ADVANCED LOGICAL TYPE SYSTEMS FOR UNTYPED LANGUAGES

Andrew M. Kent

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science,

Indiana University

October 2019

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Sam Tobin-Hochstadt, Ph.D.

Jeremy Siek, Ph.D.

Ryan Newton, Ph.D.

Larry Moss, Ph.D.

Date of Defense: 9/6/2019

To Caroline, for both putting up with all of this and helping me stay sane throughout.

Words could never fully capture how grateful I am to have her in my life.

ACKNOWLEDGEMENTS

None of this would have been possible without the community of academics and friends I was lucky enough to have been surrounded by during these past years. From patiently helping me understand concepts to listening to me stumble through descriptions of half-baked ideas, I cannot thank enough my advisor and the professors and peers that donated a portion of their time to help me along this journey.

Andrew M. Kent

ADVANCED LOGICAL TYPE SYSTEMS FOR UNTYPED LANGUAGES

Type systems with occurrence typing—the ability to refine the type of terms in a control flow sensitive way—now exist for nearly every untyped programming language that has gained popularity. While these systems have been successful in type checking many prevalent idioms, most have focused on relatively simple verification goals and coarse interface specifications. We demonstrate that such systems are naturally suited for combination with more advanced type theoretic concepts—specifically refinement types and semantic subtyping—with both formal mathematical models and experiences reports from implementing such systems at scale.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION AND BACKGROUND

For almost every untyped programming language that has gained popularity, a type system has sprung up in its wake. This is true for early systems such as Lisp[1] and Smalltalk[2], but is now widely appreciated in the context of JavaScript[3, 4, 5, 6, 7, 8], Racket[9], Clojure[10], Lua [11], PHP [12], Python [13], and more. In each of these systems, type system designers face a central challenge: *accommodating the idioms of the existing untyped language in a sound, statically-typed fashion.*

Consider the following function `move` (adapted from the TypeScript online documentation) which is capable of moving `Birds` and `Fish`:

```
function move(pet : Bird|Fish) {
  if (isFish(pet)) pet.swim();
  else             pet.fly();
}
```

Here we see two features found in TypeScript and most other type systems designed for existing untyped languages: basic set-theoretic types (i.e. ad-hoc unions), and occurrence typing. Union types such as `Bird|Fish` denote all values that are *either* a value of the first *or* second type (i.e. either a `Bird` or a `Fish`) and are essential for precisely describing the ubiquitous set-based reasoning used in untyped languages. To discriminate between the different possibilities of such a union, programmers use some form of type-based predicates. In this example, the user has written a predicate function `isFish`, which not only determines whether the input is a `Fish`, but is *known to the type system to do so.* In other words, because of the conditional test `isFish(pet)`, the type system knows the occurrence of `pet` in the first branch (i.e. the "then-branch") has type `Fish` (making `pet.swim()` well typed) and the occurrence in the second branch (i.e. the "else-branch") has type `Bird` (making `pet.fly()` well typed). This ability for the type checker to check different occurrences of the same variable at different types based on control flow-sensitive reasoning is known as *occurrence typing*[14].[1]

Because of their ability to cope with common idiomatic patterns from untyped programming, basic set-theoretic types (i.e. unions) and occurrence typing[2] are now featured in numerous languages, ranging from the logical types found in Typed Racket[9] to simple syntactic patterns[17, 18, 15] to flow-analysis driven approaches[19, 8] to TypeScript's expressive but unchecked (and thus unsound) type predicates.

1.1 Refinement Types

Although type systems with occurrence typing are capable of supporting many untyped language-specific idioms, the majority have focused on relatively simple type system features, i.e. those which can rule out dynamic type errors such as including a string in an arithmetic computation. While these guarantees are certainly a welcome improvement, we argue that such type systems—which already perform logical reasoning in a control flow-sensitive way—are well suited for verifying more precise program properties. In particular, using flow-sensitive reasoning along with *refinement types* has emerged as a popular way to provide more robust guarantees for programs while leveraging well understood off-the-shelf tools such as satisfiability modulo theories (SMT) solvers [20, 21, 22, 23].

Recall that a **refinement type** $\{v : t \mid P\}$ describes all values v of type t for which the logical predicate P holds. For a simple example, let us consider the following Haskell function which guards division with an explicit zero test:

```
divide :: Int -> Int -> Int
divide n m = if m == 0
             then error "cannot divide by zero"
             else n `div` m
```

Because the type `Int` includes `0`, Haskell's type system is unable to statically guarantee `0` is never passed as an argument. We must instead settle for a run-time check which may fail *during program execution*:

[1]This same idea is sometimes also referred to as flow-sensitive typing [15, 16] or smart casts [17].
[2]Occurrence typing can be seen as an elimination form for ad-hoc union types.

```
> divide 42 0
*** Exception: cannot divide by zero
```

By using a more advanced type system which can *refine* Haskell's types with logical predicates—e.g. Liquid Haskell [23]—we can instead encode and enforce such invariants while type checking:

```
{-@ divide :: Int -> {v: Int | v != 0 } -> Int @-}
divide :: Int -> Int -> Int
divide n m = if m == 0
             then unreachable "cannot divide by zero"
             else n `div` m
```

We cannot call *this* version of `divide` unless the type checker can prove the provided second argument (i.e. corresponding to parameter `m`) is non-zero. Furthermore, the usage of `unreachable` ensures the **`then`** branch will not type check unless it is provably unreachable. In other words, the type system must learn from the test `m == 0` and understand that in the **`then`**-branch any occurrence of `m` would have type `{v: Int | (v != 0) ∧ (v == 0)}`, which is impossible, and so that branch is in fact dead code.

With this more specific type signature, callers of `divide` must provide *provably* non-zero terms for the second argument:

```
average :: [Int] -> Int
average xs = divide (foldl (+) 0 xs) (length xs)
--Error: Type Mismatch
--   Inferred type
--      {v : Int | v >= 0 && v == len xs}
--   not a subtype of Required type
--      {v : Int | v != 0}
```

In the case of `average`, there is an implicit requirement that `xs` be a non-empty list of **`Int`**. Since this invariant is not enforced in any way (i.e. either by a refinement on the `xs` argument to `average` or with a dynamic check), the call to `divide` in the body of `average` fails.

1.2 Set-theoretic types

Although the majority of type systems with occurrence typing feature some set-theoretic types, most suffer from two notable flaws which hinder their ability to describe some language-specific idioms. First, they fail to reason completely about their types (e.g. subtyping), meaning that programs which may seem obviously correct to a programmer may fail to type check. And second, they are unable to reason about types as sets of values in ways besides unions (i.e. they omit intersection and/or negation tyoes); ways that can feel just as natural to programmers and appear in existing untyped idioms.

We can observe the first problem—that of incomplete reasoning—examining why *syntactic* subtyping rules (i.e. those used almost universally by type systems) frequently fail to recognize valid subtyping relations that occur in real programs featuring set-theoretic types. For example, consider these standard syntactic rules describing reflexivity, union, and product subtyping:

$$\frac{}{\tau <: \tau} \qquad \frac{\tau <: \sigma_1}{\tau <: \sigma_1 \cup \sigma_2} \qquad \frac{\tau <: \sigma_2}{\tau <: \sigma_1 \cup \sigma_2} \qquad \frac{\tau_1 <: \sigma \quad \tau_2 <: \sigma}{\tau_1 \cup \tau_2 <: \sigma} \qquad \frac{\tau_1 <: \sigma_1 \quad \tau_2 <: \sigma_2}{\tau_1 \times \tau_2 <: \sigma_1 \times \sigma_2}$$

With these rules, it is impossible to prove $(\alpha \cup \beta) \times \beta <: (\alpha \times \beta) \cup (\beta \times \beta)$ even though both types describe the *exact same set of values*: pairs with an α or β in the first field and a β in the second field. This unfortunately means users of such systems are explicitly encouraged by the type system to reason set-theoretically and then punished for doing so.

The unfortunate effects of the second problem—lacking intersection and/or negation types—we see when trying to precisely describe the behavior for the commonly used `hash-ref` function in Racket:

```
(define (hash-ref h k [fail (λ () (error "key not found"))])
  (cond
   [(hash-has-key? h k) (unsafe-hash-ref h k)]
   [(procedure? fail) (fail)]
   [else fail]))
```

Here are some simple REPL examples of `hash-ref` at work:

```
> (define ages (hash "Charlotte" 9
                     "Harrison"  7
                     "Sydney"    4))
> (hash-ref ages "Sydney") ;; ==> 4
> (hash-ref ages "Dwight" 10) ;; ==> 10
> (hash-ref ages "Dwight" (λ () 10)) ;; ==> 10
> (hash-ref ages "Dwight") ;; ERROR! key not found
```

We redefined `hash-ref`—a primitive in Racket— to call an invented primitive function `unsafe-hash-ref` to more clearly illustrate `hash-ref`'s semantics. It's arguments can be interpreted as follows:

- `h` is a hash table,
- `k` is the key whose associated value should be fetched from `h`, and
- `fail` is an optional argument that determines what the result is when no entry for `k` is found in `h`. In particular, if there is no entry for `k` in `h`, then:
  - if `fail` is a procedure, the result is the result of calling `fail` with no arguments, otherwise
  - if `fail` is not a procedure, the result is simply `fail` itself.

