt{M}}$ conforms to exactly one protocol—`M`—every abstract conformance has the form $[\varphi(t)\texttt{: M}]$ for some $t \in A^*$.

The same can be said about the edge sets as well:

1. An edge in (1) joins each $[\![t]\!]$ with $[\![tg]\!]$, for all $a \in A$.
2. An edge in (2) joins each $[\![\varphi(t)]\!]$ with $[\![\varphi(t)\texttt{.A}]\!]$, for all associated types `A` of `M`.
3. An edge in (3) joins each $[[\![\varphi(t)]\!]\texttt{: M}]$ with $[[\![\varphi(t)\texttt{.A}]\!]\texttt{: M}]$, for all associated conformance requirements $[\texttt{Self.A: M}]_{\texttt{M}}$ of `M`.

One final remark. Recall that in the Cayley graph of a monoid presentation $\langle A \,|\, R \rangle$, every vertex has the same number of successors, and we just showed that the same is true of the type parameter graph of $G_{\texttt{M}}$. On the other hand, in an arbitrary generic signature, the number of successors of a vertex will vary, because equivalence classes will conform to various sets of protocols with distinct associated types. The type parameter graph of a "typical" generic signature is almost never the Cayley graph of a monoid. □

16.4. The Word Problem

The previous section's Theorem 16.33 and Theorem 16.34 tell us that we can write a program that type checks if and only if two terms in a finitely-presented monoid are equivalent. The compiler must be able to verify the equivalence, or reject the program if it does not hold.

For example, we can start with our protocol encoding the bicyclic monoid, and also declare a method in a protocol extension:

```
protocol M {
  associatedtype A: M
  associatedtype B: M where Self.A.B == Self
}

extension M {
  static func testBicyclicMonoid() {
    sameType(Self.B.A.A.B.A.self, Self.B.A.B.A.A.self)  // ok
    sameType(Self.A.A.A.self, Self.B.B.B.self)          // error!
  }
}

func sameType<T>(_: T.Type, _: T.Type) {}
```

A call to `sameType()` requires that both arguments are metatypes with the same instance type. For this program to type check, the compiler must verify two statements, corresponding to each call of `sameType()`:

$$G_{\texttt{M}} \vdash [\texttt{Self.B.A.A.B.A == Self.B.A.B.A.A}]$$
$$G_{\texttt{M}} \vdash [\texttt{Self.B.A.A == Self.B.B.B}]$$

The first statement is true, because $baaba \sim babaa$, as we've seen. However, the second statement is actually false, and in fact $baa \not\sim bbb$. The compiler accepts the first call, and diagnoses an error on the second call to `sameType()`, as we would expect.

In fact, if we write a similar program for every example of a finitely-presented monoid we've seen so far, the Swift compiler will be able to correctly decide for us if any two arbitrary terms are equivalent. A natural question is, does this always work, or are there some protocol declarations constructed this way that we cannot accept?

We're asking the Swift compiler to solve the well-known *word problem*:

- **Input:** Monoid presentation $\langle A \,|\, R \rangle$, two terms x, $y \in A^*$.

  Result: True or false: $x \sim_R y$?

The word problem was perhaps first posed in the early 20th century by Axel Thue, who studied various formalisms defined by the replacement of substrings within larger strings via a fixed set of rules. Thue described what later became known as *Thue systems* and *semi-Thue systems* in a 1914 paper [146]. A Thue system is a finitely-presented monoid, while a semi-Thue system is what we get if we replace the symmetric term equivalence relation with a *reduction relation*, which we will study in the next section.

It seems that Thue's goal was in many ways similar to ours: to analyze a formal system by encoding its statements as terms in a finitely-presented monoid, such that the proof of the statement becomes a sequence of rewrite steps. Questions about the formal system then become instances of the word problem. The final task was to figure out this missing piece, and—voilá—we've solved all of mathematics.

Thue was able to solve the word problem in certain restricted instances. For example, if the rewrite rules preserve term length, then every equivalence class must be finite, and $x \sim y$ can be decided by exhaustive enumeration of $[\![y]\!]$. What eluded Thue though, was a general approach that worked in all cases.

The next twist in this saga came after the development of computability theory. We already sketched out Turing machines and the halting problem in Section 12.4, where we saw that the question of *termination checking* is undecidable in our type substitution algebra. Now, we will see another such undecidable problem. In a 1947 paper [147], Emil Post gave a proof that no effective procedure can solve the word problem. Post did this by defining an encoding of a Turing machine as a Thue system. At a very high level, this encoding works as follows:

1. At any point in time, the complete state of the Turing machine, consisting of the current state symbol and the contents of the tape, can be described by a term in the finitely-presented monoid.

2. The single-step transitions, where the machine replaces the symbol at the current head position and moves left or right, define the monoid's rewrite rules.

3. A sequential execution defines a rewrite path. If the machine halts, we know there exists a rewrite path from the initial state to the halting state. Thus, the Turing machine halts if and only if the initial and halting states are equivalent terms.

If we could solve the word problem in an *arbitrary* finitely-presented monoid, we could also solve the halting problem for an arbitrary Turing machine—a contradiction, so no such algorithm exists. Even more is true. Not only is there no "universal" algorithm that takes the monoid presentation as part of its input, but we can construct a *specific* finitely-presented monoid with an undecidable word problem.

One approach is to start with Turing's construction of a *universal machine*. This is a Turing machine that simulates the execution of another Turing machine one step at a time, receiving the standard description of the other Turing machine as input on its tape. These days, we might call this an "emulator" or "interpreter."

By applying Post's construction to a universal Turing machine, we can get a specific monoid presentation with an undecidable word problem, because we can now encode an arbitrary Turing machine as a term in this specific monoid, and then determine if the Turing machine halts by solving the word problem. The "universal Turing machine" monoid has a large number of generators and rewrite rules, though.

A much simpler monoid with an undecidable word problem appeared in a 1958 paper by G. S. Tseitin [148]. For an English translation with commentary, see [149].

Theorem 16.36. This finitely-presented monoid has an undecidable word problem:

$$\begin{aligned}\mathfrak{C}_1 := \langle a, b, c, d, e \,|\, & ac \sim ca,\ ad \sim da,\ bc \sim cb,\ bd \sim db, \\ & eca \sim ce,\ edb \sim de, \\ & cca \sim ccae \rangle\end{aligned}$$

Corollary 16.37. There does not exist an effective procedure to decide if two arbitrary type parameters are equivalent in the protocol generic signature $G_{\texttt{C1}}$:

```
protocol C1 {
  associatedtype A: C1
  associatedtype B: C1
  associatedtype C: C1
  associatedtype D: C1
  associatedtype E: C1
    where A.C == C.A, A.D == D.A, B.C == C.B, B.D == D.B,
          E.C.A == C.E, E.D.B == D.E,
          C.C.A == C.C.A.E
}
```

The Swift compiler *rejects* this protocol and diagnoses an error:

```
error: cannot build rewrite system for protocol;
rule length limit exceeded
```

The "rule length" in the error refers not to the rules we wrote down, but the rules generated while constructing a *convergent* presentation for $\mathfrak{C}_1$, something we get to in Section 16.5. If our monoid has an undecidable word problem, this cannot be done, and the procedure for generating such a presentation uses various heuristics to know when to give up. On the author's machine, the Swift compiler rejects protocol `C1` after about 30 milliseconds of processing.

To understand $\mathfrak{C}_1$, we need to introduce a new concept. A *special monoid presentation* is one where the right-hand side of each rewrite rule is ε, and a *special monoid* is one that has such a presentation:

$$\langle a_1, \ldots, a_m \,|\, u_1 \sim \varepsilon,\ \ldots,\ u_n \sim \varepsilon \rangle$$

The bicyclic monoid from Example 16.12 is special. Every group is special, because of the inverse operation; we can always replace a rule $u \sim v$ with $uv^{-1} \sim \varepsilon$.

Tseitin's monoid $\mathfrak{C}_1$ is a *universal machine for solving word problems in special monoids*. If we take a special monoid presentation $\langle A \mid R \rangle$ and a pair of terms $x, y \in A^*$, we can encode $\langle A \mid R \rangle$, x, and y as three terms W, X, and Y over the generators of $\mathfrak{C}_1$. Then, Tseitin's key result is the following:

$$x \sim y \quad \text{in } \langle A \mid R \rangle \qquad \text{if and only if} \qquad WX \sim WY \quad \text{in } \mathfrak{C}_1$$

We will sketch this out by encoding a word problem in the *dihedral group of order 12*, which describes the symmetries of a regular hexagon, as a word problem in $\mathfrak{C}_1$. Think of s as rotation by 60°, and t as reflection by an axis of symmetry:

$$D_{12} := \langle s, t \mid s^6 \sim \varepsilon,\ t^2 \sim \varepsilon,\ (st)^2 \sim \varepsilon \rangle$$

(First step.) We encode the presentation of D_{12} first. We translate each rewrite rule of D_{12} into an element of $\mathfrak{C}_1$, by replacing s with cd and t with cd^2 (if D_{12} had a third generator, it would map to cd^3, and so on):

$$\begin{aligned} s^6 &\mapsto (cd)^6 \\ t^2 &\mapsto (cd^2)^2 \\ (st)^2 &\mapsto (cdcd^2)^2 \end{aligned}$$

(Second step.) We append c at the end of each term above, concatenate them together, and add one more c at the end. We've encoded our *presentation* of D_{12} as an *element* of $\mathfrak{C}_1$. This is our term W:

$$W := \underbrace{(cd)^6}_{s^6} \cdot c \cdot \underbrace{(cd^2)^2}_{t^2} \cdot c \cdot \underbrace{(cdcd^2)^2}_{(st)^2} \cdot c \cdot c$$

(Third step.) Now we pick a pair of terms from D_{12}. Let's take $x := ts$ and $y := s^5t$. A rotation followed by a reflection is the same as a reflection followed by a rotation in the opposite direction, so $ts \sim s^5t$. We again translate these terms into $\mathfrak{C}_1$, except using a and b instead of c and d. This gives us our X and Y:

$$\begin{aligned} X &:= ab^2ab \\ Y &:= (ab)^5ab^2 \end{aligned}$$

Tseitin's rewrite rules define a (rather long) rewrite path from WX to WY.

Our example does not exhibit undecidability, because the word problem in D_{12} is very easy to solve. There are only 12 elements, so we can just write down the Cayley table. However, as with monoids, the word problem for groups is undecidable in the general case [150] (for a more accessible introduction, see [151], or Chapter 12 of [152]). Because $\mathfrak{C}_1$ can encode the word problem in *all* groups, we certainly cannot hope to write down an algorithm to solve the word problem in $\mathfrak{C}_1$.

The big picture. We've shown that every finitely-presented monoid can be encoded as a specific kind of Swift protocol, almost as if Swift's protocols are a generalization of the finitely-presented monoids! This doesn't immediately help us implement Swift generics, though. However, Chapter 17 shows that by taking a suitable alphabet and set of rewrite rules, we can go in the other direction and encode any Swift generic signature as a finitely-presented monoid. While the undecidability of the word problem prevents us from accepting *all* generic signatures, the next section will describe a large class of finitely-presented monoids where the word problem is easy to solve. All reasonable generic signatures fit in with our model.

Closing remarks. In Section 12.4, we showed that the type substitution algebra can encode arbitrary computation with recursive conformance requirements [105]. Now we see that using recursive conformance requirements and protocol `where` clauses [77], we can encode an undecidable problem in the derived requirements formalism as well.

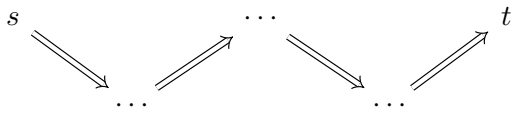
Tseitin's monoid is not a special monoid, so it cannot encode its own word problem. D. J. Collins [153] later discovered a "more" universal word problem interpreter having only a few more rules; this can encode word problems in $\mathfrak{C}_1$ or even itself:

$$\begin{aligned}\mathfrak{U} := \langle a, b, c, d, e_1, e_2, f, g \,|\, & ac \sim ca,\ ad \sim da,\ af \sim fa,\ ag \sim ga,\\ & bc \sim cb,\ bd \sim db,\ bf \sim fb,\ bg \sim gb,\\ & ce_1 \sim ec_1 a,\ de_1 \sim e_1 db,\\ & e_2 c \sim cae_2,\ e_2 d \sim dbe_2,\\ & e_1 g \sim ge_2,\\ & f \sim fe_1,\ e_2 f \sim f\rangle\end{aligned}$$

16.5. The Normal Form Algorithm

There is a "naive" solution to the word problem: we start from our first term s, and attempt different combinations of rewrite steps, until we end up at t. In fact, we can arrange the search so that if such a path does exist, the search always terminates with the appropriate rewrite path.

However, if there is no such path, we will never know when to stop. Recall that a rewrite rule $(u, v) \in R$ is a *symmetric* equivalence. A rewrite path from s to t, if it exists, may contain multiple occurrences of the same rewrite rule, applied in either or both directions over the course of the path. The rewrite graph is infinite, and the intermediate terms visited along our path might be arbitrarily long.



We now change our point of view, and think of a rewrite rule (u, v) as a directed *reduction* from u to v. That is, we're allowed to replace u with v anywhere it appears in a term, but not vice versa. Recall that a rewrite step $x(u \Rightarrow v)y$ is positive if $(u, v) \in R$, and negative if $(v, u) \in R$.

Definition 16.38. A *positive rewrite path* is one where each step taken is positive.

An empty rewrite path is positive, because it contains no steps. Positive rewrite paths are also closed under composition and whiskering, meaning that a) if p_1 and p_2 are positive, then so is $p_1 \circ p_2$; b) if p is positive and $z \in A^*$, so are $z \triangleleft p$ and $p \triangleright z$. (On the other hand, if p is positive, then p^{-1} is positive if and only if p is empty.)

Definition 16.39. Let $\langle A \,|\, R \rangle$ be a monoid presentation.

1. The *reduction relation* on A^* generated by R, denoted $s \to_R t$ or $s \to t$ when R is clear from context, relates each pair of terms s and t such that there exists a positive rewrite path with source s and destination t.

2. A term s is *irreducible* or *reduced* if $s \to t$ implies that $s = t$.

3. If $s \to t$ and t is irreducible, we say t is a *normal form* of s, denoted by $\tilde{s} = t$.

We can describe an algorithm that attempts to find normal forms.

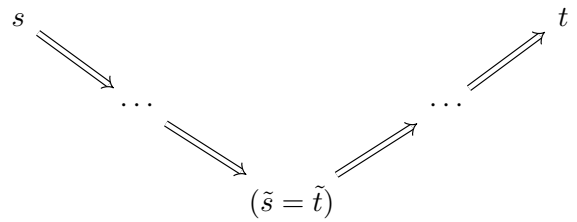
Algorithm 16A (Normal form algorithm)**.** As input, takes a term $t \in A^*$ and a list of rewrite rules $R := \{(u_1, v_1), \ldots (u_n, v_n)\}$. As output, returns some normal form of t, which we denote $\tilde{t}$; and a positive rewrite path p, such that $\mathrm{src}(p) = t$ and $\mathrm{dst}(p) = \tilde{t}$.

1. Initialize the return value p with the empty rewrite path 1_t.

2. If $t = xuy$ for some x, $y \in A^*$ and $(u, v) \in R$: set $t \leftarrow xvy$, set $p \leftarrow p \circ x(u \Rightarrow v)y$, and go back to Step 1.

3. Otherwise, return t and p.

Suppose we are given a pair of terms s, $t \in A^*$ and we wish to decide if $s \sim t$. If our algorithm tells us that both have the same normal form, it means that $s \to \tilde{s}$ via p_s, $t \to \tilde{t}$ via p_t, and also $\tilde{s} = \tilde{t}$. We can then conclude that $s \sim t$, because $p_s \circ p_t^{-1}$ is a rewrite path (no longer positive, unless p_t is empty) with source s and destination t:



Solving the word problem by comparing normal forms is very appealing, but it cannot work in the general case, because we already know the word problem is undecidable. Broadly speaking, two things can go wrong:

1. **Existence:** Our description of the normal form algorithm apparently returns a value in Step 3, so what does it mean for the normal form of some term t to not exist? The trouble is that the algorithm might not *terminate*. In this case, we say that t does not have a normal form.

2. **Uniqueness:** We demonstrated that if $\tilde{s} = \tilde{t}$ for some s, $t \in A^*$, then $s \sim t$. The converse does not always hold, however: a single equivalence class of $\sim$ may contain more than one irreducible term of $\to$. In other words, there may exist s and t such that $s \sim t$, and yet $\tilde{s} \neq \tilde{t}$.

To understand when and how we can overcome these difficulties, we turn to the theory of *string rewriting*. The text by Book and Otto [154] is a classic. Alternatively, instead of introducing string rewriting by way of finitely-presented monoids as we do here, one could instead start with an *abstract* reduction relation operating on "tree-like" expressions, of which strings are a special case. This is the theory of *term rewriting*, and we invite the reader to also read [155] and [156] for a thorough study of this approach.

Existence. Consider the monoid with two elements $\{0, 1\}$ and the binary operation $x \cdot y := \max(x, y)$. (The identity is 0.) Here is one way to present this monoid:

$$\langle a \,|\, a \sim aa \rangle$$

Every non-empty term is *equivalent* to a, but *no* non-empty term has a normal form:

$$a \to aa \to aaa \to aaaa \to \dots$$

The normal form algorithm is quite useless here. However, if we replace our rewrite rule (a, aa) with (aa, a), the normal form algorithm now *always* terminates, with no change to the term equivalence relation. For example:

$$aaaa \to aaa \to aa \to a$$

Now, every non-empty term has the normal form a. In fact, if $\langle A \,|\, R \rangle$ has the property that for each $(u, v) \in R$, either $|u| < |v|$ or $|v| < |u|$, we can always *orient* our rewrite rules so that $|v| < |u|$. We say such rewrite rules are *length-reducing*. The ensures that the intermediate term t becomes shorter on each iteration of Step 2 in Algorithm 16A, after we replace u with v. This can only go on for finitely many steps, so we must always get a normal form.

Limiting ourselves to length-reducing rewrite rules is too restrictive; for example, recall that the free commutative monoid has the *length-preserving* rewrite rule (ba, ab). Thus, we must generalize the idea of comparing term length to establish termination.

Definition 16.40. Let A be a finite alphabet of symbols. A *reduction order* $<$ is a partial order on A^* that is also well-founded (Definition 5.24) and translation-invariant (Definition 16.25).

Definition 16.41. Let $\langle A \mid R \rangle$ be a monoid presentation. A rewrite rule $(u, v) \in R$ is *oriented* with respect to some reduction order $<$, if the right-hand side is smaller with respect to this order, so $u > v$. (Of course, if $<$ is not a linear order, it may happen that we have a *non-orientable* rewrite rule $(u, v) \in R$ where $u \not< v$ and $v \not< u$.)

If we can orient the rewrite rules of our monoid presentation using a reduction order, then the normal form algorithm will always find *some* normal form in a finite number of steps. This solves the first of our two problems:

Theorem 16.42. Let $\langle A \mid R \rangle$ be a monoid presentation. If the rewrite rules of R are oriented with respect to a reduction order $<$, then every equivalence class of $\sim$ contains *at least* one reduced term. We also say that the reduction relation $\to$ is *terminating* or *Noetherian* in this case.

Proof. Assume we have a non-terminating execution of Algorithm 16A, starting with some term t. This means we have an infinite sequence of terms, each one reducing to the next by a positive rewrite step:

$$t \to t_1 \to t_2 \to \cdots \to t_n \to \cdots$$

Now, let $s = x(u \Rightarrow v)y$ be a positive rewrite step. Since $(u, v) \in R$ is oriented, we have $u > v$. Also, $<$ is translation-invariant, so $xuy > xvy$. Thus, $\mathrm{src}(s) > \mathrm{dst}(s)$. This means our infinite reduction sequence gives us an infinite descending chain of terms under our reduction order:

$$t > t_1 > t_2 > \cdots > t_n > \cdots$$

However, this contradicts the assumption that $<$ is well-founded. Thus, no such infinite sequence of terms exists, and the normal form algorithm must terminate. □

If we equip our alphabet of symbols A with a well-founded partial order, we can always get a reduction order on A^*, called the *shortlex order*:

1. Terms of shorter length precede terms of longer length.
2. Otherwise, if two terms have the same length, we compare the symbols pairwise using the order on A.

For example, if $A := \{a, b, c\}$, we might say that $a < b < c$ to define a linear order on A; then we can order various terms of A^* using the shortlex order:

$$a < b < ab < ac < aab$$

Algorithm 16B (Shortlex order)**.** Takes two terms x, $y \in A^*$ as input, and returns one of "$<$", "$>$", "$=$" or "$\perp$" as output.

1. (Shorter) If $|x| < |y|$, return "$<$".
2. (Longer) If $|x| > |y|$, return "$>$".
3. (Initialize) If $|x| = |y|$, we compare elements. Let $i := 0$.
4. (Equal) If $i = |x|$, we didn't find any differences, so $x = y$. Return "$=$".
5. (Subscript) Let x_i and $y_i \in A$ be the ith symbol of x and y, respectively.
6. (Compare) Compare x_i with y_i using the order on A. Return the result if it is "$<$", "$>$", or "$\perp$". Otherwise, we must have $x_i = y_i$, so keep going.
7. (Next) Increment i and go back to Step 4.

The shortlex order will return "$\perp$" if and only if two equal-length terms x and y have an incomparable pair of symbols at the same position, under the partial order on A. In particular, if the order on A is linear, the induced shortlex order is always a linear order on A^*. We also omit two simple proofs. It can also be shown that the shortlex order is well-founded using the same argument as in Proposition 5.25. Finally, one can show that the shortlex order is translation-invariant by induction on term length.

Example 16.43. The shortlex order is *not* the same as the lexicographic "dictionary" order on strings. For example, $b < ab$ under the shortlex order, but $ab < b$ under the lexicographic order. The lexicographic order is *not* well-founded, thus it is unsuitable for use as a reduction order:

$$b > ab > aab > aaab > aaaab > \cdots$$

At this point, we pause to observe that in all of our examples so far except the undecidable $\mathfrak{C}_1$, the normal form algorithm solves the word problem once we establish termination:

1. The monoid $\langle a \,|\, a^4 \sim \varepsilon \rangle$ from Example 16.10 has a single length-reducing rewrite rule. The normal form algorithm will output one of $\{\varepsilon, a, a^2, a^3\}$.
2. The bicyclic monoid $\langle a, b \,|\, ab \sim \varepsilon \rangle$ from Example 16.12 has a single length-reducing rewrite rule. The normal form algorithm will find the unique term of the form $b^m a^n$ in each equivalence class.
3. The free commutative monoid $\langle a, b \,|\, ba \sim ab \rangle$ from Example 16.11 can be equipped with the shortlex order where $a < b$ to establish termination. The normal form algorithm will find in each equivalence class a unique term of the form $a^m b^n$.

The three examples above all enjoy unique normal forms, something that is not true of an arbitrary monoid presentation.

Uniqueness. We now describe monoid presentations with unique normal forms:

Theorem 16.44 (Church-Rosser Theorem)**.** Let $\langle A \mid R \rangle$ be a monoid presentation. The following two conditions are equivalent:

1. (Confluence) For all $t \in A^*$, if $t \to u$ and $t \to v$ for some u, $v \in A^*$, there exists a term $z \in A^*$ such that $u \to z$ and $v \to z$.
2. (Church-Rosser property) For all x, $y \in A^*$ such that $x \sim y$, there exists $z \in A^*$ such that $x \to z$ and $y \to z$.

The Church-Rosser Theorem was discovered by Alonzo Church and John Rosser [157]. The original result was about the lambda calculus, which is a rewriting system whose terms are combinations of variables, function applications, and anonymous functions. Lambda calculus is Turing-complete, so while reduction may not terminate with a given lambda term, it is always confluent, so the Church-Rosser property holds. Haskell Curry (whom the Haskell language was named after) subsequently generalized the theorem away from the lambda calculus in [158]. We use the form of the statement where the terms are elements in a free monoid; a proof can be found in [154].

Example 16.45. Consider the following monoid presentation:

$$\langle a, b, c \mid ab \sim a,\ bc \sim b \rangle$$

The reduction relation is Noetherian, because the rules are length-reducing. However, we will see the reduction relation is *not* confluent, so the Church-Rosser property fails: we can find two irreducible terms in the same equivalence class.

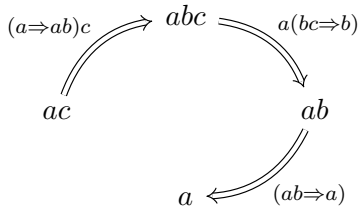
First, observe that both ac and a are irreducible: neither term has any subterms equal to the left hand side of either rule. Now, let's play with the term abc. We can begin by applying either of the two rewrite rules:

1. Applying the first rule to abc gives us ac, which is irreducible.
2. Applying the second rule to abc gives us ab, which reduces to a via the first rule, which is irreducible.

It appears that Algorithm 16A is not deterministic; in Step 2, we have a choice of rewrite rules to apply, and so we output one of two rewrite paths that both end in distinct irreducible terms:

$$p_1 := (ab \Rightarrow a)c$$
$$p_2 := a(bc \Rightarrow b) \circ (ab \Rightarrow a)$$

We've found a *confluence violation*. Note that $\mathrm{src}(p_1) = \mathrm{src}(p_2) = abc$, so $p_1^{-1} \circ p_2$ is a rewrite path from ac to a. Thus $ac \sim a$, and yet, they're distinct and irreducible. If we draw the rewrite path $p_1^{-1} \circ p_2$ as a diagram, we see that no positive rewrite path can take us from ac to a; we must go "over the hump":



Now, let's *add* the rewrite rule (ac, a) to our monoid presentation:

$$\langle a, b, c \,|\, ab \sim a,\ bc \sim b,\ ac \sim a \rangle$$

We *already had* a rewrite path from ac to a, so adding (ac, a) does not change the equivalence relation $\sim$ (we'll see this is a so-called Tietze transformation in Section 18.4). It *does* change $\to$; in particular, the reduction relation becomes confluent.

In our new presentation, the non-determinism disappears; the normal form algorithm always reduces abc to a, regardless of which rewrite rule we choose to apply first. In fact, every equivalence class now has a unique normal form $c^i b^j a^k$ for i, j, $k \in \mathbb{N}$.

We give a name to this kind of monoid presentation where the normal form algorithm always terminates and outputs a unique reduced term from each equivalence class.

Definition 16.46. A *convergent rewriting system* is a monoid presentation $\langle A \,|\, R \rangle$ whose reduction relation $\to_R$ is Noetherian and confluent.

A monoid presented by a convergent rewriting system has a decidable word problem:

Corollary 16.47. Let $\langle A \,|\, R \rangle$ be a convergent rewriting system. Then for all x, $y \in A^*$, we have $x \sim y$ if and only if $\tilde{x} = \tilde{y}$.

We add two final entries to the Rosetta stone we started in Section 16.3:

Swift generics	**F-P monoids**
Type parameter order	Shortlex order
Reduced type parameter	Reduced term

In fact, the type parameter order of Algorithm 5D is basically a specific instance of the shortlex order, except we described it with a recursive algorithm, whereas Algorithm 16B is iterative.

Completion. The process of repairing confluence violations by adding rules is called *completion*. If completion succeeds, we get a convergent rewriting system. Chapter 18 will explain the Knuth-Bendix algorithm that is used for this purpose. For example, the Swift compiler accepts a protocol `M` encoding the monoid presentation from Example 16.45; completion allows us to establish that $G_{\texttt{M}} \vdash [\texttt{Self.A.C == Self.A}]$.

Completion is only a *semi-decision procedure* that may fail to terminate; in practice, we impose an iteration limit. This cannot be improved upon, because ultimately, it is undecidable if a monoid can be presented by a convergent rewriting system [159]. A monoid with an undecidable word problem cannot have such a presentation, so completion must fail in that case. Is the converse true? That is, if a finitely-presented monoid is known to have a decidable word problem, will completion always terminate in a finite number of steps and output a convergent rewriting system? The answer is also no:

1. With a fixed presentation $\langle A \,|\, R \rangle$, completion may succeed or fail depending on the choice of reduction order $<$.

2. Two presentations $\langle A \,|\, R \rangle$ and $\langle A' \,|\, R' \rangle$ over different generating sets (alphabets) may present the same monoid up to isomorphism, but completion may succeed with one and fail with the other.

3. Some finitely-presented monoids are known to have a decidable word problem, but no convergent presentation over *any* generating set.

We will see in Section 18.5 that the avoidance of the first two issues motivates several design decisions in the lowering of generic signatures to monoid presentations. The second issue was originally explored by [160], where we learn that the monoid presentation $\langle a, b \,|\, aba \sim bab \rangle$ cannot be extended to a convergent presentation on the same alphabet, but by adding a new symbol c together with the rewrite rule $ab \sim c$, completion will quickly arrive at a convergent presentation.

The third case is more difficult. The below example is from a paper by Craig C. Squier [142]; also, see [161] and [162].

Theorem 16.48. The following finitely-presented monoid has a word problem decidable via a "bespoke" algorithm, but no convergent presentation over any generating set:

$$S_1 := \langle a, b, t, x, y \,|\, ab \sim \varepsilon,\ xa \sim atx,\ xt \sim tx,\ xb \sim bx,\ xy \sim \varepsilon \rangle$$

To prove this theorem, Squier introduced a new combinatorial property of the rewrite graph we met in Definition 16.15 (the term *derivation graph* is used in the paper), called *finite derivation type*. It can be shown that the property of having finite derivation type does not depend on the choice of presentation, so it is an invariant of the monoid itself. Furthermore, a convergent presentation necessarily has finite derivation type. (Having finite derivation type is a necessary, but not a sufficient, condition for the existence of a convergent presentation.) For example, $\langle a, b \,|\, aba \sim bab \rangle$ from above can be presented as a convergent rewriting system over a different alphabet, so this monoid necessarily has finite derivation type.

On the other hand, if it can be shown than a monoid does not have finite derivation type, we can conclude that no convergent presentation can exist over any generating set,

even if its word problem is decidable by other means. Squier was able to show that S_1 does not have finite derivation type, but it does have a decidable word problem.

So, each of the below classes is a proper subset of the next, and we shall be content to play in the shallow end of the pool from here on out:

$$\begin{array}{c}\text{convergent rewriting system}\\ \subsetneq\\ \text{decidable word problem}\\ \subsetneq\\ \text{all finitely-presented monoids}\end{array}$$

More conditions satisfied by monoids presented by convergent rewriting systems were explored in [163, 164, 165]. For a survey of results about convergent rewriting systems, see [166]. Many open questions remain. Notice how many of the example monoids we saw only had a single rewrite rule. Even with such "one-relation" monoids, the situation is far from trivial. For example, it is not known if every one-relation monoid can be presented by a convergent rewriting system, or more generally, if the word problem for such monoids is decidable or not. However, one-relation monoids are known to have finite derivation type [167]. For a survey of results about one-relation monoids and a good overview of the word problem in general, see [168]. Apart from string rewriting, another approach to solving the word problem is to use finite state automata to describe the structure of a monoid or group [169]. Other interesting undecidable problems in abstract algebra appear in [170].

Type substitution. In fact, we could have formalized the type substitution algebra as a term rewriting system. This would be a rewriting system over tree-structured terms, because the types, substitution maps and conformances in this algebra all recursively nest within one another.

The "evaluation rules" for the $\otimes$ operator would instead define a reduction relation, and the associativity of $\otimes$ would be equivalent to proving that the reduction relation is confluent. (The intrepid reader can quickly review Appendix C to see why this might be the case.) This reduction relation is not terminating, though, because as Section 12.4 demonstrates, it can encode arbitrary computation.

When viewed as a term rewriting system, type substitution is a lot like the lambda calculus—confluent but non-terminating. While this is an interesting observation, we will not pursue this direction further.

* * *

This chapter does not have a **Source Code Reference** section, because we only talked about math.

17. Symbols, Terms, and Rules

OUR MOTIVATION for translating a monoid presentation into a generic signature was to show that there are theoretical limits to what we can accept. In comparison, the translation of a generic signature into a monoid presentation is an eminently practical matter. We ended Chapter 15 by saying that a requirement machine consists of rewrite rules. We now make this precise. As we did with the derived requirements formalism in Section 5.2, we gradually reveal the full encoding. We begin with the core of the Swift generics model: unbound type parameters, conformance requirements, and same-type requirements between type parameters. We prove a correctness result, and then extend the encoding to cover bound type parameters and the other requirement kinds.

Core model. Let G be a generic signature. We recall from Section 15.1 that if `P` is some protocol, then $G \prec \texttt{P}$ means that G depends on `P`. We now introduce a certain monoid presentation $\langle A \,|\, R \rangle$, called the *requirement machine* for G.

We take our "chewy" recursive inputs—type parameters, conformance requirements, and same-type requirements—and cook them until the "connective tissue" breaks down. This yields a finite alphabet A where each symbol is one of the three kinds below:

`τ_d_i`	Generic parameter symbol	for some d, $i \in \mathbb{N}$
`A`	Name symbol	for some identifier `A`
[P]	Protocol symbol	for some `P` such that $G \prec \texttt{P}$

To get R, we define a "term" function to translate a type parameter of G into a term, and a "rule" function to translate each explicit requirement of G into a rewrite rule:

$$\begin{aligned} \mathsf{term}(\texttt{τ_d_i}) &:= \texttt{τ_d_i} \\ \mathsf{term}(\texttt{U.A}) &:= \mathsf{term}(\texttt{U}) \cdot \texttt{A} \end{aligned} \qquad \begin{aligned} \mathsf{rule}\,[\texttt{T: P}] &:= \mathsf{term}(\texttt{T}) \cdot [\texttt{P}] \sim \mathsf{term}(\texttt{T}) \\ \mathsf{rule}\,[\texttt{T == U}] &:= \mathsf{term}(\texttt{T}) \sim \mathsf{term}(\texttt{U}) \end{aligned}$$

In addition, for each protocol `P` such that $G \prec \texttt{P}$, we define a "$\mathsf{term}_{\texttt{P}}$" function to translate a type parameter of $G_{\texttt{P}}$ into a term starting with [P], and a "$\mathsf{rule}_{\texttt{P}}$" function to translate each associated requirement of `P` into a rewrite rule, in a manner analogous to "rule":

$$\begin{aligned} \mathsf{term}_{\texttt{P}}(\texttt{Self}) &:= [\texttt{P}] \\ \mathsf{term}_{\texttt{P}}(\texttt{V.A}) &:= \mathsf{term}_{\texttt{P}}(\texttt{V}) \cdot \texttt{A} \end{aligned} \qquad \begin{aligned} \mathsf{rule}_{\texttt{P}}\,[\texttt{U: Q}]_{\texttt{P}} &:= \mathsf{term}_{\texttt{P}}(\texttt{U}) \cdot [\texttt{Q}] \sim \mathsf{term}_{\texttt{P}}(\texttt{U}) \\ \mathsf{rule}_{\texttt{P}}\,[\texttt{U == V}]_{\texttt{P}} &:= \mathsf{term}_{\texttt{P}}(\texttt{U}) \sim \mathsf{term}_{\texttt{P}}(\texttt{V}) \end{aligned}$$

This completely describes a very remarkable object, where the *derived requirements* of G are *rewrite paths* in the requirement machine for G.

Example 17.1. Let G be the generic signature below:

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
                    τ_0_0.Element: Equatable,
                    τ_0_0.Element == τ_0_1.Element>
```

We met this signature in Example 5.1 and then studied it in Section 5.2 and Section 5.3. We will now construct its requirement machine.

Our generic signature depends on `Sequence`, `IteratorProtocol`, and `Equatable`. The first two declare an associated type named `Element`, and `Sequence` also declares an associated type named "`Iterator`". Our alphabet has seven symbols:

$$\{\underbrace{\texttt{τ_0_0}, \texttt{τ_0_1}}_{\text{generic param}}, \underbrace{\texttt{Element}, \texttt{Iterator}}_{\text{name}}, \underbrace{[\texttt{Sequence}], [\texttt{IteratorProtocol}], [\texttt{Equatable}]}_{\text{protocol}}\}$$

Here are the requirements of G and `Sequence` that will define our rewrite rules:

$[\texttt{τ_0_0: Sequence}]$

$[\texttt{τ_0_1: Sequence}]$

$[\texttt{τ_0_0.Element: Equatable}]$

$[\texttt{τ_0_0.Element == τ_0_1.Element}]$

$[\texttt{Self.Iterator: IteratorProtocol}]_{\texttt{Sequence}}$

$[\texttt{Self.Element == Self.Iterator.Element}]_{\texttt{Sequence}}$

We build our set of rewrite rules R by applying "rule" or "$\mathsf{rule}_\texttt{P}$" to each requirement:

$$\texttt{τ_0_0} \cdot [\texttt{Sequence}] \sim \texttt{τ_0_0} \tag{1}$$

$$\texttt{τ_0_1} \cdot [\texttt{Sequence}] \sim \texttt{τ_0_1} \tag{2}$$

$$\texttt{τ_0_0} \cdot \texttt{Element} \cdot [\texttt{Equatable}] \sim \texttt{τ_0_0} \cdot \texttt{Element} \tag{3}$$

$$\texttt{τ_0_0} \cdot \texttt{Element} \sim \texttt{τ_0_1} \cdot \texttt{Element} \tag{4}$$

$$[\texttt{Sequence}] \cdot \texttt{Iterator} \cdot [\texttt{IteratorProtocol}] \sim [\texttt{Sequence}] \cdot \texttt{Iterator} \tag{5}$$

$$[\texttt{Sequence}] \cdot \texttt{Element} \sim [\texttt{Sequence}] \cdot \texttt{Iterator} \cdot \texttt{Element} \tag{6}$$

Note that "term" is just a flattening of the linked list structure of a type parameter into an array; the resulting term is a generic parameter symbol followed by zero or more name symbols. To distinguish the two representations, we separate the components of a type parameter with "`.`" as before, while terms join their symbols with the requirement machine monoid operation "$\cdot$". The protocol variants "$\mathsf{term}_\texttt{P}$" and "$\mathsf{rule}_\texttt{P}$" are needed to encode the original protocol of each associated requirement. This is why "$\mathsf{term}_\texttt{P}$" replaces `Self` with the protocol symbol $[\texttt{P}]$ instead of the generic parameter symbol $\texttt{τ_0_0}$.

One more remark. Name symbols and protocol symbols exist in different namespaces, so even if we renamed `IteratorProtocol` to `Iterator`, the name symbol `Iterator` would remain distinct from the protocol symbol [`Iterator`]. On the other hand, two associated types with the same name in different protocols always define the *same* name symbol.

Now, let's continue with our example and think about the equivalence relation these rewrite rules generate. Notice how there is a certain symmetry behind the appearance of protocol symbols. The conformance requirements (1), (2), (3), and (5) rewrite a term that *ends* in a protocol symbol. The associated requirements (5) and (6) rewrite a term that *begins* with a protocol symbol. (The associated conformance requirement (5) is both; its left-hand side begins and ends with a protocol symbol.)

We know that $G \vdash$ [`τ_0_0.Iterator.Element == τ_0_1.Iterator.Element`] from Example 5.9, even though this requirement is not explicitly stated. Applied to this requirement, "rule" outputs this ordered pair of terms:

$$(\texttt{τ_0_0} \cdot \texttt{Iterator} \cdot \texttt{Element}, \texttt{τ_0_1} \cdot \texttt{Iterator} \cdot \texttt{Element})$$

Again, this is not an explicit requirement, so this is not one of the rules in R. However, there is a rewrite path from the first term to the second; they're equivalent via $\sim_R$:

$$\begin{aligned}
&(\texttt{τ_0_0} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{Sequence}]) \triangleright \texttt{Iterator} \cdot \texttt{Element} \\
&\circ \texttt{τ_0_0} \triangleleft ([\texttt{Sequence}] \cdot \texttt{Iterator} \cdot \texttt{Element} \Rightarrow [\texttt{Sequence}] \cdot \texttt{Element}) \\
&\circ (\texttt{τ_0_0} \cdot [\texttt{Sequence}] \Rightarrow \texttt{τ_0_0}) \triangleright \texttt{Element} \\
&\circ (\texttt{τ_0_0} \cdot \texttt{Element} \Rightarrow \texttt{τ_0_1} \cdot \texttt{Element}) \\
&\circ (\texttt{τ_0_1} \Rightarrow \texttt{τ_0_1} \cdot [\texttt{Sequence}]) \triangleright \texttt{Element} \\
&\circ \texttt{τ_0_1} \triangleleft ([\texttt{Sequence}] \cdot \texttt{Element} \Rightarrow [\texttt{Sequence}] \cdot \texttt{Iterator} \cdot \texttt{Element}) \\
&\circ (\texttt{τ_0_1} \cdot [\texttt{Sequence}] \Rightarrow \texttt{τ_0_1}) \triangleright \texttt{Iterator} \cdot \texttt{Element}
\end{aligned}$$

We can also understand this by writing down the intermediate terms between every consecutive pair of rewrite steps. We use conformance requirements to insert and delete protocol symbols before and after invoking our associated same-type requirement:

$$\begin{aligned}
&\texttt{τ_0_0} \cdot \texttt{Iterator} \cdot \texttt{Element} \\
&\sim \texttt{τ_0_0} \cdot [\texttt{Sequence}] \cdot \texttt{Iterator} \cdot \texttt{Element} \\
&\sim \texttt{τ_0_0} \cdot [\texttt{Sequence}] \cdot \texttt{Element} \\
&\sim \texttt{τ_0_0} \cdot \texttt{Element} \\
&\sim \texttt{τ_0_1} \cdot \texttt{Element} \\
&\sim \texttt{τ_0_1} \cdot [\texttt{Sequence}] \cdot \texttt{Element} \\
&\sim \texttt{τ_0_1} \cdot [\texttt{Sequence}] \cdot \texttt{Iterator} \cdot \texttt{Element} \\
&\sim \texttt{τ_0_1} \cdot \texttt{Iterator} \cdot \texttt{Element}
\end{aligned}$$

Now let's try with $G \vdash [\texttt{τ_0_0.Iterator: IteratorProtocol}]$. We apply "rule" to get an ordered pair of terms:

$$(\texttt{τ_0_0} \cdot \texttt{Iterator} \cdot [\texttt{IteratorProtocol}], \texttt{τ_0_0} \cdot \texttt{Iterator})$$

To prove this equivalence, we make use of the conformance requirement (1), and then the associated conformance requirement (5):

$$\begin{aligned}&(\texttt{τ_0_0} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{Sequence}]) \triangleright \texttt{Iterator} \cdot [\texttt{IteratorProtocol}] \\ &\circ \texttt{τ_0_0} \triangleleft ([\texttt{Sequence}] \cdot \texttt{Iterator} \cdot [\texttt{IteratorProtocol}] \Rightarrow [\texttt{Sequence}] \cdot \texttt{Iterator}) \\ &\circ (\texttt{τ_0_0} \cdot [\texttt{Sequence}] \Rightarrow \texttt{τ_0_0}) \triangleright \texttt{Iterator}\end{aligned}$$

Once again, we can show the intermediate terms in our path:

$$\begin{aligned}&\texttt{τ_0_0} \cdot \texttt{Iterator} \cdot [\texttt{IteratorProtocol}] \\ &\sim \texttt{τ_0_0} \cdot [\texttt{Sequence}] \cdot \texttt{Iterator} \cdot [\texttt{IteratorProtocol}] \\ &\sim \texttt{τ_0_0} \cdot [\texttt{Sequence}] \cdot \texttt{Iterator} \\ &\sim \texttt{τ_0_0} \cdot \texttt{Iterator}\end{aligned}$$

17.1. Correctness

We saw some examples of encoding derived requirements as word problems. We now make this precise. We prove two theorems, to establish that we can translate derivations into rewrite paths, and vice versa. We use the algebra of rewriting from Section 16.2.

We begin with a preliminary result. In Section 5.2, we defined the AssocConf and AssocSame inference rules via formal substitution: if $\texttt{T}$ is some type parameter of G known to conform to some protocol $\texttt{P}$, and $\texttt{Self.U}$ is some type parameter of $G_{\texttt{P}}$, then $\texttt{T.U}$ denotes the replacement of $\texttt{Self}$ with $\texttt{T}$ in $\texttt{Self.U}$. We can relate this to the requirement machine monoid operation.

Lemma 17.2. Let G be a generic signature, and let $\langle A \mid R \rangle$ be the requirement machine for G. Further, suppose that:

1. $\texttt{T}$ is some type parameter of G,
2. $\texttt{P}$ is some protocol, and $\mathsf{term}(\texttt{T}) \cdot [\texttt{P}] \sim \mathsf{term}(\texttt{T})$,
3. $\texttt{Self.U}$ is some type parameter of $G_{\texttt{P}}$.

Then $\mathsf{term}(\texttt{T}) \cdot \mathsf{term}_{\texttt{P}}(\texttt{Self.U}) \sim \mathsf{term}(\texttt{T.U})$.

Proof. Note that $\mathsf{term}_{\texttt{P}}(\texttt{Self.U}) = [\texttt{P}] \cdot u$ for some $u \in A^*$, by definition of "$\mathsf{term}_{\texttt{P}}$". We also let $t := \mathsf{term}(\texttt{T})$, so that $\mathsf{term}(\texttt{T.U}) = t \cdot u$. Now, if p is a rewrite path from $t \cdot [\texttt{P}]$ to t, then $p \triangleright u$ is a rewrite path from $t \cdot [\texttt{P}] \cdot u$ to $t \cdot u$. □

Example 17.3. In Example 17.1, we can see the second rewrite path in a new way:

$$\begin{aligned}
&\mathsf{term}(\texttt{τ_0_0.Iterator}) \cdot [\texttt{IteratorProtocol}] \\
&\sim \mathsf{term}(\texttt{τ_0_0}) \cdot \mathsf{term}_{\texttt{Sequence}}(\texttt{Self.Iterator}) \cdot [\texttt{IteratorProtocol}] \\
&\sim \mathsf{term}(\texttt{τ_0_0}) \cdot \mathsf{term}_{\texttt{Sequence}}(\texttt{Self.Iterator}) \\
&\sim \mathsf{term}(\texttt{τ_0_0.Iterator})
\end{aligned}$$

This idea of "cleaving" a term in two will be key in the inductive step of the below proof, when we consider the AssocConf and AssocSame inference rules.

Theorem 17.4. Let G be a generic signature, and let $\langle A \mid R \rangle$ be the requirement machine for G.

1. If $G \vdash [\texttt{T: P}]$ for some type parameter $\texttt{T}$ and protocol $\texttt{P}$, then $\mathsf{term}(\texttt{T}) \cdot [\texttt{P}] \sim \mathsf{term}(\texttt{T})$.
2. If $G \vdash [\texttt{T == U}]$ for some pair of type parameters $\texttt{T}$ and $\texttt{U}$, then $\mathsf{term}(\texttt{T}) \sim \mathsf{term}(\texttt{U})$.

Proof. We use structural induction on derived requirements (Section 11.3) to define a "path" mapping that assigns to each derived requirement $G \vdash D$ a rewrite path p such that $\mathsf{rule}(D) = (\mathrm{src}(p), \mathrm{dst}(p))$. The conclusion then follows.

Base case. To handle the elementary statements, and those inference rules that do not have any requirements among their assumptions, we construct a rewrite path directly.

A Conf step invokes an explicit conformance requirement of G:

$$[\texttt{T: P}] \tag{Conf}$$

Note that $\mathsf{rule}\,[\texttt{T: P}] = (\mathsf{term}(\texttt{T}) \cdot [\texttt{P}], \mathsf{term}(\texttt{T})) \in R$. The path is a single rewrite step:

$$\mathsf{path}\,[\texttt{T: P}] := (\mathsf{term}(\texttt{T}) \cdot [\texttt{P}] \Rightarrow \mathsf{term}(\texttt{T}))$$

A Same step invokes an explicit same-type requirement of G:

$$[\texttt{T == U}] \tag{Same}$$

Note that $\mathsf{rule}\,[\texttt{T == U}] = (\mathsf{term}(\texttt{T}), \mathsf{term}(\texttt{U})) \in R$. The path is a single rewrite step:

$$\mathsf{path}\,[\texttt{T == U}] := (\mathsf{term}(\texttt{T}) \Rightarrow \mathsf{term}(\texttt{U}))$$

A Reflex step derives a trivial same-type requirement from a valid type parameter:

$$[\texttt{T == T}] \tag{Reflex T}$$

We don't need the fact that $G \vdash \texttt{T}$ at all, because in a finitely-presented monoid, every term is already equivalent to itself via the empty rewrite path, so we let $t := \mathsf{term}(\texttt{T})$, and we set:

$$\mathsf{path}\,[\texttt{T == T}] := 1_t$$

Inductive step. In each case, the "path" of the conclusion is defined from the "path" of the step's assumptions. For an AssocConf step, the induction hypothesis gives us $p_1 := \mathsf{path}\,[\texttt{T: P}]$, so $\mathrm{src}(p_1) = \mathsf{term}(\texttt{T}) \cdot [\texttt{P}]$ and $\mathrm{dst}(p_1) = \mathsf{term}(\texttt{T})$:

$$[\texttt{T.U: Q}] \qquad (\textsc{AssocConf}\ [\texttt{Self.U: Q}]_{\texttt{P}}\ [\texttt{T: P}])$$

Note that $\mathsf{rule}_{\texttt{P}}\,[\texttt{Self.U: Q}]_{\texttt{P}} \in R$ is of the form $([\texttt{P}] \cdot u \cdot [\texttt{Q}], [\texttt{P}] \cdot u)$ for some $u \in A^*$. Using Lemma 17.2, we insert a $[\texttt{P}]$ in the right spot, apply our associated requirement, and then delete the $[\texttt{P}]$. We let $t := \mathsf{term}(\texttt{T})$, let $s := ([\texttt{P}] \cdot u \cdot [\texttt{Q}] \Rightarrow [\texttt{P}] \cdot u)$, and we set:

$$\mathsf{path}\,[\texttt{T.U: Q}] := (p_1^{-1} \triangleright u \triangleright [\texttt{Q}]) \circ (t \triangleleft s) \circ (p_1 \triangleright u)$$

For an AssocSame step, the induction hypothesis gives us $p_1 := \mathsf{path}\,[\texttt{T: P}]$:

$$[\texttt{T.U == T.V}] \qquad (\textsc{AssocSame}\ [\texttt{Self.U == Self.V}]_{\texttt{P}}\ [\texttt{T: P}])$$

Note that $\mathsf{rule}_{\texttt{P}}\,[\texttt{Self.U == Self.V}]_{\texttt{P}} \in R$ is of the form $([\texttt{P}] \cdot u, [\texttt{P}] \cdot v)$ for some $u, v \in A^*$. Again, we use Lemma 17.2. We let $t := \mathsf{term}(\texttt{T})$, let $s := ([\texttt{P}] \cdot u \Rightarrow [\texttt{P}] \cdot v)$, and we set:

$$\mathsf{path}\,[\texttt{T.U == T.V}] := (p_1^{-1} \triangleright u) \circ (t \triangleleft s) \circ (p_1 \triangleright v)$$

For a Sym or Trans step, we use the rewrite path inverse and composition operations:

$$[\texttt{U == T}] \qquad (\textsc{Sym}\ [\texttt{T == U}])$$
$$[\texttt{T == V}] \qquad (\textsc{Trans}\ [\texttt{T == U}]\ [\texttt{U == V}])$$

$$\mathsf{path}\,[\texttt{U == T}] := \mathsf{path}\,[\texttt{T == U}]^{-1}$$
$$\mathsf{path}\,[\texttt{T == V}] := \mathsf{path}\,[\texttt{T == U}] \circ \mathsf{path}\,[\texttt{U == V}]$$

For a SameConf step, the induction hypothesis gives us a pair of rewrite paths $p_1 := \mathsf{path}\,[\texttt{U: P}]$, and $p_2 := \mathsf{path}\,[\texttt{T == U}]$:

$$[\texttt{T: P}] \qquad (\textsc{SameConf}\ [\texttt{U: P}]\ [\texttt{T == U}])$$

We rewrite $\mathsf{term}(\texttt{T}) \cdot [\texttt{P}]$ into $\mathsf{term}(\texttt{U}) \cdot [\texttt{P}]$ using p_2, delete $[\texttt{P}]$ from the end of the term using p_1, and rewrite $\mathsf{term}(\texttt{U})$ back to $\mathsf{term}(\texttt{T})$ using p_2 again:

$$\mathsf{path}\,[\texttt{T: P}] := (p_2 \triangleright [\texttt{P}]) \circ p_1 \circ p_2^{-1}$$

For a SameName step, the induction hypothesis gives us $p_1 := \mathsf{path}\,[\texttt{U: P}]$, and $p_2 := [\texttt{T == U}]$:

$$[\texttt{T.A == U.A}] \qquad (\textsc{SameName}\ [\texttt{U: P}]\ [\texttt{T == U}])$$

We only need p_2 to construct our new rewrite path, by whiskering:

$$\mathsf{path}\,[\texttt{T.A == U.A}] := p_2 \triangleright \texttt{A}$$

By induction, this completely defines "path", so we apply it to our original requirement and get the desired result. Figure 17.1 shows each rewrite path we constructed. □

Figure 17.1.: Rewrite paths defined by Theorem 17.4, as paths in the rewrite graph

	Conf:
$[\mathtt{T}\colon\ \mathtt{P}]$	$\mathsf{term}(\mathtt{T}) \cdot [\mathtt{P}] \Longrightarrow \mathsf{term}(\mathtt{T})$
	Same:
$[\mathtt{T}\ ==\ \mathtt{U}]$	$\mathsf{term}(\mathtt{T}) \Longrightarrow \mathsf{term}(\mathtt{U})$
	Reflex:
$[\mathtt{T}\ ==\ \mathtt{T}]$	$\mathsf{term}(\mathtt{T})$
	AssocConf:
$[\mathtt{T.U}\colon\ \mathtt{Q}]$	$\begin{array}{ccc} \mathsf{term}(\mathtt{T.U}) \cdot [\mathtt{Q}] & \Longrightarrow \cdots \Longrightarrow & \mathsf{term}(\mathtt{T}) \cdot \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \cdot [\mathtt{Q}] \\ & & \Downarrow \\ \mathsf{term}(\mathtt{T.U}) & \Longleftarrow \cdots = & \mathsf{term}(\mathtt{T}) \cdot \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \end{array}$
	AssocSame:
$[\mathtt{T.U}\ ==\ \mathtt{T.V}]$	$\begin{array}{ccc} \mathsf{term}(\mathtt{T.U}) & \Longrightarrow \cdots \Longrightarrow & \mathsf{term}(\mathtt{T}) \cdot \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \\ & & \Downarrow \\ \mathsf{term}(\mathtt{T.V}) & \Longleftarrow \cdots = & \mathsf{term}(\mathtt{T}) \cdot \mathsf{term}_{\mathtt{P}}(\mathtt{Self.V}) \end{array}$
	Sym:
$[\mathtt{U}\ ==\ \mathtt{T}]$	$\mathsf{term}(\mathtt{T}) \Longleftarrow \cdots = \mathsf{term}(\mathtt{U})$
	Trans:
$[\mathtt{T}\ ==\ \mathtt{V}]$	$\mathsf{term}(\mathtt{T}) = \cdots \Longrightarrow \mathsf{term}(\mathtt{U}) = \cdots \Longrightarrow \mathsf{term}(\mathtt{V})$
	SameConf:
$[\mathtt{T}\colon\ \mathtt{P}]$	$\begin{array}{ccc} \mathsf{term}(\mathtt{T}) \cdot [\mathtt{P}] & = \cdots \Longrightarrow & \mathsf{term}(\mathtt{U}) \cdot [\mathtt{P}] \\ & & \Vert \\ & & \cdots \\ & & \Downarrow \\ \mathsf{term}(\mathtt{T}) & \Longleftarrow \cdots = & \mathsf{term}(\mathtt{U}) \end{array}$
	SameName:
$[\mathtt{T.A}\ ==\ \mathtt{U.A}]$	$\mathsf{term}(\mathtt{T}) \cdot \mathtt{A} = \cdots \Longrightarrow \mathsf{term}(\mathtt{U}) \cdot \mathtt{A}$

We now go in the other direction, and describe how rewrite paths in the requirement machine define derived requirements in our generic signature. Consider how we might reverse our "path" mapping. The main difficulty to overcome is the fact that we did not actually use every assumption in each step:

1. The "path" of a Reflex step [T == T] disregards the proof of the validity of the type parameter T, because in the monoid, we can immediately construct an empty rewrite path for *any* term.

   However, this means that if we start from the empty rewrite path at a "nonsensical" term, such as $\mathtt{A} \cdot \mathtt{τ_0_1} \cdot [\mathtt{Sequence}] \cdot \mathtt{τ_0_0}$, we cannot "go backwards" and hope to prove the validity of any type parameter in G.

2. The "path" of a SameName step [T.A == U.A] is defined from path [T == U] by whiskering, ignoring path [T: P], because we can whisker *any* rewrite path.

   That is, if we have an equivalent pair of terms $\mathsf{term}(\mathtt{T}) \sim \mathsf{term}(\mathtt{U})$, and A is any name symbol, it is *always* the case that $\mathsf{term}(\mathtt{T}) \cdot \mathtt{A} \sim \mathsf{term}(\mathtt{U}) \cdot \mathtt{A}$ in the monoid.

   However, we may not be able to derive $G \vdash [\mathtt{T.A == U.A}]$, because T might not conform any protocol with an associated type named "A". By the same reasoning, we might not even be able to derive $G \vdash [\mathtt{T == U}]$.

Thus, not every equivalence class of $\sim_R$ corresponds to a valid type parameter of our generic signature G. Before we can see how to translate a specific rewrite path into a derivation, we need to understand which rewrite paths are meaningful to us. We begin by describing the equivalence class of $\mathsf{term}(\mathtt{T})$ for a type parameter T.

Definition 17.5. Let $t \in A^*$ be a term.

- If the first symbol of t is a generic parameter symbol and all subsequent symbols are name symbols, then we say that t is a *standard* term.
- If the first symbol of t is a generic parameter symbol and all subsequent symbols are either name or protocol symbols, we say that t is an *admissible* term.

Lemma 17.6. Some consequences of the above:

1. Every standard term is $\mathsf{term}(\mathtt{T})$ for some unbound type parameter T.
2. Every standard term is an admissible term.
3. If (u, v) is a rewrite rule output by "rule", both u and v are admissible terms.
4. If $t \in A^*$ is an admissible term and $u \in A^*$ is a combination of name and protocol symbols (possibly empty), then tu is an admissible term.

The next lemma states that admissibility is preserved by rewriting. In particular, this means that rewriting a standard term always outputs an admissible term.

Lemma 17.7. Let G be a generic signature, and let $\langle A \mid R \rangle$ be the requirement machine for G. Let $t \in A^*$ be an admissible term. If $t \sim z$ for some other term $z \in A^*$, then z is an admissible term.

Proof. Let p be a rewrite path from t to z. We argue by induction on the length of p.

Base case. If we have an empty rewrite path, then $t = z$, so z is an admissible term.

Inductive step. Otherwise, $p = p' \circ s$ for some rewrite path p' and rewrite step s. We write $s := x(u \Rightarrow v)y$, where x, $y \in A^*$, and one of (u, v) or $(v, u) \in R$.

By the induction hypothesis, $\mathrm{dst}(p') = \mathrm{src}(s) = xuy$ is an admissible term. Because the prefix xu is non-empty, the right whisker y contains only name and protocol symbols.

If the rule applied by s is the "rule" for an explicit requirement, then the left whisker x is empty, because u starts with a generic parameter symbol. Furthermore, v is an admissible term. We see that $\mathrm{dst}(s) = vy$ is an admissible term.

If this rule applied by s is the "$\mathsf{rule_P}$" for an associated requirement, then the left whisker x is non-empty, and furthermore an admissible term, and u and v must contain only name and protocol symbols. We see that $\mathrm{dst}(s) = xvy$ is an admissible term. □

Just like a standard term describes an unbound type parameter, an admissible term more generally describes a type parameter and *a list of conformance requirements*.

Definition 17.8. The "type" function maps each admissible term to an unbound type parameter, by ignoring protocol symbols, and translating generic parameter symbols and name symbols into generic parameter types and dependent member types:

$$\begin{aligned}
\mathsf{type}(\texttt{τ_d_i}) &:= \texttt{τ_d_i} \\
\mathsf{type}(\texttt{U} \cdot \texttt{A}) &:= \mathsf{type}(\texttt{U}).\texttt{A} \\
\mathsf{type}(\texttt{U} \cdot [\texttt{P}]) &:= \mathsf{type}(\texttt{U})
\end{aligned}$$

Definition 17.9. Let $z \in A^*$ be an admissible term. The *conformance chart* of z is a *multiset* (that is, we allow duplicates) of conformance requirements:

1. Each requirement corresponds to an occurrence of a protocol symbol in z.
2. The subject type of each requirement is the prefix before the protocol symbol.
3. Duplicate elements represent identical consecutive protocol symbols.

Definition 17.10. The "chart" function maps each admissible term to its conformance chart, by ignoring generic parameter and name symbols, and converting protocol symbols into requirements:

$$\begin{aligned}
\mathsf{chart}(\texttt{τ_d_i}) &:= \{\} \\
\mathsf{chart}(\texttt{U} \cdot \texttt{A}) &:= \mathsf{chart}(\texttt{U}) \\
\mathsf{chart}(\texttt{U} \cdot [\texttt{P}]) &:= \mathsf{chart}(\texttt{U}) \cup \{[\mathsf{type}(\texttt{U})\texttt{: P}]\}
\end{aligned}$$

Example 17.11. If $z = \mathsf{term}(\mathtt{T})$ is a standard term:

$$\mathsf{type}(z) = \mathtt{T}$$
$$\mathsf{chart}(z) = \{\}$$

If $z = \mathsf{term}(\mathtt{T}) \cdot [\mathtt{P}]$ for some unbound type parameter `T` and protocol `P`:

$$\mathsf{type}(z) = \mathtt{T}$$
$$\mathsf{chart}(z) = \{[\mathtt{T}\texttt{: }\mathtt{P}]\}$$

If $z = \mathtt{τ_0_1} \cdot [\mathtt{Sequence}] \cdot [\mathtt{Sequence}] \cdot \mathtt{Iterator} \cdot [\mathtt{IteratorProtocol}]$,

$$\mathsf{type}(z) = \mathtt{τ_0_1.Iterator},$$
$$\mathsf{chart}(z) = \{[\mathtt{τ_0_1}\texttt{: }\mathtt{Sequence}],$$
$$[\mathtt{τ_0_1}\texttt{: }\mathtt{Sequence}],$$
$$[\mathtt{τ_0_1.Iterator}\texttt{: }\mathtt{IteratorProtocol}]\}.$$

We can say a lot more if we add the assumptions that `T` is a *valid* type parameter, and our generic signature G is well-formed. (We lose nothing by doing this, because otherwise we diagnose an error and reject the program; see Section 11.3.) Having done so, we see that rewriting a standard term into an admissible term is tantamount to simultaneously deriving a same-type requirement *together with every conformance requirement in the destination term's chart*.

Theorem 17.12. Let G be a well-formed generic signature, let $\langle A \mid R \rangle$ be the requirement machine for G, and let t be a standard term such that $\mathsf{type}(t)$ is a valid type parameter of G. Suppose we are given a term $z \in A^*$ such that $t \sim z$. Then z is an admissible term, and furthermore:

1. $G \vdash [\mathsf{type}(t)\ \texttt{==}\ \mathsf{type}(z)]$.
2. $G \vdash [\mathsf{type}(u)\texttt{: }\mathtt{P}]$ for each $[\mathsf{type}(u)\texttt{: }\mathtt{P}] \in \mathsf{chart}(z)$.

Proof. Lemma 17.7 already shows that z is an admissible term. To establish the principal conclusion, we reason as follows:

1. Because we started with a standard term, we know that every protocol symbol appearing in z must have been introduced by some rewrite step in our path.

   When a rewrite step introduces a protocol symbol, we must be able to derive this conformance requirement, and record it alongside our chart.

2. The other rewrite steps apply same-type requirements, and we must be able to build a derived same-type requirement corresponding to this transformation.

   We must also understand the effect that same-type requirement rules have on the conformance chart, so that we can update our collected set of derived conformance requirements when needed.

Let p be a rewrite path from t to z. We argue by induction on the length of p.

Base case. If p is empty, then $t = z$, so $\mathsf{type}(t) = \mathsf{type}(z)$, and $\mathsf{chart}(z) = \{\}$. We use the assumption that $\mathsf{type}(t)$ is a valid type parameter to derive the same-type requirement $[\mathsf{type}(t)\ \texttt{==}\ \mathsf{type}(t)]$, by applying the Reflex inference rule to a derivation of $G \vdash \mathsf{type}(t)$. We do not need to derive any conformance requirements.

Inductive step. Otherwise, $p = p' \circ s$ for some rewrite path p' and rewrite step s. Let $z' := \mathrm{dst}(p') = \mathrm{src}(s)$, and note that $z = \mathrm{dst}(s)$. By the induction hypothesis:

1. $G \vdash [\mathsf{type}(t)\ \texttt{==}\ \mathsf{type}(z')]$.
2. $G \vdash [\mathsf{type}(u')\texttt{:}\ \texttt{P}]$ for each $[\mathsf{type}(u')\texttt{:}\ \texttt{P}] \in \mathsf{chart}(z')$.

The rewrite step s applies a conformance or same-type requirement; this requirement is explicit ("rule") or associated ("$\mathsf{rule}_{\texttt{P}}$"); and the rewrite step may be positive or negative. We consider each case to derive the desired conclusion about z:

	Kind:	**Domain:**	**Sign:**	**General Form:**
1.	conformance	explicit	$+$	$(u \cdot [\texttt{P}] \Rightarrow u) \triangleright y$
2.		explicit	$-$	$(u \Rightarrow u \cdot [\texttt{P}]) \triangleright y$
3.		associated	$+$	$x \triangleleft ([\texttt{P}] \cdot u \cdot [\texttt{Q}] \Rightarrow [\texttt{P}] \cdot u) \triangleright y$
4.		associated	$-$	$x \triangleleft ([\texttt{P}] \cdot u \Rightarrow [\texttt{P}] \cdot u \cdot [\texttt{Q}]) \triangleright y$
5.	same-type	explicit	$+$	$(u \Rightarrow v) \triangleright y$
6.		explicit	$-$	$(v \Rightarrow u) \triangleright y$
7.		associated	$+$	$x \triangleleft ([\texttt{P}] \cdot u \Rightarrow [\texttt{P}] \cdot v) \triangleright y$
8.		associated	$-$	$x \triangleleft ([\texttt{P}] \cdot v \Rightarrow [\texttt{P}] \cdot u) \triangleright y$

(★ Case 1.) A positive rewrite step for an explicit conformance requirement has the following general form, where the left whisker is empty, $(u \cdot [\texttt{P}], u) \in R$ for some standard term u and protocol symbol $[\texttt{P}]$, and the right whisker y is a combination of name and protocol symbols:

$$(u \cdot [\texttt{P}] \Rightarrow u) \triangleright y$$

The rewrite step deletes an occurrence of the protocol symbol $[\texttt{P}]$:

$$z' = \mathrm{src}(s) = u \cdot [\texttt{P}] \cdot y$$
$$z = \mathrm{dst}(s) = u \cdot y$$

$$\mathsf{type}(z) = \mathsf{type}(z')$$
$$\mathsf{chart}(z) = \mathsf{chart}(z') \setminus \{[\mathsf{type}(u)\texttt{:}\ \texttt{P}]\}$$

Therefore our conclusion already holds; we just "discard" our derivation of $[\mathsf{type}(u)\texttt{:}\ \texttt{P}]$.

(★ Case 2.) A negative rewrite step for an explicit conformance requirement has the following general form, where again the left whisker is empty, $(u \cdot [\mathtt{P}], u) \in R$ for some standard term u and protocol symbol $[\mathtt{P}]$, and the right whisker y is as before:

$$(u \Rightarrow u \cdot [\mathtt{P}]) \triangleright y$$

The rewrite step inserts an occurrence of the protocol symbol $[\mathtt{P}]$:

$$z' = \mathrm{src}(s) = u \cdot y$$
$$z = \mathrm{dst}(s) = u \cdot [\mathtt{P}] \cdot y$$

$$\mathsf{type}(z) = \mathsf{type}(z')$$
$$\mathsf{chart}(z) = \mathsf{chart}(z') \cup \{[\mathsf{type}(u)\colon\ \mathtt{P}]\}$$

To establish the conclusion, we must derive $[\mathsf{type}(u)\colon\ \mathtt{P}]$. We do this with a Conf elementary statement, because $\mathsf{rule}\,[\mathsf{type}(u)\colon\ \mathtt{P}] = (u \cdot [\mathtt{P}], u) \in R$:

1. $[\mathsf{type}(u)\colon\ \mathtt{P}]$ (Conf)

(★ Case 3.) A positive rewrite step for an associated conformance requirement has the following general form, where x is an admissible term, $([\mathtt{P}] \cdot u \cdot [\mathtt{Q}], [\mathtt{P}] \cdot u) \in R$ for some u that contains only name symbols, and the right whisker y is as before:

$$x \triangleleft ([\mathtt{P}] \cdot u \cdot [\mathtt{Q}] \Rightarrow [\mathtt{P}] \cdot u) \triangleright y$$

The rewrite step deletes a protocol symbol $[\mathtt{Q}]$. Note that $\mathsf{type}(x \cdot [\mathtt{P}] \cdot u) = \mathsf{type}(x \cdot u)$:

$$z' = \mathrm{src}(s) = x \cdot [\mathtt{P}] \cdot u \cdot [\mathtt{Q}] \cdot y$$
$$z = \mathrm{dst}(s) = x \cdot [\mathtt{P}] \cdot u \cdot y$$

$$\mathsf{type}(z) = \mathsf{type}(z')$$
$$\mathsf{chart}(z) = \mathsf{chart}(z') \setminus \{[\mathsf{type}(x \cdot u)\colon\ \mathtt{Q}]\}$$

As in Case 1, we "discard" our prior derivation of $[\mathsf{type}(x \cdot u)\colon\ \mathtt{Q}]$.

(★ Case 4.) A negative rewrite step for an associated conformance requirement has the following general form, where x is an admissible term, $([\mathtt{P}] \cdot u \cdot [\mathtt{Q}], [\mathtt{P}] \cdot u) \in R$ for some u that contains only name symbols, and the right whisker y is as before:

$$x \triangleleft ([\mathtt{P}] \cdot u \Rightarrow [\mathtt{P}] \cdot u \cdot [\mathtt{Q}]) \triangleright y$$

The rewrite step inserts a protocol symbol $[\mathtt{Q}]$. Again, $\mathsf{type}(x \cdot [\mathtt{P}] \cdot u) = \mathsf{type}(x \cdot u)$:

$$z' = \mathrm{src}(s) = x \cdot [\mathtt{P}] \cdot u \cdot y$$
$$z = \mathrm{dst}(s) = x \cdot [\mathtt{P}] \cdot u \cdot [\mathtt{Q}] \cdot y$$

$$\mathsf{type}(z) = \mathsf{type}(z')$$
$$\mathsf{chart}(z) = \mathsf{chart}(z') \cup \{[\mathsf{type}(x \cdot u)\colon\ \mathtt{Q}]\}$$

We must derive $[\mathsf{type}(x \cdot u)\colon \mathtt{Q}]$. Since $[\mathsf{type}(x)\colon \mathtt{P}] \in \mathsf{chart}(z')$, the induction hypothesis tells us that $G \vdash [\mathsf{type}(x)\colon \mathtt{P}]$, and the AssocConf inference rule then gives us:

1. $[\mathsf{type}(x)\colon \mathtt{P}]$ $(\ldots)$
2. $[\mathsf{type}(x \cdot u)\colon \mathtt{Q}]$ (AssocConf 1)

(★ Case 5.) A positive rewrite step for an explicit same-type requirement has the following general form, where the left whisker is empty, $(u, v) \in R$ for standard terms u and v, and the right whisker y is a combination of name and protocol symbols:

$$(u \Rightarrow v) \triangleright y$$

The rewrite step replaces the prefix u with v:

$$z' = \mathrm{src}(s) = u \cdot y$$
$$z = \mathrm{dst}(s) = v \cdot y$$

In this case, $\mathsf{type}(z)$ and $\mathsf{type}(z')$ are not identical, so we must first derive the same-type requirement $G \vdash [\mathsf{type}(t) \ \texttt{==} \ \mathsf{type}(z)]$. We now write $u \cdot y$ instead of z', so the induction hypothesis gave us this same-type requirement:

1. $[\mathsf{type}(t) \ \texttt{==} \ \mathsf{type}(u \cdot y)]$ $(\ldots)$

We now use the assumption that G is well-formed. Because $\mathsf{type}(u \cdot y)$ appears in the above requirement, the well-formedness of G allows us to derive $G \vdash \mathsf{type}(u \cdot y)$:

2. $\mathsf{type}(u \cdot y)$ $(\ldots)$

Next, we derive $[\mathsf{type}(u) \ \texttt{==} \ \mathsf{type}(v)]$ with a Same elementary statement:

3. $[\mathsf{type}(u) \ \texttt{==} \ \mathsf{type}(v)]$ (Same)

Furthermore, $\mathsf{type}(u)$ is a prefix of $\mathsf{type}(u \cdot y)$. We apply Lemma 12.6 to $G \vdash \mathsf{type}(v \cdot y)$ and $G \vdash [\mathsf{type}(u) \ \texttt{==} \ \mathsf{type}(v)]$ to derive $G \vdash [\mathsf{type}(u \cdot y) \ \texttt{==} \ \mathsf{type}(v \cdot y)]$, or in other words, $[\mathsf{type}(z') \ \texttt{==} \ \mathsf{type}(z)]$:

4. $[\mathsf{type}(z') \ \texttt{==} \ \mathsf{type}(z)]$ $(\ldots)$

Finally, we apply the Trans inference rule to compose the above requirement with the same-type requirement given to us by the induction hypothesis:

5. $[\mathsf{type}(t) \ \texttt{==} \ \mathsf{type}(z)]$ (Trans 1 4)

We now have our same-type requirement, but we must also derive each conformance requirement in $\mathsf{chart}(z)$ from the conformance requirements in $\mathsf{chart}(z')$.

Notice how z' and z have the same number and relative ordering of protocol symbols, and they all occur in the right whisker y, so there is a one-to-one correspondence between the elements of $\mathsf{chart}(z')$ and $\mathsf{chart}(z)$.

At each occurrence of a protocol symbol in y, we split up y into a pair of terms, by writing $y = y_1 \cdot [\mathtt{P}] \cdot y_2$ for some y_1, $y_2 \in A^*$ and some protocol $\mathtt{P}$. Then:

$$[\mathsf{type}(u \cdot y_1)\colon\ \mathtt{P}] \in \mathsf{chart}(z')$$
$$[\mathsf{type}(v \cdot y_1)\colon\ \mathtt{P}] \in \mathsf{chart}(z)$$

For each $[\mathsf{type}(u \cdot y_1)\colon\ \mathtt{P}] \in \mathsf{chart}(z')$, we know that $\mathsf{type}(u \cdot y_1)$ is a prefix of $\mathsf{type}(z')$. Because G is well-formed, Proposition 11.5 says that $G \vdash \mathsf{type}(u \cdot y_1)$. Then, $\mathsf{type}(u)$ is also a prefix of $\mathsf{type}(u \cdot y_1)$, so Lemma 12.6 says that $G \vdash [\mathsf{type}(u \cdot y_1)\ \mathtt{==}\ \mathsf{type}(v \cdot y_1)]$. Finally, we flip this requirement with Sym, and apply SameConf to derive the desired conformance requirement in $\mathsf{chart}(z)$:

1. $[\mathsf{type}(u \cdot y_1)\ \mathtt{==}\ \mathsf{type}(v \cdot y_1)]$ (...)
2. $[\mathsf{type}(u \cdot y_1)\colon\ \mathtt{P}]$ (...)
3. $[\mathsf{type}(v \cdot y_1)\ \mathtt{==}\ \mathsf{type}(u \cdot y_1)]$ (Sym 1)
4. $[\mathsf{type}(u \cdot y_1)\colon\ \mathtt{P}]$ (SameConf 3 1)

(★ Case 6.) A negative rewrite step for an explicit same-type requirement has the following general form, where $(u, v) \in R$ for standard terms u and v, the left whisker is empty, and the right whisker y is a combination of name and protocol symbols:

$$(v \Rightarrow u) \triangleright y$$

We proceed as in Case 5, except that when the Same elementary statement gives us $[\mathsf{type}(u)\ \mathtt{==}\ \mathsf{type}(v)]$, we must apply the Sym inference rule to flip it around to get $[\mathsf{type}(v)\ \mathtt{==}\ \mathsf{type}(u)]$ before we proceed:

1. $[\mathsf{type}(u)\ \mathtt{==}\ \mathsf{type}(v)]$ (Same)
2. $[\mathsf{type}(v)\ \mathtt{==}\ \mathsf{type}(u)]$ (Sym 1)

We can then derive our final same-type requirement, and all conformance requirements listed in the chart, as before.

(★ Case 7.) A positive rewrite step for an associated same-type requirement has the following general form, where the left whisker x is an admissible term, $([\mathtt{P}] \cdot u, [\mathtt{P}] \cdot v) \in R$, u and v only contain name symbols, and the right whisker y is a combination of name and protocol symbols:

$$x \triangleleft ([\mathtt{P}] \cdot u \Rightarrow [\mathtt{P}] \cdot v) \triangleright y$$

The rewrite step replaces $[\texttt{P}] \cdot u$ with $[\texttt{P}] \cdot v$ in the middle of the term:

$$z' = \text{src}(s) = x \cdot [\texttt{P}] \cdot u \cdot y$$
$$z = \text{dst}(s) = x \cdot [\texttt{P}] \cdot v \cdot y$$

Note that $[\mathsf{type}(x)\texttt{: P}] \in \mathsf{chart}(z')$, which we can derive by the induction hypothesis. Also, observe that $\mathsf{type}(x \cdot [\texttt{P}] \cdot u) = \mathsf{type}(x \cdot u)$. We derive $[\mathsf{type}(x \cdot u)\ \texttt{==}\ \mathsf{type}(x \cdot v)]$ from $[\mathsf{type}(x)\texttt{: P}]$ via the AssocSame inference rule:

1. $[\mathsf{type}(x)\texttt{: P}]$ (...)
2. $[\mathsf{type}(x \cdot u)\ \texttt{==}\ \mathsf{type}(x \cdot v)]$ (AssocSame 1)

We then apply Lemma 12.6 to $G \vdash \mathsf{type}(x \cdot u \cdot y)$ and $G \vdash [\mathsf{type}(x \cdot u)\ \texttt{==}\ \mathsf{type}(x \cdot v)]$ to derive $G \vdash [\mathsf{type}(x \cdot u \cdot y)\ \texttt{==}\ \mathsf{type}(x \cdot v \cdot y)]$, or in other words, $[\mathsf{type}(z')\ \texttt{==}\ \mathsf{type}(z)]$:

4. $[\mathsf{type}(z')\ \texttt{==}\ \mathsf{type}(z)]$ (...)

Finally, we apply the Trans inference rule to compose the above requirement with the same-type requirement given to us by the induction hypothesis:

5. $[\mathsf{type}(t)\ \texttt{==}\ \mathsf{type}(z)]$ (Trans 1 4)

We must also update the conformance chart. Both x and y may contain occurrences of protocol symbols, but since $\mathsf{chart}(x) \subseteq \mathsf{chart}(z') \cap \mathsf{chart}(z)$, the requirements in $\mathsf{chart}(x)$ that are common to both remain unchanged. We derive new conformance requirements corresponding to occurrences of protocol symbols in y as in Case 5.

(★ Case 8.) A negative rewrite step for an associated same-type requirement has the following general form, where the left whisker x is an admissible term, $([\texttt{P}] \cdot u, [\texttt{P}] \cdot v) \in R$, u and v contain only name symbols, and the right whisker $y \in A^*$ is a combination of name and protocol symbols:

$$x \triangleleft ([\texttt{P}] \cdot v \Rightarrow [\texttt{P}] \cdot u) \triangleright y$$

We proceed as in Case 7, except we must flip the same-type requirement around via Sym as in Case 6. □

The specific statement we are looking for falls out from the preceding theorem:

Corollary 17.13. Let G be a generic signature, and let $\langle A \,|\, R \rangle$ be the requirement machine for G. Assume G is well-formed.

1. If $\mathsf{term}(\texttt{T}) \cdot [\texttt{P}] \sim \mathsf{term}(\texttt{T})$ for some valid type parameter $\texttt{T}$, then $G \vdash [\texttt{T: P}]$.
2. If $\mathsf{term}(\texttt{T}) \sim \mathsf{term}(\texttt{U})$ for valid type parameters $\texttt{T}$ and $\texttt{U}$, then $G \vdash [\texttt{T == U}]$.

Proof. In both cases, we have an equivalence between a standard term and an admissible term, so we can apply the previous theorem.

(★ Case 1.) Let $t := \mathsf{term}(\mathtt{T})$, and $z := \mathsf{term}(\mathtt{T}) \cdot [\mathtt{P}]$. We have:

$$\mathsf{type}(z) = \mathtt{T}$$
$$\mathsf{chart}(z) = \{[\mathtt{T}\texttt{:}\ \mathtt{P}]\}$$

Translating our rewrite path gives us $G \vdash [\mathtt{T}\ \texttt{==}\ \mathtt{T}]$ which we discard, together with $G \vdash [\mathtt{T}\texttt{:}\ \mathtt{P}]$ which we keep.

(★ Case 2.) Let $t := \mathsf{term}(\mathtt{T})$, and $z := \mathsf{term}(\mathtt{U})$. We have:

$$\mathsf{type}(z) = \mathtt{U}$$
$$\mathsf{chart}(z) = \{\}$$

Translating our rewrite path gives us $G \vdash [\mathtt{T}\ \texttt{==}\ \mathtt{U}]$ which we keep, together with an empty list of derived conformance requirements which we discard. □

The decision procedure. Recall the four basic generic signature queries that describe the equivalence class structure of a generic signature, from Section 5.6. We saw the first two are fundamental from a theoretical standpoint.

Taken together, Corollary 16.47, Theorem 17.4, and Corollary 17.13 say that if we have a well-formed generic signature G presented by a convergent rewriting system $\langle A \,|\, R \rangle$, and the given type parameters are valid, then:

1. $G \vdash [\mathtt{T}\texttt{:}\ \mathtt{P}]$ **if and only if** $\mathsf{term}(\mathtt{T}) \cdot [\mathtt{P}]$ has the same normal form as $\mathsf{term}(\mathtt{T})$.
2. $G \vdash [\mathtt{T}\ \texttt{==}\ \mathtt{U}]$ **if and only if** $\mathsf{term}(\mathtt{T})$ has the same normal form as $\mathsf{term}(\mathtt{U})$.

Also, crucially, the preconditions here are all things we can *check*, and we can compute the normal form of a term using Algorithm 16A. As we now quite clearly can see:

When implemented as below, the two basic generic signature queries always output the correct answer in a finite number of steps.

- **Query:** $\texttt{requiresProtocol}(\mathrm{G}, \mathtt{T}, \mathtt{P})$

  Apply the normal form algorithm to $\mathsf{term}(\mathtt{T}) \cdot [\mathtt{P}]$ and $\mathsf{term}(\mathtt{T})$ and check if we get a pair of identical terms.

- **Query:** $\texttt{areReducedTypeParametersEqual}(\mathrm{G}, \mathtt{T}, \mathtt{U})$

  Apply the normal form algorithm to $\mathsf{term}(\mathtt{T})$ and $\mathsf{term}(\mathtt{U})$ and check if we get a pair of identical terms.

17.2. Symbols

We've now described how the core model of Swift generics translates requirements into rewrite rules. We defined these rules over an alphabet of generic parameter, name, and protocol symbols. The remainder of this chapter describes the full model of Swift generics as it is actually implemented. Going forward, there is less mathematical rigor than before, and more of a focus on practical concerns.

We begin by expanding our alphabet with a few more symbol kinds, to encode both bound type parameters, and the other requirement kinds. While our correctness results only relied on the symmetric term equivalence relation, in the implementation we use the normal form algorithm to get something computable, so we will define a reduction order on our alphabet as well.

Symbols are formed by the rewrite context, the global singleton from Section 15.1 which manages the lifecycle of requirement machine instances. There are eight symbol kinds, with each kind having its own set of structural components. This resembles how we modeled types in Chapter 3.

For each symbol kind, a constructor function takes the structural components and returns a pointer to the unique symbol formed from those structural components. The pointer identity of a symbol is determined by its kind and structural components, so checking equality of symbols is cheap.

Definition 17.14. A *symbol* in the Requirement Machine alphabet is an instance of one of the following:

- A **generic parameter symbol** `τ_d_i`, for some depth d and index $i \in \mathbb{N}$.
- A **name symbol** `A`, where `A` is an identifier appearing in the program.
- A **protocol symbol** [`P`], where `P` is a reference to a protocol declaration.
- An **associated type symbol** [`P`|`A`], where `P` is some protocol, and `A` is the name of an associated type declaration.
- A **layout symbol** [layout: `L`], where `L` is a layout constraint, either `AnyObject` or `_NativeClass`.
- A **superclass symbol** [superclass: `C`; $t_0, \ldots, t_n$] where `C` is an interface type, called the *pattern type* of the symbol, and the t_i are terms, called *substitution terms*. In a superclass symbol, the pattern type is a class or generic class type.
- A **concrete type symbol** [concrete: `X`; $t_0, \ldots, t_n$] where the pattern type `X` is now any interface type that is not a type parameter, and the t_i are substitution terms.
- A **concrete conformance symbol** [concrete: `X`: `P`; $t_0, \ldots, t_n$] where `X` is any interface type that is not a type parameter, the t_i are substitution terms, and `P` is a reference to a protocol declaration.

Name symbols. We already introduced name symbols at the beginning of this chapter and there isn't much else to say, but we add one remark. In the requirement machine for a well-formed generic signature G, every name symbol "`A`" has to be the name of some associated type or type alias of some `P` such that $G \prec \texttt{P}$—for otherwise, "`A`" cannot appear in any valid type parameter. However, we allow name symbols for arbitrary identifiers to be formed, so that we can model invalid programs as well. In any case, only a finite set of name symbols is constructed.

Protocol symbols. The reduction order on protocol symbols has the property that if a protocol `Q` inherits from a protocol `P`, then $[\texttt{P}] < [\texttt{Q}]$.

Consider the directed graph where the vertices are protocol declarations, and for each pair of protocols `Q` and `P` such that `Q` inherits from `P`, there is an edge with source `Q` and destination `P`. Define $\mathsf{depth}(\texttt{P}) \in \mathbb{N}$ as the number of protocols we can reach from `P`, excluding `P` itself. To compute the "depth" of a protocol, we evaluate the **all inherited protocols request** with `P`, which returns the set of all protocols reachable from `P`, and then take the number of elements in this set.

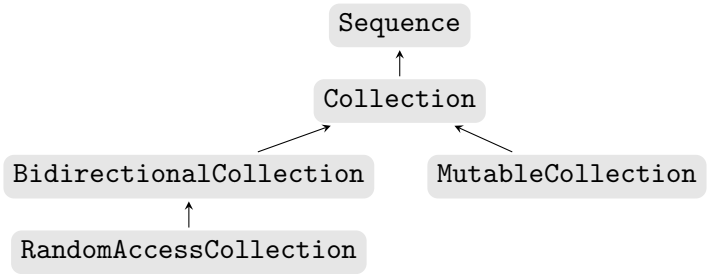
Algorithm 17A (Protocol reduction order)**.** Takes two protocol declarations `P` and `Q` as input, and returns one of "<", ">" or "=" as output.

1. If $\mathsf{depth}(\texttt{P}) > \mathsf{depth}(\texttt{Q})$, return "<". (Deeper protocols are smaller.)
2. If $\mathsf{depth}(\texttt{P}) < \mathsf{depth}(\texttt{Q})$, return ">".
3. If $\mathsf{depth}(\texttt{P}) = \mathsf{depth}(\texttt{Q})$, compare the protocols with Algorithm 5B.

Example 17.15. Various standard library protocols inherit from `Sequence`:

```
public protocol Sequence {...}
public protocol Collection: Sequence {...}
public protocol BidirectionalCollection: Collection {...}
public protocol MutableCollection: Collection {...}
public protocol RandomAccessCollection: BidirectionalCollection {...}
```

The protocol inheritance relationships above define this graph:



The corresponding protocol symbols are ordered as follows—notice how deeper protocols precede shallower protocols:

$$[\texttt{RandomAccessCollection}] < [\texttt{BidirectionalCollection}] < [\texttt{MutableCollection}] < [\texttt{Collection}] < [\texttt{Sequence}]$$

Associated type symbols. While a name symbol represents an unbound dependent member type, an *associated type symbol* represents a *bound* dependent member type. We use the notation $[\texttt{P}|\texttt{A}]$ for an associated type symbol, to evoke the "fusion" of a protocol symbol $[\texttt{P}]$ with a name symbol $\texttt{A}$. In fact, for each associated type symbol $[\texttt{P}|\texttt{A}]$, we introduce an *associated type rule* of the form $[\texttt{P}] \cdot \texttt{A} \sim [\texttt{P}|\texttt{A}]$.

There are "more" associated type symbols than there are associated type declarations, because of protocol inheritance. If $[\texttt{P}|\texttt{A}]$ is an associated type symbol, then $\texttt{A}$ is either the name of an associated type of $\texttt{P}$ itself, or an associated type of some base protocol of $\texttt{P}$. An associated type symbol does not point to an associated type *declaration* for this reason. Instead, the symbol's storage separately references the protocol declaration and identifier.

In the next section, we will see how associated type symbols arise when we translate type parameters to terms. Section 17.4 will discuss associated type rules.

Why do we need associated type symbols? There are two reasons, and we will study them in great detail in Chapter 18:

1. In Section 5.6, we saw that we can implement `isValidTypeParameter`() using the `areReducedTypeParametersEqual`() and `requiresProtocol`() generic signature queries. Once we add associated type symbols, we can instead directly decide if a type parameter is valid using the normal form algorithm.

2. More importantly, it also turns out that we need associated type symbols to obtain a convergent rewriting system when our generic signature has an infinite set of equivalence classes.

The reduction order on associated type symbols is defined from the lexicographic order on identifiers and the protocol reduction order:

Algorithm 17B (Associated type reduction order)**.** Takes two associated type symbols $[\texttt{P}|\texttt{A}]$ and $[\texttt{Q}|\texttt{B}]$ as input, and returns one of "$<$", "$>$" or "$=$" as output.

1. Compare the identifiers $\texttt{A}$ and $\texttt{B}$ using the lexicographic order. Return the result if it is "$<$" or "$>$". Otherwise, both associated types have the same name.

2. Compare the protocols $\texttt{P}$ and $\texttt{Q}$ using Algorithm 17A and return the result.

Example 17.16. We continue Example 17.15. The `Sequence` protocol declares an `Element` associated type, so protocols inheriting from `Sequence` inherit this associated type. The protocol machine for `RandomAccessCollection` has an alphabet with various associated type symbols, ordered as follows:

$$\begin{aligned}&[\texttt{RandomAccessCollection}|\texttt{Element}]\\ &< [\texttt{BidirectionalCollection}|\texttt{Element}]\\ &< [\texttt{Collection}|\texttt{Element}]\\ &< [\texttt{MutableCollection}|\texttt{Element}]\\ &< [\texttt{Sequence}|\texttt{Element}]\end{aligned}$$

We will see *why* the order must respect protocol inheritance in Section 18.5.

The next three symbol kinds appear when we build rewrite rules for layout, superclass, and concrete type requirements. Just like a conformance requirement `[T: P]` defines a rewrite rule $\mathsf{term}(\texttt{T}) \cdot [\texttt{P}] \sim \mathsf{term}(\texttt{T})$, these other requirement kinds define rewrite rules of the form $\mathsf{term}(\texttt{T}) \cdot s \sim \mathsf{term}(\texttt{T})$, where `T` is the requirement's subject type, and s is a layout, superclass, or concrete type symbol.

Layout symbols. The $[\mathsf{layout}\texttt{: AnyObject}]$ layout symbol represents the `AnyObject` layout constraint we introduced in Definition 4.1. It appears when we translate a layout requirement `[T: AnyObject]` into a rule $\mathsf{term}(\texttt{T}) \cdot [\mathsf{layout}\texttt{: AnyObject}] \sim \mathsf{term}(\texttt{T})$.

We also mentioned the `_NativeClass` layout constraint in Section 5.6. This constraint cannot be stated directly by the user, but it is implied by a superclass requirement to a Swift-native class. These two layout symbols are ordered as follows:

$$[\mathsf{layout}\texttt{: AnyObject}] < [\mathsf{layout}\texttt{: _NativeClass}]$$

Superclass and concrete type symbols. Superclass and concrete type symbols encode a concrete type that may contain type parameters. To form the symbol, we first decompose the concrete type into a *pattern type* and a list of *substitution terms*:

$$\text{concrete type} = \text{pattern type} + \text{substitution terms}$$

We're forming the symbol because we're translating an explicit or associated requirement into a rewrite rule, so we use the "term" or "$\mathsf{term}_{\texttt{P}}$" mapping as appropriate to translate each type parameter appearing in our concrete type. (The next section will introduce Algorithm 17G and Algorithm 17H used for this purpose, but for now, the definitions from the start of this chapter remain valid.) We then replace each type parameter with a "phantom" generic parameter $\tau\texttt{_0_}i$, where the depth is always zero, and the index $i \in \mathbb{N}$ is the index of the corresponding substitution term in the list.

Algorithm 17C (Build concrete type symbol)**.** Receives an interface type `X` as input, and optionally, a protocol `P`. As output, returns a pattern type together with a list of substitution terms. Note that the type `X` must not itself be a type parameter.

1. (Initialize) Set $S \leftarrow \{\}$. Set $i \leftarrow 0$.
2. (Recurse) Perform a pre-order walk over the tree structure of `X` and transform every type parameter `T` that occurs in `X` as follows, to form a new type `Y`:
   a) (Record) If we're lowering a generic signature requirement, let $t :=$ $\mathsf{term}(\texttt{T})$. Otherwise, let $t :=$ $\mathsf{term}_{\texttt{P}}(\texttt{T})$. Set $S \leftarrow S + \{t\}$. (Thus, $S[i] = t$.)
   b) (Transform) Replace this occurrence of `T` with `τ_0_i`.
   c) (Next) Set $i \leftarrow i + 1$. (Now, $|S| = i$.)
3. Return the pattern type `Y`, and the array of substitution terms S.

Example 17.17. Given the explicit requirement [`τ_0_0 == Array<τ_0_1.Element>`], we decompose the right-hand side as follows:

$$\texttt{Array<τ_0_1.Element>} = \texttt{Array<τ_0_0>} + \{\texttt{τ_0_1} \cdot \texttt{Element}\}$$

We can then form the concrete type symbol [concrete: `Array<τ_0_0>`; `τ_0_1 · Element`] from the pattern type `Array<τ_0_0>` and list of substitution terms {`τ_0_1 · Element`}.

Example 17.18. If we have a fully concrete type that does not contain type parameters, then our list of substitution terms will be empty. For example, the right-hand side of the superclass requirement [`Self: NSObject`] defines the symbol [superclass: `NSObject`].

Example 17.19. The pattern type output by Algorithm 17C satisfies certain conditions:

1. The pattern type itself cannot be a type parameter.
2. The pattern type does not contain any dependent member types.
3. Every generic parameter type appearing in the pattern type has depth zero.
4. Every generic parameter type has a unique index.
5. Indices are consecutive and start from zero.

None of the following are valid pattern types:

```
τ_0_0
Array<τ_0_0.Element>
Array<τ_1_0>
Dictionary<τ_0_0, τ_0_0>
(τ_0_1) -> τ_0_0
```

To compare a pair of superclass or concrete type symbols, we first compare their pattern types for canonical type equality, and then compare their substitution terms using the reduction order on terms, which we define in the next section. Note that this is a partial order; symbols with distinct pattern types are incomparable.

Algorithm 17D (Concrete type reduction order)**.** Takes two superclass or concrete type symbols s_1 and s_2 as input. Returns one of "<", ">", "=" or "⊥" as output.

1. (Invariant) We assume the two symbols already have the same kind; the case of comparing different kinds is handled by the general symbol order we define next.
2. (Incomparable) Compare the pattern types of s_1 and s_2 for canonical type equality. If the pattern types are distinct, return "⊥".
3. (Initialize) Both symbols have the same pattern type, so they must have the same number of substitution terms, say n. Set $i \leftarrow 0$.
4. (Equal) If $i = n$, all substitution terms are identical. Return "=".
5. (Compare) Compare the ith substitution term of s_1 and s_2 using Algorithm 17I. Return the result if it is "<" or ">".
6. (Next) Otherwise, set $i \leftarrow i + 1$ and go back to Step 4.

While superclass and concrete type symbols may contain terms, those terms cannot themselves contain superclass and concrete type symbols, because terms corresponding to type parameters only contain generic parameter, name, protocol, and associated type symbols.

Concrete conformance symbols. The notation for a concrete conformance symbol looks like a concrete type symbol, but it also stores a protocol declaration. We will see in Section 19.4 that when a type parameter `T` is subject to the combination of a conformance requirement `[T: P]` and a concrete same-type requirement `[T == X]`, we introduce a rewrite rule containing a concrete conformance symbol:

$$\mathsf{term}(\texttt{T}) \cdot [\mathsf{concrete}\texttt{: X: P; ...}] \sim \mathsf{term}(\texttt{T})$$

To compare two concrete conformance symbols, we first compare their protocol, then we compare their pattern type and substitution terms.

Algorithm 17E (Concrete conformance reduction order)**.** Takes two concrete conformance symbols s_1 and s_2 as input. Returns one of "<", ">", "=" or "⊥" as output.

1. Compare the protocol of s_1 and s_2 using Algorithm 17A. Return the result if it is "<" or ">".
2. Otherwise, compare the pattern type and substitution terms as in Algorithm 17D.

Symbol order. To compare an arbitrary pair of symbols, we first define a mapping from symbol kinds to natural numbers:

Concrete conformance:	0
Protocol:	1
Associated type:	2
Generic parameter:	3
Name:	4
Layout:	5
Superclass:	6
Concrete type:	7

Algorithm 17F (Symbol reduction order)**.** Takes two symbols as input, and returns one of "<", ">", "=" or "⊥" as output.

If the two symbols have different kinds, map their kinds to natural numbers as above, and compare the numbers with the usual linear order on $\mathbb{N}$. Otherwise, both symbols have the same kind, and we already saw how to handle each case:

- For generic parameter symbols, use Algorithm 5A.
- For name symbols, use the lexicographic order on strings.
- For protocol symbols, use Algorithm 17A.
- For associated type symbols, use Algorithm 17B.
- For layout symbols, use the order previously described.
- For superclass symbols, use Algorithm 17D.
- For concrete type symbols, use Algorithm 17D.
- For concrete conformance symbols, use Algorithm 17E.

17.3. Terms

In the theory, a term is an element of a free monoid. In the implementation, a term is a sequence of symbols. There are two flavors. A *mutable term* is a value type which stores its own heap-allocated buffer. Mutable terms are cheap to modify, but expensive to copy and store. An *immutable term* is expensive to allocate, because it is uniqued by the rewrite context, but cheap to compare for equality, because immutable terms with the same length and symbols are equal as pointers. The normal form algorithm and Knuth-Bendix completion use mutable terms for intermediate results. Immutable terms are used when storing the left-hand and right-hand side terms of a rewrite rule.

In the next section, we will see that the rewrite rules in a requirement machine always have non-empty terms on both sides. For this reason, there is no immutable term representation of the empty term ε. Mutable terms may be empty though, and that is the initial state of a mutable term upon creation, before any symbols are added.

We now describe a pair of algorithms for translating type parameters to mutable terms, implementing the "term" and "$\text{term}_\texttt{P}$" mappings from the start of this chapter. We start with the "term" mapping, used for building terms in a **query machine** or **minimization machine**. (Recall the four machine kinds from Chapter 15.) We extend this mapping to handle bound dependent member types, which translate to associated type symbols. They appear when we're building a query machine from the minimal requirements of an existing generic signature. Here is the general idea:

T:	term(T):
τ_0_0.A.B	τ_0_0 · A · B
τ_0_0.[P]A.[Q]B	τ_0_0 · [P|A] · [Q|B]

A type parameter is a single-linked list where each interior node is a dependent member type, and the tail of the list is a generic parameter type. The head of the list is the *outermost* member type, so "B" in τ_0_0.A.B, and the base type of each dependent member type is the "next" pointer.

The below is an instance of the classical algorithm for converting a linked list into an array. We traverse the list from head to tail, at each step adding a symbol at the end of the term; this leaves us with a term where the *last* element is the generic parameter symbol. We then reverse the order of symbols to get the final result.

Algorithm 17G (Build term for explicit requirement)**.** Takes a type parameter T as input, and outputs a (non-empty) mutable term.

1. (Initialize) Let t be a new empty mutable term.
2. (Base case) If T is a generic parameter type τ_d_i:
   a) First, add the generic parameter symbol τ_d_i to t.
   b) Then, reverse the symbols in t, and return t.
3. (Recursive case) Otherwise, T is a dependent member type:
   a) If T is an unbound dependent member type U.A for some type parameter U, add the name symbol A to t.
   b) Otherwise, T is a bound dependent member type U.[P]A, again for some type parameter U. Add the associated type symbol [P|A] to t.

   Set T ← U, and go back to Step 2.

Theorem 11.9 states that every valid type parameter has an equivalent bound and unbound type parameter of the same length. In the requirement machine, if we apply "term" to such a pair of equivalent type parameters, we get a pair of equivalent terms under the term equivalence relation generated by our rewrite rules. The unbound type parameter maps to a term containing name symbols, while the bound type parameter maps to a term containing associated type symbols. The equivalence is a consequence of the associated type rules, introduced in the next section.

An alternative valid implementation of Algorithm 17G would instead map both bound and unbound dependent member types to name symbols. This would create more work for the normal form algorithm and in completion, which our implementation avoids.

Next, consider the "$\mathsf{term}_{\mathtt{P}}$" mapping, used for building terms in a **protocol machine** or **protocol minimization machine**. In the core model, "$\mathsf{term}_{\mathtt{P}}$" maps an unbound type parameter of $G_{\mathtt{P}}$ to a term that starts with a protocol symbol, followed by a series of name symbols. We extend this to bound type parameters as follows: we drop the protocol symbol, and translate each dependent member type to an associated type symbol:

$\mathtt{T}$:	$\mathsf{term}_{\mathtt{P}}(\mathtt{T})$:
`Self`	$[\mathtt{P}]$
`Self.A.B`	$[\mathtt{P}] \cdot \mathtt{A} \cdot \mathtt{B}$
`Self.[P]A.[Q]B`	$[\mathtt{P}\vert\mathtt{A}] \cdot [\mathtt{Q}\vert\mathtt{B}]$

Algorithm 17H (Build term for associated requirement)**.** Takes a type parameter `T` and a protocol declaration `P` as input, and outputs a (non-empty) mutable term.

1. (Initialize) Let t be a new empty mutable term.
2. (Base case) If `T` is the protocol `Self` type (`τ_0_0`), consider the term t constructed so far:
   a) If t ends with an associated type symbol $[\mathtt{Q}\vert\mathtt{A}]$, then either `Q` is identical to `P`, or `P` inherits from `Q`. In both cases, *replace* the last symbol of t with $[\mathtt{P}\vert\mathtt{A}]$.
   b) Otherwise, either t is empty, or it ends with a name symbol. Add the protocol symbol $[\mathtt{P}]$ to t.

   Finally, reverse the symbols in t, and return t.
3. (Recursive case) Otherwise, `T` is a dependent member type.
   a) If `T` is an unbound dependent member type `U.A`, add the name symbol `A` to t.
   b) Otherwise, `T` is a bound dependent member type `U.[Q]A`. Add the associated type symbol $[\mathtt{Q}\vert\mathtt{A}]$ to t.

   Set $\mathtt{T} \leftarrow \mathtt{U}$, and go back to Step 2.

Example 17.20. Recall Example 5.17, and notice how Algorithm 17H behaves with protocol inheritance and re-stated associated types:

```
protocol Root {
  associatedtype Element
}

protocol Left: Root {}

protocol Right: Root {
  associatedtype Element
}
```

The type parameters `Self.Element` and `Self.[Root]Element` are equivalent in all three of $G_{\texttt{Root}}$, $G_{\texttt{Left}}$ and $G_{\texttt{Right}}$, via the AssocBind inference rule. In $G_{\texttt{Right}}$, this equivalence class also contains `Self.[Right]Element`, because `Right` re-states this associated type.

These type parameters map to these terms in each of the three protocol machines:

$$\begin{aligned}
&\text{term}_{\texttt{Root}}(\texttt{Self.Element}) = [\texttt{Root}] \cdot \texttt{Element}\\
&\text{term}_{\texttt{Root}}(\texttt{Self.[Root]Element}) = [\texttt{Root}|\texttt{Element}]\\
&\\
&\text{term}_{\texttt{Left}}(\texttt{Self.Element}) = [\texttt{Left}] \cdot \texttt{Element}\\
&\text{term}_{\texttt{Left}}(\texttt{Self.[Root]Element}) = [\texttt{Left}|\texttt{Element}]\\
&\\
&\text{term}_{\texttt{Right}}(\texttt{Self.Element}) = [\texttt{Right}] \cdot \texttt{Element}\\
&\text{term}_{\texttt{Right}}(\texttt{Self.[Root]Element}) = [\texttt{Right}|\texttt{Element}]\\
&\text{term}_{\texttt{Right}}(\texttt{Self.[Right]Element}) = [\texttt{Right}|\texttt{Element}]
\end{aligned}$$

Reduction order. In the previous section, we defined a partial order on symbols. We now extend this to a partial order on terms. Recall the discussion of reduction orders from Section 16.5.

We start with the standard shortlex order from Algorithm 16B, but add an additional check to ensure that terms containing name symbols are always ordered after terms without name symbols, even if the term with name symbols is shorter. This is an instance of a *weighted shortlex order*, where our chosen weight function counts the number of name symbols appearing in a term. In Section 17.5, we will see that the weighted shortlex order is necessary to get the correct behavior with protocol type aliases.

Definition 17.21. Let A be any set. We say that $w\colon A^* \to \mathbb{N}$ is a *weight function* if it satisfies $w(xy) = w(x) + w(y)$ for all x, $y \in A^*$. That is, w is a monoid homomorphism from A^* to $\mathbb{N}$, where the latter is viewed as a monoid under addition.

Algorithm 17I (Weighted shortlex order)**.** Takes two terms t and u as input, and returns one of "$<$", "$>$", "$=$" or "$\perp$" as output.

1. (Weight) Compute $w(t)$ and $w(u)$.
2. (Less) If $w(t) < w(u)$, return "$<$".
3. (More) If $w(t) > w(u)$, return "$>$".
4. (Shortlex) Otherwise $w(t) = w(u)$, so we compare the terms using Algorithm 16B and return the result.

The weighted shortlex order is a suitable reduction order under Definition 16.40.

Proposition 17.22. Let $w\colon A^* \to \mathbb{N}$ be a weight function. Then the weighted shortlex order induced by w is translation-invariant and well-founded.

Proof. Let $<$ be the standard shortlex order on A^*, and let $<_w$ be the weighted shortlex order induced by w.

We prove translation invariance first. We're given x, y, $z \in A^*$ with $x <_w y$. We will show that $zx <_w zy$; the proof that $xz <_w yz$ is similar. Since $x <_w y$, by definition we either have $w(x) < w(y)$, or $w(x) = w(y)$ and $x < y$. We consider each case:

1. If $w(x) < w(y)$, then $w(z) + w(x) < w(z) + w(y)$ because the linear order on $\mathbb{N}$ is translation-invariant, and since $w(zx) = w(z) + w(x)$ and $w(zy) = w(z) + w(y)$, we see that $w(zx) < w(zy)$, therefore $zx <_w zy$.
2. If $w(x) = w(y)$ and $x < y$, then $w(zx) = w(zy)$, and the translation invariance of $<$ on A^* implies that $zx < zy$. Once again, we conclude that $zx <_w zy$.

Now, we demonstrate that $<_w$ is well-founded, by contradiction. Assume we have an infinite descending chain:

$$t_1 >_w t_2 >_w t_3 >_w \cdots$$

If we apply the weight function w to each term t_i in this chain, we see that:

$$w(t_1) \geq w(t_2) \geq w(t_3) \geq \cdots$$

Since $w(t_i) \in \mathbb{N}$, we can only "step down" finitely many times. So there exists some index i such that for all $j > i$, $w(t_j) = w(t_i)$. Together with $t_j <_w t_i$, we conclude that actually, $t_j < t_i$. Thus, we get an infinite descending chain for the *standard* shortlex order $<$:

$$t_i > t_{i+1} > t_{i+2} > \cdots$$

However, this is a contradiction, because the standard shortlex order $<$ is well-founded. Thus, $<_w$ must also be well-founded. □

We now dispose of the notation $<_w$. We will denote the reduction order on terms by $<$, with the understanding that it is the weighted shortlex order of Algorithm 17I.

17.4. Rules

We now describe a pair of algorithms implementing the "rule" and "rule_P" mappings. We start with the "rule" mapping, which defines the local rules of a **query machine** or **minimization machine**. We can assume the input requirement is desugared already (Section 11.2), so the left-hand side is always a type parameter. We use Algorithm 17G, denoted "term", to translate type parameters to terms.

Our core model only supported conformance requirements and same-type requirements between type parameters. We saw that we translate a conformance requirement [T: P] into a rewrite rule $\text{term}(\text{T}) \cdot [\text{P}] \sim \text{term}(\text{T})$. We will now think of this as a special case of a *property rule*, which is a rewrite rule of the form $\text{term}(\text{T}) \cdot s \sim \text{term}(\text{T})$ where s is a symbol. In the full model, the other requirement kinds also translate to property rules, while same-type requirements between type parameters remain essentially distinct:

	Property rules: $\text{term}(\text{T}) \cdot s \sim \text{term}(\text{T})$	
rule [T: P]	$s = [\text{P}]$	protocol symbol
rule [T: AnyObject]	$s = [\text{layout: AnyObject}]$	layout symbol
rule [T: C]	$s = [\text{superclass: C}; \ldots]$	superclass symbol
rule [T == X]	$s = [\text{concrete: X}; \ldots]$	concrete type symbol
	Same-type rules:	
rule [T == U]	$\text{term}(\text{T}) \sim \text{term}(\text{U})$	

Algorithm 17J (Build rule from explicit requirement)**.** Takes a desugared requirement as input, and outputs a pair of terms.

1. For a **conformance requirement** [T: P], return the pair $(\text{term}(\text{T}) \cdot s, \text{term}(\text{T}))$ where s is the protocol symbol [P].
2. For a **layout requirement** [T: AnyObject], return the pair $(\text{term}(\text{T}) \cdot s, \text{term}(\text{T}))$ where s is the layout symbol [layout: AnyObject].
3. For a **superclass requirement** [T: C], return the pair $(\text{term}(\text{T}) \cdot s, \text{term}(\text{T}))$ where s is the superclass symbol from Algorithm 17C applied to C.
4. For a **same-type requirement** [T == X] with X a **concrete type**, return the pair $(\text{term}(\text{T}) \cdot s, \text{term}(\text{T}))$ where s is the concrete type symbol from Algorithm 17C applied to X.
5. For a **same-type requirement** [T == U] with U a **type parameter**, return the pair $(\text{term}(\text{T}), \text{term}(\text{U}))$.

Associated requirements. In a **protocol machine** or **protocol minimization machine**, we obtain local rules from associated requirements in a similar manner, except now, we map type parameters to terms using Algorithm 17H, denoted "$\mathsf{term}_{\mathtt{P}}$".

	Property rules: $\mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \cdot s \sim \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U})$	
$\mathsf{rule}_{\mathtt{P}}\,[\mathtt{Self.U:\ Q}]_{\mathtt{P}}$	$s = [\mathtt{Q}]$	protocol symbol
$\mathsf{rule}_{\mathtt{P}}\,[\mathtt{Self.U:\ AnyObject}]_{\mathtt{P}}$	$s = [\mathsf{layout}\mathtt{:\ AnyObject}]$	layout symbol
$\mathsf{rule}_{\mathtt{P}}\,[\mathtt{Self.U:\ C}]_{\mathtt{P}}$	$s = [\mathsf{superclass}\mathtt{:\ C};\ \ldots]$	superclass symbol
$\mathsf{rule}_{\mathtt{P}}\,[\mathtt{Self.U\ ==\ X}]_{\mathtt{P}}$	$s = [\mathsf{concrete}\mathtt{:\ X};\ \ldots]$	concrete type symbol
	Same-type rules:	
$\mathsf{rule}_{\mathtt{P}}\,[\mathtt{Self.U\ ==\ Self.V}]_{\mathtt{P}}$	$\mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \sim \mathsf{term}_{\mathtt{P}}(\mathtt{Self.V})$	

Algorithm 17K (Build rule from associated requirement)**.** Takes a desugared requirement and a protocol declaration `P` as input, and outputs a pair of terms.

1. For an **associated conformance requirement** $[\mathtt{Self.U:\ Q}]_{\mathtt{P}}$, return the pair $(\mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \cdot s, \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}))$ where s is the protocol symbol $[\mathtt{P}]$.
2. For an **associated layout requirement** $[\mathtt{Self.U:\ AnyObject}]_{\mathtt{P}}$, return the pair $(\mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \cdot s, \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}))$ where s is the symbol $[\mathsf{layout}\mathtt{:\ AnyObject}]$.
3. For an **associated superclass requirement** $[\mathtt{Self.U:\ C}]_{\mathtt{P}}$, return the pair of $(\mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \cdot s, \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}))$ where s is the superclass symbol from Algorithm 17C applied to `C`.
4. For an **associated same-type requirement** $[\mathtt{Self.U\ ==\ X}]_{\mathtt{P}}$ with `X` a **concrete type**, return the pair $(\mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}) \cdot s, \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}))$ where s is the concrete type symbol from Algorithm 17C applied to `X`.
5. For a **associated same-type requirement** $[\mathtt{Self.U\ ==\ Self.V}]_{\mathtt{P}}$ with `Self.V` a **type parameter**, return the pair $(\mathsf{term}_{\mathtt{P}}(\mathtt{Self.U})), \mathsf{term}_{\mathtt{P}}(\mathtt{Self.V}))$.

Additional rules. Protocols also have a few rules that are not from requirements:

Identity conformance rule:	
`protocol P {...}`	$[\mathtt{P}] \cdot [\mathtt{P}] \sim [\mathtt{P}]$
Associated type rules:	
`associatedtype A`	$[\mathtt{P}] \cdot \mathtt{A} \sim [\mathtt{P}|\mathtt{A}]$
Protocol type alias rules:	
`typealias A = Self.U`	$[\mathtt{P}] \cdot \mathtt{A} \sim \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U})$
`typealias A = X`	$[\mathtt{P}] \cdot \mathtt{A} \cdot [\mathsf{concrete}\mathtt{:\ X};\ \ldots] \sim [\mathtt{P}] \cdot \mathtt{A}$

Recall that in the core model, a protocol without any associated requirements does not contribute any rewrite rules to the monoid presentation. In the full model, we see that every protocol adds at least one rewrite rule.

First up, the *identity conformance rule* $[\texttt{P}] \cdot [\texttt{P}] \sim [\texttt{P}]$ conceptually states that the protocol `Self` type of `P` conforms to `P`. It plays a minor role, and we will be able to ignore it for the most part, but we will justify its existence in Example 18.10.

Then, we collect the names of the protocol's own associated type declarations and those of all inherited protocols, and for each distinct name `A`, we add an *associated type rule* $[\texttt{P}] \cdot \texttt{A} \sim [\texttt{P}|\texttt{A}]$. This states that $\texttt{term}_{\texttt{P}}(\texttt{Self.A})$ is equivalent to $\texttt{term}_{\texttt{P}}(\texttt{Self.[P]A})$, so these rules encode the AssocBind inference rule in the derived requirements formalism. Repeated application of these rules allow us to rewrite a term for a bound type parameter into one for an unbound type parameter, and vice versa. We will say more about this in Section 18.2 and Section 18.4.

Finally, we add a *protocol type alias rule* for each type alias declaration that appears in the protocol. This allows protocol type aliases to be subject to requirements, an extension of the core model. We will discuss protocol type aliases in the next section. We end this section by demonstrating how associated type rules work in practice.

Example 17.23. In the core model, the `Sequence` protocol was completely described by the two rewrite rules for its associated requirements. In the full model, these rules change form when we pass from unbound to bound type parameters, and we also add a few additional rules.

First, consider what happens when we compile the standard library itself. We type check the protocol declarations from source, so we must build the requirement signatures of `IteratorProtocol` and `Sequence`, in that order, because of the protocol dependency.

Let's look at the **protocol minimization machine** for `IteratorProtocol` first. This machine has an identity conformance rule, and an associated type rule. These rules were not present in the core model:

$$[\texttt{IteratorProtocol}] \cdot [\texttt{IteratorProtocol}] \sim [\texttt{IteratorProtocol}] \quad (1)$$

$$[\texttt{IteratorProtocol}] \cdot \texttt{Element} \sim [\texttt{IteratorProtocol}|\texttt{Element}] \quad (2)$$

Next, we build the protocol minimization machine for `Sequence`. We import the two rules from `IteratorProtocol`, then add an identity conformance rule for `Sequence` and two associated type rules for its associated types:

$$[\texttt{Sequence}] \cdot [\texttt{Sequence}] \sim [\texttt{Sequence}] \quad (3)$$

$$[\texttt{Sequence}] \cdot \texttt{Element} \sim [\texttt{Sequence}|\texttt{Element}] \quad (4)$$

$$[\texttt{Sequence}] \cdot \texttt{Iterator} \sim [\texttt{Sequence}|\texttt{Iterator}] \quad (5)$$

Finally, we resolve the associated requirements written inside the `Sequence` protocol.

We get two requirements written with unbound type parameters:

$$[\texttt{Self.Iterator: IteratorProtocol}]_{\texttt{Sequence}}$$
$$[\texttt{Self.Element == Self.Iterator.Element}]_{\texttt{Sequence}}$$

We apply "$\mathsf{rule}_{\mathsf{P}}$" to get the same two rewrite rules we had in the core model:

$$[\texttt{Sequence}] \cdot \texttt{Iterator} \cdot [\texttt{IteratorProtocol}] \sim [\texttt{Sequence}] \cdot \texttt{Iterator} \quad (6)$$
$$[\texttt{Sequence}] \cdot \texttt{Element} \sim [\texttt{Sequence}] \cdot \texttt{Iterator} \cdot \texttt{Element} \quad (7)$$

We have seven local rules in total. The next step is completion, which we describe in Chapter 18; for now, the important fact is that we transform rules (6) and (7) into a different form. The new rules contain associated type symbols instead of protocol and name symbols:

$$[\texttt{Sequence}|\texttt{Iterator}] \cdot [\texttt{IteratorProtocol}] \sim [\texttt{Sequence}|\texttt{Iterator}] \quad (\diamond 6)$$
$$[\texttt{Sequence}|\texttt{Element}] \sim [\texttt{Sequence}|\texttt{Iterator}] \cdot [\texttt{IteratorProtocol}|\texttt{Element}] \quad (\diamond 7)$$

If we consider rules (1) through (7), we see that replacing (6) and (7) with ($\diamond 6$) and ($\diamond 7$) does not change the term equivalence relation. Indeed, the two sides of ($\diamond 6$) are already equivalent via this rewrite path once we have (5) and (6),

$$([\texttt{Sequence}|\texttt{Iterator}] \Rightarrow [\texttt{Sequence}] \cdot \texttt{Iterator}) \triangleright [\texttt{IteratorProtocol}]$$
$$\circ ([\texttt{Sequence}] \cdot \texttt{Iterator} \cdot [\texttt{IteratorProtocol}] \Rightarrow [\texttt{Sequence}] \cdot \texttt{Iterator})$$
$$\circ ([\texttt{Sequence}|\texttt{Iterator}] \Rightarrow [\texttt{Sequence}] \cdot \texttt{Iterator})$$

and conversely, here is a rewrite path for (6) using (5) and ($\diamond 6$):

$$([\texttt{Sequence}] \cdot \texttt{Iterator} \Rightarrow [\texttt{Sequence}|\texttt{Iterator}]) \triangleright [\texttt{IteratorProtocol}]$$
$$\circ ([\texttt{Sequence}|\texttt{Iterator}] \cdot [\texttt{IteratorProtocol}] \Rightarrow [\texttt{Sequence}|\texttt{Iterator}])$$
$$\circ ([\texttt{Sequence}|\texttt{Iterator}] \Rightarrow [\texttt{Sequence}] \cdot \texttt{Iterator})$$

Similarly, one can show that rules (7) and ($\diamond 7$) are interchangeable via a pair of rewrite paths. Now, note that if we were to apply "$\mathsf{rule}_{\mathsf{P}}$" to the pair of requirements below, written with bound dependent member types, we would receive rules ($\diamond 6$) and ($\diamond 7$):

$$[\texttt{Self.[Sequence]Iterator: IteratorProtocol}]_{\texttt{Sequence}}$$
$$[\texttt{Self.[Sequence]Element ==}$$
$$\texttt{Self.[Sequence]Iterator.[IteratorProtocol]Element}]_{\texttt{Sequence}}$$

At the end of the rule minimization process, we translate the rules ($\diamond 6$) and ($\diamond 7$) back into requirements, and we get exactly the two requirements above (we will address this reverse translation in Section 20.4). This gives us the requirement signature of `Sequence`. We serialize this representation inside the binary module for the standard library.

Example 17.24. Suppose now the user's program states a conformance requirement to `Sequence`. We construct a **protocol machine** for each one of `IteratorProtocol` and `Sequence`. We deserialize their requirement signatures from the binary module we previously built for the standard library. The protocol machine for `IteratorProtocol` consists of rules (1) and (2) as before.

The protocol machine for `Sequence` consists of rule (1) through (5) together with ($\diamond$6) and ($\diamond$7), because the serialized associated requirements in a requirement signature are always written with bound type parameters.

Example 17.25. Finally, we revisit the generic signature from Example 17.1 in the full model. Suppose the user's program states this declaration:

```
func allEqual<T: Sequence, U: Sequence>(_ t: T, _ u: U)
    where T.Element: Equatable,
          T.Element == U.Element {...}
```

We construct a **minimization machine** to build this declaration's generic signature. We import rules from the protocol machines for `IteratorProtocol`, `Sequence`, and `Equatable` (as `Equatable` has no associated types, it only contributes a single rule):

$$[\texttt{IteratorProtocol}] \cdot [\texttt{IteratorProtocol}] \sim [\texttt{IteratorProtocol}] \quad (1)$$
$$[\texttt{IteratorProtocol}] \cdot \texttt{Element} \sim [\texttt{IteratorProtocol}|\texttt{Element}] \quad (2)$$
$$[\texttt{Sequence}] \cdot [\texttt{Sequence}] \sim [\texttt{Sequence}] \quad (3)$$
$$[\texttt{Sequence}] \cdot \texttt{Element} \sim [\texttt{Sequence}|\texttt{Element}] \quad (4)$$
$$[\texttt{Sequence}] \cdot \texttt{Iterator} \sim [\texttt{Sequence}|\texttt{Iterator}] \quad (5)$$
$$[\texttt{Sequence}|\texttt{Iterator}] \cdot [\texttt{IteratorProtocol}] \sim [\texttt{Sequence}|\texttt{Iterator}] \quad (\diamond 6)$$
$$[\texttt{Sequence}|\texttt{Element}] \sim [\texttt{Sequence}|\texttt{Iterator}] \cdot [\texttt{IteratorProtocol}|\texttt{Element}] \quad (\diamond 7)$$
$$[\texttt{Equatable}] \cdot [\texttt{Equatable}] \sim [\texttt{Equatable}] \quad (8)$$

We resolve the user-written requirements of `allEqual()`, and apply Algorithm 17J to each one, to get four more rules:

$$\tau_0_0 \cdot [\texttt{Sequence}] \sim \tau_0_0 \quad (9)$$
$$\tau_0_1 \cdot [\texttt{Sequence}] \sim \tau_0_1 \quad (10)$$
$$\tau_0_0 \cdot \texttt{Element} \cdot [\texttt{Equatable}] \sim \tau_0_0 \cdot \texttt{Element} \quad (11)$$
$$\tau_0_0 \cdot \texttt{Element} \sim \tau_0_1 \cdot \texttt{Element} \quad (12)$$

Once again, the associated type rules allow us to convert (11) and (12) into a new form that involves associated type symbols. First, notice that $\tau_0_0 \cdot [\texttt{Sequence}|\texttt{Element}]$ is

equivalent to τ_0_0 · Element via this rewrite path using rules (4) and (9):

τ_0_0 ◁ ([Sequence|Element] ⇒ [Sequence] · Element)
∘ (τ_0_0 · [Sequence] ⇒ τ_0_0) ▷ Element

We can also write the source and destination of this rewrite path as the application of "term" to two type parameters, τ_0_0.[Sequence]Element and τ_0_0.Element, so in a sense this rewrite path is analogous to the following derived requirement:

1. [τ_0_0: Sequence] (Conf)
2. [τ_0_0.[Sequence]Element == τ_0_0.Element] (AssocBind 1)

Similarly, we have an equivalence of terms τ_0_1·[Sequence|Element] ∼ τ_0_1·Element via a rewrite path involving rules (4) and (10). For this reason, completion and rule minimization replace rules (11) and (12) with (◇11) and (◇12) below:

τ_0_0 · [Sequence|Element] · [Equatable] ∼ τ_0_0 · [Sequence|Element] (◇11)
τ_0_0 · [Sequence|Element] ∼ τ_0_1 · [Sequence|Element] (◇12)

Rules (1) through (10) together with (11) and (12), generate the same term equivalence relation as rules (1) through (10) together with (◇11) and (◇12).

Here are the requirements corresponding to (11) and (12), that were input:

[τ_0_0.Element: Equatable]
[τ_0_0.Element == τ_0_1.Element]

Here are the requirements corresponding to (◇11) and (◇12) that we get at the end of minimization:

[τ_0_0.[Sequence]Element: Equatable]
[τ_0_0.[Sequence]Element == τ_0_1.[Sequence]Element]

According to Proposition 11.10, our user-written requirements generate the same theory as the reduced requirements output by minimization. We output the final generic signature:

```
<τ_0_0, τ_0_1 where τ_0_0: Sequence, τ_0_1: Sequence,
                    τ_0_0.[Sequence]Element: Equatable,
                    τ_0_0.[Sequence]Element ==
                        τ_0_1.[Sequence]Element>
```

If we subsequently build a **query machine** for this generic signature, perhaps in a different compilation session for a module that imports this one, we apply "rule" to each requirement in the generic signature, and we right away get rules (1) through (10) together with (◇11) and (◇12).

17.5. Protocol Type Aliases

Recall that protocol type aliases appear as member types of type parameters, alongside associated type declarations (Section 9.2). If a type parameter `T` conforms to a protocol `P` that declares a type alias `A`, the user can state the member type representation `T.A`.

In the interface resolution stage, we already have a generic signature, and we can issue generic signature queries against the type parameter `T` to find the declaration of `A`. We then substitute `Self` with `T` in the underlying type of `A`. However, protocol type aliases can also appear in the requirements of a `where` clause, which are resolved in the structural resolution stage, *while* we're building the current context's generic signature.

In this case, type resolution don't know anything about `T` yet, so the resolved type is always an unbound dependent member type `T.A`. Therefore, in a source program with protocol type aliases, an unbound dependent member type may refer to the name of a *type alias* declaration, and not just an associated type declaration. This gives us a new behavior that is not part of the derived requirements formalism. Instead, we just describe how it is implemented at the level of rewrite rules here.

Each protocol type alias `A` in a protocol `P` defines a *protocol type alias rule* in the protocol machine for `P` as follows. We imagine an associated same-type requirement that equates the unbound dependent member type `Self.A` on the left-hand side, with the underlying type of `A` on the right-hand side. Then, the protocol type alias rule for `A` is precisely the rule we would get if we applied "$\mathsf{rule}_{\mathtt{P}}$" to our imaginary requirement.

Algorithm 17L (Build rule for protocol type alias)**.** Takes a protocol `P`, and a type alias declaration `A`, understood to be a member of `P`, as input. Returns a pair of terms (u, v) as output.

1. Get the underlying type of `A`. How we get the underlying type depends on the requirement machine kind:
   - If this is a **protocol minimization machine**, the protocol is declared in the main module. Resolve the type representation for the underlying type in structural resolution stage.
   - If this is a **protocol machine**, we already have a requirement signature for the protocol. Look up the underlying type for `A` in the protocol's requirement signature.
2. If the underlying type of `A` is some concrete type `X`, use Algorithm 17C to build a concrete type symbol, and return this pair representing a property rule:

   $$([\mathtt{P}] \cdot \mathtt{A} \cdot [\mathsf{concrete}\colon \mathtt{X}; \ldots], [\mathtt{P}] \cdot \mathtt{A})$$

3. If the underlying type of `A` is some other type parameter `Self.U`, use Algorithm 17H to translate it into a term, and return this pair representing a same-type rule:

   $$([\mathtt{P}] \cdot \mathtt{A}, \mathsf{term}_{\mathtt{P}}(\mathtt{Self.U}))$$

Example 17.26. The `Graph` protocol below declares two type aliases:

```
protocol Pair {
  associatedtype Elt
}

protocol Graph {
  associatedtype Edge: Pair, Hashable

  typealias Vertex  = Self.Edge.Elt   // Type parameter
  typealias EdgeSet = Set<Self.Edge>  // Concrete type
}
```

We will make use of the fact that the protocol minimization machine for `Graph` contains associated type rules for `Elt` and `Edge`, as well as a rule for the associated conformance requirement $[\texttt{Self.Edge: Pair}]_{\texttt{Graph}}$:

$$[\texttt{Pair}] \cdot \texttt{Elt} \sim [\texttt{Pair}|\texttt{Elt}] \tag{1}$$

$$[\texttt{Graph}] \cdot \texttt{Edge} \sim [\texttt{Graph}|\texttt{Edge}] \tag{2}$$

$$[\texttt{Graph}] \cdot \texttt{Edge} \cdot [\texttt{Pair}] \sim [\texttt{Graph}] \cdot \texttt{Edge} \tag{3}$$

To get the protocol type alias rules for `Vertex` and `EdgeSet`, we first resolve their underlying types and write down the imaginary associated same-type requirement for each:

$$[\texttt{Self.Vertex == Self.Edge.Elt}]_{\texttt{Graph}}$$

$$[\texttt{Self.EdgeSet == Array<Self.Edge>}]_{\texttt{Graph}}$$

If we translate these requirements into rewrite rules we see that we get a same-type rule for `Vertex`, and a property rule for `EdgeSet`:

$$[\texttt{Graph}] \cdot \texttt{Vertex} \sim [\texttt{Graph}] \cdot \texttt{Edge} \cdot \texttt{Elt} \tag{4}$$

$$[\texttt{Graph}] \cdot \texttt{EdgeSet} \cdot [\texttt{concrete: Set<}\tau\texttt{_0_0>}; [\texttt{Graph}] \cdot \texttt{Edge}] \sim [\texttt{Graph}] \cdot \texttt{EdgeSet} \tag{5}$$

Next, we're going to see what happens to rule (4). Completion makes use of (2) to simplify rule (3), replacing it with ($\diamond 3$):

$$[\texttt{Graph}|\texttt{Edge}] \cdot [\texttt{Pair}] \sim [\texttt{Graph}|\texttt{Edge}] \tag{$\diamond 3$}$$

The left-hand side of rule (4) is the term $[\texttt{Graph}] \cdot \texttt{Edge} \cdot \texttt{Elt}$. This term is equivalent to $[\texttt{Graph}|\texttt{Edge}] \cdot [\texttt{Pair}|\texttt{Elt}]$ via this rewrite path using rules (1), (2) and ($\diamond 3$):

$$([\texttt{Graph}] \cdot \texttt{Edge} \Rightarrow [\texttt{Graph}|\texttt{Edge}]) \triangleright \texttt{Elt}$$
$$\circ ([\texttt{Graph}|\texttt{Edge}] \Rightarrow [\texttt{Graph}|\texttt{Edge}] \cdot [\texttt{Pair}]) \triangleright \texttt{Elt}$$
$$\circ [\texttt{Graph}|\texttt{Edge}] \triangleleft ([\texttt{Pair}] \cdot \texttt{Elt} \Rightarrow [\texttt{Pair}|\texttt{Elt}])$$

Completion discovers this fact and uses it to simplify rule (4), replacing it with ($\diamond$4):

$$[\texttt{Graph}] \cdot \texttt{Vertex} \sim [\texttt{Graph}|\texttt{Edge}] \cdot [\texttt{Pair}|\texttt{Elt}] \qquad (\diamond 4)$$

The name symbol `Vertex` remains on the left-hand side, and cannot be eliminated. The right-hand side translates to the bound type parameter `Self.[Graph]Edge.[Pair]Elt`. This becomes the underlying type we serialize as part of the protocol's requirement signature. If we subsequently build a protocol machine for `Graph`, we immediately get the rewrite rule ($\diamond$4), and not (4).

Example 17.27. We can now state a declaration with a `where` clause that refers to our type alias `Vertex`:

```
func walk<E, G: Graph>(_: G, _: (G.Vertex) -> E)
    where G.Vertex: Equatable {}
```

The derived requirements formalism does not explain protocol type aliases, and the subject type of the conformance requirement [τ_0_1.`Vertex: Equatable`] is not a consequence of the theory; there is no associated type named `Vertex`. However, in the implementation, we will see that we get the same generic signature as if we wrote the requirement as [τ_0_1.`Edge.Elt: Equatable`]:

```
func walk<E, G: Graph>(_: G, _: (G.Vertex) -> E)
    where G.Edge.Elt: Equatable {}
```

The minimization machine for `walk()` imports rules from the protocol machine `Graph`, including rule ($\diamond$4). We also add two local rules, by resolving the requirements of `walk()`:

$$\tau_0_1 \cdot [\texttt{Graph}] \sim \tau_0_1 \qquad (6)$$

$$\tau_0_1 \cdot \texttt{Vertex} \cdot [\texttt{Equatable}] \sim \tau_0_1 \cdot \texttt{Vertex} \qquad (7)$$

Notice that τ_0_1 $\cdot$ `Vertex` is equivalent to τ_0_1 $\cdot$ [`Graph`|`Edge`] $\cdot$ [`Pair`|`Elt`]:

$$(\tau_0_1 \Rightarrow \tau_0_1 \cdot [\texttt{Graph}]) \triangleright \texttt{Vertex}$$
$$\circ\, \tau_0_1 \triangleleft ([\texttt{Graph}] \cdot \texttt{Vertex} \Rightarrow [\texttt{Graph}|\texttt{Edge}] \cdot [\texttt{Pair}|\texttt{Elt}])$$

The reduction order in Algorithm 17I now becomes important. Essentially, we want the normal form algorithm to eliminate name symbols "as much as possible" when reducing a term. This is why we used a weighted shortlex order, to compare the number of name symbols in both terms *before* comparing length; that is:

$$\tau_0_1 \cdot [\texttt{Graph}|\texttt{Edge}] \cdot [\texttt{Pair}|\texttt{Elt}] < \tau_0_1 \cdot \texttt{Vertex}$$

This must be so in our example, because the unbound type parameter `τ_0_1.Vertex` is shorter than any bound type parameter in its equivalence class. In fact, the reduced type parameter here is `τ_0_1.[Graph]Edge.[Pair]Elt`. (In the derived requirements formalism, which does not describe protocol type aliases, this cannot happen, as we showed in Theorem 11.9.)

All of this is to say, that completion simplifies rule (7) and replaces it with ($\diamond$7):

$$\texttt{τ_0_1} \cdot [\texttt{Graph}|\texttt{Edge}] \cdot [\texttt{Pair}|\texttt{Elt}] \cdot [\texttt{Equatable}] \sim \texttt{τ_0_1} \cdot [\texttt{Graph}|\texttt{Edge}] \cdot [\texttt{Pair}|\texttt{Elt}] \quad (\diamond 7)$$

We translate this rule into this conformance requirement, which becomes part of the final generic signature:

$$[\texttt{τ_0_1.[Graph]Edge.[Pair]Elt: Equatable}]$$

Example 17.28. The `GenericSignatureBuilder` would lazily intertwine the processing of requirements with name lookup, and a `where` clause could refer to a type alias declared in a protocol extension without restriction. On the other hand, the Requirement Machine collects all of the rewrite rules for a protocol upfront. To avoid an expensive walk over all members of all protocol extensions while collecting rewrite rules, only type aliases declared directly inside the protocol participate in rewriting today.

This means that a type alias in a protocol extension can only be found when a type representation is resolved in the interface resolution stage. In the structural resolution stage, we instead get an unbound type parameter that is not equivalent to anything else, because no rewrite rule exists to reduce it further. In particular, it is not valid to refer to such a type alias from a `where` clause.

If we change our previous example to refer to type alias in a protocol extension, we see the compiler diagnoses an error:

```
extension Graph {
  typealias Bad = Self.Edge.Elt
}

func walk<G: Graph>(_: G) where G.Bad: Equatable {}
// error: 'Bad' was defined in extension of protocol 'Graph'
// and cannot be referenced from a 'where' clause
```

We diagnose the error via the same mechanism as for rejecting requirements which are not well-formed (Section 11.3). Recall that once we have a generic signature, we revisit each type representation in the `where` clause, and resolve it again in the interface resolution stage, to diagnose any invalid type parameters. Now, we see there is another condition to check here—if the type representation does resolve, but the resolved type declaration was a type alias in a protocol extension, we must reject the program.

Example 17.29. Another `GenericSignatureBuilder` oddity is that a protocol type alias could appear on the *right-hand side* of a conformance requirement. The Requirement Machine implements a narrow form of this for backward compatibility.

Specifically, if a protocol declares a protocol type alias `A` whose underlying type is a constraint type, we allow the dependent member type `Self.A` to appear on the right-hand side of a conformance requirement, either in the protocol itself or an extension. More general type parameters cannot appear. For example:

```
class Box {}

protocol Holder {
  typealias Wrapper = Box & Equatable
  associatedtype Contents: Self.Wrapper
}
```

We handle this in requirement decomposition (Section 11.2). In the above, the user-written requirement is $[\texttt{Self.Contents: Self.Wrapper}]_{\texttt{Holder}}$. We look up the protocol type alias `Wrapper` appearing on the right-hand side, and then proceed as if the user wrote `Box & Equatable` instead. This decomposes to yield $[\texttt{Self.Contents: Box}]_{\texttt{Holder}}$ and $[\texttt{Self.Contents: Equatable}]_{\texttt{Holder}}$.

The next algorithm for building the local rules of a protocol ties everything together.

Algorithm 17M (Build protocol rules)**.** Takes a protocol declaration `P` as input, and outputs a list of rules, represented as ordered pairs of terms.

1. Let R be a list of ordered pairs of terms, initially empty.
2. Add the identity conformance rule $[\texttt{P}] \cdot [\texttt{P}] \sim [\texttt{P}]$.
3. For each associated type `A` of `P`, add the associated type rule $[\texttt{P}] \cdot \texttt{A} \sim [\texttt{P}|\texttt{A}]$.
4. For each associated type `A` of each protocol `Q` inherited by `P` (or in other words, such that $G_{\texttt{P}} \vdash [\texttt{Self: Q}]$), add the associated type rule $[\texttt{P}] \cdot \texttt{A} \sim [\texttt{P}|\texttt{A}]$.
5. For each associated requirement of `P`, apply Algorithm 17K and add the result.
6. For each type alias declared in `P`, apply Algorithm 17L and add the result.
7. Return R.

We've completely described how we get the initial list of rewrite rules for each of the four kinds of requirement machine. We always start by collecting imported rules with Algorithm 15B. Then, to get the local rules in a query or minimization machine, we apply Algorithm 17J to each explicit requirement of our generic signature. To get the local rules in a protocol or protocol minimization machine, we apply Algorithm 17M to each protocol in the component.

The rule array. A requirement machine stores its rewrite rules in an array. Each rule is a pair of immutable terms (u, v). Rules are oriented, so $u > v$ in the reduction order, to guarantee termination of the normal form algorithm. We always insert new rules at the end of this array, so that we can refer to rules by index from other data structures. The flags associated with each rule are used by completion and minimization:

- The **permanent** flag is set if the rule is an identity conformance or an associated type rule. Permanent rules are not considered for rule minimization.
- The **left-simplified**, **right-simplified** and **substitution-simplified** flags are set when a rule is replaced by a simpler rule (Section 18.1).
- The **conflicting** and **recursive** flags are set during property map construction if the rule represents a conflicting or recursive requirement (Chapter 19).
- The **explicit** flag is set on the initial rules created from user-written requirements. Rule minimization also propagates this flag to other rules in a specific way. Explicit rules have a special behavior with the minimal conformances algorithm (Section 20.3).
- The **redundant** flag is set by rule minimization if the rewrite rule is a consequence of other non-redundant rules. When we build requirements from rules at the end of the minimization process, we only consider those rules not marked as redundant (Chapter 20.4).
- The **frozen** flag prevents any of the other flags from being set. Imported rules are always marked frozen. Local rules are marked frozen after the construction of the requirement machine is complete.

Once a flag has been set, it cannot be cleared, and the frozen flag prevents any more flags from being set.

17.6. The Normal Form Algorithm

In this section, we revisit the normal form algorithm of Section 16.5. Among other things, we can use this algorithm to decide if two type parameters are equivalent, which describes the implementation of the `areReducedTypeParametersEqual`() generic signature query. Our goal will be to address two shortcomings of the specification in Algorithm 16A:

1. If the original term contains more than one subterm equal to the left-hand side of a rewrite rule, we did not specify which rewrite step to take.
2. If there are a large number of rewrite rules, representing them as a list of pairs is inefficient, because we find ourselves comparing each subterm of our original term with the left-hand side of every rewrite rule.

Regarding the first issue, we know that completion gives us a confluent reduction relation, where the normal form of a term does not depend on any choice of rewrite steps. As a practical matter though, we still something deterministic. We do this by always choosing the rewrite step with the shortest left whisker; in other words, we scan the original term from left to right and rewrite the *left-most* matching subterm, if there is one, and repeat. To resolve the second issue, we must *find* these matching subterms efficiently. We do this with the aid of an auxiliary data structure.

The rule trie. A *trie* represents a mapping from keys to values, where the keys are strings (so, elements of a free monoid). The lookup operation finds the shortest *prefix* of an input string that is equal to some key, in linear time proportional to the length of the key. Each requirement machine has its own trie where the keys are the left-hand sides of rewrite rules, and the values are the indices of those rules in the array.

Concretely, a trie is a collection of *nodes*, organized as a tree. Each node represents a unique string that is a prefix of one or more keys stored in the trie. We denote this for a node n by $\texttt{KEY}(n)$. If $\texttt{KEY}(n)$ is the *entire* key, then n stores the value associated with this key, denoted $\texttt{VALUE}(n)$. Otherwise, $\texttt{VALUE}(n)$ is null.

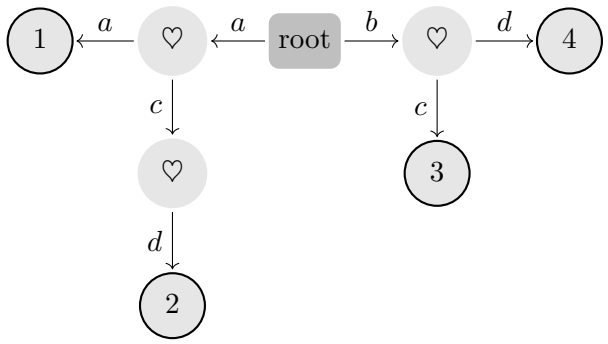
The empty string ε is a prefix of every string, so a non-empty trie always has a unique node r such that $\texttt{KEY}(r) = \varepsilon$. This is the *root* of the trie. In our application, $\texttt{VALUE}(r)$ is always null, because the left-hand side of a rewrite rule cannot be empty.

Finally, if m and n are two nodes such that $\texttt{KEY}(m) = \texttt{KEY}(n) \cdot s$ for some symbol s, we say that m is a *child* of n, or $\texttt{CHILD}(n, s) = m$. Each node stores a hashtable mapping symbols to children, and we visualize this by joining the parent with each child by an edge, labeled with the corresponding symbol. In fact, we don't need to store $\texttt{KEY}(n)$ inside n at all; the set of keys is implicitly encoded within each $\texttt{CHILD}(n, s)$.

Figure 17.2 lists four abstract rewrite rules over the alphabet $\{a, b, c, d\}$, and shows the trie we obtain by following the above description.

Figure 17.2.: Some rewrite rules, and the trie generated by their left-hand sides

$aa \sim a$ (1)
$acd \sim b$ (2)
$bc \sim b$ (3)
$bd \sim b$ (4)



To implement the trie lookup operation, we start from the root and follow the sequence of successive edges labeled by the symbols we observe while reading our term from left to right. If we reach a node with a value, we're done. On the other hand, if we either reach the end of the term, or see that a node does not have a child for the next symbol, we fail the lookup operation without returning a result. Also, we need to look for matching subterms at every position, and not just at the start of our term, so the lookup takes the index of the first symbol where the lookup begins.

Algorithm 17N (Lookup in rule trie)**.** Takes a term t and an offset i such that $0 \leq i < |t|$. If there exists a term x with $|x| = i$ and a rewrite rule (u, v) such that $t = xuy$, returns the index of this (u, v) in the array of rules. Otherwise, returns null.

1. (Initialize) Let n be the root node.
2. (End) If $i = |t|$, we've reached the end of the term. Return null.
3. (Child) Let $m := \texttt{CHILD}(n, s_i)$ where s_i is the ith symbol of t. If m is null, there are no more keys left to check. Return null.
4. (Value) Otherwise, if $\texttt{VALUE}(m)$ is not null, return $\texttt{VALUE}(m)$.
5. (Advance) Otherwise, set $n \leftarrow m$ and $i \leftarrow i + 1$. Go back to Step 2.

For example, consider the term *bacdc* and the trie shown in Figure 17.2. A lookup at position 0 will fail; we follow the edge b from the root, then see that this child does not store a value, nor does it have an outgoing edge for a. On the other hand, a lookup at position 1 will succeed; we follow the sequence of edges labeled a, c, and d, and arrive at the node whose value is rule (2). To implement the normal form algorithm then, we attempt this lookup at each position, apply the rewrite step upon finding a match, and iterate this until fixed point.

Algorithm 17O (Normal form algorithm using rule trie)**.** Takes a mutable term t as input. Mutates t, and outputs a positive rewrite path p where $\mathrm{src}(p)$ is the original term and $\mathrm{dst}(p)$ is its normal form.

1. (Initialize) Let $p := 1_t$. Let $i := 0$.
2. (Check) If $i = |t|$, the term t is now irreducible. Return p.
3. (Lookup) Invoke Algorithm 17N with t and i.
4. (Rewrite) If we found a rewrite rule (u, v) such that $t = xuy$ and $|x| = i$, then set $t \leftarrow xvy$, set $p \leftarrow p \circ x(u \Rightarrow v)y$, set $i \leftarrow 0$, and go back to Step 2.
5. (Next) Otherwise, set $i \leftarrow i + 1$, and go back to Step 2.

After adding a new rewrite rule to the array of rules, we must update the rule trie. If the new rule is (u, v), we consider the left-hand side u. We again begin at the root and follow successive edges labeled by each symbol, but now, we create new nodes if necessary. Once we reach the node for u, we store the new rule's index in this node.

Algorithm 17P (Insert rule in rule trie)**.** Takes the index of a rewrite rule (u, v) as input. Has side effects.

1. (Initialize) Let n be the root node of the trie. Let $i := 0$.
2. (End) If $i = |u|$, update $\texttt{VALUE}(n)$ with the index of (u, v), and return.
3. (Child) Let s_i be the ith symbol of u. Let $m := \texttt{CHILD}(n, s_i)$. If m is null, allocate a new node m and set $\texttt{CHILD}(n, s_i) \leftarrow m$.
4. (Next) Set $n \leftarrow m$. Set $i \leftarrow i + 1$. Go back to Step 2.

Once we collect the rewrite rules for a requirement machine, we use the below algorithm to record them. We first apply the normal form algorithm to reduce both sides with all rules added so far. If both sides already have the same normal form, we discard the rule. Otherwise, we orient the pair with the reduction order before adding it to the array and updating the trie. Note that completion also adds new rewrite rules, but with a slightly extended procedure, which we describe in Algorithm 18B.

Algorithm 17Q (Record rule)**.** Takes a pair of terms (u, v) as input. Returns a flag indicating if a new rule was actually recorded. Has side effects.

1. (Reduce) Apply Algorithm 17O to u and v to obtain $\tilde{u}$ and $\tilde{v}$.
2. (Trivial) If $\tilde{u} = \tilde{v}$, then we already have $u \sim v$. Return false.
3. (Compare) Otherwise, compare $\tilde{u}$ with $\tilde{v}$ using Algorithm 17I.
4. (Error) If $\tilde{u}$ and $\tilde{v}$ are incomparable, signal an error.
5. (Orient) If $\tilde{v} > \tilde{u}$, swap $\tilde{u}$ and $\tilde{v}$.
6. (Record) We now have $\tilde{u} > \tilde{v}$. Add $(\tilde{u}, \tilde{v})$ to the array of rules.
7. (Trie) Invoke Algorithm 17P to update the rule trie. Return true.

Note that the error in Step 4 cannot happen in our implementation, because of how we defined "rule" and "$\mathsf{rule_P}$". Indeed, if two terms are incomparable, they must have the same length, and an occurrence of a superclass, concrete type, or concrete conformance symbol at the same position. However, these symbols only appear in rules of the form $(t \cdot s, t)$, so we always have $|t \cdot s| > |t|$ and thus $t \cdot s > t$.

Rewrite paths. Algorithm 17O outputs a rewrite path describing the reduction that was performed. We now discuss the representation of rewrite paths and their rewrite steps in the implementation. While a rewrite step $x(u \Rightarrow v)y$ can be stored as a tuple of four terms (x, u, v, y) as per Definition 16.14, we use a more compact encoding.

First, either (u, v) or (v, u) is an existing rewrite rule, depending on whether this step is positive or negative. We place all rewrite rules in an array, so a rewrite step can store the index of its rewrite rule together with a "+" or "−" flag to specify the direction.

Furthermore, we only need to know the lengths of the whiskers x and y, and not the terms themselves. To see why, suppose that $s := x(u \Rightarrow v)y$, and we are given $\mathrm{src}(s) = xuy$. We can recover x and y from $\mathrm{src}(s)$ if we know $|x|$ and $|y|$. We can then use x and y to construct the term $\mathrm{dst}(s) = xvy$.

By induction, if we are given a rewrite path p represented in this way together with the term $\mathrm{src}(p)$, we can recover the left and right whisker, and thus the source and destination, of each rewrite step in p. Finally, we get $\mathrm{dst}(p)$.

Thus, we encode a rewrite step as three integers together with a boolean flag, while a rewrite path consists of the initial term together with a list of rewrite steps encoded in this manner. The *rewrite path evaluator* recovers the intermediate term at each step, as described above. This is used to print rewrite paths in debugging output.

More about tries. Algorithm 17N finds the shortest prefix of the input string that matches some key in the trie. In particular, if our trie contains two keys where one is a prefix of another, a lookup will only ever find the shorter key. We cannot simply discard the longer key though, and we will see that completion performs a different kind of trie lookup to find *all* matching prefixes, in Algorithm 18C. Finally, in Chapter 19, we will see that to implement the property map data structure, we use a different trie to find the *longest suffix* of the input string that matches a certain key. To learn more about tries, see Section 6.3 of [171].

A possible optimization. Searching an input string for occurrences of a fixed list of substrings is a common problem in programming, and our approach of using a trie is a typical solution. Our measurements show that the normal form algorithm makes up a negligible portion of total compile time, so further optimization seems unnecessary. However, in applications of this problem where the input string is very long, the paper by Alfred V. Aho and Margaret J. Corasick [172] presents a further improvement.

In our normal form algorithm, we check for a matching subterm at every position of the input term. If we're currently at some position i, we might perform a lookup that traverses some number of nodes in the trie, say j, before failing at position $i + j$. At this point, we start a new lookup from the root node, back at position $i + 1$ in the original term. Let's call this integer j the *level* of a trie node. When lookup fails at a node with level $j > 1$, we are forced to revisit symbols that we saw already.

The key idea in the Aho-Corasick algorithm is to augment the trie, which they call the *goto graph*, with an additional data structure describing the *failure function*. When a node n does not have a child for the next symbol s, the failure function sends us to some other node at the previous level, that may not be the root node. We can then repeat the lookup of the *same* symbol s at the new node. If we reach the root node, there cannot be a match, so we advance to the next symbol, and we never go back.

The tradeoff with this approach is the time taken to compute the failure function, work which would need to be redone when the rule trie changes. However, the necessary updates can be done incrementally; see [173].

17.7. Source Code Reference

Symbols

Key source files:

- `lib/AST/RequirementMachine/Symbol.h`
- `lib/AST/RequirementMachine/Symbol.cpp`

`rewriting::Symbol::Kind` *enum class*

The symbol kind. The order of the cases is significant; it is from the reduction order on symbols (Algorithm 17F).

- `ConcreteConformance`
- `Protocol`
- `AssociatedType`
- `GenericParam`
- `Layout`
- `Superclass`
- `ConcreteType`

`rewriting::Symbol` *class*

Represents an immutable, uniqued symbol. Instances are passed by value. The value wraps a single pointer to internal storage owned by the `RewriteContext`. Symbols are logically variant types, but C++ does not directly support that concept so there is a bit of boilerplate in their definition.

Building symbols. Instances of symbols are obtained via calls to a set of static factory methods which take the structural components and the `RewriteContext`:

- `forName()` takes an `Identifier`.
- `forProtocol()` takes a `ProtocolDecl *`.
- `forAssociatedType()` takes a `ProtocolDecl *` and `Identifier`.
- `forGenericParam()` takes a canonical `GenericTypeParamType *`.
- `forLayout()` takes a `LayoutConstraint`.
- `forSuperclass()` takes a pattern type and a list of substitution terms.
- `forConcreteType()` takes a pattern type and a list of substitution terms.
- `forConcreteConformance()` takes a pattern type, a list of substitution terms, and a `ProtocolDecl *`.

The last three methods take the pattern type as a `CanType` and the substitution terms as an `ArrayRef<CanType>`. The `RewriteContext::getSubstitutionSchemaFromType()` method takes an arbitrary concrete `Type` and builds the pattern type and substitution terms. Note that the pattern type is always a canonical type, so the Requirement Machine does not preserve type sugar in requirements when building a generic signature, for example.

Structural components. Various instance methods take symbols apart:

- `getKind()` returns the `Symbol::Kind`. This determines which of the remaining accessors may be called.
- `getName()` returns the `Identifier` stored in a name or associated type symbol.
- `getProtocol()` returns the `ProtocolDecl *` stored in a protocol, associated type, or concrete conformance symbol.
- `getGenericParam()` returns the `GenericTypeParamDecl *` stored in a generic parameter symbol.
- `getLayoutConstraint()` returns the `LayoutConstraint` stored in a layout symbol.
- `getConcreteType()` returns the pattern type stored in a superclass, concrete type or concrete conformance symbol.

- `getSubstitutions()` returns the substitution terms stored in a superclass, concrete type, or concrete conformance symbol.
- `getRootProtocol()` returns `nullptr` if this is a generic parameter symbol, or a protocol declaration if this is a protocol or associated type symbol. For any other symbol kind, signals an assertion.

Comparing symbols:

- `operator==` compares two symbols for equality.
- `compare()` is the symbol reduction order (Algorithm 17F). The return type of `std::optional<int>` translates into our notation as follows:

$\perp$	`std::nullopt`
$=$	`std::optional(0)`
$<$	`std::optional(-1)`
$>$	`std::optional(1)`

Debugging:

- `dump()` prints the symbol in a way that resembles our typeset notation.

`rewriting::RewriteContext` *class*

See also Section 15.3.

- `compareProtocols()` implements Algorithm 17A.
- `getGenericParamIndex()` takes a `Type` that must be a `GenericTypeParamType` with depth 0, and returns its index. This is a very specific operation, but it comes up frequently when working with the pattern type of a superclass, concrete type, or concrete conformance symbol.
- `getSubstitutionSchemaFromType()` takes an arbitrary concrete `Type` and builds the pattern type and substitution terms, used when forming a concrete type, superclass, or concrete conformance symbol. This implements Algorithm 17C.

Terms

Key source files:

- `lib/AST/RequirementMachine/Term.h`
- `lib/AST/RequirementMachine/Term.cpp`
- `lib/AST/RequirementMachine/RewriteContext.h`
- `lib/AST/RequirementMachine/InterfaceType.cpp`

Common operations. Both mutable and immutable terms implement the following subset of STL container operations:

- `size()` returns the length of the term.
- `operator[]` returns a specific element.
- `begin()` and `end()` return a pair of iterators over the `Symbol` elements of the term.
- `rbegin()` and `rend()` return a pair of iterators for reverse order iteration.
- `front()` returns the first `Symbol` in the term.
- `back()` returns the last `Symbol` in the term.

Both also provide a utility method that is useful when working with terms that represent type parameters:

- `getRootProtocol()` returns `nullptr` if the first symbol is a generic parameter symbol, otherwise returns the protocol of the first protocol or associated type symbol. Asserts if the first symbol has any other kind.

`rewriting::MutableTerm` *class*

A mutable term. Instances are passed by value or by reference, as appropriate. Each value owns a heap-allocated buffer for storing the symbols it contains, so copying mutable terms is somewhat expensive. The default constructor creates an empty `MutableTerm`. The remaining constructors initialize the `MutableTerm` from an initial list of symbols, specified as an iterator pair, an `ArrayRef`, or an immutable `Term`.

This class implements all the common methods listed above for looking at the symbols of a term, as well as a few more:

- `empty()` checks if this mutable term is empty.
- `add()` adds a single `Symbol` at the end of this term.
- `append()` appends another `Term` or `MutableTerm` at the end of this term. This is the free monoid operation.
- `rewriteSubTerm()` replaces a subterm of this term with another term. The subterm to replace is specified by a pair of iterators, which must be valid iterators into this term. The replacement term may be shorter or longer than the subterm, and the underlying storage of this term is resized as appropriate. This operation is the key step in Algorithm 17O.

`rewriting::Term` *class*

An immutable term. Instances are passed by value. Each value wraps a single pointer to internal storage owned by the `RewriteContext`. Immutable terms are created by a static factory method on `Term`:

- `get()` creates a `Term` from a `MutableTerm`, which must be non-empty.

This class implements all the common methods listed above for looking at the symbols of a term, and one more:

- `containsNameSymbols()` returns true if the term contains name symbols.

Comparing terms:

- `operator==` compares immutable terms for equality. This is implemented by comparing their pointers for identity.
- `compare()` implements the term reduction order (Algorithm 17I). The return type of `std::optional<int>` encodes the result just like `Symbol::compare()`.

Debugging:

- `dump()` prints the symbols in a term joined with ".".

`rewriting::RewriteContext` *class*

See also Section 15.3.

- `getMutableTermForType()` translates a `Type` containing a type parameter into a `MutableTerm`. This method implements both Algorithm 17G and Algorithm 17H; the second parameter is a `ProtocolDecl *`, which may be null.

Rules

Key source files:

- `lib/AST/RequirementMachine/RuleBuilder.h`
- `lib/AST/RequirementMachine/RuleBuilder.cpp`
- `lib/AST/RequirementMachine/Rule.h`
- `lib/AST/RequirementMachine/Rule.cpp`
- `lib/AST/RequirementMachine/RewriteSystem.h`

- `lib/AST/RequirementMachine/RewriteSystem.cpp`
- `lib/AST/RequirementMachine/Trie.h`

`rewriting::Rule` *class*

A rewrite rule, represented by a pair of immutable terms and some flags. The rule is always oriented, so that the right-hand side is smaller than the left-hand side under the reduction order on terms. A pair of accessor methods take a rule apart into two immutable terms:

- `getLHS()` returns the left-hand side.
- `getRHS()` returns the right-hand side.

Some utility methods for recognizing common kinds of rules:

- `isPropertyRule()` checks if this is a property rule; that is, $(t \cdot s,\ t)$ where s is a property symbol. If so, returns s, otherwise `std::nullopt`.
- `isIdentityConformanceRule()` checks if this is an identity conformance rule; that is, $([\mathtt{P}] \cdot [\mathtt{P}],\ [\mathtt{P}])$.
- `isProtocolConformanceRule()` checks if this is a protocol conformance rule; that is, $(t \cdot [\mathtt{P}],\ t)$, or in other words a property rule with a protocol symbol.
- `isAnyConformanceRule()` checks if this is a protocol conformance rule, or a property rule with a concrete conformance symbol.
- `isProtocolTypeAliasRule()` checks if this rule looks like a protocol type alias rule; that is, either a same-type rule where the left-hand side is $[\mathtt{P}] \cdot \mathtt{A}$ for some protocol $[\mathtt{P}]$ and name symbol `A`, or a property rule where the right-hand side is $[\mathtt{P}] \cdot \mathtt{A}$, and the property symbol is a concrete type symbol.
- `containsNameSymbols()` returns true if either side of the rewrite rule contains name symbols.

We defined the flags in Section 17.5:

- `isPermanent()` and `markPermanent()` test and set the **permanent** flag.
- `isExplicit()` and `markExplicit()` test and set the **explicit** flag.
- `isLHSSimplified()` and `markLHSSimplified()` test and set the **left-simplified** flag.

- `isRHSSimplified()` and `markRHSSimplified()` test and set the **right-simplified** flag.
- `isSubstitutionSimplified()` and `markSubstitutionSimplified()` test and set the **substitution-simplified** flag.
- `isConfliciting()` and `markConflicting()` test and set the **conflicting** flag.
- `isRecursive()` and `markRecursive()` test and set the **recursive** flag.
- `isFrozen()` and `freeze()` test and set the **frozen** flag.
- `isRedundant()` and `markRedundant()` test and set the **redundant** flag.

Debugging:

- `dump()` prints the rule in a way that resembles our typeset notation.

`rewriting::RewriteSystem` *class*

A list of rewrite rules in a monoid presentation together with some additional state. Every `RequirementMachine` has a `RewriteSystem`. See also Section 18.6.

- `initialize()` adds the initial list of imported and local rules.
- `simplify()` computes the normal form of a term using Algorithm 17O. Modifies the given `MutableTerm` in place, and optionally outputs a `RewritePath` from the original term to the reduced term.
- `addRule()` reduces and orients both sides, and records a new local rule. Returns `false` if both sides have the same normal form, in which case no rule is added. This implements both Algorithm 17Q, and Algorithm 18B from the next chapter.
- `addPermanentRule()` attempts to record a new rewrite rule, and immediately marks it **permanent** if a rule was added.
- `addExplicitRule()` attempts to record a new rewrite rule, and immediately marks it **explicit** if a rule was added.
- `getRules()` returns an array of all rules in this requirement machine.
- `getLocalRules()` returns an array of all local rules in this requirement machine.
- `getRule()` returns the rule at the given index.
- `dump()` prints out all rewrite rules, rewrite loops, and a few other things stored in this `RewriteSystem`.

`rewriting::RuleBuilder` *class*

A utility class to collect the imported and local rules of a requirement machine being constructed. The following methods are called from the `RequirementMachine` methods with the same name:

- `initWithGenericSignature()` builds rewrite rules for a query machine.
- `initWithWrittenRequirements()` builds rewrite rules for a minimization machine.
- `initWithProtocolSignatureRequirements()` builds rewrite rules for a protocol machine.
- `initWithProtocolWrittenRequirements()` builds rewrite rules for a protocol minimization machine.

The above share some common logic:

- `collectRulesFromReferencedProtocols()` implements Algorithm 15B from the previous chapter.
- `addRequirement()` implements Algorithm 17J and Algorithm 17K.
- `addTypeAlias()` implements Algorithm 17L.
- `addPermanentProtocolRules()` implements Algorithm 17M.

`rewriting::Trie` *template class*

A template class implementing the trie data structure. Used by the `RewriteSystem` and `PropertyMap`. The keys are terms, while the value type is a template parameter. Another template parameter selects between a shortest or longest match lookup strategy. We use shortest match for the normal form algorithm. See also Section 18.6 and Section 19.7.

- `find()` finds an existing entry with Algorithm 17N.
- `insert()` inserts a new entry with Algorithm 17P.

Rewrite Steps

Key source files:

- `lib/AST/RequirementMachine/RewriteLoop.h`
- `lib/AST/RequirementMachine/RewriteLoop.cpp`

See also Section 18.6.

`rewriting::RewriteStep::Kind` *enum*

The `RewriteStep::Kind::Rule` kind represents the application of a rewrite rule to a term. Other rewrite step kinds appear in Section 19.7.

`rewriting::RewriteStep` *struct*

A compact encoding of a rewrite step.

- `Kind` is a `RewriteStep::Kind`.
- `Inverse` is flag that encodes whether this is a positive rewrite step, or a negative rewrite step. For `Rule`, this specifies whether $(u, v) \in R$ or $(v, u) \in R$.
- `StartOffset` is the length of the left whisker $|x|$.
- `EndOffset` is the length of the left whisker $|y|$.
- `Arg` depends on kind; for `Rule`, it is the index of the rewrite rule being applied by this step in the `RewriteSystem`'s array of rules.
- `getRuleID()` is a getter method that returns the above after asserting that the kind is `Rule`.

`rewriting::RewritePath` *class*

A rewrite path, stored as a `std::vector` of `RewriteStep` elements.

- `empty()` checks if this is the empty rewrite path 1_t for some term t.
- `size()` returns the number of rewrite steps in the path.
- `begin()` and `end()` return a pair of iterators for iterating over the rewrite steps in the path.
- `dump()` prints the rewrite path using a notation that resembles what we use in this book; the correct source `Term` and `RewriteSystem` must be provided.

`rewriting::RewritePathEvaluator` *class*

A utility class to evaluate a rewrite path starting from a source term. This covers each intermediate term in the path from the compact encoding described above. Among other things, this is used to implement `RewritePath::dump()`.

18. Completion

Knuth-Bendix completion is the central algorithm in the Requirement Machine. Completion attempts to construct a convergent rewriting system from a list of rewrite rules, and a convergent rewriting system allows us to decide if two terms have the same normal form in a finite number of steps, solving the word problem. As we saw in the previous chapter, our initial rewrite rules are defined by the explicit requirements of a generic signature and its protocol dependencies. A desirable property of this mapping was given by a pair of theorems in Section 17.1: a *derived* requirement defines a rewrite path over these rewrite rules representing explicit requirements. All of this means that completion gives us a *decision procedure* for the derived requirements formalism: the question of whether any given derived requirement is satisfied—that is, if there exists a valid derivation built from explicit requirements—is easily solved by term reduction in a convergent rewriting system. This is the foundation on which we build both generic signature queries and minimization.

The algorithm. We'll give a self-contained description first, with much of the rest of the chapter devoted to examples. Our description can be supplemented with any text on the theory of rewriting, such as [154] or [155]. The algorithm is somewhat clever; to really "get it" might require several attempts. Donald E. Knuth and Peter Bendix described the algorithm for term rewriting systems in a 1970 paper [174]; a correctness proof was later given by Gérard Huet in [175]. In our application, the terms are elements of a free monoid, so we have a string rewriting system; this special case was studied in [176]. A survey of related techniques appears in [177].

The entry point into the Knuth-Bendix completion procedure is Algorithm 18E, but we break off four smaller pieces before we get there, so that only the top-level loop remains:

- Algorithm 18C finds all rules that overlap with a fixed rule at a fixed position.
- Algorithm 18D finds all pairs of rules that overlap at any position.
- Algorithm 18A builds a critical pair from a pair of overlapping rules.
- Algorithm 18B resolves a critical pair.

We begin with Algorithms 18A and 18B, proceeding from the inside out. The twin concepts of overlapping rule and critical pair are fundamental to the algorithm, and they provide the theoretical justification for the rest.

Local confluence. We would like our reduction relation $\to$ to satisfy the Church-Rosser property: if $x \sim y$ are two equivalent terms, then $x \to z$ and $y \to z$ for some term z. By Theorem 16.44, this is equivalent to $\to$ being confluent, meaning any two positive rewrite paths diverging from a common source can be extended to meet each other. This is difficult to verify directly, but a 1941 paper by Max Newman [178] shows there is a simpler equivalent condition when the reduction relation is terminating.

Definition 18.1. A reduction relation $\to$ is *locally confluent*, if whenever s_1 and s_2 are two positive rewrite steps with $\mathrm{src}(s_1) = \mathrm{src}(s_2)$, there exists a term z such that $\mathrm{dst}(s_1) \to z$ and $\mathrm{dst}(s_2) \to z$.

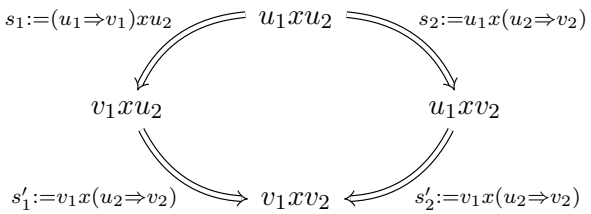
To test for local confluence, we "diverge" from a term by only one step in two different directions, and then check if both sides reduce to some common term. We will see this can be decided algorithmically, and also that we can "repair" any local confluence violations we do find. Thus, Newman's result is fundamental:

Theorem 18.2 (Newman's Lemma)**.** If a reduction relation $\to$ is terminating and locally confluent, then $\to$ is confluent.

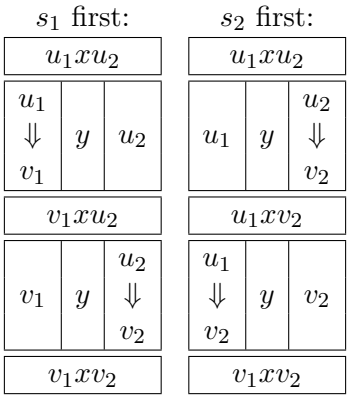
Overlapping rules. A pair of positive rewrite steps with a common source define a *critical pair*. A critical pair shows that some term can be reduced in "two different ways." We can answer if our rewrite rules define a locally confluent reduction relation by inspecting each critical pair. With any non-trivial list of rewrite rules, there are infinitely many such critical pairs, however, all but a finite subset can be disregarded. Suppose we have these two rules over some alphabet A:

$$u_1 \Rightarrow v_1$$
$$u_2 \Rightarrow v_2$$

For any term $x \in A^*$, we can form the "sandwich" term $t := u_1 x u_2$. Every such choice of x defines a new critical pair; the occurrences of u_1 and u_2 within t can be rewritten in two ways, by $s_1 := (u_1 \Rightarrow v_1)xu_2$ and $s_2 := u_1 x(u_2 \Rightarrow v_2)$. However, since s_1 and s_2 rewrite disjoint subterms of t, we say this critical pair is *orthogonal*. Orthogonal critical pairs are not interesting because they cannot witness a local confluence violation. To see why, notice that regardless of whether we apply s_1 or s_2 first, there exists a complementary rewrite step s_1' or s_2' to rewrite $\mathrm{dst}(s_1)$ or $\mathrm{dst}(s_2)$ into the "reduced sandwich" $v_1 x v_2$. In fact, we get a commutative diagram like this for any orthogonal critical pair:



We can also visualize an orthogonal critical pair using the "pictorial" notation for rewrite steps we devised in Section 16.2:



Clearly, in our quest to uncover local confluence violations, we only need to inspect critical pairs that are *not* orthogonal; that is, they must rewrite *overlapping* subterms of their common source term. There are only finitely many such critical pairs, and they are all generated by inspecting the left-hand sides of our rewrite rules. We can completely characterize them with the below definition.

Definition 18.3. Two rules (u_1, v_1) and (u_2, v_2) *overlap* if one of the following holds:

1. The left-hand side of the second rule is contained entirely within the left-hand side of the first. That is, $u_1 = xyz$ and $u_2 = y$ for some x, y, $z \in A^*$. If we write down the terms u_1 and u_2 and shift u_2 over until they line up, we get this:

$$\begin{array}{l} xyz \\ \quad y \end{array}$$

2. The left-hand side of the second rule has a prefix equal to a suffix of the left-hand side of the first. That is, $u_1 = xy$ and $u_2 = yz$ for some x, y, $z \in A^*$, with $|x| > 0$ and $|z| > 0$. If we write down the terms u_1 and u_2 and shift u_2 over until they line up, we get this:

$$\begin{array}{l} xy \\ \quad yz \end{array}$$

The above are the two ways in which a non-orthogonal critical pair can rewrite the same term. When the case distinction is important, we can talk about an overlap of the *first kind*, or *second kind*, respectively. In both cases, after a suitable assignment of x, y and z, we define the *overlap term* to be xyz, and the *overlap position* to be $|x|$.

Example 18.4. Consider these three rules:

$$\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}] \cdot [\texttt{Equatable}] \quad (1)$$

$$\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}] \Rightarrow \texttt{τ_0_1} \quad (2)$$

$$[\texttt{Collection}|\texttt{SubSequence}] \cdot [\texttt{Collection}|\texttt{Element}] \Rightarrow [\texttt{Collection}|\texttt{Element}] \quad (3)$$

There is an overlap of the first kind between (1) and (2) at position 0; the left-hand side of (2) is contained entirely in the left-hand side of (1). Here, the overlap term is $\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}] \cdot [\texttt{Equatable}]$:

$$\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}] \cdot [\texttt{Equatable}]$$
$$\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}]$$

There is an overlap of the second kind between (1) and (3) at position 1; the left-hand side of (3) begins with the prefix $[\texttt{Collection}|\texttt{SubSequence}]$, which is also a suffix of the left-hand side of (1). The overlap term is $\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}] \cdot [\texttt{Equatable}]$:

$$\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}]$$
$$[\texttt{Collection}|\texttt{SubSequence}] \cdot [\texttt{Collection}|\texttt{Element}]$$

The definition of an overlap of the second kind requires both x and z to be non-empty. If we relaxed this condition, then we would *also* have an overlap of the second kind between rule (2) and (1) at position 0:

$$\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}]$$
$$\texttt{τ_0_0} \cdot [\texttt{Collection}|\texttt{SubSequence}] \cdot [\texttt{Equatable}]$$

The overlap term and position is the same as the first case we saw already, so attempting to resolve this critical pair would not reveal anything new. We adjust our definition to avoid duplicated work in this situation.

It can happen that two rules overlap more than once at *different* positions; in this case we must consider every possible overlap. For example, these two rules generate four overlaps in total:

$$[\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \Rightarrow [\texttt{P}|\texttt{C}] \quad (4)$$

$$[\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \Rightarrow [\texttt{P}|\texttt{D}] \quad (5)$$

Rule (4) overlaps with (5) at position 1:

$$[\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}]$$
$$[\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}]$$

Rrule (4) overlaps with (5) at position 3:

$$[\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}]$$
$$[\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}]$$

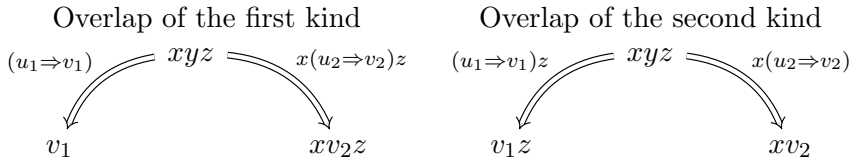
Rule (5) overlaps with (4) at position 1:

$$[\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}]$$
$$[\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}]$$

Last but not least, rule (5) overlaps with (4) at position 3:

$$[\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}]$$
$$[\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}]$$

Resolving critical pairs. A critical pair exhibits some term t being rewritten in two distinct ways. If we take the destination term of each of the two rewrite steps, we get a pair of terms that are known to be equivalent to t, and each other. For an overlap of the first kind, the two terms are $(v_1,\ xv_2z)$; for the second kind, $(v_1z,\ xv_2)$:



We *resolve* a critical pair (t_1, t_2) by reducing both sides with the reduction relation as constructed so far and comparing the reduced terms. There are four possible outcomes:

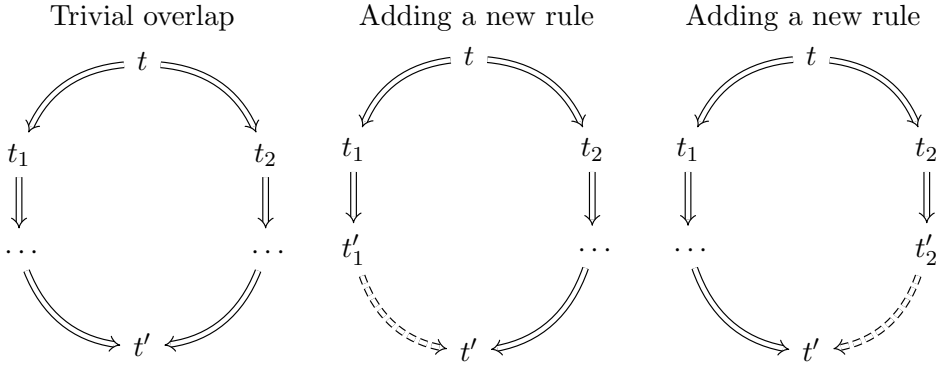
1. If t_1 and t_2 reduce to the same term t', we say the critical pair is *trivial*. (Note that this terminology offers an alternate definition of local confluence: a reduction relation is locally confluent if all critical pairs are trivial.)

2. If $t_1 \to t_1'$ and $t_2 \to t_2'$ with $t_1' \neq t_2'$, we found two distinct reduced terms in the same equivalence class: a *confluence violation*. If $t_2' < t_1'$, we repair the confluence violation by adding a new rewrite rule (t_1', t_2'). Having done so, if we then define $t' := t_2'$, we once again see that both t_1 and t_2 reduce to the same term t'.

3. If instead $t_1' < t_2'$, we have the same situation except we resolve it by adding a new rewrite rule (t_2', t_1'), and we let $t' := t_1'$.

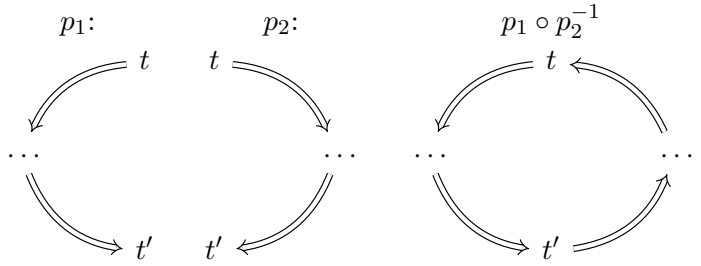
4. If t_1' and t_2' are distinct but incomparable, we have a non-orientable relation, and we must report an error. This cannot happen under the reduction order used by the Requirement Machine.

We can draw a diagram for each case, showing a subgraph of the rewrite graph; dashed arrows indicate new rules being added:



After adding a new rewrite rule if necessary, we have a pair of rewrite paths p_1 and p_2, and a term t'. Note that $\mathrm{src}(p_1) = \mathrm{src}(p_2) = t$ and $\mathrm{dst}(p_1) = \mathrm{dst}(p_2) = t'$, so both paths have the same source and destination. We say p_1 and p_2 are *parallel* rewrite paths.

Rewrite loops. Now we slightly change our point of view and introduce a new concept. If we take two parallel rewrite paths p_1 and p_2, we can form the rewrite path $\ell := p_1 \circ p_2^{-1}$ by composing the first path with the inverse of the second:



This new rewrite path ℓ has the property that it begins and ends at the *same* term t:

$$\mathrm{src}(\ell) = \mathrm{src}(p_1) = t \qquad \mathrm{dst}(\ell) = \mathrm{dst}(p_2^{-1}) = \mathrm{src}(p_2) = t$$

We say that ℓ is a *rewrite loop* with basepoint t. A rewrite loop applies some sequence of rewrite rules to the basepoint term t, then rewrites it "back" to t, via a possibly *different* mix of rewrite rules. In the rewrite graph, a rewrite loop is a cycle (sometimes called a "closed path" by graph theorists). Note that for each $t \in A^*$, the empty rewrite path 1_t can also be seen as a trivial rewrite loop with basepoint t.

Now, we give two algorithms that together form the inner loop of the completion procedure. It is also convenient to view a critical pair as a rewrite path of length 2, rather than a pair of rewrite steps with the same source term—we compose the first step with the inverse of the second. In this new formulation, critical pair resolution "completes" the critical pair to form a rewrite loop.

Algorithm 18A (Construct critical pair)**.** Takes two rules (u, v) and (u', v'), together with an overlap position i where $0 \leq i < |u|$. Returns a triple (t_1, t_2, p) where t_1 and t_2 are terms, and p is a rewrite path with $\mathrm{src}(p) = t_1$ and $\mathrm{dst}(p) = t_2$.

1. If $i + |u'| \leq |u|$, we have an overlap of the first kind; $u = xu'z$ for some x and z.
   a) Let $x := u[: i]$ (the prefix of u of length i), and $z := u[i + |u'| :]$ (the suffix of u of length $|u| - |u'| - i$).
   b) Let $t_1 := v$, the result of rewriting u with the first rule.
   c) Let $t_2 := xv'z$, the result of rewriting u with the second rule.
   d) Let $p := (v \Rightarrow u) \circ x \triangleleft (u' \Rightarrow v') \triangleright z$. This is a rewrite path from t_1 to t_2.
2. Otherwise, we have an overlap of the second kind; $u = xy$ and $u' = yz$ for some x, y and z.
   a) Let $x := u[: i]$ (the prefix of u of length i), $z := u'[|u| - i :]$ (the suffix of u' of length $|u'| - |u| + i$). (We don't actually need $y := u[i :] = u'[: |u| - i]$.)
   b) Let $t_1 := vz$, the result of rewriting xyz with the first rule.
   c) Let $t_2 := xv'$, the result of rewriting xyz with the second rule.
   d) Let $p := (v \Rightarrow u) \triangleright z \circ x \triangleleft (u' \Rightarrow v')$.
3. Return the triple (t_1, t_2, p).

Algorithm 18B (Resolve critical pair)**.** As input, takes terms u and v, and a rewrite path p with $\mathrm{src}(p) = u$ and $\mathrm{dst}(p) = v$. Records a rewrite loop, and possibly adds a new rule, returning true if a rule was added.

1. If $u = v$, then p is already a loop; record it and return false.
2. Compute the normal form of u with Algorithm 17O, to get $\tilde{u}$ and p_1.
3. Compute the normal form of v with Algorithm 17O, to get $\tilde{v}$ and p_2.
4. Compare $\tilde{u}$ and $\tilde{v}$ with Algorithm 17I.
5. If $\tilde{u} = \tilde{v}$, record a loop $p_1^{-1} \circ p \circ p_2$ with basepoint $\tilde{u} = \tilde{v}$, and return false.
6. If $\tilde{v} < \tilde{u}$, record a rule $(\tilde{u}, \tilde{v})$, record a loop $p_2^{-1} \circ p^{-1} \circ p_1 \circ (\tilde{u} \Rightarrow \tilde{v})$ with basepoint $\tilde{u}$, and return true.
7. If $\tilde{u} < \tilde{v}$, record a rule $(\tilde{v}, \tilde{u})$, record a loop $p_1^{-1} \circ p \circ p_2 \circ (\tilde{v} \Rightarrow \tilde{u})$ with basepoint $\tilde{v}$, and return true.
8. If $\tilde{u}$ and $\tilde{v}$ are incomparable, signal an error.

Rewrite loops are not just a theoretical tool; our implementation of the Knuth-Bendix algorithm follows [179] and [180] in encoding and recording the rewrite loops that describe resolved critical pairs. This enables the computation of minimal requirements in Section 20.1. Only local rules are subject to minimization, so we only record rewrite loops involving local rules. If a requirement machine instance is only to be used for generic signature queries and not minimization, rewrite loops are not recorded at all.

An optimization. Now that we know how to process a single overlap and resolve a critical pair, the next chunk of code concerns enumerating all candidate overlaps. If our rewrite rules were truly arbitrary, we would need to consider all possible combinations: for every rewrite rule $u_1 \Rightarrow v_1$, for every rewrite rule $u_2 \Rightarrow v_2$, and for every position $i < |u_1|$, we would need to check if the corresponding subterms of u_1 and u_2 are identical. However, we can do better. The bottom-up construction of a requirement machine from protocol components, and the partition of rewrite rules into imported rules and local rules, enables an optimization where overlaps between certain pairs of rules need not be considered at all:

- We don't need to look for overlaps between imported rules. While an imported rule can overlap with another imported rule, all such critical pairs are trivial and do not need to be resolved again.
- We don't need to look for overlaps between an imported rule and a local rule. An imported rule cannot overlap with a local rule.

Only two interesting pairings remain:

- A local rule can overlap with another local rule.
- A local rule can overlap with an imported rule.

We now proceed to prove that this is indeed the case. First, suppose two imported rules overlap. We will consider three possibilities in turn:

- There is an overlap of the first kind.
- There is an overlap of the second kind, and the second rule's left-hand side starts with an associated type symbol.
- There is an overlap of the second kind, and the second rule's left-hand side starts with a protocol symbol.

In all three cases, we will conclude that both rules either originate from the same protocol component, or two distinct components where one imports the other. This implies that the critical pair is now trivial, having been resolved by completion of that protocol component which contains the other.

In the first case, it is immediate that both rules were imported from the same protocol component for `P`:

$$[\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$$
$$[\texttt{P}|\texttt{A}]$$

In the second case, `P` and `Q` are either in the same protocol component, or `Q` is a protocol dependency of `P`, because we have $G_{\texttt{P}} \vdash [\texttt{Self.A: Q}]$:

$$[\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$$
$$\qquad [\texttt{Q}|\texttt{B}] \cdot [\texttt{R}|\texttt{C}]$$

The final case similarly implies that `Q` is a protocol dependency of `P`:

$$[\texttt{P}|\texttt{A}] \cdot [\texttt{Q}]$$
$$\qquad [\texttt{Q}] \cdot [\texttt{R}]$$

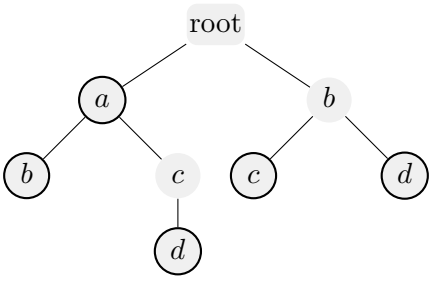
Next, we claim that imported rules cannot overlap with local rules. In a query machine or minimization machine for a generic signature, both sides of a local rule start with a generic parameter symbol. The left-hand side of an imported rule cannot start with, or contain, a generic parameter symbol, proving the claim. In a requirement machine for a protocol, we can similarly rule out overlap of the first kind between an imported rule and a local rule, because the two rules must start with the same protocol or associated type symbol, and thus originate from the same protocol component. Now, suppose a protocol machine has an overlap of the second kind. We show that if the second rule, the one with left-hand side $[\texttt{Q}|\texttt{B}] \cdot [\texttt{R}|\texttt{C}]$, is a local rule, then the first rule is as well:

$$[\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$$
$$\qquad [\texttt{Q}|\texttt{B}] \cdot [\texttt{R}|\texttt{C}]$$

As before, `Q` must be a protocol dependency of `P`. But this time, we have the further assumption that `Q` belongs to the *current* protocol component, so the appearance of `P` means that `P` must *also* be a protocol dependency of `Q`. Thus, `P` and `Q` depend on each other, and are actually part of the same component; therefore both rules are local.

Another optimization. If we fix a rule and position, we can find all overlaps involving this rule and position by performing a lookup into the rule trie, which we previously used to speed up term reduction in Section 17.6. This further cuts down the work in enumerating overlaps. While the underlying data structure is the same, the lookup algorithm here differs from that used by term reduction; we must enumerate all matches, instead of stopping after the first one.

Consider a set of rewrite rules having the terms a, ab, bc, bd and acd as their left-hand sides. The rule with left-hand side ab has overlaps with all other rules except for acd. Here is the rule trie, with thick borders denoting nodes associated with rewrite rules:



When checking the left-hand side ab for overlaps, we perform two lookups:

- At position 0, we look up ab. Starting from the root, we encounter the rule for a, followed by ab (which is the left-hand side of our rule itself, so we skip it). The latter node is a leaf so we end the search.
- At position 1, we look up b. The node for b doesn't store a rewrite rule, however, it has child nodes. We've reached the end of our input sequence, so we recursively visit all children, and find the two final overlap candidates, bc and bd.

This new kind of trie lookup can be thought of as a coroutine or an iterator, yielding zero or more results as the search proceeds. We implement it as a higher-order function taking a callback.

Algorithm 18C (Overlap lookup in rule trie)**.** Takes a term t, an offset i such that $0 \leq i < |t|$, and a callback. For each rule (u, v) where $t[i :]$ is a prefix of u or u is a prefix of $t[i :]$, invokes the callback with the rule (u, v).

1. (Initialize) Let N be the root node of the trie.
2. (End) If $i = |t|$, we've reached the end of the term. Perform a pre-order traversal of all child nodes of N, and for those children that have an associated rewrite rule, invoke the callback with that rule (this is the case where $t[i :]$ is a prefix of each u).
3. (Traverse) Let s_i be the ith symbol of t. Look up s_i in N. If there is no such child node, return.
4. (Child) Otherwise, call this child node M. If M has an associated rule $u \Rightarrow v$, invoke the callback with this rule (in this case, u is a prefix of $t[i :]$).
5. (Advance) Set $N \leftarrow M$. Set $i \leftarrow i + 1$. Go back to Step 2.

The next algorithm feeds the results of Algorithm 18C into Algorithm 18A to build a list of all critical pairs among our rewrite rules. After resolving these critical pairs, we will have to check for overlaps again, in case there are any involving the newly-added rules. To avoid repeated work, the below algorithm maintains a set of visited overlaps, so that we can avoid building and resolving a critical pair we already know has been resolved.

Algorithm 18D (Find overlapping rules)**.** Takes a list of rewrite rules as input, and outputs a list of critical pairs. Also queries and updates a set of visited overlaps V.

1. (Initialize) Set $i \leftarrow 0$. Let n be the total number of local rules. Initialize the empty output list.
2. (Outer check) If $i = n$, we're done. Return the output list.
3. (Get rule) Let (u, v) be the ith local rule. If this rule is marked as **left-simplified**, **right-simplified** or **substitution-simplified**, skip it entirely and go to Step 7. Otherwise, set $\leftarrow 0$.
4. (Inner check) If $j = |u|$, we're done. Go to Step 7.
5. (Find overlaps) Find all rules whose left-hand side is a prefix of the symbol range $u[j :]$ or vice versa, using Algorithm 18C. For each matching rule (u', v'):
   a) If (u', v') is marked as **left-simplified**, **right-simplified** or **substitution-simplified**, skip this rule.
   b) If $((u, v), (u', v'), j) \in V$, skip this rule.
   c) Otherwise, set $V \leftarrow V \cup \{((u, v), (u', v'), j)\}$, and build a critical pair for this overlap with Algorithm 18A. This returns a triple (t_1, t_2, p); add it to the output list.
6. (Inner loop) Set $j \leftarrow j + 1$, and go back to Step 3.
7. (Outer loop) Set $i \leftarrow i + 1$, and go back to Step 2.

We can now describe the main loop of the Knuth-Bendix completion procedure, which repeatedly finds and resolves critical pairs until no more non-trivial critical pairs remain. This process might not terminate, and we might find ourselves discovering new critical pairs and adding new rules to resolve them, forever. To prevent an infinite loop in the case of failure, we implement a termination check; if we think we've done too much work already, we give up on constructing a convergent rewriting system. We already mentioned the **left-simplified**, **right-simplified**, and **substitution-simplified** flags a few times; they are set by the rule simplification passes, with the first two described in Section 18.1 and the third one in Section 19.1. These passes are invoked at appropriate times in the main loop below.

Algorithm 18E (Knuth-Bendix completion procedure)**.** Takes a list of rewrite rules as input. Records new rewrite rules, as well as rewrite loops, and returns success or failure. On success, the rewrite rules define a convergent rewriting system. Failure indicates we got a non-orientable rewrite rule, or we triggered the termination check.

1. Clear the flag.
2. Build a list of critical pairs with Algorithm 18D.
3. Left-simplify all rules with Algorithm 18F.
4. Resolve each critical pair with Algorithm 18B, and set the flag if any new rewrite rules were added.
5. Right-simplify all rewrite rules with Algorithm 18G.
6. Substitution-simplify all rewrite rules with Algorithm 19A.
7. Check if the rule count, term length, or concrete nesting limits have been exceeded; if so, return failure.
8. If the flag was set, go back to Step 1. Otherwise, return success.

Termination. The termination check is controlled by a handful of frontend flags:

- `-requirement-machine-max-rule-count=<value>` sets the maximum number of local rules to `value`. Imported rules do not count toward this total, so this is hard to hit with realistic code. The default is a maximum of 4000 local rules.
- `-requirement-machine-max-rule-length=<value>` sets the maximum length of the terms in a rule to `value`. To compute the actual quantity, we add the length of the longest *user-written* rule; so the restriction in on the relative "growth" and not on the length of a type parameter written by the user. The default value is 12.
- `-requirement-machine-max-concrete-nesting=<value>` sets the upper bound on nested concrete types to `value`, to prevent substitution simplification from constructing an infinite type like `G<G<G<...>>>`. As with the limit on rule length, we add the maximum nesting depth of user-written rules to get the actual limit. The default value is 30.

While the first of the three is sufficient to detect non-termination, it takes a second or two for completion to record that many rules. The other two limits improve user experience in this case by rejecting clearly invalid programs sooner. The rule length limit being relative instead of just a total ban on terms of length 12 allows various pathological cases to succeed which would otherwise be needlessly rejected.

The following protocol, for example, presents the monoid $\mathbb{Z}_{14}$ and defines a rule of length 14, so the absolute rule length limit is really $14 + 12 = 26$. Completion does not add any longer rules so we accept it without issues:

```
protocol Z14 {
  associatedtype A: Z14
    where Self == Self.A.A.A.A.A.A.A.A.A.A.A.A.A.A
}
```

If completion fails when building a rewrite system for minimization, we have a source location associated with some protocol or generic declaration. An error is diagnosed at this source location, and we proceed with minimization producing an empty list of requirements. If completion fails on a rewrite system built from an existing generic signature or protocol component, there is no source location we can use for diagnostics; the compiler dumps the entire rewrite system and aborts with a fatal error. The latter scenario is unusual; if we successfully constructed a generic signature from user-written requirements, we should be able to build a rewrite system for it again.

Debugging flags. A pair of debugging options can help us understand the operation of the completion procedure; both can be set together[1], but be warned that they produce a large volume of output:

- `-debug-requirement-machine=completion` will dump all overlapping rules and critical pairs.
- `-debug-requirement-machine=add` will dump all rewrite rules and rewrite loops obtained while resolving critical pairs.

18.1. Rule Simplification

We impose two further conditions on our convergent rewriting system:

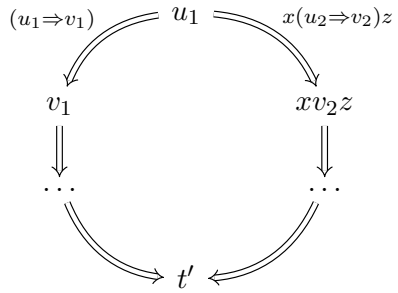
1. No rule has a left-hand side that can be reduced by any other rule.
2. No rule has a right-hand side that can be reduced by any other rule.

Such rewrite systems are called *left-reduced* or *right-reduced*, respectively, or simply *reduced* if both conditions are met. Any convergent rewriting system can be transformed into a reduced rewriting system with a pair of simplification passes that possibly delete and add rules.

[1] With a single `-debug-requirement-machine=` flag, separating the subflags with commas.

In our implementation, we don't *actually* delete rules, because we use the index of each rule as a stable reference elsewhere; instead, we set a pair of rule flags, **left-simplified** and **right-simplified**, and delete the rule from the rule trie. We've seen these flags mentioned already, so now we reveal their purpose. This will motivate the subsequent theory, setting the stage for the remaining two sections of this chapter.

Left simplification. If the left-hand side of a rewrite rule $u_1 \Rightarrow v_1$ can be reduced by another rewrite rule $u_2 \Rightarrow v_2$, then $u_1 = xu_2z$ for some x, $z \in A^*$, so we have an overlap of the first kind in the sense of Definition 18.3. Once we resolve all critical pairs, we don't need the first rule at all; we know that in a convergent rewriting system, both ways of reducing the overlap term $u_1 := xu_2z$ produce the same result:

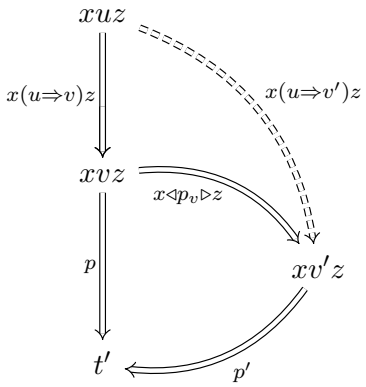


The left simplification algorithm considers the left-hand side of each rule, and marks the rule if it finds a subterm matching some other rule.

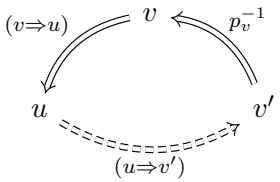
Algorithm 18F (Left-simplify rules)**.** Takes the list of local rules as input. Has side effects.

1. (Initialize) Let n be the total number of local rules, and set $i := 0$.
2. (Outer check) If $i = n$, return. Otherwise, say (u, v) is the ith local rule, and set $j := 0$.
3. (Inner check) If $j = |u|$, go to Step 8.
4. (Search) Look up $u[: j]$ in the rule trie, where $u[: j]$ is the suffix of u of length $|u| - j$.
5. (Decide) If the trie lookup returns no results, or it returns (u, v) itself (which can only happen if $j = 0$, so $u[: j] = u$), go to Step 7.
6. (Mark) Otherwise, u has a subterm equal to the left-hand side of some other rule. Mark $u \Rightarrow v$ as **left-simplified** and go to Step 7.
7. (Inner loop) Set $j \leftarrow j + 1$ and go back to Step 3.
8. (Outer loop) Set $i \leftarrow i + 1$ and go back to Step 2.

Right simplification. The other case is when we have a rule (u, v) whose right-hand side v is not reduced, so $v \to v'$ via some positive rewrite path p_v. Now suppose we have a positive rewrite path $x(u \Rightarrow v)z \circ p$, where $\mathrm{dst}(p)$ is some reduced term t'. If we reduce xuz to xvz via $x(u \Rightarrow v)z$, we have two choices on how to proceed: we can follow p, or reduce xvz to $xv'z$ via $x \triangleleft p_v \triangleright z$. By confluence, the second choice must take us to t' via a positive rewrite path p' with $\mathrm{src}(p') = xv'z$ and $\mathrm{dst}(p') = t'$. Thus, we record a new rule $u \Rightarrow v'$ that obsoletes $u \Rightarrow v$:



The right simplification algorithm outputs a right-reduced rewrite system by attempting to reduce the right-hand side of each rewrite rule. While left simplification does not need to record new rewrite rules because completion has already resolved all overlaps of the first kind, right simplification actually records new rules as well as marking existing rules as having been simplified. The new rule is related with the existing rule by a rewrite loop:



Algorithm 18G (Right-simplify rules)**.** Takes the list of local rules as input. Has side effects.

1. (Initialize) Let n be the total number of local rules, and set $i := 0$.
2. (Check) If $i = n$, return. Otherwise, let (u, v) be the ith local rule.
3. (Reduce) Apply Algorithm 17O to v to get a term $\tilde{v}$ and a rewrite path p_v. If $v = \tilde{v}$, the right-hand side v is already reduced, so go to Step 7.
4. (Record) Invoke Algorithm 17Q to add a new rewrite rule $(u, \tilde{v})$.

5. (Relate) Add the rewrite loop $(u \Rightarrow v) \circ p \circ (v' \Rightarrow u)$ with basepoint u, relating the old rule $u \Rightarrow v$ with the new rule $u \Rightarrow v'$.

6. (Mark) Mark the old rule as **right-simplified**.

7. (Loop) Increment i and go back to Step 2.

Our justification for the validity of these passes worked from the assumption that we had a convergent rewriting system; that is, that completion had already been performed. In practice, Algorithm 18E repeatedly runs both passes during completion, once per round of critical pair resolution. This is advantageous, because we can subsequently avoid considering overlaps that involve simplified rules. This strategy remains sound as long as we perform left simplification after computing critical pairs, but *before* resolving them, which might add new rules. This narrows the candidates for left simplification to those rules whose overlaps have already been considered. As for the right simplification pass, it is actually fine to run it at any point; we choose to run it after resolving critical pairs.

Related concepts. We previously saw in Section 11.4 that the same-type requirements in a generic signature are subject to similar conditions of being left-reduced and right-reduced. There is a connection here, because as we will see in Section 20.4, the minimal requirements of a generic signature are ultimately constructed from the rules of a reduced rewrite system. However, there are a few notational differences:

- The roles of "left" and "right" are reversed because requirements use a different convention; in a reduced same-type requirement [`U == V`], we have `U` $<$ `V`, whereas in a rewrite rule (u, v) we have $v < u$.

- Reduced same-type requirements have the "shortest" distance between the left-hand and right-hand side, so if `T`, `U` and `V` are all equivalent and `T` $<$ `U` $<$ `V`, the corresponding requirements are [`T == U`], [`U == V`]. On the other hand, if we have three terms t, u and v with $t < u < v$, then the two corresponding rewrite rules would be (u, t) and (v, t).

These differences are explained by the original `GenericSignatureBuilder` minimization algorithm, described as finding a minimum spanning tree for a graph of connected components. The notion of reduced requirement output by this algorithm became part of the Swift stable ABI. In Section 20.4 we show that a list of rewrite rules in a reduced rewrite system defines a list of reduced requirements via a certain transformation.

What we call reduced rewrite systems are sometimes "normalized", "canonical", or "inter-reduced" in the literature. Our rewrite system also implements a third *substitution simplification* pass for rewrite system simplification. Substitution simplification reduces substitution terms appearing in superclass, concrete type and concrete conformance symbols. We will discuss it in Chapter 19.

18.2. Associated Types

A conformance rule $t \cdot [\texttt{P}] \Rightarrow t$ always overlaps with an associated type rule $[\texttt{P}] \cdot \texttt{A} \Rightarrow [\texttt{P}|\texttt{A}]$, and resolving this critical pair defines a rule $t \cdot \texttt{A} \Rightarrow t \cdot [\texttt{P}|\texttt{A}]$ (unless the right-hand side reduces further). These rewrite rules reduce those terms representing unbound type parameters to bound type parameters. Thus, we will see how the bound and unbound type parameters from Section 5.4 manifest in our rewriting system.

Example 18.5. We're going to look at this pair of protocol declarations, and the protocol generic signature $G_{\texttt{P}}$:

```
protocol Q {
  associatedtype B
}

protocol P {
  associatedtype A: Q
}
```

Protocol `P` has an associated conformance requirement $[\texttt{Self.[P]A: Q}]_{\texttt{P}}$, and $G_{\texttt{P}}$ has a conformance requirement $[\texttt{τ_0_0: P}]$, so the protocol dependency graph has the two edges $G_{\texttt{P}} \prec \texttt{P}$ and $\texttt{P} \prec \texttt{Q}$. We first build a rewrite system for `Q`. The rewrite system for $G_{\texttt{P}}$ imports rules from `P` and `Q`:

$[\texttt{Q}] \cdot \texttt{B} \Rightarrow [\texttt{Q}|\texttt{B}]$	(1)
$[\texttt{P}] \cdot \texttt{A} \Rightarrow [\texttt{P}|\texttt{A}]$	(2)
$[\texttt{P}|\texttt{A}] \cdot [\texttt{Q}] \Rightarrow [\texttt{P}|\texttt{A}]$	(3)
$[\texttt{P}|\texttt{A}] \cdot \texttt{B} \Rightarrow [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$	($\diamond$4)
$\texttt{τ_0_0} \cdot [\texttt{P}] \Rightarrow \texttt{τ_0_0}$	(5)
$\texttt{τ_0_0} \cdot \texttt{A} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}]$	($\diamond$6)

We can categorize these rewrite rules as follows:

- Associated type rules: (1) and (2).
- Conformance requirements: (3) and (5).
- Rules added by completion are indicated with an asterisk: ($\diamond$4) and ($\diamond$6).

We omit the identity conformance rules $[\texttt{Q}] \cdot [\texttt{Q}] \Rightarrow [\texttt{Q}]$ and $[\texttt{P}] \cdot [\texttt{P}] \Rightarrow [\texttt{P}]$; in this example they would only clutter the presentation. They will serve a useful purpose later, in Example 18.10.

We now go through the construction step by step. Protocol `Q` does not depend on any other protocols. The rewrite system for `Q` is just rule (1):

$$[\mathtt{Q}] \cdot \mathtt{B} \Rightarrow [\mathtt{Q}|\mathtt{B}] \tag{1}$$

Protocol `P` imports the single rule of `Q`, and adds rules (2) and (3):

$$[\mathtt{P}] \cdot \mathtt{A} \Rightarrow [\mathtt{P}|\mathtt{A}] \tag{2}$$
$$[\mathtt{P}|\mathtt{A}] \cdot [\mathtt{Q}] \Rightarrow [\mathtt{P}|\mathtt{A}] \tag{3}$$

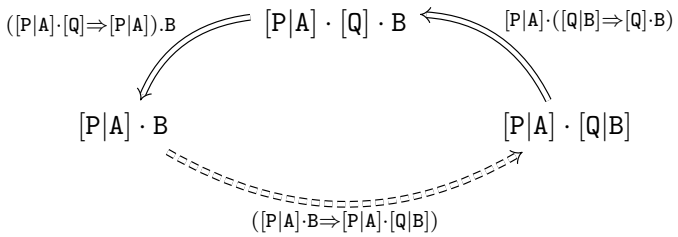
We need to check the left-hand side of rules (2) and (3) for overlaps with other rules. Rule (2) does not overlap with any other rules. Rule (3) overlaps with rule (1) on the term $[\mathtt{P}|\mathtt{A}] \cdot [\mathtt{Q}] \cdot \mathtt{B}$:

$$\begin{array}{l} [\mathtt{P}|\mathtt{A}] \cdot [\mathtt{Q}] \\ \qquad [\mathtt{Q}] \cdot \mathtt{B} \end{array}$$

Rule (3) is local to `P`, while (1) was imported from `Q`. The two sides of the critical pair reduce to $[\mathtt{P}|\mathtt{A}] \cdot \mathtt{B}$ and $[\mathtt{P}|\mathtt{A}] \cdot [\mathtt{Q}|\mathtt{B}]$. Resolving this critical pair introduces rule ($\diamond 4$):

$$[\mathtt{P}|\mathtt{A}] \cdot \mathtt{B} \Rightarrow [\mathtt{P}|\mathtt{A}] \cdot [\mathtt{Q}|\mathtt{B}] \tag{$\diamond 4$}$$

We also record a rewrite loop that defines rule ($\diamond 4$) via rules (3) and (1):



Once again, we check for overlaps in the left-hand sides of our local rules—now (2), (3) and (4). We see that no more critical pairs remain, and we have a convergent rewriting system for `P`.

Finally, we build the rewrite system for $G_{\mathtt{P}}$. We import all rules from `Q` and `P`, and add one new local rule corresponding to the conformance requirement [`τ_0_0: P`]:

$$\tau_0_0 \cdot [\mathtt{P}] \Rightarrow \tau_0_0 \tag{5}$$

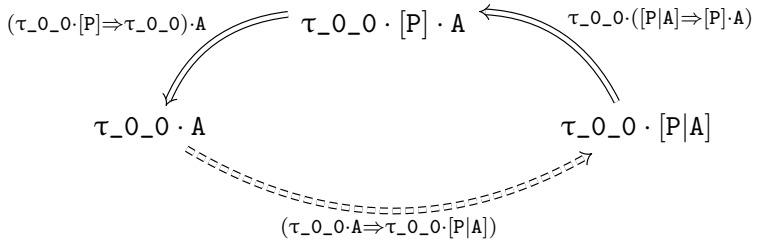
We need to check the left-hand side of rule (5) for overlaps with other rules. Indeed, rule (5) overlaps with rule (2) on the term $\tau_0_0 \cdot [\mathtt{P}] \cdot \mathtt{A}$:

$$\begin{array}{l} \tau_0_0 \cdot [\mathtt{P}] \\ \qquad [\mathtt{P}] \cdot \mathtt{A} \end{array}$$

Resolving this critical pair introduces rule ($\diamond 6$):

$$\texttt{τ_0_0} \cdot \texttt{A} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \tag{$\diamond 6$}$$

We also record a rewrite loop that defines rule ($\diamond 6$) via rules (5) and (2):



At this point, there might be overlaps involving one of rule (5) or ($\diamond 6$); a quick check shows none remain, so we have a convergent rewriting system for $G_{\texttt{P}}$.

Now, we get to the interesting part. Consider these two type parameters of $G_{\texttt{P}}$, and their corresponding terms:

1. The unbound type parameter $\texttt{τ_0_0.A.B}$, and term $\texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{B}$.
2. The bound type parameter $\texttt{τ_0_0.[P]A.[Q]B}$, and term $\texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$.

The second term is reduced, so the first term must reduce to the second term. Term reduction outputs a positive rewrite path p with $\mathrm{src}(p) = \texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{B}$ and $\mathrm{dst}(p) = \texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$. This path involves rules (4) and (6) added by completion:

$$p := (\texttt{τ_0_0} \cdot \texttt{A} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}]) \cdot \texttt{B} \circ \texttt{τ_0_0} \cdot ([\texttt{P}|\texttt{A}] \cdot \texttt{B} \Rightarrow [\texttt{P}|\texttt{A}] \cdot [\texttt{B}|\texttt{Q}])$$

Here is a diagram for p:

$$\texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{B} \Longrightarrow \texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot \texttt{B} \Longrightarrow \texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$$

There is a "telescoping" effect, as term reduction processes the name symbols from left to right, replacing them with associated type symbols. As an aside, if we have an *invalid* type parameter, such as $\texttt{τ_0_0.A.A}$, the term reduces to $\texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot \texttt{A}$ but no further; the final name symbol `A` cannot be "resolved" because the type parameter $\texttt{τ_0_0.A}$ does not *have* a member type named `A`.

The standard term $\texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{B}$ reduces to $\texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$ because the unbound type parameter $\texttt{τ_0_0.A.B}$ is equivalent to the bound type parameter $\texttt{τ_0_0.[P]A.[Q]B}$, and the latter is the reduced type of the former. The equivalence follows because we can derive a same-type requirement, and the left-hand side of this same-type requirement is the reduced type because no smaller type parameter can be shown to be equivalent:

$$G_{\texttt{P}} \vdash [\texttt{τ_0_0.[P]A.[Q]B == τ_0_0.A.B}]$$

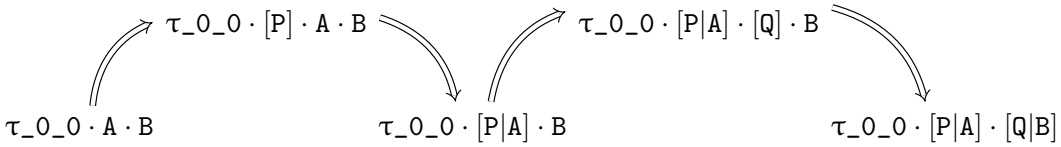
Here is one possible derivation for this same-type requirement:

1. [τ_0_0: P] (Conf)
2. [τ_0_0.[P]A == τ_0_0.A] (AssocBind 1)
3. [τ_0_0.A == τ_0_0.[P]A] (Sym 2)
4. [τ_0_0.[P]A: Q] (AssocConf 1)
5. [τ_0_0.A.B == τ_0_0.[P]A.B] (SameName 3 4)
6. [τ_0_0.[P]A.B == τ_0_0.A.B] (Sym 5)
7. [τ_0_0.[P]A.[Q]B == τ_0_0.[P]A.B] (AssocBind 4)
8. [τ_0_0.[P]A.[Q]B == τ_0_0.A.B] (Trans 7 6)

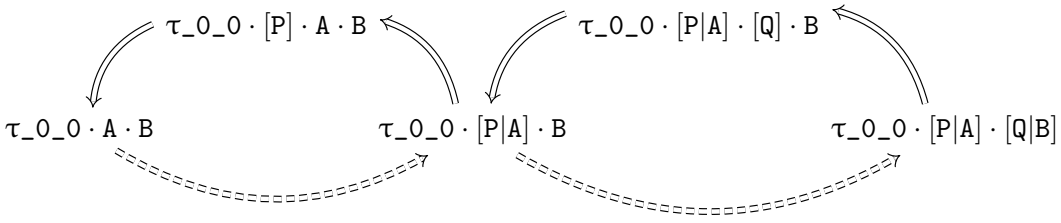
By Theorem 17.4, we can transform this derivation into a rewrite path from $\texttt{τ_0_0·A·B}$ to $\texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}]$, involving the initial rewrite rules only. We call this path p':

$$p' := (\texttt{τ_0_0} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P}]) \cdot \texttt{A} \cdot \texttt{B} \circ \texttt{τ_0_0} \cdot ([\texttt{P}] \cdot \texttt{A} \Rightarrow [\texttt{P}|\texttt{A}]) \cdot \texttt{B} \\ \circ \texttt{τ_0_0} \cdot ([\texttt{P}|\texttt{A}] \Rightarrow [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}]) \cdot \texttt{B} \circ \texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot ([\texttt{Q}] \cdot \texttt{B} \Rightarrow [\texttt{Q}|\texttt{B}])$$

Unlike p, p' is *not* a positive rewrite path, because the first and third steps are negative; each one applies a conformance rule backwards. We can visualize p' as a path in the rewrite graph, with the negative rewrite steps going up:



We now have two parallel rewrite paths: p (output by the normal form algorithm), and p' (constructed from our derivation). Their composition $p' \circ p^{-1}$ is a rewrite loop; we can represent it as a diagram, with rules ($\diamond 4$) and ($\diamond 6$) indicated by dashed arrows:



Recall the two rewrite loops we recorded in completion. We take the first rewrite loop, and whisker it on the right by B; then, we whisker the second rewrite loop on the left by τ_0_0. Now, the two rewrite loops visit the common term $\texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot \texttt{B}$. Informally, we can "glue" them together at this point, and then we get $p' \circ p^{-1}$.

In fact, given any two parallel rewrite paths in a convergent rewriting system, we can always "tile" the two-dimensional space between them if we take the finite set of rewrite loops generated by resolving critical pairs, and glue them together along common edges or vertices after putting the loops "in context" via whiskering. We will develop this idea further in Chapter 20, but here is a thought to ruminate on in the meantime:

> Terms, generated by a finite set of symbols, are the zero-dimensional objects in the rewrite graph. Rewrite paths, generated by a finite set of rewrite rules, define an equivalence relation on terms; they are the one-dimensional objects. Rewrite loops, generated by a finite set of critical pairs, define an equivalence relation on paths; they are the two-dimensional objects.

An equivalence relation on paths is called a *homotopy relation*.

Example 18.6. Let's build on our previous example and look at how unbound type parameters reduce when they appear in requirements. We add a new protocol `R`, and declare a constrained protocol extension of `P` where `Self.A.B` conforms to `R`:

```
protocol R {}

protocol Q {
  associatedtype B
}

protocol P {
  associatedtype A: Q
}

extension P where Self.A.B: R {}
```

When type checking these declarations, we need to build a generic signature for this protocol extension. We start with requirement $[\texttt{τ_0_0: P}]$ from the extended type, and add the user-written requirement $[\texttt{τ_0_0.A.B: R}]$. The rewrite system looks like the previous example but with two new rules:

$$\texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{B} \cdot [\texttt{R}] \Rightarrow \texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{B} \tag{7}$$

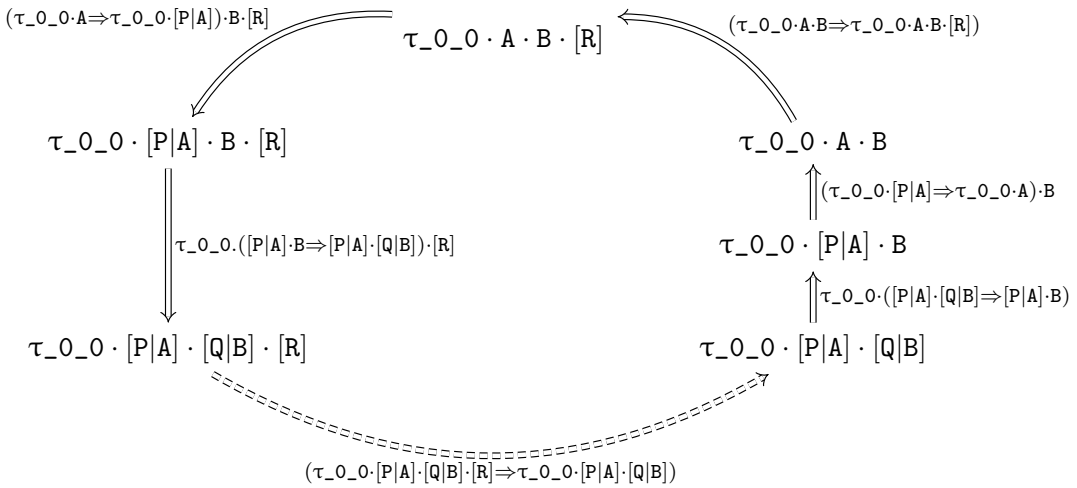
$$\texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}] \cdot [\texttt{R}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P}|\texttt{A}] \cdot [\texttt{Q}|\texttt{B}] \tag{8*}$$

Rule (7) corresponds to the user-written requirement. Rule ($\diamond$8) is added by completion, resolving the overlap between rule (7) and (6) on the term $\texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{B} \cdot [\texttt{R}]$:

$\texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{B} \cdot [\texttt{R}]$

$\texttt{τ_0_0} \cdot \texttt{A}$

We record a rewrite loop defining rule (⋄8) via rule (4), (6) and (7):



Note that this was an overlap of the first kind, so rule (7) is now marked **left-simplified**. Thus, rule (⋄8) completely supersedes rule (7). Rule (⋄8) survives minimization and maps to the requirement [τ_0_0.[P]A.[Q]B: R], which appears in the generic signature that we output for this protocol extension:

```
<τ_0_0 where τ_0_0: P, τ_0_0.[P]A.[Q]B: R>
```

If our compiler invocation is generating a serialized module, we serialize the protocol extension's generic signature and do not compute it again when this module is imported in the future. We might still need a rewrite system for generic signature queries though, in which scenario we build a "second-generation" rewrite system:

1. We originally built a convergent rewriting system from the user-written requirements of the protocol extension.
2. After finding a minimal set of rules, we convert minimal rules to requirements, and serialize the resulting generic signature.
3. In a subsequent compiler invocation, we deserialize this generic signature.
4. Then, we build a convergent rewriting system from the requirements of this generic signature; rule (8) is now one of our initial rules.

The rewrite system after step (1) is equivalent to that after step (4). That is, if we ignore **left-simplified** and **right-simplified** rules, both will have the same rules, up to permutation. (We can't completely prove this fact, because minimization is tricky. But this is the intended invariant here.)

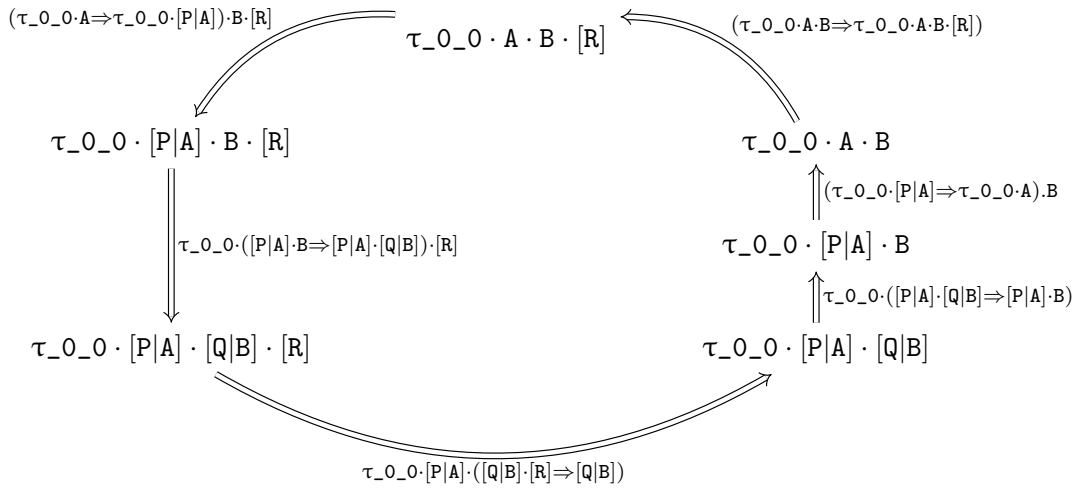
Example 18.7. Now we're going to change `Q` to add an associated requirement stating that `B` conform to `R`. The conformance requirement [τ_0_0.A.B: R] in `where` clause of our protocol extension becomes redundant as a result, because every `B` (of a type conforming to `Q`) now conforms to `R`:

```
protocol R {}

protocol Q {
  associatedtype B: R
}

protocol P {
  associatedtype A: Q
}

extension P where Self.A.B: R {}
```

The rewrite system for building the protocol extension's generic signature starts out as in the previous example, but now one more imported rule comes from protocol `Q`:

$$[\mathtt{Q}|\mathtt{B}] \cdot [\mathtt{R}] \Rightarrow [\mathtt{Q}|\mathtt{B}]$$

The conformance rule $\tau_0_0 \cdot \mathtt{A} \cdot \mathtt{B} \cdot [\mathtt{R}] \Rightarrow \tau_0_0 \cdot \mathtt{A} \cdot \mathtt{B}$ overlaps with $\tau_0_0 \cdot \mathtt{A} \Rightarrow \tau_0_0 \cdot [\mathtt{P}|\mathtt{A}]$ on the term $\tau_0_0 \cdot \mathtt{A} \cdot \mathtt{B} \cdot [\mathtt{R}]$, as before. This is an overlap of the first kind, so the original rule $\tau_0_0 \cdot \mathtt{A} \cdot \mathtt{B} \cdot [\mathtt{R}] \Rightarrow \tau_0_0 \cdot \mathtt{A} \cdot \mathtt{B}$ is marked **left-simplified**. This time though, the critical pair is trivial; both sides already reduce to $\tau_0_0 \cdot [\mathtt{P}|\mathtt{A}] \cdot [\mathtt{Q}|\mathtt{B}]$:



Example 18.8. Our next goal is to understand what happens when a type parameter conforms to two unrelated protocols that both declare an associated type with the same name:

```
protocol P1 {
  associatedtype A
}

protocol P2 {
  associatedtype A
}
```

We're going to look at this generic signature:

```
<τ_0_0 where τ_0_0: P1, τ_0_0: P2>
```

Every equivalence class of type parameters can be uniquely identified by some unbound type parameter. In the above, the type parameters $\texttt{τ_0_0.[P1]A}$ and $\texttt{τ_0_0.[P2]A}$ must therefore be equivalent to $\texttt{τ_0_0.A}$. We're going to see how this works out in the rewrite system. Here is the convergent rewriting system for the above generic signature:

$[\texttt{P1}] \cdot \texttt{A} \Rightarrow [\texttt{P1}|\texttt{A}]$	(1)
$[\texttt{P2}] \cdot \texttt{A} \Rightarrow [\texttt{P2}|\texttt{A}]$	(2)
$\texttt{τ_0_0} \cdot [\texttt{P1}] \Rightarrow \texttt{τ_0_0}$	(3)
$\texttt{τ_0_0} \cdot [\texttt{P2}] \Rightarrow \texttt{τ_0_0}$	(4)
$\texttt{τ_0_0} \cdot \texttt{A} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}]$	($\diamond$5)
$\texttt{τ_0_0} \cdot [\texttt{P2}|\texttt{A}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}]$	($\diamond$6)

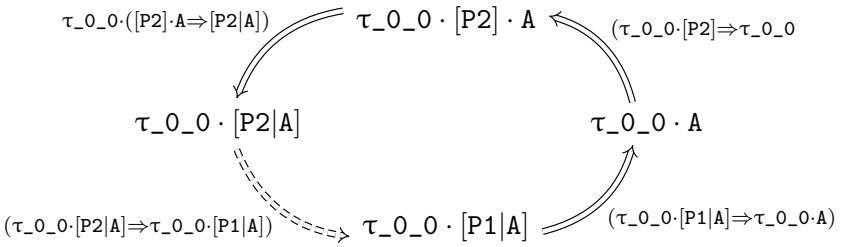
Rules (1) and (2) are imported from `P1` and `P2`, and rules (3) and (4) correspond to the conformance requirements $[\texttt{τ_0_0: P1}]$ and $[\texttt{τ_0_0: P2}]$. Completion also adds two additional rules. Rule ($\diamond$5) is added when resolving the overlap between rule (3) and rule (1), exactly as in Example 18.5.

Rule (4) overlaps with rule (2) on the term $\texttt{τ_0_0} \cdot [\texttt{P2}] \cdot \texttt{A}$:

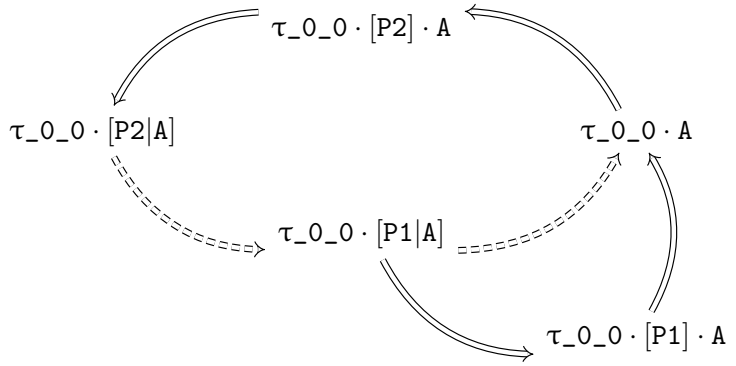
$$\begin{array}{l} \texttt{τ_0_0}.[\texttt{P2}] \\ \qquad [\texttt{P2}] \cdot \texttt{A} \end{array}$$

This gives us rule ($\diamond$6), which has a new form we haven't seen before. One side of the critical pair reduces to $\texttt{τ_0_0} \cdot [\texttt{P2}|\texttt{A}]$, and the other reduces to $\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}]$ via rule (5).

We record a rewrite loop defining rule ($\diamond 6$) via (2), (4) and (5):



We can "glue" the rewrite loops defining rule ($\diamond 5$) and ($\diamond 6$) together into a single diagram:



The two dashed arrows are rules ($\diamond 5$) and ($\diamond 6$). The solid arrows indicate a rewrite path joining `τ_0_0·[P1|A]` with `τ_0_0·[P2|A]` involving the initial rules only. This rewrite path corresponds to the following derivation:

1. [`τ_0_0: P1`] (Conf)
2. [`τ_0_0.[P1]A == τ_0_0.A`] (AssocBind 1)
3. [`τ_0_0: P2`] (Conf)
4. [`τ_0_0.[P2]A == τ_0_0.A`] (AssocBind 3)
5. [`τ_0_0.A == τ_0_0.[P2]A`] (Sym 4)
6. [`τ_0_0.[P1]A == τ_0_0.[P2]A`] (Trans 2 5)

Indeed, we see that the equivalence class with reduced type `τ_0_0.[P1]A` contains three type parameters. Here they are with corresponding terms, in type parameter order. In this case, the reduced type and the reduced term coincide:

Type	**Term**
`τ_0_0.[P1]A`	`τ_0_0 · [P1|A]`
`τ_0_0.[P2]A`	`τ_0_0 · [P2|A]`
`τ_0_0.A`	`τ_0_0 · A`

18.3. More Critical Pairs

Example 18.9. Consider this protocol inheritance hierarchy, where `Bot` inherits from `Mid`, which inherits from `Top`, and `Mid` declares an associated type:

```
protocol Top {}

protocol Mid: Top {
  associatedtype A
}

protocol Bot: Mid {}
```

We're going to examine the rewrite system for the generic signature $G_{\texttt{Bot}}$, made up rules imported from `Mid` and `Bot`, and the local rules (as before, we omit the identity conformance rules):

$[\texttt{Mid}] \cdot \texttt{A} \Rightarrow [\texttt{Mid}|\texttt{A}]$	(1)
$[\texttt{Mid}] \cdot [\texttt{Top}] \Rightarrow [\texttt{Mid}]$	(2)
$[\texttt{Bot}] \cdot \texttt{A} \Rightarrow [\texttt{Bot}|\texttt{A}]$	(3)
$[\texttt{Bot}] \cdot [\texttt{Mid}] \Rightarrow [\texttt{Bot}]$	(4)
$[\texttt{Bot}] \cdot [\texttt{Mid}|\texttt{A}] \Rightarrow [\texttt{Bot}|\texttt{A}]$	($\diamond$5)
$[\texttt{Bot}] \cdot [\texttt{Top}] \Rightarrow [\texttt{Bot}]$	($\diamond$6)
$\tau_0_0 \cdot [\texttt{Bot}] \Rightarrow \tau_0_0$	(7)
$\tau_0_0 \cdot [\texttt{Mid}] \Rightarrow \tau_0_0$	($\diamond$8)
$\tau_0_0 \cdot [\texttt{Top}] \Rightarrow \tau_0_0$	(*9)
$\tau_0_0 \cdot \texttt{A} \Rightarrow \tau_0_0 \cdot [\texttt{Bot}|\texttt{A}]$	(*10)
$\tau_0_0 \cdot [\texttt{Mid}|\texttt{A}] \Rightarrow \tau_0_0 \cdot [\texttt{Bot}|\texttt{A}]$	($\diamond$11)

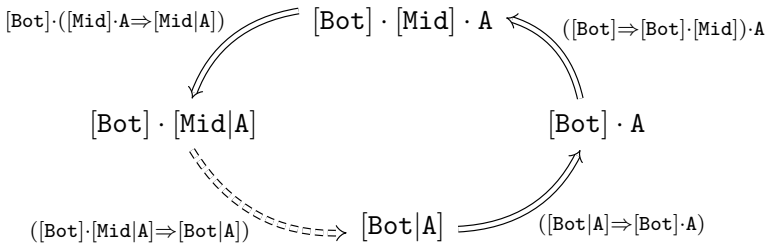
We're going to focus on a handful of interesting rules:

- While `Bot` does not declare any associated types, it inherits `A` from `Mid`; rule (3) is the associated type rule for `A`.
- Rules ($\diamond$5) and ($\diamond$11) are added by completion as are a consequence of the associated type rule.
- Rules ($\diamond$6), ($\diamond$8) and (*9) are also from completion. They express the "transitive" conformance requirements $[\texttt{Self: Top}]_{\texttt{Bottom}}$, $[\tau_0_0\texttt{: Top}]$ and $[\tau_0_0\texttt{: Mid}]$.

Rule (4) overlaps with rule (1) on the term $[\texttt{Bot}] \cdot [\texttt{Mid}] \cdot \texttt{A}$:

$$\begin{array}{l} [\texttt{Bot}] \cdot [\texttt{Mid}] \\ \qquad [\texttt{Mid}] \cdot \texttt{A} \end{array}$$

Resolving this critical pair records a rewrite loop defining rule ($\diamond 5$) via (1), (3) and (4):

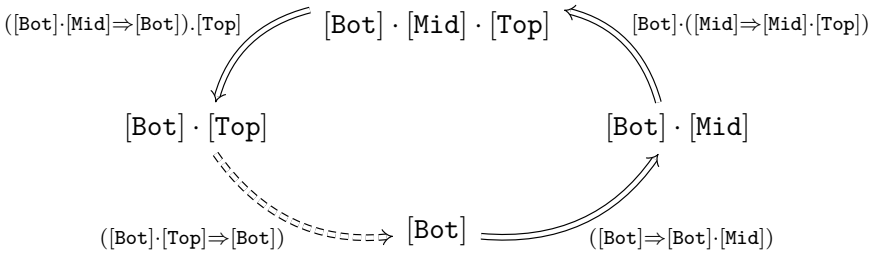


Rule ($\diamond 5$) says, if a term conforming to `Bot` is followed by $[\texttt{Mid}|\texttt{A}]$, we can reduce it to $[\texttt{Bot}|\texttt{A}]$.

Rule (4) overlaps with rule (3) on the term $[\texttt{Bot}] \cdot [\texttt{Mid}] \cdot [\texttt{Top}]$:

$$\begin{array}{l} [\texttt{Bot}] \cdot [\texttt{Mid}] \\ \qquad [\texttt{Mid}] \cdot [\texttt{Top}] \end{array}$$

Resolving this critical pair records a rewrite loop defining rule ($\diamond 6$) via rule (2) and (4):

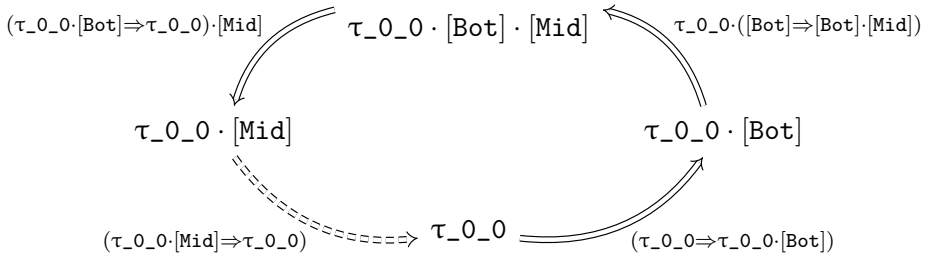


Rule ($\diamond 6$) says, if a term conforms to `Bot`, it also conforms to `Top`. In general, after completion, rewrite rules of the form $[\texttt{P}] \cdot [\texttt{Q}] \Rightarrow [\texttt{P}]$ will encode the transitive closure of the protocol inheritance relation.

This completes the rewrite system for `Bot`. Moving on to $G_{\texttt{Bot}}$, we see that rule (7) overlaps with rule (4) on the term $\texttt{τ_0_0} \cdot [\texttt{Bot}] \cdot [\texttt{Mid}]$:

$$\begin{array}{l} \texttt{τ_0_0} \cdot [\texttt{Bot}] \\ \qquad [\texttt{Bot}] \cdot [\texttt{Mid}] \end{array}$$

Resolving this critical pair defines rule (⋄8) via rules (4), (7) and (⋄8):



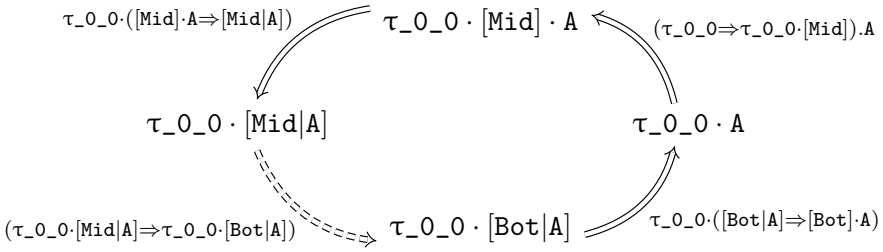
Rule (⋄8) is the derived requirement [τ_0_0: Mid]. Rule (7) overlaps with rule (5) in the same way, defining rule (*9), which is the derived requirement [τ_0_0: Top]. Once again, we see the transitive closure of the protocol inheritance relation being computed.

Rule (7) overlaps with rule (3) on the term τ_0_0 · [Bot] · A, defining rule (*10), by the same general principle as in Example 18.5.

Finally, rule (⋄8) overlaps with rule (1) on the term τ_0_0 · [Mid] · A:

τ_0_0 · [Mid]
[Mid] · A

Resolving this critical pair defines rule (⋄11) via rule (1), (3) and (⋄8):



In this example, the relationship between type parameters and terms is more subtle than we've previously seen, because the two distinct associated type symbols [Mid|A] and [Bot|A] both correspond to the same associated type *declaration*. On one hand, the equivalence class of τ_0_0.[Mid]A contains one other type parameter, τ_0_0.A. However, we have *three* equivalent terms: τ_0_0 · [Mid|A], τ_0_0 · [Bot|A], and τ_0_0 · A. Not only that, but applying Algorithm 17G to the reduced type parameter τ_0_0.[Mid]A outputs τ_0_0 · [Mid|A], which is *not* reduced; the reduced term here is τ_0_0 · [Bot|A], because [Bot|A] < [Mid|A] in the reduction order (Algorithm 17A):

Type	Term
τ_0_0.[Mid]A	τ_0_0 · [Bot|A]
	τ_0_0 · [Mid|A]
τ_0_0.A	τ_0_0 · A

Now, suppose we change the declaration of `Bot` to *re-state* the associated type:

```
protocol Bot: Mid {
  associatedtype A
}
```

The rewrite system is identical; we now call rule (3) an associated type rule instead of an *inherited* associated type rule, but the rule remains the same. Our equivalence class has three type parameters and three terms, because now every associated type symbol corresponds to a declaration:

Type	Term
$\tau_0_0.[\texttt{Bot}]\texttt{A}$	$\tau_0_0 \cdot [\texttt{Bot}|\texttt{A}]$
$\tau_0_0.[\texttt{Mid}]\texttt{A}$	$\tau_0_0 \cdot [\texttt{Mid}|\texttt{A}]$
$\tau_0_0.\texttt{A}$	$\tau_0_0 \cdot \texttt{A}$

We defined Algorithm 5C of the type parameter order so that a root associated type declaration always precedes such a re-stated associated type declaration; therefore the reduced type parameter of our equivalence class remains $\tau_0_0.[\texttt{Mid}]\texttt{A}$. The reduced term is also unchanged, $\tau_0_0 \cdot [\texttt{Bot}|\texttt{A}]$. This means that re-stating an associated type (or removing a re-stated associated type) does not change either the rewrite system or reduced type relation, and in particular, has no effect on calling convention, witness table layout, or any other aspect of the ABI.

Note that because the correspondence is not immediate, the mapping of terms to type parameters must be defined in such a way to output a reduced type parameter given a reduced term, by a careful choice of associated type declaration at each step. This will be explained in Section 19.5. This minor complication could be avoided if it weren't for inherited associated type symbols, and the fact that the reduction order on associated type symbols differs from the type parameter order on associated type declarations. However, both of those behaviors have important redeeming qualities, as we will see in Section 18.5.

Example 18.10. We now ask if `f()` type checks; is `T.C.B` equivalent to `T.A`?

```
protocol S {
  associatedtype A
  associatedtype B
  associatedtype C: S where Self == Self.C.C, Self.B == Self.C.A
}

func f<T: S>(val: T.C.B) {
  let val2: T.A = val
}
```

Our protocol states three associated requirements:

$$[\texttt{Self.C: S}]_{\texttt{S}}$$
$$[\texttt{Self == Self.C.C}]_{\texttt{S}}$$
$$[\texttt{Self.B == Self.C.A}]_{\texttt{S}}$$

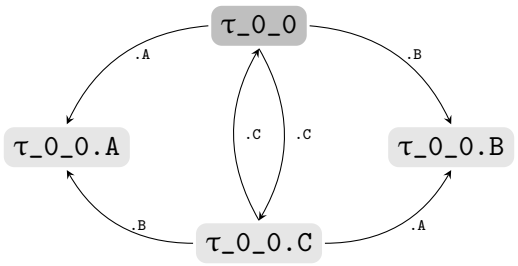
A consequence of the two same-type requirements is that `τ_0_0.A` is equivalent to `τ_0_0.C.C.A`, which is equivalent to `τ_0_0.C.B`. Here is the full derivation:

1. [`τ_0_0: S`] (Conf)
2. [`τ_0_0.C: S`] (AssocConf 1)
3. [`τ_0_0.C.B == τ_0_0.C.C.A`] (AssocSame 2)
4. [`τ_0_0 == τ_0_0.C.C`] (AssocSame 1)
5. [`τ_0_0.C.C == τ_0_0`] (Sym 4)
6. [`τ_0_0.C.C.A == τ_0_0.A`] (SameName 1 5)
7. [`τ_0_0.C.B == τ_0_0.A`] (Trans 3 6)

Our generic signature defines four infinite equivalence classes:

`τ_0_0`	`τ_0_0.A`	`τ_0_0.C`	`τ_0_0.B`
`τ_0_0.C.C`	`τ_0_0.C.B`	`τ_0_0.C.C.C`	`τ_0_0.C.A`
`τ_0_0.C.C.C.C`	`τ_0_0.C.C.A`	`τ_0_0.C.C.C.C.C`	`τ_0_0.C.C.B`
...	...	...	...

For a visual perspective, we turn to the type parameter graph of $G_{\texttt{S}}$:



Here are two concrete conforming types:

```
struct X: S {
  typealias A = Int
  typealias B = String
  typealias C = Y
}
```

```
struct Y: S {
  typealias A = String
  typealias B = Int
  typealias C = X
}
```

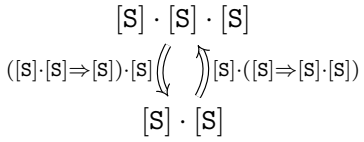
Now, let's look at our fundamental source of truth, the rewrite system. We start with an identity conformance rule, three associated type rules, and finally, three rules corresponding to user-written requirements:

$[\mathtt{S}] \cdot [\mathtt{S}] \Rightarrow [\mathtt{S}]$	(1)
$[\mathtt{S}] \cdot \mathtt{A} \Rightarrow [\mathtt{S}\vert\mathtt{A}]$	(2)
$[\mathtt{S}] \cdot \mathtt{B} \Rightarrow [\mathtt{S}\vert\mathtt{B}]$	(3)
$[\mathtt{S}] \cdot \mathtt{C} \Rightarrow [\mathtt{S}\vert\mathtt{C}]$	(4)
$[\mathtt{S}] \cdot \mathtt{C} \cdot [\mathtt{S}] \Rightarrow [\mathtt{S}] \cdot \mathtt{C}$	(5)
$[\mathtt{S}] \cdot \mathtt{C} \cdot \mathtt{A} \Rightarrow [\mathtt{S}] \cdot \mathtt{B}$	(6)
$[\mathtt{S}] \cdot \mathtt{C} \cdot \mathtt{C} \Rightarrow [\mathtt{S}]$	(7)

Completion also adds *ten* new rules; we will study a few at a time. Let's consider the first rule, $[\mathtt{S}] \cdot [\mathtt{S}] \Rightarrow [\mathtt{S}]$. Every protocol has an identity conformance rule, but in the previous examples it didn't play an important role so we ignored it. The identity conformance rule always overlaps with itself:

$$[\mathtt{S}] \cdot [\mathtt{S}]$$
$$[\mathtt{S}] \cdot [\mathtt{S}]$$

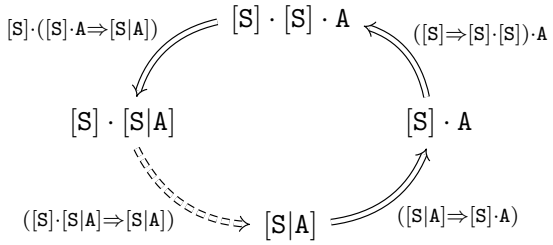
The critical pair is always trivial:



Identity conformance rules also overlap with associated type rules. For example, rule (1) overlaps with (2) on the term $[\mathtt{S}] \cdot [\mathtt{S}] \cdot \mathtt{A}$:

$$[\mathtt{S}] \cdot [\mathtt{S}]$$
$$[\mathtt{S}] \cdot \mathtt{A}$$

One side of this critical pair reduces to $[\mathtt{S}] \cdot [\mathtt{S}\vert\mathtt{A}]$, while the other reduces to $[\mathtt{S}\vert\mathtt{A}]$. We define a new rule $[\mathtt{S}] \cdot [\mathtt{S}\vert\mathtt{A}] \Rightarrow [\mathtt{S}\vert\mathtt{A}]$ and record a rewrite loop:



In the same way we get such a rule for every associated type:

$[\mathtt{S}] \cdot [\mathtt{S}|\mathtt{A}] \Rightarrow [\mathtt{S}|\mathtt{A}]$	($\diamond$ 8)
$[\mathtt{S}] \cdot [\mathtt{S}|\mathtt{B}] \Rightarrow [\mathtt{S}|\mathtt{B}]$	(*9)
$[\mathtt{S}] \cdot [\mathtt{S}|\mathtt{C}] \Rightarrow [\mathtt{S}|\mathtt{C}]$	(*10)

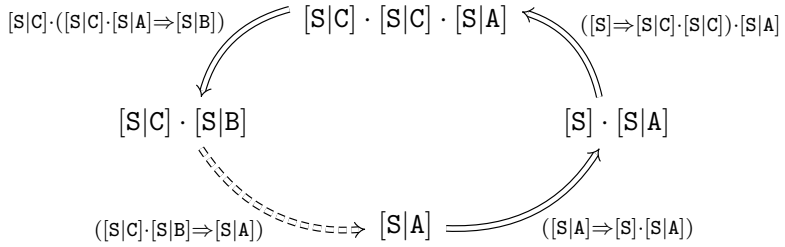
While we could have also written down similar rules in previous examples, they would not have illustrated anything of consequence. They will shortly; but first, we have a few overlaps between conformance rules and associated type rules. As in previous examples, we get the usual rules ($\diamond$ 11), ($\diamond$ 12), and ($\diamond$ 13) for reducing unbound type parameters; we also left-simplify (5), (6) and (7), replacing them with rules ($\diamond$ 14), ($\diamond$ 15), and ($\diamond$ 16):

$[\mathtt{S}|\mathtt{C}] \cdot \mathtt{A} \Rightarrow [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{A}]$	($\diamond$ 11)
$[\mathtt{S}|\mathtt{C}] \cdot \mathtt{B} \Rightarrow [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{B}]$	($\diamond$ 12)
$[\mathtt{S}|\mathtt{C}] \cdot \mathtt{C} \Rightarrow [\mathtt{S}]$	($\diamond$ 13)
$[\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}] \Rightarrow [\mathtt{S}|\mathtt{C}]$	($\diamond$ 14)
$[\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{C}] \Rightarrow [\mathtt{S}]$	($\diamond$ 15)
$[\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{A}] \Rightarrow [\mathtt{S}|\mathtt{B}]$	($\diamond$ 16)

Here is the main event: rule ($\diamond$ 15) overlaps with ($\diamond$ 16) on the term $[\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{A}]$:

$$\begin{array}{l} [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{C}] \\ \qquad [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{A}] \end{array}$$

The right-hand side of this critical pair reduces the overlap term $[\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{A}]$ with rule ($\diamond$ 15), which gives us $[\mathtt{S}] \cdot [\mathtt{S}|\mathtt{A}]$. Previously, terms for bound type parameters would begin with *either* a protocol symbol or an associated type symbol, but not both; the associated type symbol already encodes the protocol. Indeed, this funny-looking term reduces to just $[\mathtt{S}|\mathtt{A}]$ via rule ($\diamond$ 8), defined as a consequence of that mysterious identity conformance rule (1).



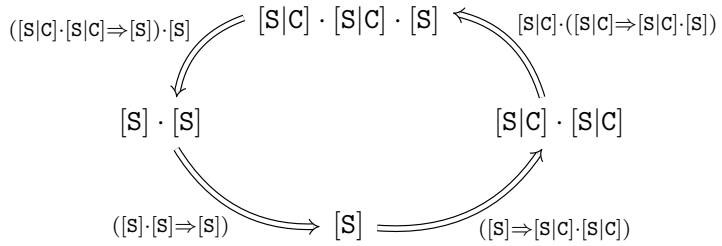
At last, and as predicated, we get the rule that proves $G_{\mathtt{S}} \vdash [\tau_0_0.\mathtt{A} == \tau_0_0.\mathtt{C.B}]$:

$$[\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{B}] \Rightarrow [\mathtt{S}|\mathtt{A}] \qquad (\diamond 17)$$

Now the *encore*: rule ($\diamond 15$) overlaps with rule ($\diamond 14$) on the term $[\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}]$:

$$\begin{array}{l} [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}|\mathtt{C}] \\ \qquad [\mathtt{S}|\mathtt{C}] \cdot [\mathtt{S}] \end{array}$$

This critical pair resolves trivially, by way of the identity conformance rule:



If the identity conformance rule had not been part of that initial set, this last critical pair would *define* the identity conformance rule, and we would obtain the same rewrite system in the end. We don't need to explicitly add the identity conformance rule, after all. There is a practical consideration though. By making this rule part of the initial set, and crucially, marking it **permanent**, we remove it from consideration in the rewrite system minimization algorithm. This cuts out a lot of unnecessary work.

Protocol `S` is an interesting test case demonstrating the rewrite system's ability to discover non-trivial identities. It originates from a developer's bug report in 2020 [181]; at the time, it broke the `GenericSignatureBuilder`'s minimization algorithm. Amusingly, the Rust compiler's generics implementation is unable to prove the derived requirement:

```
trait S {
  type A;
  type B;
  type C: S<A = Self::B, C = Self>;
}

fn f<T: S>(val: <T::C as S>::B) {
  let val2: T::A = val;
  // note: expected associated type `<T as S>::A`
  // found associated type `<<T as S>::C as S>::B`
}
```

18.4. Tietze Transformations

Both the Knuth-Bendix completion and the rule simplification algorithms preserve the equivalence relation on terms. When completion adds a new rewrite rule, it is because the corresponding pair of terms are already joined by a rewrite path. There is an analogous guarantee when rule simplification deletes a rewrite rule: the two terms are known to be joined by at least one other rewrite path that does not involve this rule. These transformations knead the reduction relation into a better form, while the equivalence relation on terms remains completely determined by the rewrite rules of our initial monoid presentation.

Mathematically speaking, both algorithms are defined as a transformation on monoid presentations. This transformation decomposes into a composition of sequential steps, where at each step the monoid presentation is changed in such a way that monoid isomorphism is preserved. So far, we've seen two of the four isomorphism-preserving transformations below, so named after Heinrich Tietze, who introduced them in 1908:

Definition 18.11. Let $\langle A \,|\, R \rangle$ be a finitely-presented monoid. An *elementary Tietze transformation*, or just Tietze transformation, is one of the following:

1. (Adding a rewrite rule) If a pair of terms u, $v \in A^*$ are already joined by a rewrite path from u to v—that is, if $u \sim v$ as elements of $\langle A \,|\, R \rangle$—we can add (u, v):

   $$\langle A \,|\, R \cup \{(u, v)\} \rangle$$

2. (Removing a rewrite rule) If $(u, v) \in R$ and we have a rewrite path from u to v that does not contain the rewrite step $x(u \Rightarrow v)y$ or $x(v \Rightarrow u)y$ for any x, $y \in A^*$, we can remove (u, v):

   $$\langle A \,|\, R \setminus \{(u, v)\} \rangle$$

3. (Adding a generator) If a is some symbol distinct from all other symbols of A, and $t \in A^*$ is any term, we can simultaneously add a and make it equivalent to t:

   $$\langle A \cup \{a\} \,|\, R \cup \{(t, a)\} \rangle$$

4. (Removing a generator) If $a \in A$, $(t, a) \in R$ for some term t not involving a, and no other $(u, v) \in R$ has a term u or v involving a, we can simultaneously remove a and (t, a):

   $$\langle A \setminus \{a\} \,|\, R \setminus \{(t, a)\} \rangle$$

The following is the key result here: two finitely-presented monoids are isomorphic if and only if they are *Tietze-equivalent*, meaning we can obtain one from the other by a finite sequence of elementary Tietze transformations.

Some finer points:

- For every Tietze transformation, there is a complementary transformation which undoes the change; in this way (1) and (2) are inverses, and similarly (3) and (4).
- We already know that changing the order of the terms in a rewrite rule—replacing $(u, v) \in R$ with (v, u)—does not change the set of rewrite steps generated, and thus presents the same monoid. Now we see this operation is actually a composition of two elementary Tietze transformations; we first add (v, u), and remove (u, v).
- Some definitions of (4) drop the condition that the removed symbol a not appear in any other rewrite rule $(u, v) \in R$. Our restriction does not cost us any generality, because if a occurs in any other rewrite rule, we can always first perform a series of Tietze transformations to eliminate such occurrences: for each such (u, v), we replace occurrences of a with t in u and v, add the new rewrite rule, and finally remove the old rule (u, v). However, we must still require that a not occur in t; otherwise, replacing a with t does not eliminate all occurrences of a.

Associated type symbols. Tietze transformations give us a new way to understand associated type symbols. The ultimate equivalence between rewrite systems built from user-written requirements and minimal requirements, shown in Section 18.2, means we could have defined Algorithms 17G and 17H to only ever build terms from protocol and name symbols. After completion, we end up with the same rewrite system, it just takes a little bit more work to get there. In this setup, among the initial rewrite rules, the only occurrences of associated type symbols are on the right-hand sides of associated type rules:

$$[\texttt{P}] \cdot \texttt{A} \Rightarrow [\texttt{P}|\texttt{A}]$$

Thus, we can understand the associated type rules as having been added by a sequence of Tietze transformations of the third kind, applied to some monoid presentation. This "primordial" monoid presentation involves only protocol and name symbols, and it encodes the same monoid that ours does, up to isomorphism. So why do we bother with associated type symbols at all? In the next section, we will see the answer has to do with the interaction between recursive conformance requirements and completion.

Further discussion. The elementary Tietze transformations are "higher dimensional" rewrite steps, in that they define an edge relation on a graph whose vertices are *monoid presentations*. A path in this graph witnesses the fact that the source and destination define an isomorphic pair of monoids. The problem of deciding if two presentations are joined by a path is the *monoid isomorphism problem*. Like the word problem, this is of course undecidable in general.

Tietze transformations are fundamental to the study of *combinatorial group theory*, and are described in any book on the subject, such as [182]. Recall that in a group presentation, a rewrite rule (u, v) can always be written as (uv^{-1}, ε), with the identity element on the right hand side. The term uv^{-1} is called a *relator*; a set of relators takes the place of rewrite rules in a group presentation. Tietze transformations of monoid presentations are described in [154] and [183].

18.5. Recursive Conformances

From previous examples, it may look like completion essentially enumerates *all* derived requirements, but this cannot be true in general. We know from Section 12.3 that recursive conformance requirements allow us to define generic signatures with an infinite theory of derived requirements. If the compiler is to allow such generic signatures, this infinite set of derived requirements must be encoded by a finite number of rewrite rules.

We will see that with recursive conformance requirements, successful termination of the Knuth-Bendix algorithm depends both on the existence of associated type symbols (and rules), and our choice of reduction order. This furnishes the concrete example for a scenario mentioned in Section 16.4.

Example 18.12. Our first example is the protocol `N` that we encountered several times already, most recently in Section 16.3:

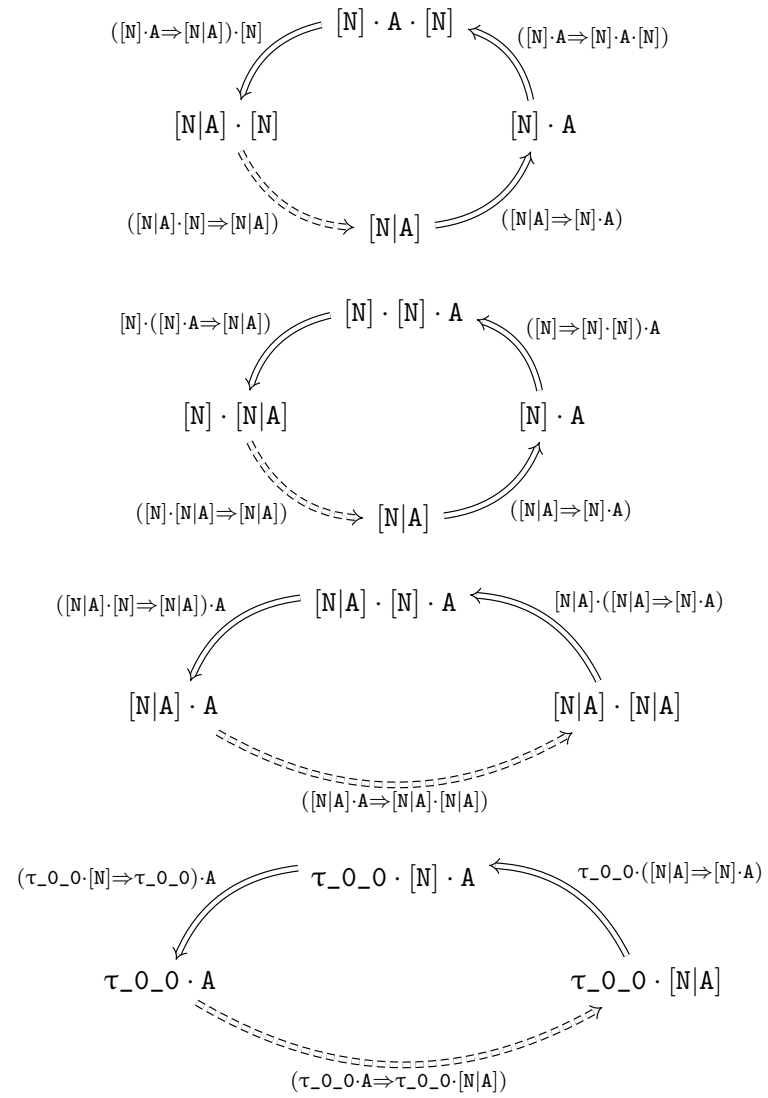
```
protocol N {
  associatedtype A: N
}
```

Let's look at the convergent rewriting system for $G_{\texttt{N}}$:

$[\texttt{N}] \cdot [\texttt{N}] \Rightarrow [\texttt{N}]$	(1)
$[\texttt{N}] \cdot \texttt{A} \Rightarrow [\texttt{N}|\texttt{A}]$	(2)
$[\texttt{N}] \cdot \texttt{A} \cdot [\texttt{N}] \Rightarrow [\texttt{N}] \cdot \texttt{A}$	(3)
$[\texttt{N}|\texttt{A}] \cdot [\texttt{N}] \Rightarrow [\texttt{N}|\texttt{A}]$	($\diamond$4)
$[\texttt{N}] \cdot [\texttt{N}|\texttt{A}] \Rightarrow [\texttt{N}|\texttt{A}]$	($\diamond$5)
$[\texttt{N}|\texttt{A}] \cdot \texttt{A} \Rightarrow [\texttt{N}|\texttt{A}] \cdot [\texttt{N}|\texttt{A}]$	($\diamond$6)
$\tau_0_0 \cdot [\texttt{N}] \Rightarrow \tau_0_0$	(7)
$\tau_0_0 \cdot \texttt{A} \Rightarrow \tau_0_0 \cdot [\texttt{N}|\texttt{A}]$	($\diamond$8)

Rules ($\diamond$4), ($\diamond$5), ($\diamond$6) and ($\diamond$8) are defined by the rewrite loops shown in Figure 18.1. Rule (3) is marked **left-simplified** because its left-hand side $[\texttt{N}] \cdot \texttt{A} \cdot [\texttt{N}]$ can always be reduced by the rewrite path $([\texttt{N}] \cdot \texttt{A} \Rightarrow [\texttt{N}|\texttt{A}]).[\texttt{N}] \circ ([\texttt{N}|\texttt{A}] \cdot [\texttt{N}] \Rightarrow [\texttt{N}|\texttt{A}])$.

Figure 18.1.: Critical pairs in $G_{\mathtt{N}}$



In our rewrite system, the type parameter terms have $\texttt{τ_0_0}$ as the first symbol, and either $\texttt{A}$ or $[\texttt{N}|\texttt{A}]$ as each subsequent symbol. Any such term reduces to a term of the same length:

$$\texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}] \cdot \texttt{A} \cdot \texttt{A} \to \texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}] \cdot [\texttt{N}|\texttt{A}] \cdot [\texttt{N}|\texttt{A}]$$

The reduced terms are those that do not contain $\texttt{A}$; that is, a reduced term has the form $\texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}]^n$. Also, every type parameter of $G_{\texttt{N}}$ conforms to $\texttt{N}$. We have two infinite families of derived conformance requirements, where the subject types are the bound and unbound type parameters from each equivalence class:

$[\texttt{τ_0_0.[N]A: N}]$
$[\texttt{τ_0_0.[N]A.[N]A: N}]$
$[\texttt{τ_0_0.[N]A.[N]A.[N]A: N}]$
...
$[\texttt{τ_0_0.A: N}]$
$[\texttt{τ_0_0.A.A: N}]$
$[\texttt{τ_0_0.A.A.A: N}]$
...

Each derived requirement corresponds to an equivalence of terms $t \cdot [\texttt{N}] \sim t$, where t is a type parameter term. We can explicitly construct rewrite paths that witness these equivalences. Let's temporarily denote the positive rewrite step for rules $(\diamond 4)$, $(\diamond 6)$ and $(\diamond 8)$ by α, β and γ respectively:

$$\alpha := ([\texttt{N}|\texttt{A}] \cdot [\texttt{N}] \Rightarrow [\texttt{N}|\texttt{A}])$$
$$\beta := ([\texttt{N}|\texttt{A}] \cdot \texttt{A} \Rightarrow [\texttt{N}|\texttt{A}] \cdot [\texttt{N}|\texttt{A}])$$
$$\gamma := (\texttt{τ_0_0} \cdot \texttt{A} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}])$$

For a derived requirement with bound subject type, $t = \texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}]^n$. We can rewrite $t \cdot [\texttt{N}]$ to t with a single rewrite step, using rule α to eliminate the suffix $[\texttt{N}]$ while leaving the rest of the term unchanged:

$$\texttt{τ_0_0} \triangleleft \alpha$$
$$\texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}] \triangleleft \alpha$$
$$\texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}] \cdot [\texttt{N}|\texttt{A}] \triangleleft \alpha$$
$$\ldots$$

For those derived requirements with unbound subject types, $t = \texttt{τ_0_0·A}$. To rewrite $t \cdot [\texttt{N}]$ to t, we first reduce $t \to t'$, eliminate $[\texttt{N}]$ with rule α, and reverse the reduction to obtain t.

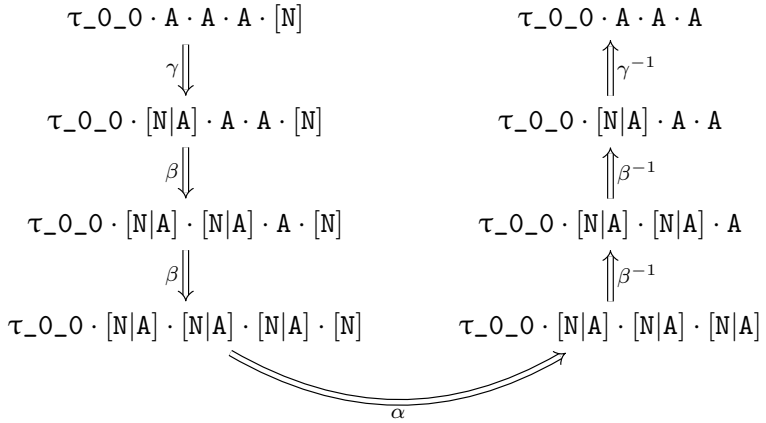
The reduction is given by a path p_n, where each p_n rewrites $\texttt{τ_0_0} \cdot \texttt{A}^n \to \texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}]^n$:

$$
\begin{aligned}
&p_1 := \gamma \\
&p_2 := (\gamma \rhd \texttt{A} \cdot [\texttt{N}]) \circ (\texttt{τ_0_0} \lhd \beta \rhd [\texttt{N}]) \\
&p_3 := (\gamma \rhd \texttt{A} \cdot \texttt{A} \cdot [\texttt{N}]) \circ (\texttt{τ_0_0} \lhd \beta \rhd \texttt{A} \cdot [\texttt{N}]) \circ (\texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}] \lhd \beta \rhd [\texttt{N}]) \\
&\ldots
\end{aligned}
$$

Thus, the derived requirements with unbound subject types correspond to the following rewrite paths:

$$
\begin{aligned}
&(p_1 \rhd [\texttt{N}]) \circ (\texttt{τ_0_0} \lhd \alpha) \circ p_1^{-1} \\
&(p_2 \rhd [\texttt{N}]) \circ (\texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}] \lhd \alpha) \circ p_2^{-1} \\
&(p_3 \rhd [\texttt{N}]) \circ (\texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}] \cdot [\texttt{N}|\texttt{A}] \lhd \alpha) \circ p_3^{-1} \\
&\ldots
\end{aligned}
$$

For example, $\texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{A} \cdot \texttt{A} \cdot [\texttt{N}]$ and $\texttt{τ_0_0} \cdot \texttt{A} \cdot \texttt{A} \cdot \texttt{A}$ both reduce to $\texttt{τ_0_0} \cdot [\texttt{N}|\texttt{A}] \cdot [\texttt{N}|\texttt{A}] \cdot [\texttt{N}|\texttt{A}]$; we can join them with this rewrite path:



Thus, our eight rules encode two infinite families of derived requirements.

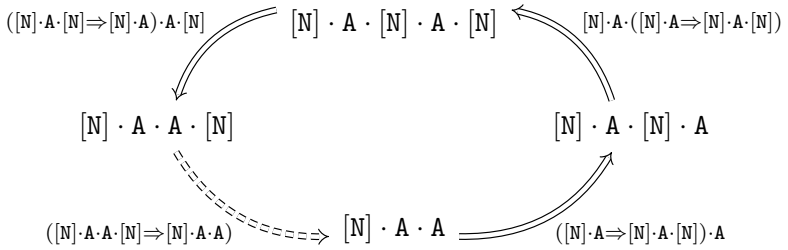
Now, we will show that the associated type rule $[\texttt{N}] \cdot \texttt{A} \Rightarrow [\texttt{N}|\texttt{A}]$ was essential. Let's start over with our the initial rules, but delete the symbol $[\texttt{N}|\texttt{A}]$ and rule (2) before attempting completion. Only two rules remain:

$[\texttt{N}] \cdot [\texttt{N}] \Rightarrow [\texttt{N}]$	(1)
$[\texttt{N}] \cdot \texttt{A} \cdot [\texttt{N}] \Rightarrow [\texttt{N}] \cdot \texttt{A}$	(3)

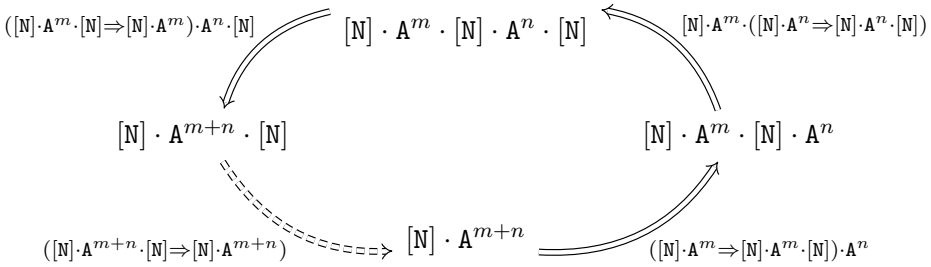
We see that rule (3) rule overlaps with itself on the term $[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$:

$$\begin{array}{l} [\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \\ \qquad [\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \end{array}$$

Resolving this critical pair introduces a new rule $[\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \Rightarrow [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A}$:



After adding this rule, we check for overlaps again. The new rule overlaps with rule (3) on $[\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$, and also with itself on $[\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}]$. Rule (3) also overlaps with the new rule on $[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}]$. It is apparent that resolving those critical pairs will introduce new rules, and this process will never end. We get an infinite family of critical pairs, indexed by m, $n \in \mathbb{N}$:



These critical pairs define an infinite sequence of rewrite rules:

$$\begin{array}{l} [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \Rightarrow [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \\ [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \Rightarrow [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \\ [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \Rightarrow [\mathtt{N}] \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \\ \ldots \end{array}$$

Adding the symbol $[\mathtt{N}|\mathtt{A}]$ together with the rule $[\mathtt{N}] \cdot \mathtt{A} \Rightarrow [\mathtt{N}|\mathtt{A}]$ is a Tietze transformation, so we know it does not change the equivalence relation on terms. However, it ensures convergence. Convergence also depends on associated type symbols preceding name symbols, so $[\mathtt{N}|\mathtt{A}] < \mathtt{A}$. Otherwise, we once again end up with an infinite sequence of

rewrite rules:

$$[\texttt{N}|\texttt{A}] \cdot \texttt{A} \cdot [\texttt{N}] \Rightarrow [\texttt{N}|\texttt{A}] \cdot \texttt{A}$$
$$[\texttt{N}|\texttt{A}] \cdot \texttt{A} \cdot \texttt{A} \cdot [\texttt{N}] \Rightarrow [\texttt{N}|\texttt{A}] \cdot \texttt{A} \cdot \texttt{A}$$
$$[\texttt{N}|\texttt{A}] \cdot \texttt{A} \cdot \texttt{A} \cdot \texttt{A} \cdot [\texttt{N}] \Rightarrow [\texttt{N}|\texttt{A}] \cdot \texttt{A} \cdot \texttt{A} \cdot \texttt{A}$$
$$\ldots$$

Let's summarize this example using monoid presentations and Tietze transformations. We write n, a, b instead of $[\texttt{N}]$, $\texttt{A}$, $[\texttt{N}|\texttt{A}]$. Together with the identity conformance rule, the user-written requirement $[\texttt{Self.A: N}]_{\texttt{N}}$, with an unbound type parameter as the subject type, defines this presentation over the alphabet a, n:

$$M_1 := \langle a, n \,|\, nn \sim n,\ nan \sim na \rangle$$

We saw that completion does not terminate on M_1; abstractly, we can understand it as producing an *infinite* convergent presentation M_1', containing a family of rewrite rules parameterized by $i \in \mathbb{N}$:

$$M_1' := \langle a, n \,|\, na^i n \sim na^i \rangle$$

To ensure convergence, we added the symbol b and its defining rewrite rule $na \sim b$ to M_1; a Tietze transformation which gave us the presentation M_2 over a, b, n:

$$M_2 := \langle a, b, n \,|\, nn \sim n,\ nan \sim na,\ na \sim b \rangle$$

Performing completion on M_2 with the reduction order $n < b < a$ succeeds after adding the two rewrite rules $bn \sim b$ and $ba \sim bb$, giving us the finite convergent presentation M_2':

$$M_2' := \langle a, b, n \,|\, nn \sim n,\ nan \sim na,\ na \sim b,\ bn \sim b,\ ba \sim bb \rangle$$

Rule simplification removes the rewrite rule $nan \sim na$, giving us a reduced presentation:

$$M_2'' := \langle a, b, n \,|\, nn \sim n,\ na \sim b,\ bn \sim b,\ ba \sim bb \rangle$$

We haven't seen rewrite system minimization yet, but to minimize M_2'', we eliminate all rewrite rules added by completion except for $bn \sim b$, which we need because the original rewrite rule $nan \sim na$ was deleted. This leaves us with M_3:

$$M_3 := \langle a, b, n \,|\, nn \sim n,\ na \sim b,\ bn \sim b \rangle$$

The rule $bn \sim b$ is the minimal conformance requirement $[\texttt{Self.[N]A: N}]_{\texttt{N}}$ with a bound type parameter as the subject type. All of the above presentations define the *same* monoid, up to isomorphism. Furthermore, if we perform completion on M_3, we get M_2''.

Example 18.13. Let's say we inherit from `N`, and impose our own recursive conformance requirement on `A`:

```
protocol Q: N where A: Q {}
```

Even though protocol `Q` does not declare an associated type named `A`, we define an associated type symbol $[\texttt{Q}|\texttt{A}]$ and corresponding rule $[\texttt{Q}] \cdot \texttt{A} \Rightarrow [\texttt{Q}|\texttt{A}]$. We will see that convergence is contingent on this rule.

The generic signature $G_{\texttt{Q}}$ has all the derived requirements of $G_{\texttt{N}}$, and adds two more infinite families of conformances to `Q`:

$$\begin{array}{l}
\hline
[\tau_0_0.[\texttt{N}]\texttt{A}: \texttt{Q}] \\
[\tau_0_0.[\texttt{N}]\texttt{A}.[\texttt{N}]\texttt{A}: \texttt{Q}] \\
[\tau_0_0.[\texttt{N}]\texttt{A}.[\texttt{N}]\texttt{A}.[\texttt{N}]\texttt{A}: \texttt{Q}] \\
\ldots \\
\hline
[\tau_0_0.\texttt{A}: \texttt{Q}] \\
[\tau_0_0.\texttt{A}.\texttt{A}: \texttt{Q}] \\
[\tau_0_0.\texttt{A}.\texttt{A}.\texttt{A}: \texttt{Q}] \\
\ldots \\
\hline
\end{array}$$

Listing 18.1 shows the convergent rewrite system for `Q`, which imports rules from protocol `N`. Completion discovers the following critical pairs; we will be content to just summarize because they are similar to previous examples:

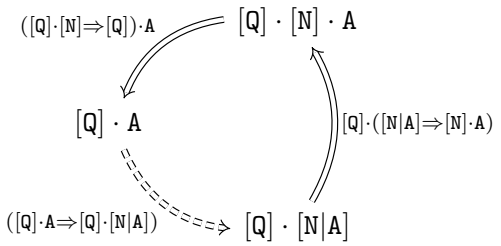
- Rule (6) overlaps with (7) on $[\texttt{Q}] \cdot [\texttt{Q}] \cdot \texttt{A}$. We define rule ($\diamond$10) (this general principle was established in Example 18.10).
- Rule (8) overlaps with (2) on $[\texttt{Q}] \cdot [\texttt{N}] \cdot \texttt{A}$. We define rule ($\diamond$11) (Example 18.9).
- Rule (9) overlaps with (7) on $[\texttt{Q}] \cdot \texttt{A} \cdot [\texttt{Q}]$. We define rule ($\diamond$12), and mark rule (9) as **left-simplified** (Example 18.6).
- Rule ($\diamond$12) overlaps with (8) on $[\texttt{Q}|\texttt{A}] \cdot [\texttt{Q}] \cdot [\texttt{N}]$. We define rule ($\diamond$13).
- Rule ($\diamond$12) overlaps with (7) on $[\texttt{Q}|\texttt{A}] \cdot [\texttt{Q}] \cdot \texttt{A}$. We define rule ($\diamond$14) (Example 18.5).
- Rule ($\diamond$13) overlaps with (2) on $[\texttt{Q}|\texttt{A}] \cdot [\texttt{N}] \cdot \texttt{A}$. We define rule ($\diamond$15).

Here's a question. What if we only had associated type symbols that directly correspond to associated type declarations? That is, what if we got rid of the symbol $[\texttt{Q}|\texttt{A}]$, together with rule (7), the inherited associated type rule $[\texttt{Q}] \cdot \texttt{A} \Rightarrow [\texttt{Q}|\texttt{A}]$? The resulting monoid would again be equivalent, because this is a Tietze transformation. However, we will see that completion fails.

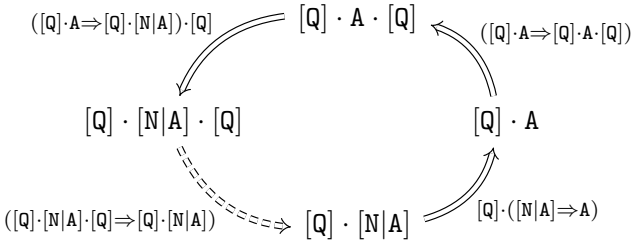
Listing 18.1.: Rewrite system for protocol `Q`

$[\mathtt{N}] \cdot [\mathtt{N}] \Rightarrow [\mathtt{N}]$	(1)
$[\mathtt{N}] \cdot \mathtt{A} \Rightarrow [\mathtt{N}\vert\mathtt{A}]$	(2)
$[\mathtt{N}\vert\mathtt{A}] \cdot [\mathtt{N}] \Rightarrow [\mathtt{N}\vert\mathtt{A}]$	(3)
$[\mathtt{N}] \cdot [\mathtt{N}\vert\mathtt{A}] \Rightarrow [\mathtt{N}\vert\mathtt{A}]$	($\diamond$ 4)
$[\mathtt{N}\vert\mathtt{A}] \cdot \mathtt{A} \Rightarrow [\mathtt{N}\vert\mathtt{A}] \cdot [\mathtt{N}\vert\mathtt{A}]$	($\diamond$ 5)
$[\mathtt{Q}] \cdot [\mathtt{Q}] \Rightarrow [\mathtt{Q}]$	(6)
$[\mathtt{Q}] \cdot \mathtt{A} \Rightarrow [\mathtt{Q}\vert\mathtt{A}]$	(7)
$[\mathtt{Q}] \cdot [\mathtt{N}] \Rightarrow [\mathtt{Q}]$	(8)
$[\mathtt{Q}] \cdot \mathtt{A} \cdot [\mathtt{Q}] \Rightarrow [\mathtt{Q}] \cdot \mathtt{A}$	(9)
$[\mathtt{Q}] \cdot [\mathtt{Q}\vert\mathtt{A}] \Rightarrow [\mathtt{Q}\vert\mathtt{A}]$	($\diamond$ 10)
$[\mathtt{Q}] \cdot [\mathtt{N}\vert\mathtt{A}] \Rightarrow [\mathtt{Q}\vert\mathtt{A}]$	($\diamond$ 11)
$[\mathtt{Q}\vert\mathtt{A}] \cdot [\mathtt{Q}] \Rightarrow [\mathtt{Q}\vert\mathtt{A}]$	($\diamond$ 12)
$[\mathtt{Q}\vert\mathtt{A}] \cdot [\mathtt{N}] \Rightarrow [\mathtt{Q}\vert\mathtt{A}]$	($\diamond$ 13)
$[\mathtt{Q}\vert\mathtt{A}] \cdot \mathtt{A} \Rightarrow [\mathtt{Q}\vert\mathtt{A}] \cdot [\mathtt{Q}\vert\mathtt{A}]$	($\diamond$ 14)
$[\mathtt{Q}\vert\mathtt{A}] \cdot [\mathtt{N}\vert\mathtt{A}] \Rightarrow [\mathtt{Q}\vert\mathtt{A}] \cdot [\mathtt{Q}\vert\mathtt{A}]$	($\diamond$ 15)

Rule (8) still overlaps with rule (2), but instead of rule ($\diamond 11$), we now get a rule $[\mathtt{Q}] \cdot \mathtt{A} \Rightarrow [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}]$ when we resolve this critical pair:



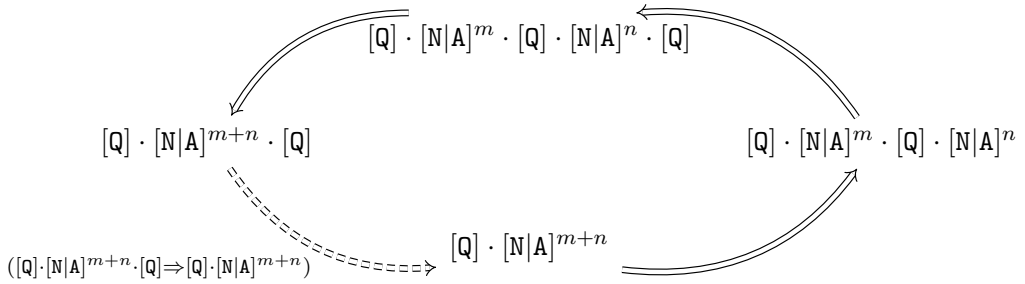
Rule (9) now overlaps with $[\mathtt{Q}] \cdot \mathtt{A} \Rightarrow [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}]$. We get a new rule $[\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}] \Rightarrow [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}]$:



Now, the rule $[\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}] \Rightarrow [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}]$ overlaps with itself on the term $[\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}]$:

$$\begin{array}{l} [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}] \\ \qquad\quad [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}] \end{array}$$

The overlaps of this rule with itself generate an infinite family of critical pairs indexed by m, $n \in \mathbb{N}$:



These critical pairs define an infinite family of rewrite rules:

$$\begin{array}{l} [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}] \Rightarrow [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \\ [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}] \Rightarrow [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{N}|\mathtt{A}] \\ [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{Q}] \Rightarrow [\mathtt{Q}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{N}|\mathtt{A}] \cdot [\mathtt{N}|\mathtt{A}] \\ \ldots \end{array}$$

Recall that the reduction order on protocols was defined to compare the number of elements in the protocol inheritance closure before comparing names. So if neither `N` nor `Q` inherited any other protocols, we would have $[\texttt{N}] < [\texttt{Q}]$. However, since `Q` inherits from `N`, we have $[\texttt{Q}] < [\texttt{N}]$ and thus $[\texttt{Q|A}] < [\texttt{N|A}]$. We won't go into details, but the convergence of the above rewrite system *also* depends on the reduction order being so. If instead $[\texttt{N|A}] < [\texttt{Q|A}]$, our rewrite system would not have a (finite) convergent presentation (we will see below that we can still describe an *infinite* convergent presentation; but we have no way to compute with one).

To summarize the theory, let's write a, b, c, p, q instead of `A`, `[N|A]`, `[Q|A]`, `[P]`, `[Q]`. The identity conformance rules and user-written requirements of `N` and `Q` define the following presentation:

$$M_1 := \langle a, n, q \,|\, nn \sim n,\ nq \sim n,\ qq \sim q,\ nan \sim na,\ qaq \sim qa \rangle$$

Completion doesn't terminate with the above; abstractly, we get an infinite convergent presentation over the same generators:

$$M_1' := \langle a, n, q \,|\, na^i n \sim na^i,\ qa^i n \sim qa^i,\ qa^i q \sim qa^i \rangle$$

The first Tietze transformation adds b with defining rewrite rule $na \sim b$, as in the previous example:

$$M_2 := \langle a, b, n, q \,|\, nn \sim n,\ qq \sim q,\ qn \sim q,\ nan \sim na,\ qaq \sim qa,\ na \sim b \rangle$$

We saw that completion still fails; we get the following infinite convergent presentation:

$$M_2' := \langle a, b, n, q \,|\, nn \sim n,\ na \sim b,\ bn \sim b,\ ba \sim bb,\ qb^i q \sim qb^i,\ qb^i a \sim qb^{i+1} \rangle$$

However, we then add c with defining rewrite rule $qa \sim c$:

$$M_3 := \langle a, b, c, n, q \,|\, nn \sim n,\ qq \sim q,\ qn \sim q,\ nan \sim na,\ qaq \sim qa,\ na \sim b,\ qa \sim c \rangle$$

Completion succeeds on M_3 using a reduction order where $c < b < a$ and $p < n$, and we get a finite convergent rewriting system. We've already listed the numerous rules once, so we won't do it again. Let's call this finite convergent presentation M_2':

$$M_3' := \langle a, b, c, n, q \,|\, \ldots 15 \text{ rewrite rules } \ldots \rangle$$

The minimized rewrite system corresponds to this presentation:

$$M_4 := \langle a, b, c, n, q \,|\, nn \sim n,\ na \sim b,\ bn \sim b,\ qq \sim q,\ qa \sim c,\ qn \sim q,\ cq \sim c \rangle$$

If we start with M_3 and perform completion, we again get M_2'.

Example 18.14. Our final example lays bare a limitation of the Requirement Machine in the form a generic signature whose rewrite system is not convergent. It is probably the simplest possible such instance, so we will study it in detail. We will attempt to combine the recursion of protocol `N`, with an equivalence between two associated types that have the same name in unrelated protocols, as in Example 18.8. The setup is the same as in that example, but now both associated types are subject to recursive conformance requirements:

```
protocol P1 {
  associatedtype A: P1
}

protocol P2 {
  associatedtype A: P2
}
```

We define a type parameter conforming to both `P1` and `P2`:

```
<τ_0_0 where τ_0_0: P1, τ_0_0: P2>
```

We have the following initial rewrite rules:

$[\texttt{P1}] \cdot [\texttt{P1}] \Rightarrow [\texttt{P1}]$	(1)
$[\texttt{P1}] \cdot \texttt{A} \Rightarrow [\texttt{P1}\vert\texttt{A}]$	(2)
$[\texttt{P1}\vert\texttt{A}] \cdot [\texttt{P1}] \Rightarrow [\texttt{P1}\vert\texttt{A}]$	(3)
$[\texttt{P1}\vert\texttt{A}] \cdot \texttt{A} \Rightarrow [\texttt{P1}\vert\texttt{A}] \cdot [\texttt{P1}\vert\texttt{A}]$	(4)
$[\texttt{P2}] \cdot [\texttt{P2}] \Rightarrow [\texttt{P2}]$	(5)
$[\texttt{P2}] \cdot \texttt{A} \Rightarrow [\texttt{P2}\vert\texttt{A}]$	(6)
$[\texttt{P2}\vert\texttt{A}] \cdot [\texttt{P2}] \Rightarrow [\texttt{P2}\vert\texttt{A}]$	(7)
$[\texttt{P2}\vert\texttt{A}] \cdot \texttt{A} \Rightarrow [\texttt{P2}\vert\texttt{A}] \cdot [\texttt{P2}\vert\texttt{A}]$	(8)
$\tau_0_0 \cdot [\texttt{P1}] \Rightarrow \tau_0_0$	(9)
$\tau_0_0 \cdot [\texttt{P2}] \Rightarrow \tau_0_0$	(10)

In our generic signature, there are infinitely many type parameters, all of the form `τ_0_0` followed by a combination of `A`, `[P1]A` and `[P2]A`:

```
τ_0_0.A
τ_0_0.[P2]A.[P1]A
τ_0_0.[P1]A.A.[P2]A
...
```

Membership in an equivalence class is determined by the number of `A`'s appearing in the type parameter, so `τ_0_0.[P1]A.[P1]A`, `τ_0_0.A.A` and `τ_0_0.[P2]A.[P2]A` are all equivalent, so we can derive same-type requirements for these equivalences.

Rules (9) and (10) overlap with (2) and (6), respectively. Resolving these overlaps proceeds as in Example 18.8:

$$\texttt{τ_0_0} \cdot \texttt{A} \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \tag{11}$$
$$\texttt{τ_0_0} \cdot [\texttt{P2}|\texttt{A}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \tag{12}$$

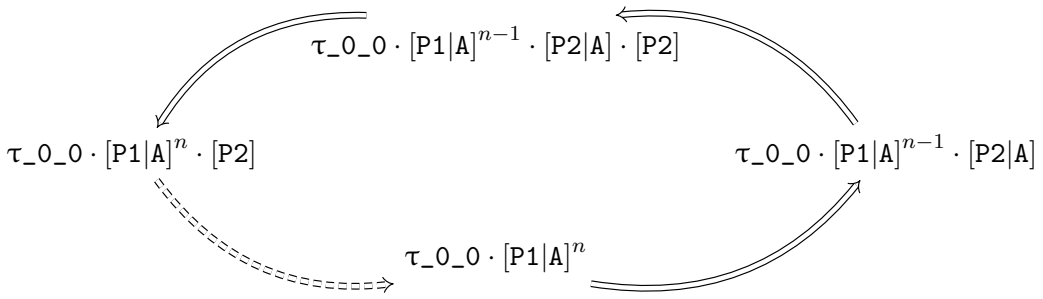
We added new rules, so we must check for overlapping rules again. Now, (12) overlaps with (7) on the term $\texttt{τ_0_0} \cdot [\texttt{P2}|\texttt{A}] \cdot [\texttt{P2}]$, and rule (12) also overlaps with (8) on the term $\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P2}|\texttt{A}] \cdot \texttt{A}$. Resolving both critical pairs adds two new rules:

$$\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P2}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \tag{13}$$
$$\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P2}|\texttt{A}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P1}|\texttt{A}] \tag{14}$$

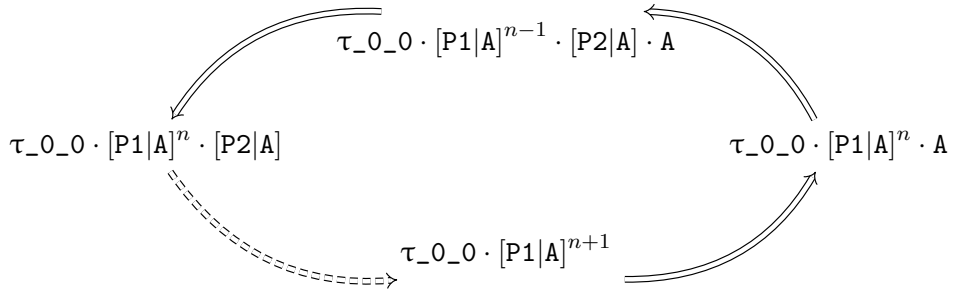
We now begin the third round, with another two critical pairs: rule (14) overlaps with (7) on the term $\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P2}|\texttt{A}] \cdot [\texttt{P2}]$, and with (8) on the term $\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P2}|\texttt{A}] \cdot \texttt{A}$. A pattern begins to emerge. We add two new rules, but we have two new critical pairs:

$$\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P2}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P1}|\texttt{A}] \tag{15}$$
$$\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P2}|\texttt{A}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P1}|\texttt{A}] \cdot [\texttt{P1}|\texttt{A}] \tag{16}$$

Completion eventually hits a limit and fails. Our rewrite system actually has two infinite families of critical pairs; the first family defines a rule $\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}]^n \cdot [\texttt{P2}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}]^n$ for each $n \in \mathbb{N}$:



The second defines $\texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}]^n \cdot [\texttt{P2}|\texttt{A}] \Rightarrow \texttt{τ_0_0} \cdot [\texttt{P1}|\texttt{A}]^{n+1}$:



In protocol `N`, associated type rules allowed us an infinite set of derived requirements with a finite set of rewrite rules. In the next example of protocol `Q` inheriting from `N`, inherited associated type rules saved the day. This time though, we don't have any tricks left.

Let's write a, b, c, p, q, t instead of `A`, [P|A], [Q|A], [P], [Q], τ_0_0. Our generic signature defines the following monoid presentation over the alphabet a, p, q, t:

$$M_1 := \langle a, p, q, t \,|\, pp \sim p,\ pap \sim pa,\ qq \sim q,\ qaq \sim qa,\ tp \sim t,\ tq \sim t \rangle$$

The Requirement Machine added the symbols b and c with defining rewrite rules $pa \sim b$ and $qa \sim c$, respectively:

$$\begin{aligned} M_2 := \langle a, b, c, p, q, t \,|\, & pp \sim p,\ pap \sim pa,\ qq \sim q,\ qaq \sim qa, \\ & tp \sim t,\ tq \sim t,\ pa \sim b,\ qa \sim c \rangle \end{aligned}$$

We saw that completion fails; we got an infinite convergent presentation over the same alphabet:

$$\begin{aligned} M_2' := \langle a, b, c, p, q, t \,|\, & pp \sim p,\ pa \sim b,\ bp \sim b,\ ba \sim bb, \\ & qq \sim q,\ qa \sim c,\ cq \sim c,\ ca \sim cc, \\ & ta \sim tb,\ tb^i q \sim tb^i,\ tb^i c \sim tb^{i+1} \rangle \end{aligned}$$

In fact, the associated type rules don't simplify our understanding of this monoid at all. We can also write down an infinite convergent presentation for the completion of M_1:

$$M_1' := \langle a, p, q, t \,|\, pa^i p \sim pa^i,\ qa^i q \sim qa^i,\ ta^i p \sim ta^i,\ ta^i q \sim ta^i \rangle$$

This infinite presentation is simple enough that one can implement a one-off algorithm for solving the word problem in M_1'. We can simplify the original monoid presentation further by removing $pp \sim p$ and $qq \sim q$; the result is *not* Tietze-equivalent, but it runs into the same problems with completion.

Future directions. We will now state an open question about this strikingly simple monoid presentation of only four symbols and rewrite rules:

> Can the monoid $\langle a, p, q, t \,|\, pap \sim pa,\ qaq \sim qa,\ tp \sim t,\ tq \sim t \rangle$ be presented by a finite convergent rewriting system?

Either of the possible outcomes would be of interest:

- If we can answer in the affirmative, then the Requirement Machine might be able to support our generic signature by tweaking how we construct the symbols and rules of our rewrite system.
- If instead we prove that no such convergent rewriting system exists, we have another example of a finitely-presented monoid with decidable word problem that cannot be presented by a finite convergent rewriting system, much like Theorem 16.48.

One approach to the study of infinite convergent rewriting systems is via formal language theory; this is explored in [184]. We recall the standard definitions, which can be found in [185]. A *language* over an alphabet A is a subset of the free monoid A^*; a language is *recursive* if there is a total computable function to determine membership in this set, and a *regular language* is a set of strings recognized by a finite state automaton (or equivalently, matching a regular expression—the classical kind, without back references). The left-hand sides of the rules in a (finite or infinite) convergent rewriting system define a language in this sense, and a finite convergent rewriting system is then the special case where this language is finite; this direction is explored in.

Every finitely-presented monoid can be presented by an infinite convergent rewriting system; but if the word problem in this monoid is undecidable, the rewrite system's language is not recursive, so of course this viewpoint does not fundamentally change the difficulty of the problem. However, if we consider infinite convergent presentations with a sufficiently restricted class of language (for instance, regular languages), we can find an effective procedure for term reduction with a rewrite system of this type. Completion then becomes a highly non-trivial problem. A completion procedure for a certain kind of infinite convergent rewriting system was proposed in [186]; there, the left-hand sides involve exponents, much like the notation $ta^ip \sim ta^i$ and $ta^iq \sim ta^i$ used in our examples here. In principle, such an extension could one day be considered for the Requirement Machine.

Example 18.15. We end this chapter with one final curiosity. We proved the derived requirements formalism to be undecidable in Section 16.3 by showing that an arbitrary finitely-presented monoid $\langle A \,|\, R \rangle$ can be encoded in the form of a protocol declaration. In Chapter 17 we defined a lowering of a generic signature and its protocol dependencies into a finitely-presented monoid. If we chain both transformations, we see that we can

map a finitely-presented monoid to a protocol declaration and then back to a finitely-presented monoid. In what sense does the latter monoid encode the original monoid?

Let $A^* := \{a, b, c\}$, $R := \{(ab, c), (bc, \varepsilon)\}$, and consider the monoid $M := \langle A \,|\, R \rangle$. Written down as a Swift protocol, $M := \langle a, b, c \,|\, ab \sim c,\ bc \sim \varepsilon \rangle$ looks like this:

```
protocol P {
  associatedtype A: P
  associatedtype B: P
  associatedtype C: P
    where A.B == C, B.C == Self
}
```

Now we pretend to comment out the `where` clause, so now our protocol `P` just presents the free monoid A^*. We will list the convergent rewriting system for `P`, but partition the rules into two sets in a new way. The first set of rules involves name symbols; we call this set $\mathcal{N}$:

$$[\mathtt{P}] \cdot \mathtt{A} \Rightarrow [\mathtt{P}|\mathtt{A}]$$
$$[\mathtt{P}] \cdot \mathtt{B} \Rightarrow [\mathtt{P}|\mathtt{B}]$$
$$[\mathtt{P}] \cdot \mathtt{C} \Rightarrow [\mathtt{P}|\mathtt{C}]$$
$$[\mathtt{P}|\mathtt{A}] \cdot \mathtt{A} \Rightarrow [\mathtt{P}|\mathtt{A}] \cdot [\mathtt{P}|\mathtt{A}]$$
$$[\mathtt{P}|\mathtt{A}] \cdot \mathtt{B} \Rightarrow [\mathtt{P}|\mathtt{A}] \cdot [\mathtt{P}|\mathtt{B}]$$
$$[\mathtt{P}|\mathtt{A}] \cdot \mathtt{C} \Rightarrow [\mathtt{P}|\mathtt{A}] \cdot [\mathtt{P}|\mathtt{C}]$$
$$[\mathtt{P}|\mathtt{B}] \cdot \mathtt{A} \Rightarrow [\mathtt{P}|\mathtt{B}] \cdot [\mathtt{P}|\mathtt{A}]$$
$$[\mathtt{P}|\mathtt{B}] \cdot \mathtt{B} \Rightarrow [\mathtt{P}|\mathtt{B}] \cdot [\mathtt{P}|\mathtt{B}]$$
$$[\mathtt{P}|\mathtt{B}] \cdot \mathtt{C} \Rightarrow [\mathtt{P}|\mathtt{B}] \cdot [\mathtt{P}|\mathtt{C}]$$
$$[\mathtt{P}|\mathtt{C}] \cdot \mathtt{A} \Rightarrow [\mathtt{P}|\mathtt{C}] \cdot [\mathtt{P}|\mathtt{A}]$$
$$[\mathtt{P}|\mathtt{C}] \cdot \mathtt{B} \Rightarrow [\mathtt{P}|\mathtt{C}] \cdot [\mathtt{P}|\mathtt{B}]$$
$$[\mathtt{P}|\mathtt{C}] \cdot \mathtt{C} \Rightarrow [\mathtt{P}|\mathtt{C}] \cdot [\mathtt{P}|\mathtt{C}]$$

The second set involves protocol and associated type symbols only. We call this set $\mathcal{S}$:

$$[\mathtt{P}] \cdot [\mathtt{P}] \Rightarrow [\mathtt{P}]$$
$$[\mathtt{P}] \cdot [\mathtt{P}|\mathtt{A}] \Rightarrow [\mathtt{P}|\mathtt{A}]$$
$$[\mathtt{P}] \cdot [\mathtt{P}|\mathtt{B}] \Rightarrow [\mathtt{P}|\mathtt{B}]$$
$$[\mathtt{P}] \cdot [\mathtt{P}|\mathtt{C}] \Rightarrow [\mathtt{P}|\mathtt{C}]$$
$$[\mathtt{P}|\mathtt{A}] \cdot [\mathtt{P}] \Rightarrow [\mathtt{P}|\mathtt{A}]$$
$$[\mathtt{P}|\mathtt{B}] \cdot [\mathtt{P}] \Rightarrow [\mathtt{P}|\mathtt{B}]$$
$$[\mathtt{P}|\mathtt{C}] \cdot [\mathtt{P}] \Rightarrow [\mathtt{P}|\mathtt{C}]$$

The union $\mathcal{N} \cup \mathcal{S}$ is a rewrite system for a protocol encoding the free monoid $\{a, b, c\}^*$. Here, $\mathcal{N}$ contains 12 elements, and $\mathcal{S}$ contains 7 elements. More generally, if $|A| = n$, we have $|\mathcal{N}| = n(n+1)$ and $|\mathcal{S}| = 2n + 1$. Now, if we identify a, b, c with [P|A], [P|B], [P|C], and define a new symbol $e :=$ [P], we see that $\langle A \cup \{e\} \,|\, \mathcal{S} \rangle$ is a convergent presentation. But to understand the object it presents, we need a few related definitions.

A *semigroup* is a set together with an associative binary operation, but without a distinguished identity element. The *free semigroup* over a set A, denoted A^+, is the set of all *non-empty* strings from the symbols of A. We can define a *finitely-presented semigroup* analogously to a finitely-presented monoid; here the rewrite rules have the form (u, v) where u, $v \in A^+$.

Every monoid trivially satisfies the semigroup axioms, and thus a free monoid A^* is also a semigroup. However a free monoid A^* is not a *free* semigroup. For example, taking $A := \{a, b, c\}$, we see among the elements of A^* *viewed as a semigroup*, we have non-trivial identities involving ε, such as $a = \varepsilon a$, $b\varepsilon = b$, and so on. Thus, this cannot be a free semigroup, where $xy \neq x$ and $xy \neq y$ for all x, $y \in A^+$. The free monoid A^* *is* a finitely-presented semigroup though; we must add the symbol e to the alphabet, and a pair of rewrite rules $ex \sim e$, $xe \sim e$ for each $x \in A$. For example, as a semigroup, $\{a, b, c\}^*$ has four generators and seven rewrite rules:

$$\langle a, b, c, e \,|\, e \sim e,\ ea \sim a,\ eb \sim b,\ ec \sim c,\ ae \sim a,\ be \sim b,\ ce \sim c \rangle$$

Indeed, we see this set of rewrite rules is just $\mathcal{S}$, so our embedding of the free monoid as a Swift protocol ends up encoding the free monoid as the finitely-presented semigroup $\langle A \cup \{e\} \,|\, \mathcal{S} \rangle$, but embedded in an even larger semigroup that also contains name symbols and the set of rules $\mathcal{N}$.

Now, we imagine that we uncomment the `where` clause of our protocol `P`, with the two requirements [`Self.A.B == C`] and [`Self.B.C == Self`]. These requirements contribute two new rewrite rules to our convergent rewriting system; we will call this set $\mathcal{R}$:

$$[\texttt{P}|\texttt{A}] \cdot [\texttt{P}|\texttt{B}] \Rightarrow [\texttt{P}|\texttt{C}]$$
$$[\texttt{P}|\texttt{B}] \cdot [\texttt{P}|\texttt{C}] \Rightarrow [\texttt{P}]$$

We can construct $\mathcal{R}$ from R by replacing ε with e anywhere that it appears in a rewrite rule. This gives us a set of semigroup rewrite rules, and we see that $\langle A \cup \{e\} \,|\, \mathcal{S} \cup \mathcal{R} \rangle$ is a semigroup presentation of $\langle A \,|\, R \rangle$:

$$\langle a, b, c, e \,|\, ee \sim e,\ ea \sim a,\ eb \sim b,\ ec \sim c,\ ae \sim a,\ be \sim b,\ ce \sim c,\ ab \sim c,\ bc \sim e \rangle$$

Neither the monoid presentation M nor the semigroup presentation $\langle A \cup \{e\} \,|\, \mathcal{S} \cup \mathcal{R} \rangle$ is convergent. Now consider the following set of rewrite rules $\hat{R}$:

$$\hat{R} := \{(cc, a),\ (ba, c),\ (ca, a),\ (cb, \varepsilon)\}$$

A trivial but technical calculation shows that $\langle A \,|\, R \cup \hat{R} \rangle$ is a convergent monoid presentation of $\langle A \,|\, R \rangle$. Next, we transform $\hat{R}$ into a set of semigroup rewrite rules, replacing ε with e. We call this set $\hat{\mathcal{R}}$:

$$[\mathtt{P}|\mathtt{C}] \cdot [\mathtt{P}|\mathtt{C}] \Rightarrow [\mathtt{P}|\mathtt{A}]$$
$$[\mathtt{P}|\mathtt{B}] \cdot [\mathtt{P}|\mathtt{A}] \Rightarrow [\mathtt{P}|\mathtt{C}]$$
$$[\mathtt{P}|\mathtt{C}] \cdot [\mathtt{P}|\mathtt{A}] \Rightarrow [\mathtt{P}|\mathtt{A}]$$
$$[\mathtt{P}|\mathtt{C}] \cdot [\mathtt{P}|\mathtt{B}] \Rightarrow [\mathtt{P}]$$

We now claim that $\langle A \cup \{e\} \,|\, \mathcal{S} \cup \mathcal{R} \cup \hat{\mathcal{R}} \rangle$ is a convergent semigroup presentation, because we can mechanically transform any positive rewrite path over $\langle A \,|\, R \cup \hat{R} \rangle$ into a positive rewrite path over this semigroup presentation. The only rewrite step that does not map trivially is $x(cb \Rightarrow \varepsilon)y$, for some x, $y \in A^*$. The corresponding rewrite step over the semigroup presentation, $x(cb \Rightarrow e)y$, has destination term xey instead of xy. If either one of x or y is non-empty, we can eliminate the occurrence of e by adding a single positive rewrite step from $\mathcal{S}$. Otherwise, if $xey = e$, there is nothing to do. In any given positive rewrite path, this transformation can only add finitely many new rewrite steps involving rewrite rules from $\mathcal{S}$. Thus, confluence and termination are preserved.

It is of course not true in general that the union of two convergent rewriting systems is convergent, however in our construction it happens to work. There are no non-trivial overlaps between the elements of $\mathcal{N}$, $\mathcal{S}$ and $\mathcal{R}$. We've proven the following:

Theorem 18.16. Suppose $\langle A \,|\, R \rangle$ is a finitely-presented monoid, $\hat{R}$ is a finite set of rewrite rules such that $\langle A \,|\, R \cup \hat{R} \rangle$ is a convergent monoid presentation, and both R and $\hat{R}$ are oriented with respect to the shortlex order on A^*. We define a protocol $\mathtt{P}$ encoding $\langle A \,|\, R \rangle$, naming the associated types in correspondence with the fixed linear order on the generating set A (for example, if $a < b < c < \cdots$, we have $[\mathtt{P}|\mathtt{A}] < [\mathtt{P}|\mathtt{B}] < [\mathtt{P}|\mathtt{C}]$). Then our protocol $\mathtt{P}$ has the following *convergent* presentation:

$$\langle A \cup \{e\} \cup B \,|\, \mathcal{N} \cup \mathcal{S} \cup \mathcal{R} \cup \hat{\mathcal{R}} \rangle$$

Here, $e := [\mathtt{P}]$, B is a set of name symbols, $\mathcal{N}$ and $\mathcal{S}$ only depend on A, and $\mathcal{R}$ and $\hat{\mathcal{R}}$ are constructed from R and $\hat{R}$ as above. The semigroup presented above also contains a sub-semigroup with the following convergent presentation:

$$\langle A \cup \{e\} \,|\, \mathcal{S} \cup \mathcal{R} \cup \hat{\mathcal{R}} \rangle$$

This sub-semigroup is isomorphic to the monoid $\langle A \,|\, R \cup \hat{R} \rangle$ and thus $\langle A \,|\, R \rangle$.

Hence, we see that if we encode a finitely-presented monoid $\langle A \,|\, R \rangle$ as a protocol $\mathtt{P}$, the Requirement Machine will accept $\mathtt{P}$ and solve the word problem in $\langle A \,|\, R \rangle$ if and only if $\langle A \,|\, R \rangle$ has a convergent presentation compatible with the shortlex order on A^*. That's a satisfying conclusion to "Swift type checking is undecidable."

Our double encoding of "a monoid as a protocol as a rewrite system" introduces many new symbols and rewrite rules not found in the original presentation. It is a remarkable fact of our construction then, that a convergent monoid presentation always maps to a convergent rewriting system where the added detritus can always be "factored out."

18.6. Source Code Reference

Key source files:

- `lib/AST/RequirementMachine/KnuthBendix.cpp`
- `lib/AST/RequirementMachine/RewriteSystem.cpp`
- `lib/AST/RequirementMachine/Trie.h`

`rewriting::RewriteSystem` *class*

See also Section 17.7.

- `addRule()` implements Algorithm 18B.
- `recordRewriteLoop()` records a rewrite loop if this rewriting system is used for minimization.
- `computeCriticalPair()` implements Algorithm 18A.
- `performKnuthBendix()` implements Algorithm 18E.
- `simplifyLeftHandSides()` implements Algorithm 18F.
- `simplifyRightHandSides()` implements Algorithm 18G.

`rewriting::Trie` *template class*

See also Section 17.7.

- `findAll()` finds all overlapping rules using Algorithm 18D.

19. The Property Map

19.1. Substitution Simplification

Algorithm 19A (Simplify substitution terms)**.**

19.2. Property Unification

19.3. Concrete Type Unification

19.4. Concrete Conformances

19.5. Generic Signature Queries

19.6. Reduced Types

19.7. Source Code Reference

20. Minimization

20.1. Homotopy Reduction

20.2. The Elimination Order

20.3. Conformance Minimization

20.4. Building Requirements

20.5. Concrete Contraction

20.6. Source Code Reference

A. Mathematical Conventions

The more formal sections in this book are written in "Euclidean style", where the text is further subdivided into elements of the below kinds. All elements are numbered consecutively within a single chapter, except for algorithms which use a letter:

- A **Definition** introduces terminology or notation.
- An **Example** shows how this terminology and notation arises in practice.
- A **Proposition** states a logical consequence of one or more definitions.
- A **Lemma** is an intermediate proposition in service of proving another theorem.
- A **Theorem** is a "deeper" proposition which is more profound in some sense.
- A **Corollary** is an immediate consequence of an earlier theorem.
- An **Algorithm** is a step-by-step description of a computable function, defined in terms of inputs and outputs.

A handful of Greek letters are used for variable names and other notation: lowercase α (alpha), β (beta), γ (gamma), ε (epsilon), η (eta), π (pi), σ (sigma), τ (tau), φ (phi); and uppercase Σ (sigma). The "$=$" operator says that two existing things are identical or equivalent; "$\neq$" means that they are not. The "$:=$" operator *defines* the new thing on the left as a combination of existing things on the right. The "$\leftarrow$" operator denotes an imperative assignment statement in an algorithm.

Sets. A *set* is a collection of elements without regard to order or duplicates. Sets can be finite or infinite. A finite set can be specified by listing its elements in any order, for example $\{a, b, c\}$. The empty set $\varnothing$ is the unique set with no elements. The set of *natural numbers* $\mathbb{N} := \{0, 1, 2, \ldots\}$ is the infinite set containing zero and all of its successors. The set of *integers* $\mathbb{Z} := \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ also contains the negative whole numbers.

The notation $x \in S$ means "x is an element of a set S," and $x \notin S$ is its negation. Properties of sets can be defined using *existential* quantification ("there exists (at least one) $x \in S$ such that x has this property…") or *universal* quantification ("for all $x \in S$, the following property is true of x…").

A set X is a *subset* of another set Y, written as $X \subseteq Y$, if for all $x \in X$, it is also true that $x \in Y$. If $X \subseteq Y$ and $Y \subseteq X$, then the two sets have the same elements, so they're equal; $X = Y$. If $X \subseteq Y$ and $X \neq Y$, then X is a *proper* subset of Y, written as $X \subsetneq Y$. The *union* $X \cup Y$ is the set of all elements belonging to either X or Y. The *intersection* $X \cap Y$ is the set of all elements belonging to both X and Y. The *difference* $X \setminus Y$ is the set of all elements of X not also in Y.

The *Cartesian product* of two sets X and Y, denoted $X \times Y$, is the set of all *ordered pairs* (x, y) where $x \in X$, $y \in Y$. Note that the ordered pair (x, y) is not the same as the set $\{x, y\}$, because $(x, y) \neq (y, x)$. The Cartesian product construction generalizes to any finite number of sets, to give *ordered tuples* or *sequences*.

Functions. A *function* (or *mapping*) $f \colon X \to Y$ assigns to each $x \in X$ a unique element $f(x) \in Y$. If the sets X and Y are equipped with some kind of additional structure (which will be explicitly defined), then f is a *homomorphism* if it preserves this structure.

A function $f \colon X \times Y \to Z$ defined on the Cartesian product can be thought of as taking a pair of values $x \in X$, $y \in Y$ to an element $f(x, y) \in Z$. A *binary operation* is a function named by a symbol like "$\otimes$", "$\triangleleft$", or "$\cdot$" defined on the Cartesian product of two sets. The application of a binary operation is denoted by writing the symbol in between the two elements, like $x \otimes y$.

The *cardinality* of a finite set S, denoted $|S|$, is the number of elements in S. The notation $|x|$ sometimes denotes certain other functions taking values in $\mathbb{N}$, such as the length of a sequence; this will always be explicitly defined when needed.

More

- Formal systems (Section 5.2).
- Equivalence relations (Section 5.3, Section 16.2).
- Partial and linear orders (Section 5.5, Section 16.5, Section 17.3).
- Category theory (Section 6.3).
- Boolean satisfiability (Section 7.6).
- Directed graphs (Section 8.3, Section 12.2, Section 12.3, Section 15.1).
- Proof by induction (Section 11.3).
- Computability theory (Section 12.4, Section 16.4).
- Finitely-presented monoids and string rewriting (Chapter 16, Chapter 18).

B. Derived Requirements

Let G be a generic signature. We generate the theory of G by repeated application of inference rules, starting from a finite set of elementary statements. A derivation proves that some element belongs to this set by giving a list of derivation steps where the assumptions in each step are conclusions of previous steps. Nomenclature:

Symbol	Description
`T`, `U`, and `V`	type parameters
`Self.U` and `Self.V`	type parameters rooted in the protocol `Self` type
`X`	a concrete type
`C`	a concrete class type
`X`′ and `C`′	obtained from `X` and `C` by replacing `Self` with `T`
`P` and `Q`	protocols
`A`	the name of an associated type of `P`
`[P]A`	an associated type declaration of `P`
`T.[P]A` and `T.A`	bound and unbound dependent member type
`[T: P]`	a conformance requirement
`[T == U]`	a same-type requirement between type parameters
`[T == X]`	a concrete same-type requirement
`[T: C]`	a superclass requirement
`[T: AnyObject]`	a layout requirement
$\texttt{[Self.U: Q]}_{\texttt{P}}$	an associated requirement of protocol `P`

See Section 5.2, Section 5.3, and Section 5.4 for details.

Elementary statements. For each generic parameter `τ_d_i` of G:

`τ_d_i` (Generic)

For each explicit requirement of G by kind:

`[T: P]` (Conf)

`[T == U]` (Same)

`[T == X]` (Concrete)

`[T: C]` (Super)

`[T: AnyObject]` (Layout)

Requirement signatures. Assume $G \vdash$ [T: P]. For each associated type A of P:

T.A	(AssocName [T: P])
T.[P]A	(AssocDecl [T: P])
[T.[P]A == T.A]	(AssocBind [T: P])

For each associated requirement of P by kind:

[T.U: Q]	(AssocConf [Self.U: Q]$_{\texttt{P}}$ [T: P])
[T.U == T.V]	(AssocSame [Self.U == Self.V]$_{\texttt{P}}$ [T: P])
[T.U == X′]	(AssocConcrete [Self.U == X]$_{\texttt{P}}$ [T: P])
[T.U: C′]	(AssocSuper [Self.U: C]$_{\texttt{P}}$ [T: P])
[T.U: AnyObject]	(AssocLayout [Self.U: AnyObject]$_{\texttt{P}}$ [T: P])

Equivalence. Same-type requirements generate an equivalence relation:

[T == T]	(Reflex T)
[U == T]	(Sym [T == U])
[T == V]	(Trans [T == U] [U == V])

Compatibility. A derived conformance, superclass or layout requirement applies to all type parameters in an equivalence class:

[T: P]	(SameConf [U: P] [T == U])
[T == X]	(SameConcrete [U == X] [T == U])
[T: C]	(SameSuper [U: C] [T == U])
[T: AnyObject]	(SameLayout [U: AnyObject] [T == U])

If two type parameters are equivalent, so are all corresponding member types:

[T.A == U.A]	(SameName [U: P] [T == U])
[T.[P]A == U.[P]A]	(SameDecl [U: P] [T == U])

C. Substitution Algebra

This is a summary of our algebraic notation for various operations on types, substitution maps, and conformances. Details in Chapter 6, Chapter 7, and Chapter 12.

G, H, I, ...	Generic signatures	Σ, Σ_1, ...	Substitution maps
`T`	Type parameter	`X`	Interface type
`τ_d_i`	Generic parameter	$[\texttt{X}_d\texttt{: P}]$	Normal conformance
d	Nominal type declaration	`[X: P]`	Specialized conformance
$\texttt{X}_d$	Declared interface type of d	`[T: P]`	Abstract conformance

$\textsc{Type}(G)$	All valid interface types for G
$\textsc{Sub}(G, H)$	Substitution maps with input signature G and output signature H
$\textsc{Conf}(G)$	Conformances with output generic signature G
$\textsc{Req}(G)$	Requirements containing interface types of G

$\textsc{Type}(G) \otimes \textsc{Sub}(G, H) \to \textsc{Type}(H)$	Type substitution	Chapter 6
$\textsc{Sub}(G, H) \otimes \textsc{Sub}(H, I) \to \textsc{Sub}(G, I)$	Substitution map composition	Section 6.3
$\textsc{Conf}(G) \otimes \textsc{Sub}(G, H) \to \textsc{Conf}(H)$	Conformance substitution	Section 7.2
$\textsc{Req}(G) \otimes \textsc{Sub}(G, H) \to \textsc{Req}(H)$	Requirement substitution	Section 9.3

Substitution maps. A substitution map $\Sigma \in \textsc{Sub}(G, H)$ contains an array of interface types from $\textsc{Type}(H)$, and an array of conformances from $\textsc{Conf}(H)$.

For each generic parameter `τ_d_i` and root abstract conformance `[T: P]` of G:

$\texttt{τ_d_i} \otimes \Sigma = \texttt{τ_d_i} \otimes \{\ldots, \texttt{τ_d_i} \mapsto \texttt{X}, \ldots\} = \texttt{X}$

$[\texttt{T: P}] \otimes \Sigma = [\texttt{T: P}] \otimes \{\ldots, [\texttt{T: P}] \mapsto [\texttt{X: P}], \ldots\} = [\texttt{X: P}]$ Section 7.2

Substitution map composition:

$\texttt{X} \otimes (\Sigma_1 \otimes \Sigma_2) = (\texttt{X} \otimes \Sigma_1) \otimes \Sigma_2$

$\Sigma_1 \otimes (\Sigma_2 \otimes \Sigma_3) = (\Sigma_1 \otimes \Sigma_2) \otimes \Sigma_3$

Identity substitution map:

$\texttt{X} \otimes 1_G = \texttt{X}$	for all $\texttt{X} \in \textsc{Type}(G)$
$[\texttt{X: P}] \otimes 1_G = [\texttt{X: P}]$	for all $[\texttt{X: P}] \in \textsc{Conf}(G)$
$1_G \otimes \Sigma = \Sigma \otimes 1_H = \Sigma$	for all $\Sigma \in \textsc{Sub}(G, H)$

Every generic nominal type `X` is can be written as $\texttt{X} = \texttt{X}_d \otimes \Sigma$ for some nominal type declaration d and substitution map Σ (Section 6.1).

Conformances. A normal conformance $[\mathtt{X}_d\mathtt{:\ P}]$ declares a series of type witnesses and associated conformances. If $\mathtt{X} = \mathtt{X}_d \otimes \Sigma$, then $[\mathtt{X:\ P}] = [\mathtt{X}_d\mathtt{:\ P}] \otimes \Sigma$ is a specialized conformance formed from the normal conformance $[\mathtt{X}_d\mathtt{:\ P}]$ and substitution map Σ.

$\textsc{Proto}$	All protocol declarations
$\textsc{AssocType}_\mathtt{P}$	All associated type declarations of $\mathtt{P} \in \textsc{Proto}$
$\textsc{AssocConf}_\mathtt{P}$	All associated conformance requirements declared of $\mathtt{P} \in \textsc{Proto}$
$\textsc{Conf}_\mathtt{P}(G)$	The set of all $[\mathtt{X:\ P}] \in \textsc{Conf}(G)$ for a fixed $\mathtt{P} \in \textsc{Proto}$
$\langle\mathtt{P}]$	Protocol declaration named $\mathtt{P}$
$\langle\mathtt{P}\vert\mathtt{A}$	Associated type declaration named $\mathtt{A}$ in protocol $\mathtt{P}$
$\langle\mathtt{Self.U:\ Q}]$	Associated conformance requirement
$\mathtt{T.[P]A}$	Dependent member type with base type $\mathtt{T}$ and associated type $\mathtt{A}$ of $\mathtt{P}$
$\Sigma_{[\mathtt{X:\ P}]}$	Protocol substitution map $\{\tau_0_0 \mapsto \mathtt{X};\ [\tau_0_0\mathtt{:\ P}] \mapsto [\mathtt{X:\ P}]\}$

$\textsc{Proto} \otimes \textsc{Type}(G) \to \textsc{Conf}(G)$	Global conformance lookup	Section 7.1
$\textsc{AssocType}_\mathtt{P} \otimes \textsc{Conf}_\mathtt{P}(G) \to \textsc{Type}(G)$	Type witness projection	Section 7.3
$\textsc{AssocConf}_\mathtt{P} \otimes \textsc{Conf}_\mathtt{P}(G) \to \textsc{Conf}(G)$	Associated conformance proj.	Section 7.5

Global conformance lookup:

$\langle\mathtt{P}] \otimes \mathtt{X}_d := [\mathtt{X}_d\mathtt{:\ P}]$	normal
$\langle\mathtt{P}] \otimes (\mathtt{X}_d \otimes \Sigma) := [\mathtt{X}_d\mathtt{:\ P}] \otimes \Sigma$	specialized
$\langle\mathtt{P}] \otimes \mathtt{T} := [\mathtt{T:\ P}]$	abstract
$(\langle\mathtt{P}] \otimes \mathtt{T}) \otimes \Sigma = \langle\mathtt{P}] \otimes (\mathtt{T} \otimes \Sigma)$	Section 7.4

Specialized conformance substitution:

$([\mathtt{X}_d\mathtt{:\ P}] \otimes \Sigma_1) \otimes \Sigma_2 := [\mathtt{X}_d\mathtt{:\ P}] \otimes (\Sigma_1 \otimes \Sigma_2)$

For each $\langle\mathtt{P}\vert\mathtt{A} \in \textsc{AssocType}_\mathtt{P}$:

$\langle\mathtt{P}\vert\mathtt{A} \otimes [\mathtt{X}_d\mathtt{:\ P}] :=$ declared in source	normal
$\langle\mathtt{P}\vert\mathtt{A} \otimes ([\mathtt{X}_d\mathtt{:\ P}] \otimes \Sigma) := (\langle\mathtt{P}\vert\mathtt{A} \otimes [\mathtt{X}_d\mathtt{:\ P}]) \otimes \Sigma$	specialized
$\langle\mathtt{P}\vert\mathtt{A} \otimes [\mathtt{T:\ P}] := \mathtt{T.[P]A}$ Section 7.4	abstract

For each $\langle\mathtt{Self.U:\ Q}] \in \textsc{AssocConf}_\mathtt{P}$:

$\langle\mathtt{Self.U:\ Q}] \otimes [\mathtt{X}_d\mathtt{:\ P}] := \langle\mathtt{Q}] \otimes \mathtt{Self.U} \otimes \Sigma_{[\mathtt{X}_d\mathtt{:\ P}]}$	normal
$\langle\mathtt{Self.U:\ Q}] \otimes ([\mathtt{X}_d\mathtt{:\ P}] \otimes \Sigma) := (\langle\mathtt{Self.U:\ Q}] \otimes [\mathtt{X}_d\mathtt{:\ P}]) \otimes \Sigma$	specialized
$\langle\mathtt{Self.U:\ Q}] \otimes [\mathtt{T:\ P}] := [\mathtt{T.U:\ Q}]$	abstract

Dependent member type substitution:

$\mathtt{T.[P]A} \otimes \Sigma := \langle\mathtt{P}\vert\mathtt{A} \otimes ([\mathtt{T:\ P}] \otimes \Sigma)$ Section 7.4

Local conformance lookup using a conformance path:

$[\mathtt{T:\ P}] \otimes \Sigma := \langle\mathtt{Self.U}_n\mathtt{:\ P}_n] \otimes \cdots \otimes \langle\mathtt{Self.U}_1\mathtt{:\ P}_1] \otimes [\mathtt{T}_0\mathtt{:\ P}_0] \otimes \Sigma$ Chapter 12

Bibliography

[1] "The Swift programming language," 2014. https://docs.swift.org/swift-book/
(Cited on page 11.)

[2] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
https://www.goodreads.com/en/book/show/887908
(Cited on pages 11 and 39.)

[3] K. Cooper and L. Torczon, *Engineering a Compiler*. Elsevier Science, 2004.
https://dl.acm.org/doi/pdf/10.5555/2737838
(Cited on pages 11 and 39.)

[4] R. Nystrom, *Crafting Interpreters*. Genever Benning, 2021. https://craftinginterpreters.com
(Cited on page 11.)

[5] J. G. Siek, *Essentials of Compilation*. The MIT Press, 2023.
https://mitpress.mit.edu/9780262047760/essentials-of-compilation/
(Cited on page 11.)

[6] "Swift evolution process," 2016. https://www.swift.org/swift-evolution/
(Cited on page 14.)

[7] J. McCall and S. Pestov, "Implementing Swift generics," 2017.
https://www.youtube.com/watch?v=ctS8FzqcRug
(Cited on page 14.)

[8] D. Rexin, "Compact value witnesses in Swift," 2023. https://www.youtube.com/watch?v=hjgDwdGJIhI
(Cited on page 14.)

[9] H. Borla, J. McCall, and S. Pestov, "SE-0393: Value and type parameter packs," 2023.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0393-parameter-packs.md
(Cited on page 14.)

[10] S. Pestov and H. Borla, "SE-0398: Allow generic types to abstract over packs," 2023.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0398-variadic-types.md
(Cited on page 14.)

[11] S. Poirier and H. Borla, "SE-0399: Tuple of value pack expansion," 2023. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0399-tuple-of-value-pack-expansion.md
(Cited on page 14.)

[12] J. Groff, M. Gottesman, A. Trick, and K. Farvardin, "SE-0390: Noncopyable structs and enums," 2023.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0390-noncopyable-structs-and-enums.md
(Cited on pages 14 and 21.)

[13] K. Farvardin, T. Kientzle, and S. Pestov, "SE-0427: Noncopyable generics," 2024.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0427-noncopyable-generics.md
(Cited on pages 14 and 21.)

[14] A. Trick and T. Kientzle, "SE-0446: Nonescapable types," 2025.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0446-non-escapable.md
(Cited on page 14.)

[15] A. Alonso and J. Groff, "SE-0452: Integer generic parameters," 2025. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0452-integer-generic-parameters.md (Cited on page 14.)

[16] R. Harper and G. Morrisett, "Compiling polymorphism using intensional type analysis," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. Association for Computing Machinery, 1995, p. 130–141. https://www.cs.cmu.edu/~rwh/papers/intensional/popl95.pdf (Cited on page 35.)

[17] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. Association for Computing Machinery, 1989, p. 60–76. https://www.researchgate.net/profile/Stephen-Blott/publication/2710954_How_to_Make_Ad-Hoc_Polymorphism_Less_Ad_Hoc/links/553e01f20cf2fbfe509b81f8/How-to-Make-Ad-Hoc-Polymorphism-Less-Ad-Hoc.pdf (Cited on page 35.)

[18] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler, "Type classes in Haskell," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, p. 109–138, mar 1996. https://dl.acm.org/doi/pdf/10.1145/227699.227700 (Cited on page 35.)

[19] S. Peyton Jones, M. Jones, and E. Meijer, "Type classes: an exploration of the design space," in *Haskell workshop*, January 1997. https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/ (Cited on page 35.)

[20] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow, "Associated types with class," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. Association for Computing Machinery, 2005, p. 1–13. https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/assoc.pdf (Cited on page 35.)

[21] M. M. T. Chakravarty, G. Keller, and S. P. Jones, "Associated type synonyms," in *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '05. Association for Computing Machinery, 2005, p. 241–253. https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/at-syns.pdf (Cited on page 35.)

[22] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann, "Type checking with open type functions," in *ICFP 2008*, April 2008. https://www.microsoft.com/en-us/research/publication/type-checking-with-open-type-functions/ (Cited on page 35.)

[23] O. Kiselyov, S. Peyton Jones, and C. Shan, "Fun with type functions," April 2009. https://www.microsoft.com/en-us/research/publication/fun-type-functions/ (Cited on page 35.)

[24] D. Vandevoorde, N. Josuttis, and D. Gregor, *C++ Templates: The Complete Guide*. Pearson Education, 2017. http://www.tmplbook.com/ (Cited on page 35.)

[25] A. Stepanov and P. McJones, *Elements of Programming*. Semigroup Press, 2019. https://www.elementsofprogramming.com (Cited on page 36.)

[26] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, "Concepts: Linguistic support for generic programming in C++," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. Association for Computing Machinery, 2006, p. 291–310. https://www.stroustrup.com/oopsla06.pdf
(Cited on page 36.)

[27] J. G. Siek and A. Lumsdaine, "Essential language support for generic programming," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 73–84. https://www.researchgate.net/publication/213886648_Essential_Language_Support_for_Generic_Programming
(Cited on page 36.)

[28] Rust Traits Working Group, "Rust RFC 1598: Generic associated types," 2016. https://rust-lang.github.io/rfcs/1598-generic_associated_types.html
(Cited on page 36.)

[29] Rust Traits Working Group, "Rust RFC 2000: Const generics," 2017. https://rust-lang.github.io/rfcs/1598-generic_associated_types.html
(Cited on page 36.)

[30] J. Roesch, "Parse and accept type equality constraints in "where" clauses," 2014. https://github.com/rust-lang/rust/issues/20041
(Cited on page 36.)

[31] A. Turon, ""where" clauses are only elaborated for supertraits, and not other things," 2015. https://github.com/rust-lang/rust/issues/20671
(Cited on page 36.)

[32] J. Milewski, "Formalizing Rust traits," Ph.D. dissertation, University of British Columbia, 2015. https://open.library.ubc.ca/collections/ubctheses/24/items/1.0220521
(Cited on page 36.)

[33] Rust Traits Working Group, "The Chalk book," 2015. https://rust-lang.github.io/chalk/book/
(Cited on page 36.)

[34] A. Langer, "Java generics FAQs," 2004. http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html
(Cited on page 36.)

[35] "Project Valhalla," 2024. https://openjdk.org/projects/valhalla/
(Cited on page 36.)

[36] D. Abrahams and D. Racordon, "The Hylo programming language," 2021. https://www.hylo-lang.org
(Cited on page 36.)

[37] D. Racordon, "Implement a variant of Slava Pestov's requirement machine," 2024. https://github.com/hylo-lang/hylo/pull/1482
(Cited on page 36.)

[38] N. Cook, N. Chandler, and M. Ricketson, "SE-0281: `@main`: Type-based program entry points," 2020. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0281-main-attribute.md
(Cited on page 37.)

[39] R. Widmann, "SE-0383: Deprecate @UIApplicationMain and @NSApplicationMain," 2023. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0383-deprecate-uiapplicationmain-and-nsapplicationmain.md
(Cited on page 37.)

[40] "Swift intermediate language (SIL)," 2024.
https://github.com/swiftlang/swift/blob/main/docs/SIL/SIL.md
(Cited on pages 39 and 195.)

[41] J. Groff and C. Lattner, "Swift's high-level IR: A case study," 2015.
https://www.youtube.com/watch?v=Ntj8ab-5cvE
(Cited on page 39.)

[42] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis and transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86. https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf
(Cited on page 39.)

[43] A. Zhilin, "SE-0077: Improved operator declarations," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0077-operator-precedence.md
(Cited on page 45.)

[44] T. Allevato and D. Gregor, "SE-0091: Improving operator requirements in protocols," 2016. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0091-improving-operators-in-protocols.md
(Cited on page 45.)

[45] D. Gregor, "Request evaluator," 2018.
https://github.com/swiftlang/swift/blob/main/docs/RequestEvaluator.md
(Cited on pages 49 and 54.)

[46] A. Aho, B. Kernighan, and P. Weinberger, *The AWK Programming Language*, ser. Addison-Wesley Professional Computing Series. Pearson Education, 2023. https://awk.dev
(Cited on page 51.)

[47] G. Hoare, "Swift compiler performance," 2017.
https://github.com/swiftlang/swift/blob/main/docs/CompilerPerformance.md
(Cited on page 51.)

[48] J. Rose, "Dependency analysis," 2015.
https://github.com/swiftlang/swift/blob/main/docs/DependencyAnalysis.md
(Cited on page 54.)

[49] S. Pestov, "Library evolution in Swift," 2020. https://www.swift.org/blog/library-evolution/
(Cited on pages 56 and 196.)

[50] J. Rose and S. Pestov, "Library evolution," 2015.
https://github.com/swiftlang/swift/blob/main/docs/LibraryEvolution.rst
(Cited on pages 56 and 196.)

[51] S. Pestov, "SE-0193: Cross-module inlining and specialization," 2018. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0193-cross-module-inlining-and-specialization.md
(Cited on page 57.)

[52] J. Rose and B. Cohen, "SE-0260: Library evolution for stable ABIs," 2019.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0260-library-evolution.md
(Cited on page 57.)

[53] A. Wagner, Z. Eisbach, and A. Ahmed, "Realistic realizability: Specifying ABIs you can count on," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. https://doi.org/10.1145/3689755
(Cited on page 57.)

[54] A. Zubarev and A. Zheng, "SE-0095: Replace `protocol<P1,P2>` syntax with `P1 & P2` syntax," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0095-any-as-existential.md
(Cited on page 73.)

[55] P. Yaskevich, H. Borla, and S. Pestov, "SE-0346: Lightweight same-type requirements for primary associated types," 2022. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0346-light-weight-same-type-syntax.md (Cited on pages 73 and 105.)

[56] H. Borla, "SE-0335: Introduce existential `any`," 2021. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0335-existential-any.md (Cited on page 73.)

[57] J. McCall and D. Gregor, "SE-0296: Async/await," 2020. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0296-async-await.md (Cited on page 75.)

[58] D. Racordon, D. Shabalin, D. Zheng, D. Abrahams, and B. Saeta, "Mutable value semantics," *Journal of Object Technology*, vol. 21, 2022. http://www.jot.fm/issues/issue_2022_02/article2.pdf (Cited on page 76.)

[59] M. Gottesman and J. Groff, "SE-0377: `borrowing` and `consuming` parameter ownership modifiers," 2023. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0377-parameter-ownership-modifiers.md (Cited on page 76.)

[60] D. Gregor, "SE-0021: Naming functions with argument labels," 2016. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0021-generalized-naming.md (Cited on page 78.)

[61] A. Zheng, "SE-0111: Remove type system significance of function argument labels," 2016. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0111-remove-arg-label-type-significance.md (Cited on page 78.)

[62] C. Lattner, "SE-0029: Remove implicit tuple splat behavior from function applications," 2016. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0029-remove-implicit-tuple-splat.md (Cited on page 78.)

[63] C. Lattner, "SE-0066: Standardize function type argument syntax to require parentheses," 2016. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0066-standardize-function-type-syntax.md (Cited on page 78.)

[64] V. S. and A. Zheng, "SE-0110: Distinguish between single-tuple and multiple-argument function types," 2016. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0110-distinguish-single-tuple-arg.md (Cited on page 78.)

[65] J. Wells, "Typability and type checking in System F are equivalent and undecidable," *Annals of Pure and Applied Logic*, vol. 98, no. 1, pp. 111–156, 1999. https://www.sciencedirect.com/science/article/pii/S0168007298000475 (Cited on page 79.)

[66] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields, "Practical type inference for arbitrary-rank types," *J. Funct. Program.*, vol. 17, no. 1, p. 1–82, jan 2007. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/putting.pdf (Cited on page 79.)

[67] F. Kellison-Linn, "SE-0315: Type placeholders," 2021. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0315-placeholder-types.md (Cited on page 79.)

[68] D. Gregor, P. Yaskevich, and H. Borla, "Type checker design and implementation," 2020. https://github.com/swiftlang/swift/blob/main/docs/TypeChecker.md (Cited on page 82.)

[69] P. Yaskevich, "New diagnostic architecture overview," 2019.
https://www.swift.org/blog/new-diagnostic-arch-overview/
(Cited on page 82.)

[70] J. McCall, D. Gregor, K. Malawski, and C. Lattner, "SE-0306: Actors," 2020.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0306-actors.md
(Cited on page 94.)

[71] D. Hart, R. Widmann, and P. Jahkola, "SE-0081: Move `where` clause to end of declaration," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0081-move-where-expression.md
(Cited on page 103.)

[72] C. Lattner, "SE-0048: Generic type aliases," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0048-generic-typealias.md
(Cited on page 103.)

[73] D. Hart and A. Zheng, "SE-0156: Class and subtype existentials," 2017.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0156-subclass-existentials.md
(Cited on page 103.)

[74] C. Eidhof, "SE-0148: Generic subscripts," 2017.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0148-generic-subscripts.md
(Cited on page 103.)

[75] A. Latsis, "SE-0261: `where` clauses on contextually generic declarations," 2019. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0267-where-on-contextually-generic.md
(Cited on page 103.)

[76] D. Gregor, "SE-0341: Opaque parameter declarations," 2022.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0341-opaque-parameters.md
(Cited on pages 103 and 442.)

[77] D. Hart, J. Bandes-Storch, and D. Gregor, "SE-0142: Permit where clauses to constrain associated types," 2017.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0142-associated-types-constraints.md
(Cited on pages 107, 273, and 523.)

[78] J. Groff, "SE-0035: Limiting `inout` capture to `@noescape` contexts," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0035-limit-inout-capture.md
(Cited on page 113.)

[79] B. Royal-Gordon, "SE-0254: Static and class subscripts," 2019.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0254-static-subscripts.md
(Cited on page 115.)

[80] H. Curry, *Foundations of Mathematical Logic*. Dover Publications, 1977.
https://store.doverpublications.com/products/9780486634623
(Cited on page 134.)

[81] R. Grimaldi, *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison-Wesley Longman, 1998. https://www.goodreads.com/en/book/show/1575542
(Cited on pages 159, 263, and 375.)

[82] K. Wagner, "SE-0404: Nested protocols," 2023.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0404-nested-protocols.md
(Cited on page 185.)

[83] M. Sulzmann, T. Schrijvers, and P. J. Stuckey, "Principal type inference for GHC-style multi-parameter type classes," in *Asian Symposium on Programming Languages and Systems*, 2006. https://lirias.kuleuven.be/retrieve/25382/
(Cited on page 185.)

[84] B. Milewski, *Category Theory for Programmers*. Blurb, Incorporated, 2018. https://github.com/hmemcpy/milewski-ctfp-pdf/
(Cited on page 189.)

[85] "SIL type lowering," 2024. https://github.com/swiftlang/swift/blob/main/docs/SIL/Types.md
(Cited on pages 195 and 199.)

[86] "Type layout," 2017. https://github.com/swiftlang/swift/blob/main/docs/ABI/TypeLayout.rst
(Cited on page 197.)

[87] Aleksandr, "Value wrapped in a thunk recursively(?)," 2024. https://forums.swift.org/t/value-wrapped-in-a-thunk-recursively/75925
(Cited on page 203.)

[88] J. McCall, "Abstract the object type of optional types," 2016. https://github.com/swiftlang/swift/pull/4689
(Cited on page 207.)

[89] S. Pestov, "SIL: Stop imploding parameter list into a single value with opaque abstraction pattern," 2018. https://github.com/swiftlang/swift/pull/19578
(Cited on page 207.)

[90] J. Groff, "SILGen: Emit literal closures at the abstraction level of their context. [take 3]," 2021. https://github.com/swiftlang/swift/pull/39233
(Cited on page 207.)

[91] J. Groff, "Improving the representation of polymorphic interfaces in SIL with "substituted function types"," 2019. https://forums.swift.org/t/improving-the-representation-of-polymorphic-interfaces-in-sil-with-substituted-function-types/29711
(Cited on page 211.)

[92] H. Haskins, "SE-0364: Warning for retroactive conformances of external types," 2023. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0364-retroactive-conformance-warning.md
(Cited on page 216.)

[93] "SR-2235: Redeclared associatedtype inference not working," 2016. https://github.com/swiftlang/swift/issues/44842
(Cited on page 243.)

[94] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, ser. Mathematical Sciences Series. Freeman, 1979. https://www.goodreads.com/book/show/284369.Computers_and_Intractability
(Cited on page 246.)

[95] J. MacCormick, *What Can Be Computed?: A Practical Guide to the Theory of Computation*. Princeton University Press, 2018. https://whatcanbecomputed.com
(Cited on pages 246 and 423.)

[96] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158. https://dl.acm.org/doi/pdf/10.1145/800157.805047
(Cited on page 246.)

[97] S. Aaronson, "P $\stackrel{?}{=}$ NP," 2017. https://www.scottaaronson.com/papers/pnp.pdf
(Cited on page 247.)

[98] E. Lippert, "Lambda expressions vs. anonymous methods, part five," 2007. https://learn.microsoft.com/en-us/archive/blogs/ericlippert/lambda-expressions-vs-anonymous-methods-part-five
(Cited on page 247.)

[99] D. E. Knuth, *The Art of Computer Programming: Volume 4B: Combinatorial Algorithms*. Addison-Wesley, 2022. https://www-cs-faculty.stanford.edu/~knuth/taocp.html
(Cited on page 247.)

[100] U. Schöning and J. Torán, *The Satisfiability Problem: Algorithms and Analyses*, ser. Mathematik für Anwendungen. Lehmanns Media, 2013.
https://www.goodreads.com/book/show/27394636-the-satisfiability-problem
(Cited on page 247.)

[101] U. Knauer, *Algebraic Graph Theory: Morphisms, Monoids and Matrices*. De Gruyter, 2019.
https://www.degruyter.com/document/doi/10.1515/9783110617368/html?lang=en
(Cited on page 263.)

[102] D. E. Knuth, *The Art of Computer Programming: Volume 1: Fundamental Algorithms*. Addison-Wesley, 1997. https://www-cs-faculty.stanford.edu/~knuth/taocp.html
(Cited on pages 273 and 418.)

[103] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Commun. ACM*, vol. 7, no. 5, p. 301–303, may 1964. https://dl.acm.org/doi/pdf/10.1145/364099.364331
(Cited on page 273.)

[104] Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Comput. Surv.*, vol. 23, no. 3, p. 319–344, sep 1991. https://dl.acm.org/doi/pdf/10.1145/116873.116878
(Cited on page 273.)

[105] D. Gregor, E. Sadun, and A. Zheng, "SE-0157: Support recursive constraints on associated types," 2017.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0157-recursive-protocol-constraints.md
(Cited on pages 273, 415, and 523.)

[106] D. Gregor, "Implementing recursive protocol constraints," 2016.
https://gist.github.com/DougGregor/e7c4e7bb4465d6f5fa2b59be72dbdba6
(Cited on page 273.)

[107] T. Nadeau, "SE-0103: Make non-escaping closures the default," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0103-make-noescape-default.md
(Cited on page 277.)

[108] E. Sadun, "SE-0068: Expanding swift `Self` to class members and value types," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0068-universal-self.md
(Cited on page 282.)

[109] L. Lecrenier, "SE-0011: Replace `typealias` keyword with `associatedtype` for associated type declarations," 2015.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0011-replace-typealias-associated.md
(Cited on page 295.)

[110] D. Hart and D. Gregor, "SE-0092: Typealiases in protocols and protocol extensions," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0092-typealiases-in-protocols.md
(Cited on page 295.)

[111] "SR-631: Extensions in different files do not recognize each other," 2016.
https://github.com/swiftlang/swift/issues/43248
(Cited on page 323.)

[112] H. Borla, "SE-0361: Extensions on bound generic types," 2022.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0361-bound-generic-extensions.md
(Cited on page 331.)

[113] D. Gregor, "SE-0143: Conditional conformances," 2016.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0143-conditional-conformances.md
(Cited on page 333.)

[114] "SR-6724: Swift 4.1 crash when using conditional conformance," 2018.
https://github.com/swiftlang/swift/issues/49273
(Cited on page 339.)

[115] S. Leffler, "Rust's type system is Turing-Complete," 2017.
https://sdleffler.github.io/RustTypeSystemTuringComplete/
(Cited on page 340.)

[116] J. Revuelta, T. Lehmann, and D. Gregor, "SE-0413: Typed throws," 2023.
https://github.com/swiftlang/swift-evolution/blob/main/proposals/0413-typed-throws.md
(Cited on page 355.)

[117] A. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007. https://link.springer.com/book/10.1007/978-3-540-74113-8
(Cited on page 375.)

[118] D. Gregor, "Generic signatures," 2018.
https://github.com/swiftlang/swift/blob/main/docs/ABI/GenericSignature.md
(Cited on page 377.)

[119] D. Kőnig, *Theory of Finite and Infinite Graphs*. Birkhäuser Boston, 1936.
https://www.goodreads.com/book/show/3359970-theory-of-finite-and-infinite-graphs
(Cited on page 418.)

[120] N. Cutland, *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980. https://www.cambridge.org/us/universitypress/subjects/computer-science/programming-languages-and-applied-logic/computability-introduction-recursive-function-theory?format=PB
(Cited on page 423.)

[121] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 01 1937.
https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf
(Cited on page 423.)

[122] A. Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, vol. 58, p. 345, 1936. https://www.semanticscholar.org/paper/An-Unsolvable-Problem-of-Elementary-Number-Theory-Church/60400c043b2624f9cfc2d8daa0f45f3c1d524de3
(Cited on page 425.)

[123] E. L. Post, "Formal reductions of the general combinatorial decision problem," *American Journal of Mathematics*, vol. 65, no. 2, pp. 197–215, 1943. http://www.jstor.org/stable/2371809
(Cited on page 426.)

[124] M. L. Minsky, "Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines," *Annals of Mathematics*, vol. 74, no. 3, pp. 437–455, 1961. http://www.jstor.org/stable/1970290
(Cited on page 426.)

[125] M. L. Minsky, *Computation: Finite and Infinite Machines*. USA: Prentice-Hall, Inc., 1967. https://www.goodreads.com/en/book/show/326791.Computation
(Cited on page 426.)

[126] L. De Mol, "Tag systems and Collatz-like functions," *Theoretical Computer Science*, vol. 390, no. 1, pp. 92–101, 2008. https://www.sciencedirect.com/science/article/pii/S0304397507007700
(Cited on page 427.)

[127] K. Bauer, "BrainF*** in the Swift type system," 2023. https://forums.swift.org/t/brainf-in-the-swift-type-system/68301
(Cited on page 435.)

[128] D. Gregor and P. Yaskevich, "SE-0252: Key path member lookup," 2019. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0252-keypath-dynamic-member-lookup.md
(Cited on page 435.)

[129] R. Tate, A. Leung, and S. Lerner, "Taming wildcards in Java's type system," *SIGPLAN Not.*, vol. 46, no. 6, p. 614–627, Jun. 2011. https://rosstate.org/publications/tamewild/tamewild-tate-pldi11-tr.pdf
(Cited on page 435.)

[130] R. Grigore, "Java generics are Turing complete," *SIGPLAN Not.*, vol. 52, no. 1, p. 73–85, Jan 2017. https://arxiv.org/abs/1605.05274
(Cited on page 435.)

[131] R. Ostrow, "Taking types too far," 2019. https://ostro.ws/post-taking-types-too-far
(Cited on page 435.)

[132] J. Lagarias, *The Ultimate Challenge: The* $3x + 1$ *Problem*, ser. Monograph Bks. American Mathematical Society, 2010. https://bookstore.ams.org/mbk-78
(Cited on page 435.)

[133] S. Wolfram, "After 100 years, can we finally crack Post's problem of "Tag?" A story of computational irreducibility, and more." https://writings.stephenwolfram.com/2021/03/after-100-years-can-we-finally-crack-posts-problem-of-tag-a-story-of-computational-irreducibility-and-more/
(Cited on page 435.)

[134] D. Gregor and J. Groff, "SE-0244: Opaque result types," 2019. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0244-opaque-result-types.md
(Cited on page 442.)

[135] B. Driscoll and H. Borla, "SE-0328: Structural opaque result types," 2021. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0328-structural-opaque-result-types.md
(Cited on page 442.)

[136] P. Yaskevich, "SE-0360: Opaque result types with limited availability," 2022. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0360-opaque-result-types-with-availability.md
(Cited on page 442.)

[137] G. Roué, "Swift 5.7: Incorrect compiler error with opaque result types and primary associated types," 2022. https://github.com/swiftlang/swift/issues/59391
(Cited on page 452.)

[138] D. Zarzycki, "[NFC] Change magic emoji to __," 2021. https://github.com/swiftlang/swift/pull/35734
(Cited on page 457.)

[139] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972. https://github.com/tpn/pdfs/blob/master/Depth-First%20Search%20and%20Linear%20Graph%20Algorithms%20-%20Tarjan%20(1972).pdf
(Cited on page 484.)

[140] J. Howie, *Fundamentals of Semigroup Theory*, ser. LMS monographs. Clarendon Press, 1995. https://www.goodreads.com/book/show/3420594-fundamentals-of-semigroup-theory (Cited on page 495.)

[141] J. Smith and A. Romanowska, *Post-Modern Algebra*. Wiley, 2011. https://www.wiley.com/en-us/Post+Modern+Algebra-p-9780471127383 (Cited on page 495.)

[142] C. C. Squier, F. Otto, and Y. Kobayashi, "A finiteness condition for rewriting systems," *Theoretical Computer Science*, vol. 131, no. 2, pp. 271–294, 1994. https://www.sciencedirect.com/science/article/pii/0304397594901759 (Cited on pages 503 and 530.)

[143] L. Brodie, *Thinking Forth: A Language and Philosophy for Solving Problems*. Punchy Publishing, 2004. https://www.forth.com/wp-content/uploads/2018/11/thinking-forth-color.pdf (Cited on page 511.)

[144] M. von Thun, "Joy programming language." https://hypercubed.github.io/joy/joy.html (Cited on page 511.)

[145] S. Pestov, D. Ehrenberg, and J. Groff, "Factor: a dynamic stack-based programming language," *SIGPLAN Not.*, vol. 45, no. 12, p. 43–58, oct 2010. https://factorcode.org/slava/dls.pdf (Cited on page 511.)

[146] J. F. Power, "Thue's 1914 paper: a translation," *CoRR*, vol. abs/1308.5858, 2013. http://arxiv.org/abs/1308.5858 (Cited on page 520.)

[147] E. L. Post, "Recursive unsolvability of a problem of Thue," *The Journal of Symbolic Logic*, vol. 12, no. 1, p. 1–11, 1947. https://www.wolframscience.com/prizes/tm23/images/Post2.pdf (Cited on page 520.)

[148] G. S. Tseitin, "An associative calculus with an insoluble problem of equivalence," in *Problems of the constructive direction in mathematics. Part 1*. Moscow–Leningrad: Acad. Sci. USSR, 1958, vol. 52, pp. 172–189. https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=tm&paperid=1317&option_lang=eng (Cited on page 521.)

[149] C.-F. Nyberg-Brodda, "G. S. Tseytin's seven-relation semigroup with undecidable word problem," 2024. https://arxiv.org/abs/2401.11757 (Cited on page 521.)

[150] W. W. Boone, "The word problem," *Annals of Mathematics*, vol. 70, no. 2, pp. 207–265, 1959. http://www.jstor.org/stable/1970103 (Cited on page 522.)

[151] W. Cravitz, "An introduction to the word problem for groups," 2021. https://math.uchicago.edu/~may/REU2021/REUPapers/Cravitz.pdf (Cited on page 522.)

[152] J. J. Rotman, *An Introduction to the Theory of Groups*. Springer, 1994. https://link.springer.com/book/10.1007/978-1-4612-4176-8 (Cited on page 522.)

[153] D. J. Collins, "A universal semigroup," *Algebra and Logic*, vol. 9, no. 6, pp. 442–446, 1970. https://m.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=al&paperid=1278&option_lang=rus (Cited on page 523.)

[154] R. Book and F. Otto, *String-Rewriting Systems*, ser. Monographs in Computer Science. Springer, 2012. https://link.springer.com/book/10.1007/978-1-4613-9771-7
(Cited on pages 525, 528, 585, and 620.)

[155] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998. https://www21.in.tum.de/~nipkow/TRaAT/
(Cited on pages 525 and 585.)

[156] N. Dershowitz and J.-P. Jouannaud, "Chapter 6 - Rewrite Systems," in *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*. Amsterdam: Elsevier, 1990. https://www.goodreads.com/book/show/2305428.Handbook_of_Theoretical_Computer_Science_Vol_B
(Cited on page 525.)

[157] A. Church and J. B. Rosser, "Some properties of conversion," *Transactions of the American Mathematical Society*, vol. 39, no. 3, pp. 472–482, 1936. https://www.ams.org/journals/tran/1936-039-03/S0002-9947-1936-1501858-0/S0002-9947-1936-1501858-0.pdf
(Cited on page 528.)

[158] H. Curry and R. Feys, *Combinatory Logic*. North-Holland Publishing Company, 1958, vol. 1. https://www.goodreads.com/book/show/65701927-combinatory-logic
(Cited on page 528.)

[159] C. Ó'Dúnlaing, "Undecidable questions related to Church-Rosser Thue systems," *Theoretical Computer Science*, vol. 23, no. 3, pp. 339–345, 1983. https://www.sciencedirect.com/science/article/pii/0304397583900397
(Cited on page 530.)

[160] D. Kapur and P. Narendran, "A finite Thue system with decidable word problem and without equivalent finite canonical system," *Theoretical Computer Science*, vol. 35, pp. 337–344, 1985. https://www.sciencedirect.com/science/article/pii/0304397585900234
(Cited on page 530.)

[161] Y. Lafont and A. Prouté, "Church-Rosser property and homology of monoids," *Mathematical Structures in Computer Science*, vol. 1, pp. 297 – 326, 1991. https://www.irif.fr/~mellies/mpri/mpri-ens/articles/lafont-proute-church-rosser-property-and-homology-of-monoids.pdf
(Cited on page 530.)

[162] Y. Lafont, "A new finiteness condition for monoids presented by complete rewriting systems (after Craig C. Squier)," *Journal of Pure and Applied Algebra*, vol. 98, no. 3, pp. 229–244, 1995. https://www.sciencedirect.com/science/article/pii/002240499400043I
(Cited on page 530.)

[163] C. C. Squier, "Word problems and a homological finiteness condition for monoids," *Journal of Pure and Applied Algebra*, vol. 49, no. 1, pp. 201–217, 1987. https://www.sciencedirect.com/science/article/pii/0022404987901290
(Cited on page 531.)

[164] R. Cremanns and F. Otto, "Finite derivation type implies the homological finiteness condition FP3," *Journal of Symbolic Computation*, vol. 18, no. 2, pp. 91–112, 1994. https://www.sciencedirect.com/science/article/pii/S074771718471039X
(Cited on page 531.)

[165] Y. Kobayashi, "A finitely presented monoid which has solvable word problem but has no regular complete presentation," *Theoretical Computer Science*, vol. 146, no. 1, pp. 321–329, 1995. https://www.sciencedirect.com/science/article/pii/030439759400264J
(Cited on page 531.)

[166] F. Otto and Y. Kobayashi, "Properties of monoids that are presented by finite convergent string-rewriting systems — a survey," in *Advances in Algorithms, Languages, and Complexity*. Springer, 1997, pp. 225–266. https://static.aminer.org/pdf/PDF/000/066/293/properties_of_monoids_that_are_presented_by_finite_convergent_string.pdf
(Cited on page 531.)

[167] Y. Kobayashi, "Finite homotopy bases of one-relator monoids," *Journal of Algebra*, vol. 229, no. 2, pp. 547–569, 2000. https://www.sciencedirect.com/science/article/pii/S0021869399982510
(Cited on page 531.)

[168] C.-F. Nyberg-Brodda, "The word problem for one-relation monoids: a survey," *Semigroup Forum*, vol. 103, no. 2, pp. 297–355, 2021. https://link.springer.com/article/10.1007/s00233-021-10216-8
(Cited on page 531.)

[169] D. Epstein, *Word Processing in Groups*. CRC Press, 1992. https://www.taylorfrancis.com/books/mono/10.1201/9781439865699/word-processing-groups-david-epstein
(Cited on page 531.)

[170] A. Tarski, A. Mostowski, and R. Robinson, *Undecidable Theories*, ser. Studies in logic and the foundations of mathematics. North-Holland, 1953. https://www.goodreads.com/book/show/7439448-undecidable-theories
(Cited on page 531.)

[171] D. E. Knuth, *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley, 1998. https://www-cs-faculty.stanford.edu/~knuth/taocp.html
(Cited on page 575.)

[172] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, p. 333–340, jun 1975. https://doi.org/10.1145/360825.360855
(Cited on page 575.)

[173] B. Meyer, "Incremental string matching," *Information Processing Letters*, vol. 21, no. 5, pp. 219–227, 1985. https://se.inf.ethz.ch/~meyer/publications/string/string_matching.pdf
(Cited on page 576.)

[174] D. E. Knuth and P. B. Bendix, "Simple word problems in universal algebras," in *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Springer, 1983, pp. 342–376. https://www.semanticscholar.org/paper/Simple-Word-Problems-in-Universal-Algebras-Knuth-Bendix/94877bdf8313565b90758a5e664764139857b358
(Cited on page 585.)

[175] G. Huet, "A complete proof of correctness of the Knuth-Bendix completion algorithm," *Journal of Computer and System Sciences*, vol. 23, no. 1, pp. 11–21, 1981. https://www.sciencedirect.com/science/article/pii/0022000081900027
(Cited on page 585.)

[176] D. Kapur and P. Narendran, "The Knuth-Bendix completion procedure and Thue systems," *SIAM Journal on Computing*, vol. 14, no. 4, pp. 1052–1072, 1985. https://doi.org/10.1137/0214073
(Cited on page 585.)

[177] B. Buchberger, "History and basic features of the critical-pair/completion procedure," *Journal of Symbolic Computation*, vol. 3, no. 1, pp. 3–38, 1987. https://www.sciencedirect.com/science/article/pii/S0747717187800202
(Cited on page 585.)

[178] M. H. A. Newman, "On theories with a combinatorial definition of equivalence," *Annals of Mathematics*, vol. 43, no. 2, pp. 223–243, 1942. http://www.ens-lyon.fr/LIP/REWRITING/TERMINATION/NEWMAN/Newman.pdf
(Cited on page 586.)

[179] A. Heyworth and M. Johnson, "Logged rewriting for monoids," 2005. https://arxiv.org/abs/math/0507344
(Cited on page 592.)

[180] Y. Guiraud, P. Malbos, and S. Mimram, "A homotopical completion procedure with applications to coherence of monoids," in *RTA - 24th International Conference on Rewriting Techniques and Applications - 2013*, vol. 21, Jun. 2013, pp. 223–238. https://hal.inria.fr/hal-00818253
(Cited on page 592.)

[181] "SR-12120: Compiler forgets some constraints of P within extension to P, known bug?" 2020.