When ascribing a type to `hash-ref`, the first two argument types are straightforward: `h` should be of type `(Hashtable A B)` and `k` should be of type `A`. If no third argument is provided, the function will return a `B` or error. But what about when the third argument (i.e. `fail`) is provided? From the definition, we see the behavior of `hash-ref` can vary depending on whether `fail` is a procedure or a *non-procedure*. If it is a procedure, it must take no arguments and it will return some value, say of type `C`, making `fail` have type `(-> C)`. If it is any non-procedure—i.e. a value of type `C` and `(¬ Procedure)`—then that value is simply returned. The following is a polymorphic, overloaded function type that captures these three possibilities:

```
  (All (A B C)
    (case->
      [(Hashtable A B) A                     -> B]
      [(Hashtable A B) A (-> C)              -> (U B C)]
      [(Hashtable A B) A (∩ C (¬ Procedure)) -> (U B C)]))
```

Because negation types are not present in Typed Racket, however, the following incomplete type is used instead:[3]

[3]the real type is actually slightly more complex for non-interesting reasons, but we present the key ideas.

```
  (All (A B C)
    (case->
      [(Hashtable A B) A         -> B]
      [(Hashtable A B) A (-> C) -> (U B C)]
      [(Hashtable A B) A False  -> (U B False)]))
```

And while this type is indeed sound, it limits what programmers can provide for the third argument, interfering with many basic uses (e.g. storing booleans in the hash table and using a symbol to signal no entry was found).

Both of these issues are illustrative of the challenges that often occur in occurrence typing systems which do not completely reason about and/or do not feature the complete spectrum of set-theoretic types. As we will argue in more detail shortly, a type system can overcome such challenges by considering a *semantic* interpretation of types, in which the meaning of a type *is* the set of values it denotes. In fact, we will show how embracing semantic subtyping and the full range of set-theoretic types can enrich and simplify occurrence typing systems.

1.3 Thesis Statement and Outline

With occurrence typing defined and some intuition for how it might relate to refinement and set-theoretic types generally, we can present our thesis statement:

> *Occurrence typing combines with refinement and set-theoretic types to form more expressive and more complete type systems.*

In particular, by "more expressive and more complete" we mean the resulting systems are capable of accurately describing more untyped idioms with types and can successfully check more programs because of this. This thesis is defended in the remainder of this document as follows:

- chapter 2 lays out a foundational calculus for occurrence typing and gives an overview of various occurrence typing approaches;
- chapter 3 describes an occurrence typing calculus with refinement types and the results of scaling the approach to larger systems along with related work;

- chapter 4 gives an overview of set-theoretic types, their semantic interpretation, and how to actually implement such systems;
- chapter 5 describes an occurrence typing calculus built on set-theoretic types and semantic subtyping and examines how non-trivial real world systems—such as Typed Racket's numeric tower—might benefit from the added expressiveness.

CHAPTER 2
OCCURRENCE TYPING

Stated broadly, occurrence typing is the ability for a type system to check different occurrences of the same variable at different types. This is highlighted in the name perhaps because it is the most "obvious" feature necessary for effectively type checking untyped programs and it stands in stark contrast to the standard practice (i.e. where a variable has exactly on type for the entirety of its lexical scope). In practice however, we find that occurrence typing is necessary but not sufficient for type checking many untyped programs. To cover many of the idioms used by programmers in the wild, we argue a *collection* of type system features similar to the following is desirable:

- occurrence typing;
- type predicates;
- untagged (i.e. true) union types;
- positive and negative reasoning about the results of type predicates;
- idiomatic reasoning about "null-checks";
- logical reasoning; and
- structural reasoning about certain values.

We give descriptions and justification for these features in the following section, which reviews examples given by Tobin-Hochstadt and Felleisen [9] as demonstrative examples of untyped idioms a type checker of untyped languages should handle. Admittedly, some system designers may choose a slightly different feature set, however the above list seems both foundational and large enough to be useful in practice, and so we will use it for our study of occurrence typing generally in this work. The key ideas can be easily adapted to similar systems.

2.1 Occurrence Typing Examples

Here we review some examples from prior work [9] which help illustrate why the aforementioned type system features make a reasonable starting point when studying "real-world" occurrence typing system. Although these examples are drawn from a Lisp/Scheme-like language (Racket), the same fundamental patterns arise in most untyped languages. If the reader needs no convincing or is already familiar with the aforementioned features, this section can be safely skipped. A core calculus for a type checker well-suited for checking these kinds of programs is given later in this chapter (see section 2.2).

The most basic example of occurrence typing can be seen in example 1. Here, regardless of what value `x` has, the expression will return a number:

Example 1

```
(if (number? x)
    (add1 x)
    0)
```

We believe this expression is well typed for primarily two reasons: (1) the `number?` function is known to be a **type predicate** for numbers (i.e. it accepts *any* value and returns `true` if and only if the input is a number, otherwise it returns **`false`**) and (2) we know that in the "then-branch" the test expression `(number? x)` was **positive** (i.e. produced a non-**`false`** value) and so the occurrence of `x` in `(add1 x)` is indeed a number.

While example 1 shows how simple positive reasoning about type predicates can be of use, example 2 shows how the **negative** result of a type predicate (i.e. when it returns **`false`**) and an **untagged union** can work together to inform the type system:

Example 2

```
(: magnitude (-> (U String Number) Number))
(define (magnitude x)
  (if (number? x)
      (abs x)
      (string-length x)))
```

Because of the type annotation we know that the parameter `x` has the union type `(U String Number)`, i.e. it is *either* a `String` or `Number`. The then-branch checks as it did in example 1, while the "else-branch" type checks because of the negative information that is gleaned regarding `x`. In particular, when the predicate returns **`false`**, we learn that `x` is *not* a number. This combined with what we initially we knew about the type of `x` allows us to conclude in the else-branch that `x` must be a `String` and thus `(string-length x)` too is well typed.

Example 3 is similar to examples 1 and 2, except that instead of an explicit predicate, we perform a **null-check** of sorts on `x` by testing directly whether or not it is **`false`**:

Example 3

```
(let ([x (assoc v l)])
  (if x
      ;; compute with x ...
      (error (format "~v not in ~v" v l))))
```

In the non-**`false`** case (the commented out then-branch), we expect the type system to conclude that `x` is indeed the key/value pair for `v` in the association list `l` (since `assoc` returns either that or **`false`**).

Now let us consider why **logical reasoning** is necessary for an effective occurrence typing system. To type check this next example we must reason correctly about logical disjunctions and their implications:

Example 4

```
(if (or (number? x)
        (string? x))
    (magnitude x)
    0)
```

In particular, in example 4 we expect the type system to understand that in the then-branch exactly one of the predicates for `x` produced `true` (although precisely which is uncertain). Because of this, we should be able to pass `x` to a function which accepts *either* type (such as `magnitude` from example 2). Similarly, the type system should be able to reason about the logical implications of conjunctions as well:

Example 5

```
(if (and (number? x) (string? y))
    (+ x (string-length y))
    0)
```

In example 5 we use predicate tests over multiple variables and the type system should learn something about all applicable variables when the conjunction of those tests returns `true`. Example 6 is similar to example 5, except with a subtle "programmer error" that must be highlighted by the type system:

Example 6

```
;; x is a Number or String
(if (and (number? x) (string? y))
    (+ x (string-length y))
    (string-length x))
```

Here the test-expression and then-branch should type check successfully, but the else branch should fail because it is not clear *why* the test-expression produced **`false`**: it could be because `x` is a string instead of a number, *or* it could simply be that `y` is not a string.

Example 7 is also similar to example 5 except that the **`and`**-macro has been expanded. The type system should still be able to reason about this control flow to see that the then-branch of the outer-most **`if`** is only executed when `x` is a `Number` and `y` is a `String`.

Example 7

```
(if (if (number? x) (string? y) false)
    (+ x (string-length y))
    0)
```

Example 8 demonstrates an important feature: the ability for the user to abstract over a type predicate (in this case for the type `(U String Number)`):

Example 8

```
(: str-num? (-> Any Boolean : (U String Number)))
(define (str-num? x)
  (or (string? x) (number? x)))
```

This ability is key because predicates for untagged union types—a staple in reasoning about most any untyped language—simply do not exist a priori: they must be constructed and/or synthesized by composing more primitive pre-existing predicates.

Example 9 is the macro-expansion of example 4, i.e. the **`or`**-macro connecting the type tests for `x` has been expanded into the equivalent **`let`**-expression:

Example 9

```
(if (let ([tmp (number? x)])
       (if tmp tmp (string? x)))
    (magnitude x)
    0)
```

Similarly here the type checker should still be able to conclude that in the then-branch `x` must have type `Number` or `String` since the logically equivalent expansion of the **`or`**-macro must have produced `true`.

Example 10 introduces the need for **structural reasoning** about a term:[1]

Example 10

```
(if (and (pair? p) (number? (fst p)))
    (add1 (fst p))
    7)
```

In other words, initially `p` can have any type. If `(pair? p)` produces `true`, we know at a minimum it is of type `(Pair Any Any)` and that the other expression in the conjunct will be executed. With `p` having type `(Pair Any Any)`, we can successfully type check `(fst p)` and ask if it is a number. If this conjunction produces `true`, we know in the then-branch `(fst p)` now has type `Number`, and we can thus check `(add1 (fst p))` successfully.

Example 11 is like example 10 but shows that not only should the type of expressions such as `(fst p)` be able to be updated via occurrence typing, but that those updates should impact the type of `p` itself. I.e., in this example, testing the types of both `fst` and `snd` of `p` has the effect up updating the type of `p` itself.

[1]We use `fst` and `snd` in lieu of the actual (and more opaque) pair accessor names `car` and `cdr`.

Example 11

```
(: norm (-> (Pair Number Number) Number))
(define (norm p)
  (sqrt (+ (expt (fst p) 2) (expt (snd p) 2))))

(λ ([p : (Pair Any Any)])
  (if (and (number? (fst p)) (number? (snd p)))
      (norm p)
      (error "non-number pair!")))
```

Example 12 indicates that predicate abstraction should also work for the structural subcomponents of arguments when desired. I.e., `fst-num?` is essentially a number predicate for the `fst` field of its argument.

Example 12

```
(: fst-num? (-> (Pair Any Any) Boolean : (Pair Number Any)))
(define (fst-num? p)
  (number? (fst p)))
```

Example 13 creates a setting where all of this logical reasoning should work together in a complex conditional expression:

Example 13

```
(cond [(and (number? x) (string? y))
       ;; clause 1
       ...]
      [(number? x)
       ;; clause 2
       ...]
      [else
       ;; clause 3
       ...])
```

In particular, in clause 1, we know that `x` is a `Number` and `y` is a `String`. In clause 2, `x` is known to be a `Number` and `y` is known to not be a `String` (since the previous predicate regarding `x` and `y` produced **`false`**). Finally, in clause 3 `y` is known to be a `String` (since we now know `x` is not a `Number`).

Finally, example 14 is like example 13 but includes structural reasoning, presenting a function whose correctness depends on almost all of the aforementioned features working together within the type checker:

Example 14

```
(λ ([input : (U String Number)]
    [extra : (Pair Any Any)])
  (cond [(and (number? input) (fst-num? extra))
         (+ input (fst extra))]
        [(fst-num? extra)
         (+ (string-length input) (fst extra))]
        [else 0]))
```

2.2 λ_{OT}: A Calculus for Occurrence Typing

We define a λ-calculus—dubbed λ_{OT}—which acts as a foundation upon which we can build and experiment. λ_{OT} is roughly equivalent to the λ_{TR} calculus introduced by Tobin-Hochstadt and Felleisen [9] with a slightly modified syntax. We use this formalism because it is expressive enough to cope with many real-world idioms that arise in untyped programming (i.e. those found in section 2.1) and it still accurately describes how some non-trivial occurrence typing systems in use today—such as Typed Racket and Typed Clojure—fundamentally operate.

Because the formalisms introduced in sections 3.2 and 5.2 closely resemble the approach taken by λ_{OT}, it will be a useful to first understand how λ_{OT} works before examining how refinements and semantic set-theoretic types might affect such a system.

2.2.1 λ_{OT} Syntax

The syntax of expressions, types, propositions, and other forms are given in figure 2.1.

Constants and Expressions in λ_{OT} describe a relatively simple lambda calculus: it has integers, booleans, and unary primitive operations as its constants; variables and function application are standard; λ-abstractions are annotated with their argument type for simplicity of type checking; conditionals and local binding forms are standard; pair construction and projection is explicit, allowing us to omit polymorphism for simplicity.

Indices abstract over valid pair indices, simplifying some rules and metafunctions.

Types deserve some detailed description. The universal "top" type `Any` is the type which describes all well typed terms. `Int` is the type of integers, while `True` and `False` are

$i ::=$	$1 \mid 2$	**Indices**	$c ::=$		**Constants**
$e ::=$		**Expressions**		$\mid$ *int*	integer value
	$\mid c$	constant		$\mid \texttt{true}$	true value
	$\mid x, y, z$	variables		$\mid \texttt{false}$	false value
	$\mid (e\ e)$	application		$\mid$ *uop*	primitive ops
	$\mid (\lambda (x\!:\!\tau)\, e)$	abstraction	$\pi ::=$		**Paths**
	$\mid (\texttt{if}\ e\ e\ e)$	conditional		$\mid x$	variable
	$\mid (\texttt{let}\,(x\ e)\ e)$	local binding		$\mid (\mathit{proj}\ i\ \pi)$	field access
	$\mid (\texttt{pair}\ e\ e)$	pair construction	$o ::=$		**Symbolic Objects**
	$\mid (\texttt{proj}\ i\ e)$	pair projection		$\mid \pi$	path object
$\tau, \sigma ::=$		**Types**		$\mid \top^o$	empty object
	$\mid \texttt{Any}$	universal type	$p, q ::=$		**Propositions**
	$\mid \texttt{Int}$	integer type		$\mid \mathbb{tt}$	trivial prop
	$\mid \texttt{True}$	true types		$\mid \mathbb{ff}$	absurd prop
	$\mid \texttt{False}$	false type		$\mid p \wedge p$	conjunction
	$\mid \tau \times \tau$	product type		$\mid p \vee p$	disjunction
	$\mid (x\!:\!\tau) \to \mathrm{R}$	arrow type		$\mid \pi \in \tau$	π is of type τ
	$\mid (\bigcup \vec{\tau})$	type union		$\mid \pi \notin \tau$	π is not of type τ
$R ::=$	$\langle \tau, p, q, o \rangle$	**Type-Results**	$\Gamma ::=$	$\vec{p}$	**Type Env**

Figure 2.1: λ_{OT} Syntax

the types of the boolean values `true` and `false`. Pair types are written $\tau \times \sigma$, describing pairs whose first field is of type τ and whose second field is of type σ. Function types consist of a named argument x, a domain type τ, and codomain type-result R in which x is bound. $(\bigcup \vec{\tau})$ describes a "true" (i.e. untagged) union of the components in $\vec{\tau}$. For convenience we write the boolean type $(\bigcup\ \texttt{True}\ \texttt{False})$ as `Bool` and the uninhabited 'bottom' type $(\bigcup)$ as `Empty`.

Propositions are a key component of λ_{OT}, providing for a standard propositional logic with some type-specific features. $\mathbb{tt}$ and $\mathbb{ff}$ are the trivial and absurd propositions, while $p \wedge q$ and $p \vee q$ represent the conjunction and disjunction of propositions p and q. Type information is expressed by propositions of the form $\pi \in \tau$ and $\pi \notin \tau$, which state that the path π is and is not of type τ respectively.

Paths. Our type system supports occurrence typing by allowing logical propositions about the types of *certain pure terms* in our language, which we dub *paths*. λ_{OT} supports variable and pair-projection paths. One can think of them as representing some known, pure computational path to a value about which we can make type-related claims.

Symbolic Objects. This syntactic abstraction can be thought of as a "maybe path", i.e. either some path π or $\top^o$ to indicate no path. $\top^o$ is used to identity expressions whose type the logic will not reason about. In λ_{OT} we essentially use $\top^o$ to indicate a term's value will not necessarily correspond to a named value (i.e. a variable or variable's subfield), but in more complex systems $\top^o$ could also be used for potentially impure computations (allowing mutation to be soundly supported without reasoning about effects explicitly, for example).

Type-Results allow λ_{OT} to easily reason about more than the just the type τ of an expression. I.e., in addition to describing an expression's type, a type-result further informs the system by explicitly describing (1) what can be inferred in the respective branches of a conditional if this expression is used as the test-expression—described by the pair of propositions p (the 'then proposition') and q (the 'else proposition') in the type result—and (2) which symbolic object o the expression's evaluation would corresponds to.

Environments are simply collections of propositions. Note that in an efficient implementation of such a system it is useful to separate the environment into two portions: a traditional mapping of variables to types along with a set of currently known propositions. The latter can then be used to refine the former during type checking.

2.2.2 λ_{OT} Type System

Instead of assigning types, λ_{OT}'s typing judgment assigns *type-results* to expressions:

$$\Gamma \vdash e : \langle \tau, p, q, o \rangle$$

This judgment states that in environment Γ

- e has type τ;
- if e evaluates to a non-`false` (i.e. treated as true) value, "then proposition" p holds;
- if e evaluates to `false`, "else proposition" q holds;
- e's value corresponds to the symbolic object o.

$\boxed{\Gamma \vdash e : \mathrm{R}}$

T-Const
$$\Gamma \vdash c : \Delta(c)$$

T-Abs
$$\frac{\Gamma, x \in \tau \vdash e : \mathrm{R}}{\Gamma \vdash (\lambda(x{:}\tau)\, e) : \langle (x{:}\tau) \to \mathrm{R}, \mathbb{tt}, \mathbb{ff}, \top^o \rangle}$$

T-Var
$$\frac{\Gamma \vdash x \in \tau}{\Gamma \vdash x : \langle \tau, x \notin \mathtt{False}, x \in \mathtt{False}, x \rangle}$$

T-If
$$\frac{\Gamma \vdash e_1 : \langle \mathtt{Any}, p_1, q_1, \top^o \rangle \qquad \Gamma, p_1 \vdash e_2 : \mathrm{R} \qquad \Gamma, q_1 \vdash e_3 : \mathrm{R}}{\Gamma \vdash (\mathtt{if}\ e_1\ e_2\ e_3) : \mathrm{R}}$$

T-Subsume
$$\frac{\Gamma \vdash e : \mathrm{R}' \qquad \Gamma \vdash \mathrm{R}' <: \mathrm{R}}{\Gamma \vdash e : \mathrm{R}}$$

T-Let
$$\frac{\Gamma \vdash e_1 : \langle \tau_1, p_1, q_1, o_1 \rangle \qquad p_x = (x \notin \mathtt{False} \wedge p_1) \vee (x \in \mathtt{False} \wedge q_1) \qquad \Gamma, x \in \tau_1, p_x \vdash e : \mathrm{R}_2}{\Gamma \vdash (\mathtt{let}\, (x\ e_1)\ e_2) : \mathrm{R}_2[x \mapsto o_1]}$$

T-App
$$\frac{\Gamma \vdash e_1 : \langle (x{:}\tau) \to \mathrm{R}, \mathbb{tt}, \mathbb{tt}, \top^o \rangle \qquad \Gamma \vdash e_2 : \langle \tau, \mathbb{tt}, \mathbb{tt}, o_2 \rangle}{\Gamma \vdash (e_1\ e_2) : \mathrm{R}[x \mapsto o_2]}$$

T-Pair
$$\frac{\Gamma \vdash e_1 : \langle \tau_1, \mathbb{tt}, \mathbb{tt}, \top^o \rangle \qquad \Gamma \vdash e_2 : \langle \tau_2, \mathbb{tt}, \mathbb{tt}, \top^o \rangle}{\Gamma \vdash (\mathtt{pair}\ e_1\ e_2) : \langle \tau_1 \times \tau_2, \mathbb{tt}, \mathbb{ff}, \top^o \rangle}$$