https://github.com/swiftlang/swift/issues/54555
(Cited on page 617.)

[182] W. Magnus, A. Karrass, and D. Solitar, *Combinatorial Group Theory: Presentations of Groups in Terms of Generators and Relations*. Dover Publications, 1976.
https://www.goodreads.com/book/show/331129.Combinatorial_Group_Theory
(Cited on page 620.)

[183] S. Henry and S. Mimram, "Tietze equivalences as weak equivalences," 2021.
https://arxiv.org/abs/2101.03591
(Cited on page 620.)

[184] F. Otto, M. Katsura, and Y. Kobayashi, "Infinite convergent string-rewriting systems and cross-sections for finitely presented monoids," *Journal of Symbolic Computation*, vol. 26, no. 5, pp. 621–648, 1998.
https://www.sciencedirect.com/science/article/pii/S0747717198902309
(Cited on page 633.)

[185] D. Perrin, "Chapter 1 - Finite Automata," in *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*. Amsterdam: Elsevier, 1990.
https://www.goodreads.com/book/show/2305428.Handbook_of_Theoretical_Computer_Science_Vol_B
(Cited on page 633.)

[186] R. E. Needham, "Infinite complete group presentations," *Journal of Pure and Applied Algebra*, vol. 110, no. 2, pp. 195–218, 1996. https://www.sciencedirect.com/science/article/pii/0022404995001085
(Cited on page 633.)

Index

A **bold** page number is where the concept is first defined; an *italic* page number is where the implementation is shown in a "Source Code Reference" section. A plain page number is where the concept is used.

Symbols

1_G, 181
    see also identity substitution map
1_t, 504
    see also empty rewrite path
$[\![x]\!]$, 141
    see also equivalence class
$1_{[\![G]\!]}$, 261
    see also forwarding substitution map
$\mathbb{N}$, 643
    see also natural numbers
$[\![\circlearrowleft \mathtt{T}]\!]$, 442
    see also opaque archetype
$\Rightarrow$, 503
    see also rewrite path, rewrite step
$\triangleleft$, 507
    see also rewrite path whiskering
$\triangleright$, 507
    see also rewrite path whiskering
$[\![\mathtt{T}]\!]$, 256
    see also archetype
$\bot$, 392, 527, 554, 559
    see also partial order
$\cap$, 644
    see also intersection
$\cdot$, 495, 534
    see also monoid
$\circ$, 506
    see also rewrite path
$\cup$, 644
    see also union
$\in$, 643
    see also set
$\mathbb{Z}$, 643
    see also integers
$\notin$, 643
$\otimes$, 177, 186, 218, 222, 225, 232, 298, 647
    see also type substitution
$\prec$, 487
    see also protocol dependency
$\rightarrow$, 524
    see also reduction relation
$\setminus$, 644
    see also set difference
$\subseteq$, 644
    see also subset
$\subsetneq$, 644
    see also proper subset
$\times$, 644
    see also Cartesian product
$\varnothing$, 643
    see also empty set
$\vdash$, 135, 645
    see also derivation

A

Abelian group, 502
ABI, 57, 103, 158, 183, 354, 377, 383, 385, 600, 613
    conformance path, 411
abstract conformance, 213, **227**, *250*, 286, 301
    associated conformance, 232
    equivalence, **404**
    identity substitution map, 181
    monoid, 518
    opaque archetype, 443, 447
    recursive, 421
    with archetype, 260
abstract generic signature request, **350**, 359, *394*, 475, *492*
    opaque result type, 438

abstract syntax tree, **39**, 40, 45, 52, 55, *57*, 195, 462
abstraction pattern, **200**, 201, 205, *211*
access control, 463
accessor declaration, 114, **116**, *126*
active request, **50**
address-only type, **196**, 198
Aho-Corasick algorithm, **576**
Alan Turing, 423
Alfred Aho, 575
all inherited protocols request, 550
Alonzo Church, 425, 528
alphabet
  in requirement machine, **549**
  see also generating set
`Any`, **73**, 102, 358, 439
`AnyObject`, 73, 100, 164, 196, 301, 549
`AnyObject` lookup, *see* dynamic lookup
archetype type, 20, 30, **72**, 178, 297
  opaque archetype, 442
  SIL type lowering, 196, 198
`ArchetypeBuilder`, **269**
`areReducedTypeParametersEqual`(), **159**, *168*, 273, 571
argument label, 77
array sugared type, **78**, *86*
assembly language, 40
AssocBind derivation step, **150**, 374, 376, 558, 562
  summary, 646
AssocConcrete derivation step, **139**
  summary, 646
AssocConfp, **232**, 648
AssocConf derivation step, **137**, 232, 374, 405, 408, 481, 513, 536, 538, 545
  summary, 646
AssocDecl derivation step, **150**, 227, 374
  summary, 646
associated conformance, 34, **231**, *250*
  of abstract conformance, 260
  opaque archetype, 454
  summary, 648
associated conformance projection, 34, **232**, 402, 416, 419
  conformance path graph, 409
  summary, 648
associated conformance request, **231**, *254*
associated conformance requirement, 33, 231, 403, 601
  free monoid, 512
  in requirement machine, 533, 544
  inference rule, 137
  protocol dependency, 359, 477, 480, 484
  protocol dependency graph, 414
  protocol inheritance, 336
  recursive, 415
  summary, 648
associated layout requirement, 104
  inference rule, 139
associated requirement, 32, **104**, 340, 351, 392
  conformance checker, 231
  in requirement machine, 533, 561
  inference rule, 139, 374
  requirement signature, 132
  summary, 645
associated same-type requirement, 33, 240
  associated type inference, 241
  finitely-presented monoid, 514
  in requirement machine, 533, 546
  inference rule, 137, 139
  protocol inheritance, 353
  protocol type alias, 566
associated superclass requirement, 390
  inference rule, 139
associated type declaration, 28, **104**, *125*
  bound dependent member type, 366
  compared to associated type symbol, 551
  dependent member type, 71
  in requirement machine, 562
  inference rule, 136, 150
  inheritance clause, 351
  requirement signature, 132
  summary, 645, 648
  type resolution, 292
  type witness, 222, 224, 386
  unqualified lookup, 283, 285
associated type inference, 28, 222, **236**, *254*
  extension binding, 324
  opaque result type, 454
  request cycle, 490
  textual interfaces, 56
  type resolution, 283
associated type order, **156**
associated type reduction order, **551**
associated type rule, 551, **562**, 567, 571, 610, 619
associated type symbol, **549**, 555, 556, *577*, 593, 612, 619
associative operation, 188, 219, 220, **495**, 531
AssocLayout derivation step, **139**, 374

summary, 646
AssocName derivation step, **136**, 150, 374, 513
summary, 646
AssocSame derivation step, **137**, 375, 516, 517, 538, 547
summary, 646
AssocSuper derivation step, **139**, 375
summary, 646
AssocTypeP, **225**, 648
AST, *see* abstract syntax tree
AST context, *58*, *490*
AST lowering request, **48**, 52
AST printer, **56**, *64*, 457
asymptotic complexity, 246, 413, 572
autoclosure function type, **76**
availability, 440, 460
`awk`, 51
Axel Thue, 520

B

base case, **372**
basepoint, **590**
batch mode, **38**, 53
bicyclic monoid, **501**, 514, 519, 527
special monoid presentation, 522
binary module, *see* serialized module, 462, 563
binary operation, 177, 232, 495, 513, **644**
finitely-presented monoid, 499
free monoid, 496
in requirement machine, 534
Boolean formula, **243**
bound dependent member type, 30, **71**, 406
associated type declaration, 104
equivalence, 342
in requirement machine, 556
inference rule, 150
linear order, 157
summary, 645
type resolution, 288
type substitution, 149, 377
validity, 160, 227, 366
bound type parameter, **149**
bound and unbound form, 375
in requirement machine, 556, 601
type substitution, 154
boxing, 27, 36, 94, 113
breadth-first search, 411, 484
bridging header, **56**
built-in module, **83**
built-in type, **83**

C

C, 47, 56
C♯, 247
C++, 13, 20, 35, 56, 190
call expression, 21, 30, 74, 75, 109, 257
calling convention, 20, 26, 35, 158, 195, 199, 204
candidate value witness, **237**, *254*
opaque result type, 454
canonical conformance, *208*, **250**
canonical generic signature, **131**, *169*, 474
canonical SIL, **39**
canonical substitution map, *208*, 252
canonical type, **68**, *87*, *208*, 282
in requirement machine, 549
reduced type, 163
canonical type equality, **68**, *84*, 99, 140, 163, 206, 258, 301, 342, 554
same-type requirement, 301, 387
capture info request, **111**
captured value, 75, **110**, *126*, 199
Java inner class, 109
Cartesian product, 140, 500, **644**
category, **189**
Cayley graph, 500, 501, 503, 512, 518
finitely-presented monoid, 499
free monoid, 496, 498
trivial monoid, 498
Cayley table, **499**, 522
Church-Rosser property, **528**, 586
circular inheritance, 50
circular reference, 50, 464
Clang, 56, 345
Clang file unit, **55**
Clang importer, **55**, *64*
class declaration, **94**, 97, *119*, 194, *209*, 301, 365
extension, 322
inherited conformance, 214
superclass requirement, 389
superclass type, 190
class hierarchy diagram, 190
class type, **70**, 99, 114, 139, *209*, 389
constraint type, 100
extended type, 322
opaque result type, 438
SIL type lowering, 198, 205
summary, 645

superclass type, 192
class-constrained protocol, **104**, *170*
clause in SAT, 244
closure context, 75, **113**, 199
closure conversion, **110**, *126*
closure expression, 76, 77, 93, 110, *120*, 201
re-abstraction thunk, 201
SIL type lowering, 199, 200
closure function, 111, 112
CNF, *see* conjunctive normal form
coarser relation, 163
Collatz conjecture, 427
commutative diagram, **188**, 586
conditional conformance, 335
global conformance lookup, 216
type witness projection, 226
commutative operation, 496, 500, 502
compiler intrinsic, **83**
completion, 395, 474, 489, **529**, 563, 571
see also Knuth-Bendix algorithm
complex numbers, 502
composition
of morphisms, 189
of substitution maps, 186
computable number, 423
concatenative language, **511**
concrete conformance, **213**, *250*, 287, 301, 365, 391
concrete conformance symbol, **549**, 554, *577*, *581*
concrete contraction, 289, **641**
CONCRETE derivation step, **139**
summary, 645
concrete metatype type, **74**
concrete type symbol, **549**, 552, 560, *577*
conditional conformance, *248*, 301, 304, **333**, *346*, 356, 363
conditional requirement, *248*, *315*, **334**, *346*, 363, 365
CONF(G), **219**, 647
CONF derivation step, **136**, 405, 481, 513, 537, 544
summary, 645
conflicting requirement, 349, 362, 386, **389**, 392, *394*, 475, 476, 571
conflicting rule, 571, *582*
confluence, **528**, 531, 586
confluence violation, **528**, 586, 587, 589
confluent reduction relation, 572
conformance, 26, **213**, 236, *249*, 479, 647
conformance substitution graph, 419
replace opaque archetype, 463
conformance checker, 26, **224**, 231, 304, *315*, 337, 341
extension binding, 323
conformance lookup callback, *253*
conformance lookup table, **214**, 215, *248*, 337
conformance path, 233, **403**, *435*
equivalence, **411**
linear order, **411**
conformance path evaluation, 403
conformance path graph, **409**, 417
monoid, 518
conformance path length, **403**
conformance path order, **411**
conformance requirement, 24, **99**, *169*, *252*, 401
checking, 72, 301, 303
concrete conformance, 391, 554
conflicts, 386, 389
decomposition, 359
desugaring, 365
elementary statement, 136
formal system, 136
generic signature query, 159
in archetype builder, 270
in requirement machine, 533, 543, 552, 560, 561
opaque result type, 438
protocol dependency, 474, 487
reduced, 381
summary, 645
type parameter graph, 265
type substitution, 178, 227, 228, 298
well-formed, **367**
witness table, 158
conformance rule, *581*
conformance substitution, **218**, *250*
opaque archetype, 447
conformance substitution graph, **419**
conformance substitution map, **219**, 219, *251*, 335, *346*
summary, 647
conforming type, 26, 103, *249*, *250*, 283
inherited conformance, 215
normal conformance, 214
CONFP(G), 648
conjunction, 243
conjunctive normal form, 244
connected components, **414**

constrained extension, 103, 291, **330**, 342, *345*, 348, 605
constrained protocol type, *see* parameterized protocol type
constraint requirement representation, **102**
constraint solver arena, **82**, *89*
constraint type, 72, **100**, 361, 570
  opaque parameter, 101
  protocol inheritance clause, 104
constructor declaration, 97, **110**, *125*
context substitution map, 23, 70, **180**, 182, 198, *207*
  class member, 193
  conditional conformance, 304
  conformance lookup, 218
  for a declaration context, **294**, 335
  non-generic type, 182
  of constrained extension, 291
  of local type, 183
  opaque archetype, 445
  superclass type, 192
  type resolution, 290, 302
contextual type, 72, 182, **255**, *310*
  opaque archetype, 448
contextually-generic declaration, **102**, 301, *315*
  history, 103
convention
  SIL type lowering, 199
convergent rewriting system, 395, 474, **529**, 548, 585, 597, 605
coroutine, 594
Craig Squier, 530
critical pair, **586**, 590, 600, *637*
cycle, **264**, 414, 419, 590
  protocol dependency graph, 415
cycle graph, **267**

D

DAG, *see* directed acyclic graph
debug build, 38
decision procedure, 134
declaration, **93**, *116*
declaration context, *62*, **93**, 97, *119*, *166*, 277, 286, 447
  normal conformance, 214
declaration kind, *116*
declared interface type, **93**, *119*, *343*
  associated type declaration, 104, 283
  class declaration, 191
  extension declaration, 321
  generic parameter declaration, 98, 100, 281
  global conformance lookup, 215
  nested nominal type, 98
  nominal type declaration, **70**, 180, 181, 218, 279, 281, 304, 335
  normal conformance, 214
  opaque result type, 444
  self interface type, 108
  SIL type lowering, 198
  summary, 647
  type alias declaration, 282
  type resolution, 284, 290, 294, 296
  type substitution, 280
default type witness, 222, 224, 240, 242
default witness, 223, **321**, 455
definite initialization, **39**
delayed parsing, **45**, 326
demangler, 458
Dénes Kőnig, 418
dependency file, **51**
dependency sink, **52**, *59*
dependency source, **52**, *59*
dependent member type, 29, **71**, *171*, 231, 408
  associated type declaration, 104
  base type, 450
  free monoid, 512
  identity substitution map, 181
  in archetype builder, 270
  in requirement machine, 533, 541, 556
  in requirements, 101
  inference rule, 136, 150
  linear order, 157
  opaque archetype, 457
  recursive conformance, 417
  type resolution, 277
  type substitution, 178, **228**, 229, 399
  unqualified lookup, 283
  validity, 227, 366
dependent type, 72
depth, 67, **97**, *121*, *122*, 296, 320
  in generic parameter symbol, 549
  opaque result type, 438, 442, 445
  substitution term, 552
depth-first search, 484
derivation, *see also* derived requirement, **135**
derivation step, **135**
derived requirement, 33, **134**, 178, 298, 401, 480
  abstract conformance, 227

conflicts, 389
decision procedure, 473, 585
decomposition, 359
encoding monoid, 512
enumeration, 273
equivalence, 378
in requirement machine, 536, 537, 542
protocol type alias, 566
structural induction, 373, 407
summary, 645
type resolution, 286
well-formed, 365
derived requirements
type substitution, 227
destination vertex, **263**, 550
conformance path, 409
conformance path graph, 409
path, **264**
protocol dependency graph, 414
destructor declaration, **110**
desugared requirement, **365**, 378, 389, *396*
in requirement machine, 475
diagnostic, **41**, 447
conditional conformance, 337
conflicting requirement, 377, 388, 390
conformance checker, 231, 341
extension binding, 323
from abstract generic signature request, 351
from inferred generic signature request, 349
invalid member type, 278
invalid requirement, 361
invalid same-type requirement, 364
invalid type parameter, 288, 368
malformed generic signature, 365
multiple solutions, 82, 242
opaque result type, 441, 464
overlapping conformance, 334
printing archetype type, 72
printing error type, 82
printing generic parameter type, 66, 71, 165
printing unbound generic type, 80
protocol inheritance clause, 353
redundant requirements, 383
retroactive conformance, 217
substitution failure, 303
sugared type alias type, 357
type resolution, 278, 294
undecidable generic signature, 521
unsatisfied requirement, 32, 297
dictionary sugared type, **78**, *86*
dihedral group, 522
direct lookup, 42, *63*, **326**, *344*
directed acyclic graph, **264**, 414, 415, 485
directed graph, **263**
Cayley graph, 496
conformance path graph, 409
conformance substitution graph, 419
protocol dependency graph, 414
protocol inheritance graph, 550
rewrite graph, 503
strongly connected components, 481
type parameter graph, 264
disjoint set, *see* union-find
disjunction, 243
distributive law, 502, 507
domain, 140
Donald Knuth, 585
dynamic cast, 217
dynamic lookup, **43**, 46
dynamic `Self` type, **80**, 108, *208*, 281, 290, 313

E

edge, **263**, 480
Cayley graph, 499
conformance path graph, 409
conformance substitution graph, 419
graph homomorphism, 415
rewrite graph, 503
effective procedure, **424**, 520, 521
elementary derivation step
summary, 645
elementary statement, **134**, 383, 537
structural induction, 373
Emil Post, 426, 520
emoji, 457
empty generic signature, **131**, *166*, 182, 295, 348
empty path, **264**, 418, 481
empty protocol composition, *see* `Any`
empty rewrite path, **504**, 505, 506, 508, 516, 524, 537, 540, *584*, 590
empty set, 111, 362, 498, **643**
empty substitution map, **182**, 194, *208*
opaque archetype, 444
empty term, *579*
enum declaration, **94**, 97, *119*
extension, 322

SIL type lowering, 198
enum element declaration, **94**, 110
enum type, **70**, 196, 281, 372
SIL type lowering, 198, 205
equivalence class, **141**, 483
abstract conformances, 404, 409
monoid operation, 509
of terms, 589, 612
of type parameters, 31, 264, 376, 608
terms, 499
type parameters, 515
equivalence relation, **140**, 392, 404, 419, 481
interface types, 162
error, 41, 447
error type, 49, **82**, 177, 182, 278, 291, 295, 302, 303, 368
escaping function type, 75, 113, 277
eta expansion, 109
evaluation function, **48**, *58*
exact cover, 247
exhaustive switch, 88, 116
existential abstract conformance, 463
existential archetype, *208*, 257, 463
existential generic environment, 257
existential metatype type, 74
existential type, 27, 73
SIL type lowering, 196, 198
explicit requirement, 134, 487, 533
structural induction, 373
explicit rule, **571**, *581*
expression, 18, 21, 23, 30, 45, 55, 68, 72, 76, 77, 79, 80, 82, 93, 94, 99, 109, 116, 255, 257, 294
opaque result type, 440
expression type checker, 43, 66, **80**, 173, 190, 242, 247, 256, 261, 304
extended nominal request, *343*
extended type, 304, 305, **319**, *343*, 348, 605
extension binding, **322**, *344*
extension declaration, 93, 213, 305, **319**, *343*, 391

F

Factor, 511
field, 502
file unit, **55**, 93, *120*
finitely-presented monoid, **499**, 514
finitely-presented semigroup, **635**
forest, 483, 485
formal substitution, 404, 480, 513
formal system, **134**, 372
formal type, **195**, 200, 205, *210*
abstraction pattern, 201, *211*
type substitution, 206
Forth, 511
forward reference, 47, 50
forwarding substitution map, **261**, *274*, 334, 340
opaque archetype, 448
free commutative monoid, **500**, 527
free monoid, **496**, 512, 555, 579
free semigroup, **635**
frontend flag, 41, 50, 61, 465, 596
frontend job, **38**, 224
frozen rule, 571, *582*
fully-concrete type, 70, **131**, 175, 182, 280, 290, 340, 356, 387, 553
reduced type, 163
function, 159, 378, **644**
function declaration, 18, 79, 97, **107**, *125*, 199, 355, 386
associated type inference, 236
opaque result type, 437
override, 190
SIL type lowering, 200
function type, 24, 67, **75**, 78, 107, 176
SIL type lowering, 195, 198, 205, 207

G

G_{P}, *see* protocol generic signature
generating set, **496**, 530
generic argument, 22, 67, 180, 290, 354, 360
type resolution, 295
generic arguments, 70, 278, 279, *312*, *314*
generic class, 355
generic class type, 191, 365
generic context, **97**, *121*, 166
generic declaration, **97**, *121*, 129, *314*
Generic derivation step, **136**, 373, 378, 513
summary, 645
generic environment, **257**, *274*, 295
opaque result type, 442
generic function type, 19, **79**, 107, 445
generic nominal type, **70**, 176, 179, 218, 356
extended type, 330
SIL type lowering, 197, 204
generic parameter declaration, 18, 71, **97**, 100, *122*, 281, 319, 355
associated type inference, 240
type witness, 222

generic parameter list, 18, **97**, 100, *122*, 279, 281, 295, 296, 320, 348, 456
generic parameter list request, **99**, *122*
generic parameter order, **155**, *171*
generic parameter symbol, 533, 540, **549**, 555, *577*, 593
generic parameter type, 19, 66, **71**, *123*, 134, 155, 281, 379
  elementary statement, 136
  opaque archetype, 457
  opaque result type, 438, 452
  structural induction, 373
  type sugar, 67, 133, 257
  validity, 160, 366
generic signature, 19, 25, **129**, *166*, 227, 236, 347
  calling convention, 158
  checking requirements, 295
  elementary statement, 134
  extension binding, 323
  infinite, 269, 340, 620
  normal conformance, 214
  opaque result type, 438
  query machine, 474
  requirement machine, 479, *491*, 533
  SIL function type, 199, *211*
  SIL type lowering, 198, 205
  summary, 645
  superclass type, 192
  type parameter graph, 264
  type resolution, 277, 286
  type substitution, 149, *207*
  well-formed, 367
generic signature constructor, **347**, 377, *393*
generic signature equality, **131**, *167*
generic signature equivalence, **378**, 476
generic signature query, **159**, *168*, 259, 269, 277, 286, 294, 352, 473, *491*, 585
generic signature request, 48, 278, 345, **348**, *393*
generic type alias, 68, **97**, 293, 324, 331, 357
  type substitution, 175
`GenericSignatureBuilder`, **273**, 377, 383, 385, 391, 480, 569, 570, 600, 617
Gérard Huet, 585
`getConcreteType`(), **161**, *168*, 286
`getLayoutConstraint`(), **164**, *168*, 259
`getLocalRequirements`(), **165**, 259, 275
`getReducedType`(), **163**, *168*, 273, 288, 391, 412, 413, 450
`getRequiredProtocols`(), **160**, *168*, 259, 273, 286
`getSugaredType`(), **165**, *168*
`getSuperclassBound`(), **164**, *168*, 259, 286
global conformance lookup, 27, **215**, 225, *248*, *276*, *346*, 401, 447
  associated conformance, 231
  building substitution map, 228
  compatibility, 228
  conditional conformance, 336
  conformance requirement, 301
  opaque archetype, 443
  requirement desugaring, 362, 365
  substitution map, 294, 335
  summary, 648
  type resolution, 284, 291, 294
  with archetype, 260
global declaration, 322, 348
global variable
  opaque archetype, 447
graph, *see* directed graph
graph homomorphism, **415**
  local conformance lookup, 419
Grigori Tseitin, 521
group, 502
  special monoid presentation, 522

H

halting problem, 79, **425**, 520
Haskell, 35, 76, 185, 528
Haskell Curry, 528
heap-allocated box, 114, 198
Heinrich Tietze, 618
higher-rank polymorphism, 79
histogram, 489
history, **14**, 14, 21, 37, 38, 45, 49, 57, 73, 75, 76, 78, 79, 94, 103, 105, 107, 113, 115, 185, 207, 216, 256, 269, 277, 282, 295, 323, 330–333, 355, 383, 415, 442, 457, 480, 617
homomorphism, 415, 514, **644**
horse, 56, 94, 156, 213, 224, 324, 360, 440, 454
Hylo, 36

I

identifier, 18, 41, 42, 49, *171*
  dependent member type, 71
  in name symbol, 549
identifier type representation, **279**, 323
identity conformance rule, **562**, 571, *581*, 615

identity element, **495**
  Cayley graph, 496
  finitely-presented monoid, 499
  free monoid, 496
identity morphism, **189**
identity substitution map, **181**, 188, *207*, 219, 229, 261, 262, 404, 430
  opaque archetype, 445, 448
  superclass type, 192
immutable term, **555**, 571, *580*
implied conformance, 337
import declaration, **55**, 284
imported module, **55**, *64*, 326, 345
imported rule, 487, 571, *582*, 592
incomparable elements, **154**, 391, 527, 554, 574
incremental build, 39, **51**
index, 67, **97**, *122*, 320, 552
  in generic parameter symbol, 549
  opaque result type, 438, 457
indirect enum case, 197, 198
induction, **372**, 376, 406, 497, 513, 516, 517, 527, 541, 575
  path length, 411
inductive step, **372**
inference rule, **134**, 144, 372, 386
  concrete types, 387
  structural induction, 373
inferred generic signature request, **348**, 359, *393*, 475, *492*
infinite descending chain, 526
infinite generic signature, 269
infinite graph, **263**
  conformance path graph, 414, 417
  rewrite graph, 503
  type parameter graph, 266, 417
infinite monoid presentation, 502, 625, 629, 632, 633
infinite path, *see* ray
infinite set, 177
infix operator, 43
inheritance, *see* subclassing
inheritance clause
  associated type declaration, **105**, *125*, 351
  class declaration, 190
  extension declaration, 213
  generic parameter declaration, **100**, *123*, 129, 349, 355
  nominal type declaration, 213
  protocol declaration, 104, *124*, 352, 353
inherited associated type, 152, 242, 558
inherited conformance, 213, **215**, *251*, 260
  output generic signature, 219
  substitution, 220
inherited protocol, **104**, *124*, 152, 156, 232, 336, 352, 558, 611
  associated types, 562
  circularity, 416
  in requirement machine, 550, 551
initial value expression, 23, 48, 68, 94, 110, **114**, 305
initializer interface type, **110**
inlinable function, 22, **55**, 463
  opaque result type, 438, 462
  textual interfaces, 56
input generic signature, **173**, 186, *208*, 229, 234, *252*, 291, 340, 356, 369
  conformance requirement, 178
  opaque archetype, 444
  opaque result type, 441
  summary, 647
instance type of metatype, **74**, *89*
integers, 140, 157, 502, **643**
interface resolution stage, 71, 278, 286, *310*, 368, 394
  checking requirements, 296
  protocol type alias, 566, 569
interface type, 20, 72, *119*, 129, 149, *171*, **176**
  associated type inference, 236
  empty generic signature, 182
  extension member, 319, 321
  function declaration, 79
  identity substitution map, 182
  opaque archetype, 442, 446, 448
  opaque result type, 442
  self parameter, 108
  SIL function type, 200
  SIL type lowering, 198
  subscript declaration, 79
  summary, 647
  superclass type, 192, 193
  type resolution, 278, 294
  value declaration, **93**
interface type request, 48, 49, **107**, 278, 459
intersection, 141, **644**
invalid conformance, **213**, 362
invalid type representation, 278
inverse rewrite path, **505**
inverse rewrite step, **505**
IRGen, **39**, 56, 68, 164, 255
irreducible term, *see* reduced term

`isConcreteType`(), **161**, *168*
`isReducedType`(), **163**, *168*
`isValidTypeParameter`(), **160**, *168*, 273, 288
iterable declaration context, **326**

J

Java, 36, 109, 190, 435
John Rosser, 528
Joy, 511
JSON, 51

K

Knuth-Bendix algorithm, **585**, 618, *637*
    mutable terms, 555
Kőnig's infinity lemma, 418

L

l-value type, **82**, 114
labeled edge, **263**
lambda calculus, 109, 425, 528, 531
lambda cube, 72
language, 633
layout constraint, 100, 301, 549
Layout derivation step, **139**
    summary, 645
layout requirement, **99**, *169*
    checking, 301
    conditional conformance, 336
    conflicts, 386, 389
    desugaring, 365
    elementary statement, 139
    generic signature query, 164
    in requirement machine, 552, 560, 561
    reduced, 381
    summary, 645
    type substitution, 298
    well-formed, **367**
layout symbol, **549**, 552, 560, *577*
lazy member loader, **326**, *345*
left simplification, **598**, 618, *637*
left whisker, **503**, 572, 575, *584*
left-reduced rewriting system, **597**, *637*
left-simplified rule, 571, *581*, 595, 598, 606, **607**, 626
length-preserving rewrite rule, 520, **525**
length-reducing rewrite rule, **525**, 528
lexer, 65
lexicographic order, 527, 551, 555
library evolution, **56**
    opaque result types, 462
limitation
    associated type inference, 243
    completion termination check, 596
    conditional conformance, 326, 339, 341
    conformance coherence and subclassing, 217
    extension binding, 324
    generic type alias, 293, 306
    higher-rank polymorphism, 79
    nominal type nesting, 182
    of derived requirements, 139, 161, 386, 566
    opaque result type, 447, 453, 456, 465
    protocol inheritance clauses, 352
    re-abstraction thunk, 203
    requirement inference in protocols, 359
    type alias in protocol extension, 569
    undecidable generic signature, 521
    unsupported generic signature, 630
linear order, **154**, 509, 526
    conformance path, 411
linked list, 556
Lisp, 40, 113
literal
    in SAT, 244
LLVM, 39
loadable type, **196**, 198
local confluence, **586**
local conformance, **214**, *248*
local conformance lookup, **228**, 300, 399, *435*, 447
    graph homomorphism, 419
    recursive, 423
local conformance lookup callback, *253*
local context, **110**
local declaration context, 182
local function declaration, **107**, 111, 112
local requirements, 165, 269, *275*
local rule, **474**, 571, *582*, 592
local type declaration, **96**, 109, 182, 326
    type resolution, 284
local variable
    opaque archetype, 447
local variable declaration, *126*
locally finite graph, **268**, 418
loop, **410**, 414
    graph homomorphism, 415
    seecycle, 268
Lothar Collatz, 427
lowered captures, **112**
lowered optional type, **204**

lowered parameter, **196**, *211*
abstraction pattern, 201
tuple type, 199
lowered position, **206**
lowered result, **196**, *211*
abstraction pattern, 201
tuple type, 199
lowered type, **195**, *210*
type substitution, 206

M

macOS, 441
main function, **37**
main module, **37**, 55, 345, 368, 377, 462, 476
main source file, **37**, *64*, 447
mangling, 103, 158, 377, *467*
conformance path, 411
contextually-generic declaration, 103
opaque result type, 457–459
map replacement types out of environment, **262**, *275*
map type into environment, **255**, *274*, 295
map type out of environment, **255**, *275*
opaque archetype, 448
mapping, *see* function
Margaret Corasick, 575
matrices, 502
member declaration, **94**
member lookup table, **327**, *345*
member reference expression, 25, 41, 107, 160, 181, 187, 193, 294
member type declaration, **94**
member type representation, **285**, 323
opaque archetype, 457
protocol type alias, 566
metadata access function, 23, 462
metatype type, 24, **74**, 78, 94, 108, 176, 519
SIL type lowering, 198
method declaration, **94**, 107
minimal requirement, 347, 357, *395*, *397*, 473, 475
minimal requirement signature, **392**
minimization machine, **475**, *490*, *491*, 556, 560, *583*, 593
summary, 479
ML, 76
modular arithmetic, 499, 527
module declaration, **55**, *63*, 93, 155
conformance, 216
type resolution, 284
module interface, *see* textual interface
module lookup, **42**, *63*
monoid, **495**
monoid congruence, **509**
monoid homomorphism, **514**, 558
monoid isomorphism, 498, **514**, 530, 618
monoid isomorphism problem, 619
monoid operation
in requirement machine, 536
monoid presentation, **498**, 533, 618, 625
of requirement machine, 533, *582*
rewrite graph, 503
monomorphization, *see* specialization
morphism in category, **189**
multi-parameter type class, 185
multiset, 541
mutable term, **555**, *579*

N

name lookup, 18, **41**, *61*, 326, 352
access control, 463
name symbol, 533, 540, **549**, 555, 556, *577*, 619
natural numbers, **643**
addition, 495
induction, 372
linear order, 158, 509, 559
multiplication, 502
negation, 243
negative rewrite step, **503**, 516, 543, 604
in requirement machine, 575, *584*
nested type declaration, 35, 182
extension, 322
Newman's lemma, 586
Noetherian reduction relation, *see* terminating reduction relation
nominal type, **70**, 72, 95, 176, 179, 182, 279, 290, 323
nominal type declaration, 70, **94**, *119*, 213, 231, 294
generic arguments, 181
global conformance lookup, 215
summary, 647
non-deterministic polynomial time, 246
non-escaping function type, 75, 113, 277
non-orientable relation, 589
non-orientable rewrite rule, **526**
non-terminating computation, 339, 464
noncopyable type, 14, 21
nonescapable type, 14

normal conformance, 26, **213**, *251*, 303, 304, 319, 386
  associated conformance, 232
  associated type inference, 236
  opaque type witness, 454
  output generic signature, 219
  recursive, 420
  substitution, 219
  type witness, 224
normal form, **524**
normal form algorithm, **524**, 549, 551, 571
  mutable terms, 555
  protocol type alias, 568
NP problem, 246
NP-complete problem, 246
NP-hard problem, 246

O

object file, 41, 51, 463
object in category, **189**
object type, **82**
Objective-C, 43, 46, 56, 75, 80, 164, 327
Objective-C existential, 301
opaque abstract conformance, **443**, 447, 454, 463
  underlying conformance, 460
opaque abstraction pattern, 202, 206, 207, *211*
opaque archetype, 257, **442**, 449, 463, *466*
  interface type, 447
  textual interface, 457
  type substitution, 446, 455
  type witness, 454
opaque archetype order, **453**
opaque generic environment, 257, **449**, 450, *467*
opaque parameter, **98**, 101, *122*, 129, 437
  history, 103
opaque result declaration, **437**, 459, *465*
opaque result generic signature, **438**, 450, 459, *465*
  weight, 452
opaque result type, 60, **438**
  circularity, 464
  weight, 452
  with availability, 441
opaque result type request, **437**, *466*
opaque type descriptor, **459**, *468*
operator declaration, **43**
operator lookup, **43**, 45
operator symbol, **43**
optional sugared type, 68, **78**, *86*, *89*, 197, *309*
optional type, 197
  lowered position, 206
  re-abstraction, 204
  SIL type lowering, 205, 207
ordered pair, 140, 263, **644**
ordered tuple, 403, 503, **644**
oriented rewrite rule, **525**, **526**, 571
original-to-substituted thunk, **200**, 202, 203, *212*
  tuple type, 204
orthogonal rewrite step, **586**
outer generic signature, **438**, 444, 445, 456, 459
  runtime type metadata, 461
outer type parameter
  opaque result type, 450
  weight, 452
output generic signature, **175**, 186, 234, 284, 356, 369
  of conformance, 219
  of substitution map, 219
  opaque archetype, 447
  opaque result type, 446
  summary, 647
overlap position, **587**
overlap term, **587**
overlapping conformance, 334
overlapping rules, **587**, 598
overload resolution, *see* expression type checker
override, 43, **190**, 322
owner declaration, **438**, 442, 444, 456, 459, *465*
  mangling, 457
  opaque result type, 462
ownership, 20, 199
ownership specifier, 76, *91*
  self parameter, 108

P

parallel rewrite paths, **590**, 604
parameter declaration, 93, **114**
parameter pack, 14
parameterized protocol type, 36, **73**, 105
  constraint type, 100
  decomposition, 360, 361
  opaque result type, 438, 439, 450
  simulation, 358
parent type, 70, 72, 179, 184
parsed generic parameter list, **97**, *121*
parser, 18, **39**, 43, 45, 65, 277, 326
  opaque archetype, 459

partial application, 113
partial function, 423
partial order, 43, **153**, 158, 391, 526, 554
Pascal, 47
pass-through type alias, **331**, *345*
path, **263**, 480
  Cayley graph, 497, 500
  conformance path graph, 409, 416
  graph homomorphism, 416
  in tree, 264
  in type parameter graph, 265
  protocol dependency graph, 416
  rewrite graph, 503
path length, **264**
pattern, **114**
pattern binding declaration, **114**
pattern binding entry, **114**
pattern type, 549, **552**, *577*
payload type of optional, **78**, *89*, 197
Peano arithmetic, 372
permanent rule, **571**, *581*, 617
Peter Bendix, 585
placeholder type, **79**, 304
polynomials, 502
positive rewrite path, **524**, 524, 573, 586, 599, 603
positive rewrite step, **503**, 516, 526, 543
  in requirement machine, 575, *584*
postfix operator, 43
potential archetype, **269**
pre-check expression pass, **43**
precedence group, **43**
predecessor, **268**
prefix, 366
prefix operator, 43
preorder, 159
primary archetype, 188, *208*, **255**, 255, 257, 338
  type substitution, 177
primary associated type, **105**, *124*, 360, 361
  opaque result type, 451
primary file, **38**, 45, 48, *63*, *64*, 224, 323, 394, 438, 462
  conformance checker, 231
primary generic environment, **255**, 257, *275*, 297
prime number, 502
principal abstract conformance, **404**, 409
principal derivation, **405**
Prolog, 36
proof by induction, *see* induction
proper subset, 531, **643**
property declaration, **94**, *126*
property rule, **560**, *581*
Proto, **222**, 648
protocol component, 351, 392, 477, **481**, 487, *490*
  completion, 592, 597
protocol component graph, **483**
protocol composition type, **73**
  constraint type, 100
  decomposition, 360
  history, 103
  opaque result type, 438
protocol conformance, *see* conformance
protocol declaration, **103**, *124*, *170*, 474, 487
  conformance, *250*
  generic signature, 348
  global conformance lookup, 215
  in protocol symbol, 549
  protocol dependency graph, 414
  summary, 648
  unqualified lookup, 283
protocol dependencies request, **484**, *491*
protocol dependency, **487**
protocol dependency graph, 359, **414**, 480, *490*, *491*, 593, 601
protocol dependency set, 367, **480**, 533
protocol extension, *62*, 182, 214, 289, 321, *343*, 605–607
  associated type inference, 238
  generic signature, 348
  opaque type witness, 455
  type alias member, 569
protocol generic signature, **131**, 132, 231, *252*, 347, 348, 369, 392, 403, 480, 487, 601
  free monoid, 512
  generic protocol, 185
  in requirement machine, 533
  opaque result type, 455
  protocol inheritance, 352
  well-formed, 368
protocol inheritance, *see* inherited protocol, 137
protocol inheritance closure, 629
protocol machine, **477**, *490*, *491*, 552, 557, 561, 566, *583*, 593
  summary, 479
protocol minimization machine, **477**, *490*, *491*, 557, 561, 567, *583*

summary, 479
protocol node, **487**
protocol order, **155**, *171*
protocol reduction order, **550**, 629
protocol `Self` type, 33, 98, **103**, 108, *122*, 131, 137, 184, 185, 231, 281, 293, 313, 321, *343*, 534
free monoid, 512
summary, 645
protocol substitution map, 184, **231**, *252*, 286, 304, 337, 361, 418
opaque type witness, 455
summary, 648
type resolution, 284, 294
protocol symbol, 533, 540, **549**, 560, *577*, *581*, 593, 619
reduction order, 550
protocol type, **72**, 103, 293, 298, 365
constraint type, 100
decomposition, 360
opaque result type, 438, 439
protocol type alias, 132, *170*, **282**, 287, 289, 295, 351, 566
type resolution, 292
with protocol base, 293
protocol type alias rule, **566**, *581*

Q

qualified lookup, 26, **41**, *62*, 160, 237, 280, 284, 285, 294, *314*, 319, 322
archetype base, 259
protocol inheritance, 104
superclass, 190
type resolution, 290
type witness, 224
qualified lookup request, **48**
quaternions, 502
query machine, **474**, 476, *490*, *491*, 556, 560, *583*, 593
summary, 479

R

rational numbers, 141, 502
raw SIL, **39**
ray, 266, 418
graph homomorphism, 419
re-abstraction thunk, **200**, 201, *212*
reachability relation, **409**, 480, 481
recursive conformance requirement, **415**, 620
in archetype builder, 269
recursive language, 633
recursive rule, 571, *582*
reduced abstract conformance, **404**
conformance path graph, 409
reduced conformance path, **411**
reduced requirement, 347, **381**, *397*
reduced rewriting system, *637*
reduced term, **524**, 589, 612
reduced type, 31, 68, 149, **163**, 167, 258, 300, 363
type parameter graph, 264
reduced type equality, 68, 140, **142**, 258, 269, 349, 363, 386, 404, 419
finitely-presented monoid, 515
generic signature query, 159
on interface types, 163
reduced type parameter, *276*, 349, 376, 380, 410, 612
abstract conformance, 404
opaque archetype, 442, 449
reduction order, **526**, 581
in requirement machine, 571, *580*, 612
reduction relation, **524**, 586, 618
redundant requirement, 145, **383**
redundant rule, 571, *582*
reference count, 20, 75, 99, 100, 104, 114, 164
reference counting, 20, 79, 112, 113
reference storage type, **79**, 114
Reflex derivation step, **142**, 142, 373, 374, 407, 516, 517, 537, 543
summary, 646
reflexive relation, **140**, 480, 481, 508
regular language, 496, 633
relation, **140**
release build, 38
replace opaque archetypes with underlying types, **463**, *468*
replacement type, **174**, *252*
SIL type lowering, 206
summary, 647
Req(G), **298**, 647
request, **47**, 52, 348, 350, 351
request cycle, **50**, 277, 352, 490
request evaluator, **47**, *59*, 277, 348
requirement, **99**, *169*, *315*, 348, 359, 647
requirement decomposition, **360**, 438, 570
requirement desugaring, **362**
requirement inference, 68, 348, 351, **354**, *396*
opaque result type, 453

requirement kind, *170*, 298, 301, 365, 379, 381, 386
  summary, 645
requirement machine, **473**, 473, *491*
  monoid presentation, **533**
requirement minimization, 130, 145, **377**, *see* minimal requirement, 585, 592, 597, 606, 619, 625
requirement order, **392**, *397*, 411
requirement representation, **101**, *123*, 348
requirement resolution, 349, *393*
requirement signature, 33, **132**, *170*, 236, 351, 352, 368, 392, 404, 477, *491*, 566
  conformance checker, 231
  requirement machine, 479
  summary, 645
  witness table, 158
requirement signature constructor, *395*
requirement signature request, **351**, 359, 392, *395*, 477, 484, *492*
`requiresClass`(), **164**, *168*, 198, 259
`requiresProtocol`(), **159**, *168*, 273, 412, 413
resilience, 17, **56**
  opaque result type, 462
  SIL type lowering, 196
resolved type
  see type resolution
resolving critical pair, 588, **589**, 595, 600, 605
retroactive conformance, **216**
return statement
  opaque result type, 440
rewrite context, **474**, 476, 477, 487, *490*, 549
  immutable terms, 555
rewrite graph, **503**, 530, 590
  in requirement machine, 538
rewrite loop, 489, **590**, 637
rewrite path, **504**, 541, 618
  in requirement machine, 533, 536, 538, 540, 575, *584*
  to derivation, 516
rewrite path composition, **506**, 517, 524
  in requirement machine, 538
rewrite path evaluator, **575**, *584*
rewrite path inverse, 517
rewrite path length, **504**, 508, 516
rewrite path whiskering, **507**, 517, 524, 540
  in requirement machine, 538
rewrite rule, **498**
  in requirement machine, 533, 552, *581*
  protocol type alias, 558
  special monoid presentation, 521
rewrite step, **503**, 604
  critical pair, 586, 589
  in requirement machine, 543, 575, *584*
  local confluence, 586
  normal form algorithm, 571
  orthogonal, 586
  Tietze transformation, 618
rewrite step whiskering, **507**
right simplification, **599**, 618, *637*
right whisker, **503**, 575, *584*
right-reduced rewriting system, **597**, *637*
right-simplified rule, 571, *582*, 595, 598
ring, 502
Robert Tarjan, 484
root abstract conformance, **228**, 229, 401, 403
  conformance path, 409
root associated type, **156**, 613
root conformance, 178, **228**, 229, 402
  substitution map composition, 186
  summary, 647
root generic parameter, **450**
root vertex
  Cayley graph, 496
  of tree, 264
  of trie, 572
  of type parameter graph, 264
rule, *see* rewrite rule
rule array, **571**, 573, 574, *582*, 584
rule trie, **572**, 593, 598, 637
runtime type metadata, 20, 22, 68, 183, 255
  circularity, 465
  opaque result type, 459, 461, *468*
Rust, 36, 340, 435, 617

S

s-expression, **40**, 84
Same derivation step, **136**, 537, 545
  summary, 645
same-type requirement, **99**, *169*
  checking, 301
  concrete conformance, 391, 554
  conditional conformance, 336
  conflicting, 387
  conflicts, 386, 389
  desugaring, 363–365
  elementary statement, 136, 139, 537
  equivalence, 140
  formal system, 136
  generic signature query, 161

in archetype builder, 269, 271
in requirement machine, 533, 545, 552, 560, 561
inference rule, 142, 144
opaque result type, 439, 441, 451
recursive, 163
reduced, 381
summary, 645
type resolution, 294
type substitution, 179, 298
well-formed, **367**
same-type requirement representation, **102**
SameConcrete derivation step, **144**
SameConf derivation step, **144**, 405, 408, 538, 546
summary, 646
SameDecl derivation step, **150**, 374, 376, 406, 407
summary, 646
SameLayout derivation step, **144**
summary, 646
SameName derivation step, **144**, 144, 150, 374, 376, 406, 407, 515, 517, 538
summary, 646
SameSuper derivation step, **144**
summary, 646
SAT problem, 243
satisfied requirement, **301**, 349, 362, 364, 386
conflicts, 389
opaque result type, 441
well-formed generic signature, 366
SCC, *see* strongly connected component
scope tree, **42**, *61*, *64*, 100, 284, 319
secondary file, **38**, 45, *64*, 224, 438
self conformance, **213**
self interface type, **108**, *119*, *343*
extension declaration, 321
`self` parameter, **107**, *125*, 195
constructor declaration, 110
nested nominal type, 109
Sema, **39**
semi-Thue system, 520
see also reduction relation
semigroup, **635**
sequence expression, 43
serialized AST file unit, **55**
serialized module, **55**, *64*, 326, 345, 347, 352, 368, 478, 484, 606
serialized SIL, 55
set, **643**
captures, 111, 112
edges, 263
finitely-presented monoid, 499
free monoid, 496
generators, 496
monoid, 495
relation, 140
types, 176, 193
vertices, 263
set difference, **644**
shortlex order, 159, 411, **526**
side effect, 284
SIL, **39**, 72, 79, 195, 255, 295
SIL function type, **195**, 199, 202, 205, *211*
abstraction pattern, 201
lowered position, 206
optional payload, 204
type substitution, 201
SIL mandatory pass, **39**
SIL optimizer, **39**, 183, 203, 206, 462, 465
SIL performance pass, **39**
SIL type lowering, **195**, *211*
opaque archetype, 462
opaque result type, 463
SILGen, **39**, 48, 49, 68, 75, 76, 80, 82, 108, 110, 179, 195, 203, 207, *212*, 214, 255, 261, 462
simple path, **418**
single file mode, **38**
source file, 38, 45, 47, 51, **55**, *120*, 284, 319, 322, 326, 348, 462
source location, 42, *62*, 218, 277, 279, 303, *312*, 348, 349, *394*, *395*
source range, 42, 97, 98, 100, 281, 283, 284, *312*
source vertex, **263**, 550
Cayley graph, 496
conformance path graph, 409
path, **264**
protocol dependency graph, 414
spanning tree, **485**, 600
special monoid presentation, 521
specialization, **22**, 36, 55, 206
opaque result type, 462
specialized conformance, 213, **219**, *251*
associated conformance, 232
conditional conformance, 335
opaque archetype, 463
opaque type witness, 456
output generic signature, 219
recursive, 420, 423

substitution, 220
type witness, 225
specialized type, 70, **179**, 183, 191, 335
global conformance lookup, 215
SSA form, 39
stack, 50, 52, 485, 510
stack language, **510**
standard description, **424**, 520
standard term, 603
statement, 55, 109, 116, 440
static method declaration, **108**
Stephen Cook, 246
storage declaration, **114**, *126*
stored property, 196
opaque archetype, 447
SIL type lowering, 198
stored property declaration, 22, **94**, 321
string rewriting, 525
strongly connected component, **481**, *490*
struct declaration, 22, 97, *119*, 321
struct type, **70**, 279, 281
SIL type lowering, 196, 198, 205
structural components, **65**, 358, 549
structural induction, **373**, 407, 537
structural requirements request, **351**, *395*, 484
structural resolution stage, 71, 278, 288, *310*, 349, 356, 368, 379, 394, 475
checking requirements, 296
protocol type alias, 566, 569
structural type, 24, **73**, 176
Sub(G, H), **177**, 261, 647
subclassing, **190**
subgraph, **263**, 266, 274, 484, 590
rewrite graph, 503
subject type
of requirement, **99**
subscript declaration, 79, 97, 110, **115**, *126*, 355, 386
associated type inference, 236
history, 103
opaque result type, 437
override, 190
subset, 140, 263, **643**
substituted opaque archetype, **445**
substituted requirement, 32, 297, *315*, 335, 340, 351, 354, 362
summary, 647
substituted type, *207*
substituted underlying type, 357
substituted-to-original thunk, **200**, 202, 203, *212*
tuple type, 204
substitution failure, **179**, 291, 303
substitution map, 21, 70, 191, *207*, *208*, 236, *252*, 280, 341, 356, 399
conformance substitution graph, 419
conformances, 228
containing type variables, 82
lowered type, 206, *212*
opaque archetype, 445, 447
opaque generic environment, 449
opaque result type, *466*
replace opaque archetype, 463
runtime type metadata, 22, 461
summary, 647
substitution map composition, **186**, *209*, 220, 261, 446
summary, 647
superclass type, 194
substitution map equality, *209*
substitution simplification, 596
substitution term, 549, **552**, *577*
substitution-simplified rule, 571, *582*, 595
subterm, **496**, 575, *579*, 586, 587, 592, 598
normal form algorithm, 571
subtree, 485
subtype, 36, 190
successor, 372, 484, 643
of potential archetype, 270
of vertex, **268**
vertex, 518
sugared type, **67**, 78, *87*, 98, *122*, 133, 157, 166, 174, 216, 257, 357
in requirement machine, 474
Super derivation step, **139**
summary, 645
superclass declaration, **192**, *209*, 215, *248*, 259
opaque type witness, 455
superclass requirement, **99**, *169*, 280
checking, 301
conditional conformance, 336
conflicts, 386, 389
desugaring, 365
elementary statement, 139
generic signature query, 164
in archetype builder, 269
in requirement machine, 552, 560, 561
opaque result type, 438
reduced, 381

requirement inference, 355
summary, 645
type resolution, 294
type substitution, 298
well-formed, **367**
superclass requirements, 190
superclass substitution map, **195**
opaque type witness, 455
type resolution, 280, 284, 290, 294
superclass symbol, **549**, 552, 560, *577*
superclass type, 94, **190**, *209*, 214, 280, 301, 365, 389
inherited conformance, 214, 220
type witness, 222
Swift, 11
2.2, 73, 78, 295
3.0, 45, 73, 78, 103, 113, 207, 256, 269, 277, 295, 331
3.1, 207
4.0, 78, 103, 107
4.1, 38, 330, 332, 415
4.2, 38, 49, 57, 332, 333
5.0, 57, 78, 207, 323
5.1, 57, 115, 282, 442
5.3, 37, 103
5.5, 75, 94, 457
5.6, 73, 79, 207, 480
5.7, 73, 103, 105, 331, 383, 442, 480
5.8, 480
5.9, 14, 21, 76
5.10, 37, 185
6.0, 14, 21, 216, 355
6.2, 14
Swift driver, **37**, *57*
Swift frontend, **38**, *57*
Swift package manager, 37
SYM derivation step, **142**, 142, 377, 407, 516, 517, 538, 546
summary, 646
symbol, *576*
in requirement machine, 533, **549**
symmetric relation, **140**, 363, 481, 508
syntax tree, *see* abstract syntax tree
synthesized declaration, 40, *116*, 223, 324, 454, 457
textual interfaces, 56

T

tag system, **426**, 464
Tarjan's algorithm, **484**, *490*
TBD, 41
term, **496**
in requirement machine, 533, **555**, *580*, 612
term equivalence relation, 498, 505, **508**, 509, 586
in requirement machine, 535, 549, 557
term length, **496**, 500, 501, 513, 520, 525, *579*
term rewriting, 525, 531
terminating reduction relation, **526**, 528, 531, 586
termination, 163, 525
textual interface, 41, **56**, *64*, 386
opaque archetype, 459
opaque result type, 438, 457
theory, **135**, 476
finite, 269
infinite, 148, 267, 269, 620
thick function, **75**, 113
SIL type lowering, 199
thin function, **75**
SIL type lowering, 199
Thue system, 520
see also finitely-presented monoid
Tietze transformation, 529, **618**, 624–626
top-level code declaration, **37**, **93**, 110
top-level declaration, **93**
top-level lookup, **42**, *61*, *63*
top-level type declaration
type resolution, 284
total order, *see* linear order
trailing `where` clause, *see* `where` clause
TRANS derivation step, **142**, 142, 376, 377, 408, 515–517, 538, 545, 547
summary, 646
transitive closure, 112, 487, 611
transitive relation, **140**, 153, 196, 352, 387, 480, 481, 501, 508
translation-invariant relation, **508**, 526, 559
tree, 39, 42, 43, 65, 84, **264**, 277, 483, 550, 572
Cayley graph of free monoid, 496
type parameter graph, 265
trie, **572**, *583*, 593, 598
trivial critical pair, **589**, 591, 592, 607, 615, 617
trivial monoid, **498**
trivial type, 20, 199, *212*
tuple pattern, 115
tuple splat, 77
tuple type, 24, 67, **75**, 78, 174, 176
lowered position, 206

SIL type lowering, 196, 198, 199, 204, 205, 207
Turing machine, 424, 520
Turing-complete language, 425
type, **65**, *83*, 101, 277
containing archetypes, 255
containing type parameters, 72, 205
containing type variables, 82
replace opaque archetype, 463
Type(G), **176**, 258, 647
type alias declaration, 67, **96**, *119*, 222, 282, 287, 291, 305
opaque archetype, 454
textual interface, 457
type alias requirements request, **351**, *395*
type alias type, 28, **78**, *86*, 96, 282, 324, 356, 357, 360
extended type, 330
type declaration, **93**, *119*, *171*
type erasure, 191
type expansion context, **463**, *468*
type kind, **65**, *88*
type lowering, 49, *see* SIL type lowering
type parameter, 30, **72**, 99, 140, *171*, 202, 227, 286, 612
in requirement machine, 533, 541, 556, *580*
opaque archetype, 442
prefix, 366
SIL type lowering, 198
summary, 645
type resolution, 294
validity, 366
type parameter graph, **264**, 270, 272, 273, 382, 417, 614
Cayley graph, 496, 512
conformance path graph, 409
monoid, 518
type parameter length, **154**, 376, 406, 413, 452
type parameter order, *171*, 385
opaque result type, 452, 453
reduced requirement, 381
type parameter type
SIL type lowering, 196, 197
type pointer equality, **68**, *84*, *90*, 216, 258
generic environment, 257
type representation, 18, 65, 82, 93, 101, 116, 256, **277**, *311*, 322, 355
opaque result type, 438
type representation kind, **65**
type resolution, 19, 67, 82, 101, 149, 175, **277**, *307*, 348, 352, 354
opaque archetype, 442, 459
opaque result type, 438
type resolution context, **277**, 304, *307*
type resolution flags, **277**, *309*
type resolution options, *307*
type resolution stage, **277**, 286, 295, *310*, *311*, 368, 394
type substitution, **177**, *207*, 218, 224, 386, 399, 523, 531
lowered type, 195, 206, *212*
opaque archetype, 443, 446
summary, 647
type resolution, 280
well-formed generic signature, 366
type variable type, **80**, 306
type witness, 132, **222**, *250*, 283, 303, 340, 341, 361, 401, 479
of abstract conformance, 260
of conditional conformance, 335
opaque archetype, 443
opaque result type, 454
summary, 648
type witness projection, **225**
type witness request, **224**, *254*
type witnesses request, **224**, *254*
type-check primary file request, **48**, 52, 63, 224, 278, 323, 394, 465
typed pattern, 115
TypeScript, 435

U

unapplied function reference, 77, 109
unbound dependent member type, 30, **71**
equivalence, 342
in requirement machine, 533, 556
in requirements, 349, 368, 379, 475
inference rule, 136
linear order, 157
protocol type alias, 566
summary, 645
type resolution, 278, 288
type substitution, 229
validity, 160, 366
unbound generic type, **80**, *119*, 304
unbound type parameter, **149**
bound and unbound form, 375
in requirement machine, 533, 540, 601, 608
type parameter graph, 265

unconstrained extension, **330**, 369
undecidable problem, 530
  halting problem, 425, 465
  monoid isomorphism, 619
  rank-3 polymorphism, 79
  word problem, 520
underlying conformance, **460**
underlying type
  in requirement machine, 566
  of type alias declaration, 67, **96**, *119*, *170*, 282, 287, 305
  of type alias type, **78**
  opaque archetype, **460**
  opaque result type, **437**, 440, 444
  type substitution, 175
underlying type substitution map, **441**, 446, 449, 460, 462, *466*
union, **644**
union-find, **273**
universal Turing machine, 520
unowned reference type, **79**, 114
unqualified lookup, **41**, *61*, 97, 279, 280, *313*
unqualified lookup request, **48**

V

valid abstract conformance, **227**, 401, 405
valid type parameter, **134**, 178, 399, 416
  bound and unbound form, 376, 557
  enumeration, 273
  equivalence, 378
  generic signature query, 160, 548
  in requirement machine, 540, 542
  interface type, 176
  prefix, 366, 367
  protocol generic signature, 369
  structural induction, 373
  summary, 645
  type substitution, 229
value declaration, **93**, *118*, 278, 319, 326, 438
  opaque result type, *466*
value interface type, **114**
value requirement, **103**, 214, 222, 236, *254*, 454
variable declaration, **114**, *126*, 305, 386
  associated type inference, 236
  opaque result type, 437
  override, 190
variadic generics, 14
vertex, **263**, 414, 480
  Cayley graph, 499
  conformance path graph, 409
  conformance substitution graph, 419
  graph homomorphism, 415
  of type parameter graph, 264
  rewrite graph, 503
visitor pattern, 88, 116
`Void` type, **67**
vtable, **191**, 322

W

warning, 41, 217, 353, 383
weak reference type, 20, **79**, 114
  SIL type lowering, 196, 198
weight function, **558**
weighted shortlex order, 568
well-defined, 483, 499, 509
well-formed generic signature, **367**, 376, 379, 389, 406, 407, 416
  name symbols, 550
well-formed requirement, **367**, *394*, 476, 569
well-formed substitution map, **340**, 377, 386, 387, 404
  well-formed generic signature, 366
well-founded order, **157**, 375, 411, 526, 559
  opaque result type, 453
`where` clause, **101**, *123*, 129, 348, 349, *393*
  associated type declaration, 105, *125*, 351
  contextually-generic declaration, 301
  extension declaration, 319, 330
  history, 103
  opaque parameter, 101
  protocol declaration, 106, *124*, 351, 353, 514
  protocol type alias, 566
  requirement inference, 355
  scope tree, 285
  type resolution, 288
whiskering, 541, 572, 575, 599, 604
whole module optimization, **38**, 462
witness table, 26, 34, 158, 419
  opaque result type, 461, *468*
word problem, **519**, 619

X

XCC problem, 247
Xcode, 37

Z

zero element, 502

Command Line Flags

Symbols

-###, 39

A

-analyze-request-evaluator, 61
-analyze-requirement-machine, 489, 493

D

-debug-constraints, 82
-debug-cycles, 50
-debug-generic-signatures, 130, 132, 380
-debug-requirement-machine, 488, 493
    add, 597
    completion, 597
    concrete-contraction, 641
    concrete-unification, 639
    concretize-nested-types, 639
    conflicting-rules, 639
    homotopy-reduction, 641
    homotopy-reduction-detail, 641
    property-map, 639
    protocol-dependencies, 483
    timers, 483
-disable-batch-mode, 38
-disable-requirement-machine-concrete-contraction, 641
-disable-requirement-machine-reuse, 477
-dump-ast, 40
-dump-parse, 40
-dump-requirement-machine, 489
-dump-scope-maps, 42

E

-emit-ir, 40
-emit-library, 38
-emit-module, 38, 55
-emit-module-interface, 56, 457
-emit-sil, 40
-emit-silgen, 40, 199, 211
-enable-batch-mode, 38
-enable-library-evolution, 56, 457
-enable-requirement-machine-opaque-archetypes, 639

F
`-fine-grained-timers`, 50
`-frontend`, 41

I
`-import-objc-header`, 56
`-incremental`, 51

J
`-j`, 39

L
`-language-mode`, 207, 349

M
`-max-substitution-count`, 465
`-max-substitution-depth`, 465

O
`-O`, 39
`-o`, 40

P
`-parse-stdlib`, 55, 83
`-print-ast`, 40

R
`-requirement-machine-max-concrete-nesting`, 596
`-requirement-machine-max-rule-count`, 596
`-requirement-machine-max-rule-length`, 596

S
`-S`, 40
`-stats-output-dir`, 50

T
`-trace-stats-events`, 51

W
`-wmo`, 38

X
`-Xfrontend`, 41