T-Proj
$$\frac{\Gamma \vdash e : \langle \tau_1 \times \tau_2, \mathbb{tt}, \mathbb{tt}, o \rangle \qquad o' = (proj\ i\ x) \qquad \mathrm{R} = \langle \tau_i, o' \notin \mathtt{False}, o' \in \mathtt{False}, o' \rangle}{\Gamma \vdash (\mathtt{proj}\ i\ e) : \mathrm{R}[x \mapsto o]}$$

Figure 2.2: λ_{OT} Typing Judgment

$\boxed{\Delta : c \to \mathrm{R}}$

$\Delta(int)$	$= \langle \mathtt{Int}, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$
$\Delta(\mathtt{true})$	$= \langle \mathtt{True}, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$
$\Delta(\mathtt{false})$	$= \langle \mathtt{False}, \mathbb{ff}, \mathbb{tt}, \top^o \rangle$
$\Delta(\mathtt{not})$	$= \langle (x{:}\mathtt{Any}) \to \langle \mathtt{Bool}, x \in \mathtt{False}, x \notin \mathtt{False}, \top^o \rangle, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$
$\Delta(\mathtt{zero}?)$	$= \langle (x{:}\mathtt{Int}) \to \langle \mathtt{Bool}, \mathbb{tt}, \mathbb{tt}, \top^o \rangle, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$
$\Delta(\mathtt{sub1})$	$= \langle (x{:}\mathtt{Int}) \to \langle \mathtt{Int}, \mathbb{tt}, \mathbb{ff}, \top^o \rangle, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$
$\Delta(\mathtt{add1})$	$= \langle (x{:}\mathtt{Int}) \to \langle \mathtt{Int}, \mathbb{tt}, \mathbb{ff}, \top^o \rangle, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$
$\Delta(\mathtt{int}?)$	$= \langle (x{:}\mathtt{Any}) \to \langle \mathtt{Bool}, x \in \mathtt{Int}, x \notin \mathtt{Int}, \top^o \rangle, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$
$\Delta(\mathtt{bool}?)$	$= \langle (x{:}\mathtt{Any}) \to \langle \mathtt{Bool}, x \in \mathtt{Bool}, x \notin \mathtt{Bool}, \top^o \rangle, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$
$\Delta(\mathtt{pair}?)$	$= \langle (x{:}\mathtt{Any}) \to \langle \mathtt{Bool}, x \in \mathtt{Any} \times \mathtt{Any}, x \notin \mathtt{Any} \times \mathtt{Any}, \top^o \rangle, \mathbb{tt}, \mathbb{ff}, \top^o \rangle$

Figure 2.3: λ_{OT} Constant Type-Results

T-Const is used for type checking the respective constants values, consulting the Δ metafunction described in figure 2.3. Note that the then- and else-propositions are consistent with whether the constant is non-`false` or `false`. They all have the symbolic object $\top^o$ since we gain nothing from reasoning about the type of language constants (i.e. they are already assigned their most precise type).

T-Var may assign any type τ to variable x so long as the system can derive $\Gamma \vdash x \in \tau$. The then- and else-propositions reflect the self evident fact that if x evaluates to a non-`false` value then x is not of type `False`, otherwise it is of type `False`. The symbolic object informs the type system that this expression corresponds to the path x.

T-Abs, the rule for checking lambda abstractions, checks the body of the abstraction in the extended environment which maps x to τ. We use the standard convention of choosing fresh names not currently bound when extending Γ with new bindings. The type-result from checking the body is then used as the range for the function type, and the then- and else-propositions report the non-falseness of a function value.

T-App handles function application, first checking that e_1 and e_2 are well-typed individually and then ensuring the type of e_2 is a subtype of the domain of e_1. The overall type-result of the application is R from the type of e_1, with the symbolic object of the operand, o_2, substituted for the name x (this allows the type result to now specifically talk about the argument which was provided in this case: o_2).

T-If is used for conditionals, describing the important process by which information learned from evaluating test-expressions is projected into the respective branches. After ensuring e_1 is well-typed at some type, we make note of the then- and else-propositions p_1 and q_1. We then extend the environment with the appropriate proposition, dependent upon which branch we are checking: p_1 is assumed while checking the then-branch and q_1 for the else-branch. The type result of a conditional is simply the type result implied by both branches (which can be determined by pointwise unioning their respective type-results).

T-Let first checks whether the expression e_1 whose value will be bound to x is well typed. When checking the body, the environment is extended with the type for x and a proposition describing x's then- and else- propositions (i.e. what we can learn from testing x). Since x is chosen to be fresh it is unbound outside the body; we then substitute o_1 for

x on the result as we do with function application, since we know what object corresponds to x and x will not be in scope outside of this expression.

In order to omit polymorphism we use explicit pair introduction and elimination rules. T-Pair introduces pairs, first checking the types for e_1 and e_2. The type-result then includes the product of these individual types, propositions reflecting the non-`false` nature of the value, and a trivial symbolic object (note that in principle we could have symbolic objects for pairs as well). Pair elimination forms are checked with T-Proj, which ensure its argument is indeed a pair before returning the respective type and a symbolic object describing which field was accessed.

Object Substitution on type-results is performed pointwise and structurally on the respective subcomponents. When substituting the object $\top^o$ for a variable x in a path π, if $x \in \mathsf{fvs}(\pi)$ then $\pi[x \mapsto \top^o] = \top^o$. When substituting in a proposition $(\pi \in \tau)$ or $(\pi \notin \tau)$, if π becomes $\top^o$ then the entire proposition becomes $\mathbb{tt}$.

Well Formedness. For any judgment $\Gamma \vdash e : \mathrm{R}$, we require that the free variables in e and R be a subset of those found in Γ.

2.2.3 λ_{OT} Subtyping

Figure 2.4 describes the subtyping relation <: for types and symbolic objects.

For objects, the null object $\top^o$ is the top object and the relation is reflexive.

All types are subtypes of themselves and of the top type `Any`. A type is a subtype of a union if it is a subtype of any element of the union. Unions are only subtypes of a type if *every* member of the union is a subtype of that type. Function subtyping has the standard contra- and co-variance in the domain and range. Pair subtyping is standard.

Type-result subtyping is the pointwise subtyping/implication of the respective parts (with the environment contributing for the propositions).

2.2.4 λ_{OT} Logic and Type Metafunctions

The logic for λ_{OT} is a standard natural deduction-style propositional logic with a few additional type-related rules which are described in figure 2.5. We omit introduction and elimination rules for the atomic propositions and logical connectives as they are entirely

$\boxed{o <: o}$

$$\textsc{SO-Refl}\quad o <: o \qquad\qquad \textsc{SO-Null}\quad o <: \top^{o}$$

$\boxed{\tau <: \tau}$

$$\textsc{S-Refl}\quad \tau <: \tau \qquad \textsc{S-Top}\quad \tau <: \mathsf{Any} \qquad \textsc{S-Union1}\ \frac{\forall \tau' \text{ in } \vec{\tau}.\ \tau' <: \sigma}{(\bigcup \vec{\tau}) <: \sigma} \qquad \textsc{S-Union2}\ \frac{\exists \sigma' \text{ in } \vec{\sigma}.\ \tau <: \sigma'}{\tau <: (\bigcup \vec{\sigma}))}$$

$$\textsc{S-Pair}\ \frac{\tau_1 <: \tau_2 \qquad \sigma_1 <: \sigma_2}{\tau_1 \times \sigma_1 <: \tau_2 \times \sigma_2} \qquad \textsc{S-Fun}\ \frac{\tau_2 <: \tau_1 \qquad \Gamma \vdash \mathrm{R}_1 <: \mathrm{R}_2}{(x{:}\tau_1) \rightarrow \mathrm{R}_1 <: (x{:}\tau_2) \rightarrow \mathrm{R}_2}$$

$\boxed{\Gamma \vdash \mathrm{R} <: \mathrm{R}}$

$$\textsc{SR-Sub}\ \frac{\tau <: \tau' \qquad o <: o' \qquad \Gamma, p \vdash p' \qquad \Gamma, q \vdash q'}{\Gamma \vdash \langle \tau, p, q, o \rangle <: \langle \tau', p', q', o' \rangle}$$

Figure 2.4: λ_{OT} Subtyping

standard. Like the typing judgment, we require variables mentioned on the right-hand side of the turnstile to be a subset of those mentioned on the left, i.e. for $\Gamma \vdash p$ to be well-formed, $\mathsf{fvs}(p) \subseteq \mathsf{fvs}(\Gamma)$ must hold.

T-Empty is analogous to the traditional principle of "ex falso quodlibet", i.e. if we can prove a term has the uninhabited type we can prove anything.

Rules T-Update+ and T-Update- allow us to *refine* the type of a term by combining its known type with some other known positive or negative information about that term. For example, suppose we know a path π is of type τ and that some field further into that path $(proj\ \vec{i}\ \pi)$[2] is or is not of type σ. We roughly want to update the type along that potentially deeper path as follows: if we know $(proj\ \vec{i}\ \pi) \in \sigma$—that the field along $\vec{i}$ within π *is* of type σ—we update that field's type τ' to be $\mathsf{restrict}(\tau', \sigma)$ (i.e. the conservative "intersection" of the two types); conversely, updating a field's type τ' with the knowledge that the field *is not* σ updates the field to be $\mathsf{remove}(\tau', \sigma)$ (i.e. the conservative "difference" between the two).

The update metafunction—also described in figure 2.5—essentially describes how to

[2]Here we abbreviate $(proj\ i_n\ (...\ (proj\ i_1\ \pi)))$ as $(proj\ \vec{i}\ \pi)$ where $\vec{i}$ is the (potentially empty) sequence of field accesses $i_n :: ... :: i_1$.

$$\textsc{L-Empty}\quad \frac{\Gamma \vdash \pi \in \texttt{Empty}}{\Gamma \vdash p} \qquad \textsc{L-Sub}\quad \frac{\Gamma \vdash \pi \in \sigma \qquad \sigma <: \tau}{\Gamma \vdash \pi \in \tau} \qquad \textsc{L-SubNot}\quad \frac{\Gamma \vdash \pi \notin \sigma \qquad \tau <: \sigma}{\Gamma \vdash \pi \notin \tau}$$

$$\textsc{L-Update+}\quad \frac{\begin{array}{c}\Gamma \vdash \pi \in \tau \\ \Gamma \vdash (proj\ \vec{i}\ \pi) \in \sigma\end{array}}{\Gamma \vdash \pi \in \mathsf{update}^{+}(\tau, \vec{i}, \sigma)} \qquad \textsc{L-Update–}\quad \frac{\begin{array}{c}\Gamma \vdash \pi \in \tau \\ \Gamma \vdash (proj\ \vec{i}\ \pi) \notin \sigma\end{array}}{\Gamma \vdash \pi \in \mathsf{update}^{-}(\tau, \vec{i}, \sigma)}$$

$$\boxed{\mathsf{update} : \pm\ \tau\ \vec{i}\ \tau \to \tau}$$

$$\begin{array}{ll}
\mathsf{update}^{\pm}(\tau_1 \times \tau_2, \vec{i}::1, \sigma) & = \mathsf{update}^{\pm}(\tau_1, \vec{i}, \sigma) \times \tau_2 \\
\mathsf{update}^{\pm}(\tau_1 \times \tau_2, \vec{i}::2, \sigma) & = \tau_1 \times \mathsf{update}^{\pm}(\tau_2, \vec{i}, \sigma) \\
\mathsf{update}^{+}(\tau, \epsilon, \sigma) & = \mathsf{restrict}(\tau, \sigma) \\
\mathsf{update}^{-}(\tau, \epsilon, \sigma) & = \mathsf{remove}(\tau, \sigma) \\
\mathsf{update}^{\pm}((\bigcup \vec{\tau}), \vec{i}, \sigma) & = (\bigcup \overrightarrow{\mathsf{update}^{\pm}(\tau, \vec{i}, \sigma)})
\end{array}$$

$$\boxed{\mathsf{restrict} : \tau\ \tau \to \tau}$$

$$\begin{array}{lll}
\mathsf{restrict}(\tau, \sigma) & = \texttt{Empty}\ \text{ if } \nexists v.\vdash v : \tau \text{ and } \vdash v : \sigma & \\
\mathsf{restrict}((\bigcup \vec{\tau}), \sigma) & = (\bigcup \overrightarrow{\mathsf{restrict}(\tau, \sigma)}) & \\
\mathsf{restrict}(\tau, \sigma) & = \tau & \text{if} \vdash \tau <: \sigma \\
\mathsf{restrict}(\tau, \sigma) & = \sigma & \text{otherwise}
\end{array}$$

$$\boxed{\mathsf{remove} : \tau\ \tau \to \tau}$$

$$\begin{array}{ll}
\mathsf{remove}(\tau, \sigma) & = \texttt{Empty}\ \text{if} \vdash \tau <: \sigma \\
\mathsf{remove}((\bigcup \vec{\tau}), \sigma) & = (\bigcup \overrightarrow{\mathsf{remove}(\tau, \sigma)}) \\
\mathsf{remove}(\tau, \sigma) & = \tau \qquad \text{otherwise}
\end{array}$$

Figure 2.5: λ_{OT} Type-related Logic Rules and Metafunctions

syntactically combine type information from two propositions in a conservative syntactic manner. update itself traverses the necessary fields so the correct part of the type is updated before calling either restrict (to combine positive type information) or remove (to subtract negative information from a type). We provide some relatively straightforward derivations to further clarify the utility of the update-related rules in the logic:[3]

Given $\Gamma_1 = \{x \in (\bigcup \texttt{Int}\,\texttt{True}), x \in (\bigcup \texttt{Int}\,\texttt{False})\}$, we can derive $\Gamma_1 \vdash x \in \texttt{Int}$ by using L-Update+ to combine the positive information regarding x:

$$\dfrac{\dfrac{\Gamma_1 \vdash x \in (\bigcup \texttt{Int}\,\texttt{True}) \qquad \Gamma_1 \vdash x \in (\bigcup \texttt{Int}\,\texttt{False})}{\Gamma_1 \vdash x \in \mathsf{update}^{+}((\bigcup \texttt{Int}\,\texttt{True}), \epsilon, (\bigcup \texttt{Int}\,\texttt{False}))}\,\textsc{L-Update+} \qquad \texttt{Int} = \mathsf{update}^{+}((\bigcup \texttt{Int}\,\texttt{True}), \epsilon, (\bigcup \texttt{Int}\,\texttt{False}))}{\Gamma_1 \vdash x \in \texttt{Int}}$$

Given $\Gamma_2 = \{x \in \texttt{Bool}, x \notin \texttt{False}\}$, we can derive $\Gamma_2 \vdash x \in \texttt{True}$ by combining the known type with the negative information about x with L-Update-:

$$\dfrac{\dfrac{\Gamma_2 \vdash x \in \texttt{Bool} \qquad \Gamma_2 \vdash x \notin \texttt{False}}{\Gamma_2 \vdash x \in \mathsf{update}^{-}(\texttt{Bool}, \epsilon, \texttt{False})}\,\textsc{L-Update-} \qquad \texttt{True} = \mathsf{update}^{-}(\texttt{Bool}, \epsilon, \texttt{False})}{\Gamma_2 \vdash x \in \texttt{True}}$$

Given $\Gamma_3 = \{x \in \texttt{Int} \times (\bigcup \texttt{Int}\,\texttt{True}), (\mathit{proj}\ 2\ x) \in \texttt{Bool}\}$, we can derive $\Gamma_3 \vdash x \in \texttt{Int} \times \texttt{True}$ by using L-Update+ to update the second field of the product type which is known for x:

[3]For convenience we automatically flatten and remove Empty from union types when calling update; we could instead add a usage of L-Sub, which would perform the simplification explicitly.

$$\dfrac{\dfrac{\Gamma_3 \vdash x \in \mathtt{Int} \times (\bigcup\ \mathtt{Int}\ \mathtt{True}) \qquad \Gamma_3 \vdash (\mathit{proj}\ 2\ x) \in \mathtt{Bool}}{\Gamma_3 \vdash x \in \mathsf{update}^{+}(\mathtt{Int} \times (\bigcup\ \mathtt{Int}\ \mathtt{True}), 2, \mathtt{Bool})}\ \textsc{L-Update+} \qquad \mathtt{Int} \times \mathtt{True} = \mathsf{update}^{+}(\mathtt{Int} \times (\bigcup\ \mathtt{Int}\ \mathtt{True}), 2, \mathtt{Bool})}{\Gamma_3 \vdash x \in \mathtt{Int} \times \mathtt{True}}$$

Given $\Gamma_4 = \{x \in \mathtt{Int} \times (\mathtt{Bool} \times \mathtt{False}), (\mathit{proj}\ 1\ (\mathit{proj}\ 2\ x)) \notin \mathtt{False}\}$, we can derive $\Gamma_4 \vdash x \in \mathtt{Int} \times (\mathtt{True} \times \mathtt{False})$ by using L-Update- to update the first field of the second field of the known type for x:

$$\dfrac{\dfrac{\Gamma_4 \vdash x \in \mathtt{Int} \times (\mathtt{Bool} \times \mathtt{False}) \qquad \Gamma_4 \vdash (\mathit{proj}\ 1\ (\mathit{proj}\ 2\ x)) \notin \mathtt{False}}{\Gamma_4 \vdash x \in \mathsf{update}^{-}(\mathtt{Int} \times (\mathtt{Bool} \times \mathtt{False}), 1 :: 2, \mathtt{False})}\ \textsc{L-Update-} \qquad \mathtt{Int} \times (\mathtt{True} \times \mathtt{False}) = \mathsf{update}^{-}(\mathtt{Int} \times (\mathtt{Bool} \times \mathtt{False}), 1 :: 2, \mathtt{False})}{\Gamma_4 \vdash x \in \mathtt{Int} \times (\mathtt{True} \times \mathtt{False})}$$

2.2.5 λ_{OT} Semantics

λ_{OT} uses the big-step reduction semantics described in figure 2.6, calling the metafunction in figure 2.7 for primitive ops. To understand these semantics, first we describe values (v) and runtime environments (ρ) for λ_{OT}:

$$\begin{array}{ll} v & ::= c \mid (\mathtt{pair}\ v\ v) \mid [\rho,\ (\lambda(x{:}\tau)\ e)] \\ \rho & \overrightarrow{x := v} \end{array}$$

Note that higher-order values are represented with closures—this is because it suits the unique model-theoretic proof technique introduced to prove soundness for this style of calculus [9].

The evaluation judgment $\rho \vdash e \Downarrow v$ states that in runtime-environment ρ, expression e evaluates to the value v. Like many untyped languages, λ_{OT} treats all non-`false` values as “true” for the purposes of conditional test-expressions.

$\boxed{\rho \vdash e \Downarrow v}$

$$\textsc{B-Const}\quad \rho \vdash c \Downarrow c$$

$$\textsc{B-Var}\quad \frac{\rho(x) = v}{\rho \vdash x \Downarrow v}$$

$$\textsc{B-Let}\quad \frac{\rho \vdash e_1 \Downarrow v_1 \qquad \rho[x := v_1] \vdash e_2 \Downarrow v}{\rho \vdash (\texttt{let}\,(x\ e_1)\ e_2) \Downarrow v}$$

$$\textsc{B-Abs}\quad \rho \vdash (\lambda(x\!:\!\tau)\,e) \Downarrow [\rho,\ (\lambda(x\!:\!\tau)\,e)]$$

$$\textsc{B-Proj}\quad \frac{\rho \vdash e \Downarrow (\texttt{pair}\ v_1\ v_2)}{\rho \vdash (\texttt{proj}\ i\ e) \Downarrow v_i}$$

$$\textsc{B-Pair}\quad \frac{\rho \vdash e_1 \Downarrow v_1 \qquad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash (\texttt{pair}\ e_1\ e_2) \Downarrow (\texttt{pair}\ v_1\ v_2)}$$

$$\textsc{B-Beta}\quad \frac{\rho \vdash e_1 \Downarrow [\rho_c, (\lambda(x\!:\!\tau)\,e)] \qquad \rho \vdash e_2 \Downarrow v_2 \qquad \rho_c[x := v_2] \vdash e \Downarrow v}{\rho \vdash (e_1\ e_2) \Downarrow v}$$

$$\textsc{B-Prim}\quad \frac{\rho \vdash e_1 \Downarrow uop \qquad \rho \vdash e_2 \Downarrow v_2 \qquad \delta(uop,\ v_2) = v}{\rho \vdash (e_1\ e_2) \Downarrow v}$$

$$\textsc{B-IfTrue}\quad \frac{\rho \vdash e_1 \Downarrow v_1 \qquad v_1 \neq \texttt{false} \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash (\texttt{if}\ e_1\ e_2\ e_3) \Downarrow v}$$

$$\textsc{B-IfFalse}\quad \frac{\rho \vdash e_1 \Downarrow \texttt{false} \qquad \rho \vdash e_3 \Downarrow v}{\rho \vdash (\texttt{if}\ e_1\ e_2\ e_3) \Downarrow v}$$

Figure 2.6: λ_{OT} Big-step Reduction Relation

$\boxed{\delta : uop\ v \to v}$

$$
\begin{aligned}
\delta(\texttt{not},\ v) &= \begin{cases} \texttt{true} & \text{if } v = \texttt{false} \\ \texttt{false} & \text{otherwise} \end{cases} \\
\delta(\texttt{zero?},\ int) &= \begin{cases} \texttt{true} & \text{if } int = 0 \\ \texttt{false} & \text{otherwise} \end{cases} \\
\delta(\texttt{sub1},\ int) &= int - 1 \\
\delta(\texttt{add1},\ int) &= int + 1 \\
\delta(\texttt{int?},\ v) &= \begin{cases} \texttt{true} & \text{if } v \text{ is an integer} \\ \texttt{false} & \text{otherwise} \end{cases} \\
\delta(\texttt{bool?},\ v) &= \begin{cases} \texttt{true} & \text{if } v \text{ is a boolean} \\ \texttt{false} & \text{otherwise} \end{cases} \\
\delta(\texttt{pair?},\ v) &= \begin{cases} \texttt{true} & \text{if } v \text{ is a pair} \\ \texttt{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 2.7: λ_{OT} Primitive Types

$\boxed{\rho \vDash p}$

$$\textsc{M-Top}\quad \rho \vDash \mathbb{tt} \qquad \textsc{M-Or}\ \frac{\rho \vDash p_1 \text{ or } \rho \vDash p_2}{\rho \vDash p_1 \vee p_2} \qquad \textsc{M-And}\ \frac{\rho \vDash p_1 \qquad \rho \vDash p_2}{\rho \vDash p_1 \wedge p_2}$$

$$\textsc{M-Type}\ \frac{\vdash \rho(\pi) : \tau}{\rho \vDash \pi \in \tau} \qquad \textsc{M-TypeNot}\ \frac{\vdash \rho(o) : \sigma \qquad \nexists v.\vdash v : \tau \text{ and } \vdash v : \sigma}{\rho \vDash o \notin \tau}$$

Figure 2.8: λ_{OT} Models Relation

2.2.6 λ_{OT} Soundness

Because our formalism is described as a type-theory aware logic, it is convenient to examine its soundness using a model-theoretic approach similar to those used in the study of proof theory. For λ_{OT} a model is any runtime-value environment ρ (i.e. a mapping from variables to values) and is said to *satisfy* a proposition p (written $\rho \vDash p$) when its assignment of values to the free variables of p make the proposition a tautology. The details of satisfaction are defined in figure 2.8. The satisfaction relation extends to environments in a pointwise manner.

In order to complete our definition of satisfaction and prove our soundness theorem, we also require a typing rule for closures:

$$\textsc{T-Closure}\ \frac{\exists\Gamma.\ \rho \vDash \Gamma \qquad \Gamma \vdash (\lambda(x\!:\!\tau)\, e) : \mathrm{R}}{\vdash [\rho,\, (\lambda(x\!:\!\tau)\, e)] : \mathrm{R}}$$

The satisfaction rules are mostly straightforward. $\mathbb{tt}$ is always satisfied, while the logical connectives $\vee$ and $\wedge$ are satisfied in the standard ways.

From M-Type we see propositions stating a path π is of type τ are satisfied when the value of π in ρ is a subtype of τ. Similarly M-TypeNot tells us if a path π's value in ρ has a type which does not overlap with τ, then the proposition $\pi \notin \tau$ is satisfied.

Soundness

Our first lemma states that our proof theory respects models.

Lemma 1. *If $\rho \models \Gamma$ and $\Gamma \vdash p$ then $\rho \models p$.*

Proof. By structural induction on $\Gamma \vdash p$. □

With our proof theory and models in sync and our operational semantics defined, we can state and prove the next key lemma for type soundness which deals with evaluation.

Lemma 2. *If $\Gamma \vdash e : \langle \tau, p, q, o \rangle$, $\rho \models \Gamma$ and $\rho \vdash e \Downarrow v$ then all of the following hold:*

1. *either $o = \top^o$, or for some path π, $o = \pi$ and $\rho(\pi) = v$,*
2. *$v \neq$ `false` and $\rho \models p$, or $v =$ `false` and $\rho \models q$, and*
3. *$\Gamma \vdash v : \langle \tau, \mathbb{tt}, \mathbb{tt}, \top^o \rangle$*

Proof. By induction on the derivation of $\rho \vdash e \Downarrow v$. □

Now we can state our soundness theorem for λ_{OT}.

Theorem 1. *(Type Soundness for λ_{OT}). If $\vdash e : \tau$ and $\vdash e \Downarrow v$ then $\vdash v : \tau$.*

Proof. Corollary of lemma 2. □

Although this model-theoretic proof technique works quite naturally, it includes the standard drawbacks of big-step soundness proofs, saying nothing about diverging or stuck terms. We could address this by adding an `error` value of type `Empty` that is propagated upward during evaluation and modify our soundness claim to show `error` is not derived from evaluating well-typed terms.

2.2.7 Scaling Up λ_{OT}

The descriptions in previous sections sought to be straightforward and declarative to highlight the key ideas governing how λ_{OT} operates. In the following sections, we review additional features and implementation strategies which are useful in practice.

Mutation

The simplest way to support mutation in λ_{OT} is to note which variables are mutated and then simply allow no propositions to be introduced for those variables aside from those declaring their initial type when they are bound.

In other words, once we know which variables are immutable and which are mutable, we simply use two typing rules for variables:

$$\textsc{T-ImmutableVar}\quad \frac{\Gamma \vdash x \in \tau \qquad x \text{ is immutable}}{\Gamma \vdash x : \langle \tau, x \notin \texttt{False}, x \in \texttt{False}, x \rangle} \qquad \textsc{T-MutableVar}\quad \frac{\Gamma \vdash x \in \tau \qquad x \text{ is mutable}}{\Gamma \vdash x : \langle \tau, \mathbb{tt}, \mathbb{tt}, \top^{o} \rangle}$$

T-ImmutableVar is identical to T-Var since for immutable variables we can always safely learn facts about them when examining their value. T-MutableVar on the other hand is conservative and provides no propositions or symbolic object, since facts learned from testing a mutable variable's value may not always hold (i.e. the variable may be mutated at some future point).

We also will need two rules for local bindings: one for when the locally bound variable is potentially mutated and one for when it is not:

$$\textsc{T-ImmutableLet}\quad \frac{\begin{array}{c} x \text{ is immutable} \\ \Gamma \vdash e_1 : \langle \tau_1, p_1, q_1, o_1 \rangle \\ p_x = (x \notin \texttt{False} \wedge p_1) \vee (x \in \texttt{False} \wedge q_1) \\ \Gamma, x \in \tau_1, p_x \vdash e : \mathrm{R}_2 \end{array}}{\Gamma \vdash (\texttt{let}\,(x\ e_1)\ e_2) : \mathrm{R}_2[x \mapsto o_1]} \qquad \textsc{T-MutableLet}\quad \frac{\begin{array}{c} x \text{ is mutable} \\ \Gamma \vdash e_1 : \langle \tau_1, \mathbb{tt}, \mathbb{tt}, \top^{o} \rangle \\ \Gamma, x \in \tau_1 \vdash e : \mathrm{R}_2 \end{array}}{\Gamma \vdash (\texttt{let}\,(x\ e_1)\ e_2) : \mathrm{R}_2}$$

Like the variable rules, T-ImmutableLet is the same as T-Let, but T-MutableLet omits the propositions for the variable except for the initial type declaration.

A more expressive approach to supporting mutation would be to add an affect system and allow the refining of mutable variables' types so long as no effects may change their value (Flow [8] uses this approach), but we will not discuss that here since it would involve many changes to the type system.

Aliasing

For simplicity in λ_{OT} we do not reason about how variable's values relate to one another. Doing so, however, can greatly reduce the number of propositions that need to be accounted for when checking the body of local binding forms. For example, in the following program the inner-most body of the **`let`**-expressions has three variables in scope which can be logically reasoned about: `p`, `x`, and `y`.

```
(let ([p ...])
  (let ([x (fst p)]
        [y (snd p)])
    ...)
```

However, the expressions being bound to `x` and `y` both have non-trivial symbolic objects which simply refer to subcomponents of `p`, i.e. `x` is literally just an *alias* for the value at (*proj* 1 `p`) and `y` for the value at (*proj* 2 `p`). If, instead of reasoning about `x` and `y`, we note this aliasing and instead reason about (*proj* 1 `p`) and (*proj* 2 `p`) whenever `x` and `y` occur in that lexical portion of the program, we can reduce the size of our logical environment dramatically. When done consistently, this can be viewed as a form of "copy propagation" which reduces the number of unique variables in a lexical context. Furthermore, tracking these aliasing relations between local objects and their symbolic objects allows for more programs to type check, because information learned about the types of any of the three identifiers can refine the type of the other two.

This explicit aliasing is included in the calculus described in chapter 3, but the technique can be useful even in the absence of refinement types. (In fact, this feature was added to Typed Racket long before refinements.)

Checking vs Synthesizing Logical Information

As may come as no surprise, an implementation of a system like λ_{OT} is often done as a bidirectional type checker in the style first formally introduced by Pierce and Turner [24]. I.e., when an expected type for an expression is present, the type checker can propagate that information downward into the appropriate subcomponents of that expression which can simplify type checking. When there is no expected type for an expression, however, the type must be synthesized and (in our case) all logical information that may be gleaned from that expression's evaluation must be propagated outward by the typing derivation. For example, suppose the following expression (example 9 from section 2.1) were being type checked with the expected type `Number`:

```
(if (let ([tmp (number? x)])
       (if tmp tmp (string? x)))
    (magnitude x)
    0)
```

Because both the then- and else-branches can contribute to the result, both are visited by the type checker in "check mode" with expected type of the entire expression (`Number`) and no "learned propositions" from those expressions (if any existed) need to be considered. However, the test-expression does not have an expected type, and therefore it must be checked in "synthesis mode", reporting both its type and any learned logical propositions. In particular, this means the expression `(if tmp tmp (string? x))` will be checked in synthesis mode and thus its propositions will be the disjunction of the propositions from its then- and else-branches, since we must know what can be learned in either case to understand what can be learned from the entire expression.

While the difference in this case may seem trivial, in large expressions with nested conditionals—like those arising in the expansion of **`match`** statements in Racket—the difference in the size and complexity of the generated propositions can be vast: each nested conditional will result in a disjunction of the learned information from each branch which can lead to fruitless exponential blowup in proposition size. If, however, the nesting of conditionals has an expected type, the logical information can be ignored entirely since we only care that each branch is indeed of the expected type.

Context Conjunctive Normal Form

The following steps help keep Γ in conjunctive normal form (CNF), which in our experience makes logical inquiries during type checking easier to decide. First, continually combine and remove redundant propositions from the environment, i.e. $(\mathit{proj}\ 1\ \pi) \in \tau$ can update $\pi \in \texttt{Any} \times \sigma$ to be $\pi \in \tau \times \sigma$, at which point $(\mathit{proj}\ 1\ \pi) \in \tau$ is redundant (since it should be implied by $\pi \in \texttt{Any} \times \sigma$) and can be removed, etc. Second, perform standard logical transformations so Γ is in CNF. Note that atoms in disjunctions with the same subject often can be combined or "cancel eachother out" by making the union a tautology, and that statements such as $\pi \in \texttt{Any}$ and $\pi \in \texttt{Empty}$ are logically equivalent to $\mathbb{tt}$ and $\mathbb{ff}$ respectively. Once in CNF, use the principle of disjunctive syllogism to simplify and reduce disjunctions by eliminating their provably absurd atoms.

More Subsumption

Subsumption in λ_{OT} relies on the simple "sub-result" relation defined in figure 2.4. This relation can be naturally extended with two obviously sound rules which help simplify type checking in practice by eliminating propositions the type indicates will not hold:

$$\textsc{SR-NonFalse}\quad \frac{\nexists v.\ \Gamma \vdash v : \tau \text{ and } \Gamma \vdash v : \texttt{False}}{\Gamma \vdash \langle \tau, p, q, o\rangle <: \langle \tau, p, \mathbb{ff}, o\rangle} \qquad \textsc{SR-False}\quad \frac{\tau <: \texttt{False}}{\Gamma \vdash \langle \tau, p, q, o\rangle <: \langle \tau, \mathbb{ff}, q, o\rangle}$$

Essentially SR-NonFalse allows us to discard the else-proposition when we can tell from the type that the result cannot be `false`, and SR-False allows us to discard the then-proposition when the result must be `false`.

Additionally, in practice subsumption should be used each time the environment is extended with a logical proposition (i.e. T-If and T-Let) so the information from that proposition is included in the overall output type-result.

2.2.8 Related Work in Occurrence Typing

In this section we give an overview of the ways different systems support occurrence typing and discuss how they relate to λ_{OT}.

Syntactic Type Tests

The simplest method for supporting occurrence typing involves syntactically recognizing type-tests and updating the type environment appropriately for the respective branches. This approach has the advantages of being both easy to understand and implement and directly pairing with the syntactic type tests many languages feature. To demonstrate, let us consider several TypeScript[4] examples which make use of this feature in similar ways.

The simplest and perhaps most common syntactic test we would want an occurrence typing system to understand is the ubiquitous "null check":

```
function louder(s: string | null): string {
    if (s) return s.concat("!");
    else   return "";
}
```

This check could also be written as `(s !== null)`; in either case we are expecting the type system to learn that in the then-branch since `s` is non-**`null`** and therefore a `string`.

In other cases, it may be that certain functions are known a priori to be type predicates (e.g., PHP has a slew of these functions). Some occurrence typing systems recognize when these particular functions appear in a conditional test and will appropriately inform the type system in either branch:

```
function louder(s: string | null): string {
    if (is_string(s)) return s.concat("!");
    else              return "";
}
```

Finally, some languages have special syntactic constructs which inspects a value's type. Occurrence typing systems may syntactically recognize this pattern and determine which type is being tested for in each instance:

```
function louder(s: string | null): string {
    if (typeof(s) == "string") return s.concat("!");
    else                       return "";
}
```

The main limitation of this approach is that it is inherently first order and thus may be at odds a more functional style of programming. For example, consider the `filter` function which takes a predicate `p` and a list `l` and returns a list containing each element `x` in `l` for which `(p x)` produced a non-**`false`** value. The type for `filter` ideally would reflect the fact that when given a *type predicate*, the type of the resulting list is determined by what type the predicate was for. E.g., if we use `number?` as the predicate, the result should have type `(Listof Number)`:

[4]TypeScript 3.2 with the `strictNullChecks` option enabled so types are non-nullable by default.

```
> (define l (list 1 "2" 3 'four (list 5)))
> (filter number? l)
'(1 3)
```

However this is only possible when the *type* of a predicate—not merely its syntactic occurrence in a program—somehow meaningfully witnesses its "predicate-ness"; this allows the type of a function which takes higher-order values (like `filter`) to accurately describe its behavior.

Languages with Syntactic Type Tests

The syntactic approach has been used extensively in both industrial and academic type systems over the years.

Untyped languages such as JavaScript seem well-suited for this approach with their C-like style and idiomatic type-tag tests. The industrial languages TypeScript [3], Flow[25], and Hack[25] all use this approach as the means of supporting occurrence typing in their respective systems. Many academic systems for these languages also use this approach: Safe TypeScript[6] and the "flow-typing" approach introduced by Guha et al. [26], both in the context of JavaScript.

Statically typed languages which support type-testing are also amenable to this kind of reasoning. Ceylon[15] and Kotlin[17] are JVM-based statically typed languages which use occurrence typing to statically reason about their respective JVM **`instanceof`** tests, which among other things allow them to statically rule out the ubiquitous null-pointer errors that occur in Java-like languages.

Finally, several advanced type systems have built-in syntactic type-case constructs which the type checker is aware of. The Whiley programming language[16] features untagged union types and type-case constructs the type system is aware of to support occurrence typing (called in their work "flow typing"). Several projects based on semantic subtyping have featured syntactic-based occurrence typing as a means for effectively working with their untagged unions [21, 27], while others have merely support type-based dispatch while introducing a *fresh* name instead of refining the type of an existing name[28]. Generalized algebraic data types[29] also—perhaps surprisingly—admit a form of occurrence typing since in their presence “pattern matching causes type refinement”.

Dependent Types

Dependent types—types which may depend on non-type program terms—are capable of supporting occurrence typing as well. However, this approach is not as common as the syntactic technique and has certain trade-offs. For example, with dependent types the user is able to create their own abstractions which can act as predicates, however this comes at the price of having a more complicated type system for the user to reason about and the language designer to implement.

Our calculus λ_{OT} uses a special dependent function type capable of describing what the result of a function's application entails; these unique function types also appear in Typed Racket [9] and Typed Clojure[30]. Other calculi[31]—such as Dependent JavaScript[4]—instead *heavily* invests in refinement types to support occurrence typing, allowing for a predicate type such as the following:

$$x : \mathit{Any} \rightarrow \{\nu : \texttt{bool} \mid \nu \,?\, (\mathit{tag}(x) = \texttt{"string"}) : (\mathit{tag}(x) \neq \texttt{"string"})\}$$

where the codomain is a boolean with a refinement relating its truth-value to whether the type tag of the argument equals a particular value (in this case `"string"`).

Finally, dependently typed systems supporting a somewhat automated form of language-integrated verification—such as Liquid Haskell[23], F*[22], Sage[32], and others[20, 33, 34, 35]—fundamentally support occurrence typing since they utilize the control flow of the program when attempting to automatically prove the subtyping constraints for the occurrences of the same terms at various points in the program.

Untagged Union Normalization

The final approach we discuss allows the programmer to reason via occurrence typing while indirectly attacking the issue in the type checker. It comes from the insight that any program with untagged unions can be expanded into an equivalent set of possible programs with no untagged unions. For instance, we could expand the aforementioned example function `louder`—whose parameter `s` had type `string | null`—into the following two programs (i.e. one for each possibility in the union type):

```
function louder1(s: string): string {
    if (s) {
        return s.concat("!");
    } else {
        return ""; // dead code
    }
}

function louder2(s: null): string {
    if (s) {
        return s.concat("!"); // dead code
    } else {
        return "";
    }
}
```

If all of the branches in this program are either well-typed or proven unreachable, then the original program (where `s` had type `string | null`) contains no type errors. Note however that there will be an exponentional number of possible programs to type check as the number of variables with union types increases. This “two-phase” approach to supporting occurrence typing has been used thus far to target JavaScript with both simple types [7] and refinement types[36].

CHAPTER 3
OCCURRENCE TYPING WITH REFINEMENT TYPES

Applying a static type discipline to an existing code base written in a dynamically-typed language such as JavaScript, Python, or Racket requires a type system tailored to the idioms of the language. When building gradually–typed systems, designers have focused their attention on type systems with relatively simple goals, e.g. ruling out dynamic type errors such as including a string in an arithmetic computation. These systems—ranging from widely-adopted industrial efforts such as TypeScript [3], Hack [12], and Flow [25] to more academic systems such as Typed Racket [37], Typed Clojure [10], Reticulated Python [13], and Gradualtalk [38]—have been successful in this narrow goal.

However, advanced type systems can express more powerful properties and check more significant invariants than merely the absence of dynamic type errors. Refinement and dependent types, as well as sophisticated encodings in the type systems of languages such as Haskell and ML [39, 40], allow programmers to capture more precise correctness criteria for their programs such as those for balanced binary trees, sized vectors, and much more.

In this chapter, we combine these two strands of research, producing a system we dub *Refinement Typed Racket*, or RTR. RTR follows in the tradition of Dependent ML [41] and Liquid Haskell [23] by supporting dependent and refinement types over a limited but extensible expression language. Experience with these languages has already demonstrated that expressive and rich program properties can be captured by a fully-decidable type system.

Furthermore, by building on the existing framework of *occurrence typing*, refinement types prove to be a natural addition to the implementation, formal model, and soundness results. As we discuss in chapter 2, occurrence typing is designed to reason about dynamic type tests and control flow in existing untyped programs, using propositions about the types of terms and simple rules of logical inference. Extending this logic to encompass refinements of types as well as propositions drawn from solver-backed theories produces an expressive

```
(: max : (-> ([x : Integer] [y : Integer])
             (Refine [z : Integer] (and (>= z x) (>= z y))))
(define (max x y) (if (> x y) x y))
```

Figure 3.1: `max` with refinement types

system which scales to real programs. In this chapter, we show examples drawn from the theory of linear inequalities and the theory of bitvectors.

Figure 3.1 presents a simple demonstration of integrating refinement types with linear arithmetic. The `max` function takes two integers and returns the larger, but instead of describing it as a simple binary operator on values of type `Integer`, as the current Typed Racket implementation specifies, we give a more precise type specifically stating that the result is greater than or equal to both inputs.

The syntax for function types in RTR allows for explicit dependencies between the domain and codomain by giving names to arguments which are in scope in any logical refinements. Note that the `max` function definition does not require any changes to accommodate the stronger type, nor do clients of `max` need to care that the type provides more guarantees than before; the conditional in the body of `max` enables the use of the refinement type in the result, as in most refinement type systems. Occurrence typing's pre-existing ability to reason about conditionals means that abstraction and combination of conditional tests properly works in RTR without requiring anything more from solvers for various theories.

The rest of this chapter is structured as follows: in section 3.1 we give additional examples of taking occurrence typing beyond simple type tests and into the realm of refinement types; in section 3.2 we give a formal model λ_{RTR} which illustrates how to combine occurrence typing and refinement types, proving it sound; in section 3.3 we talk about scaling the ideas found in λ_{RTR} into a full language; in section 3.4 we discuss a case study testing an RTR prototype on more than 56,000 lines of Typed Racket code; in section 3.5 we discuss our experiences adding refinement types to Typed Racket proper; and in section 3.6 we discuss related work.

3.1 Beyond Occurrence Typing

In chapter 2 we laid out a general technique for supporting occurrence typing; in this section we begin to examine ways in which occurrence typing can support more expressive control-flow based reasoning.

3.1.1 Occurrence Typing with Linear Arithmetic

Consider how a standard vector access function `vector-ref` might be implemented in a relatively simply-typed language (e.g. standard Typed Racket). In order to ensure we only access valid indices of the vector, our function must conduct a runtime check before performing the raw, unsafe memory access at the user-specified index:

```
(: vector-ref (∀ (A) (-> (Vectorof A) Integer A)))
  (define (vector-ref v i)
    (if (<= 0 i (sub1 (vector-length v)))
        (unsafe-vector-ref v i)
        (error 'vector-ref "invalid vector index ~a" i)))
```

Although the type for `vector-ref` prevents some runtime errors, invalid indices remain a potential problem. In order to eliminate these, we can extend our new system to consider propositions from the theory of linear integer arithmetic (with a simple implementation of Fourier-Motzkin elimination [42] as a lightweight solver). This allows us to give `<=` a dependent function type where the truth-value of the result reports the intuitively implied linear inequality. We can then design a safe function `safe-vector-ref`:

```
(: safe-vector-ref
   (∀ {A} (-> ([v : (Vectorof A)]
               [i : Integer])
               #:pre (v i) (and (<= 0 i)
                                (< i (vector-length v)))
               A)))
(define safe-vector-ref unsafe-vector-ref)
```

Now the type guarantees only provably valid indices are used. While replacing *all* occurrences of `vector-ref` with `safe-vector-ref` in a program may seem desirable, such a change would likely result is programs that no longer type check! One reason for this is the validity of an index is not always apparent at the actual use site. For example, consider a standard vector dot product function:

```
(: safe-dot-product
  (-> (Vectorof Integer) (Vectorof Integer) Integer))
(define (safe-dot-product A B)
  (let loop ([i : Natural 0])
    (cond
      [(< i (vector-length A)) (+ (* (safe-vector-ref A i)
                                     (safe-vector-ref B i))
                                  (loop (add1 i)))]
        [else 0])))
```

Because there is no explicit knowledge about the length of `B`, our attempt verify one of the indices in `safe-dot-product` will not type check:

```
Type Checker error in (safe-vector-ref B i)

unable to prove precondition: (and (< i (vector-length B)) (<= 0 i))
```

In order to type check `safe-dot-product`, the types for the domain must either be enriched to include the assumption that the vectors are of equal length, or a dynamic check must be added which verifies the assumption at runtime. Also note that without carefully examining the use sites of this function it is difficult to know which solution would be ideal—demanding clients statically verify the property at every call may be an unreasonable requirement. Fortunately a middle ground can be achieved by allowing for both:

```
(: dot-product (-> (Vectorof Integer)
                   (Vectorof Integer)
                   Integer))
(define (dot-product A B)
  (unless (= (vector-length A) (vector-length B))
    (error 'dot-product "invalid vector lengths!"))
  (safe-dot-product A B))
```

Legacy code and clients who cannot easily verify their vectors' lengths may continue to call `dot-product` while clients wishing to statically eliminate this error may call a safe version which uses a stronger type.

Safe vector access is a simple example of the program invariants expressible with occurrence typing extended with the theory of linear integer arithmetic—we have chosen it for thorough examination because it relates directly to our sizable case study on existing Typed Racket code. Xi [43], however, demonstrates at length in the presentation of Dependent ML how the invariants of far richer programs, such as balanced red-black trees and simple type-preserving evaluators, can be expressed using this same class of refinements.

3.1.2 Occurrence Typing with Bitvectors

Linear arithmetic, however, is merely one example of extending RTR with an external theory. To illustrate, we additionally experimented by adding the theory of bitvectors to RTR. By leveraging Z3's bitvector reasoning [44] we were able to type check the helper function `xtime` found in many implementations of AES [45] encryption. This function computes the result of multiplying the elements of the field $\mathbb{F}_{2^8}$ by x (i.e. polynomials of the form $\mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$, which AES conveniently represents using a byte):

```
(: xtime (-> Byte Byte))
(define (xtime num)
  (let ([n (bitwise-and (* #x02 num) #xff)])
    (cond
      [(= #x00 (bitwise-and num #x80)) n]
      [else (bitwise-xor n #x1b)])))
```

In this example the type `Byte` is a shorthand for the type `(Refine [b : BitVector] (<= #x00 b #xff))`. To verify this program, we enrich the types of the relevant bitwise operations (e.g. `=`, `bitwise-and`, etc.) to include propositions and refinements relating the values to bitvector-theoretic statements and add bitvector literals to the set of terms which may be lifted to the type level. Adding the theory of bitvectors and verifying this program proved to be a relatively straightforward process; in section 3.2.4 we discuss in detail our general strategy for adding new theories to RTR.

3.2 Formal Model

Our base system λ_{RTR} is a natural extension of λ_{OT}; new language forms and judgments are highlighted.

The typing judgment for λ_{RTR} resembles a standard typing judgment except that instead of assigning types, it assigns *type-results* to well-typed expressions:

$$\Gamma \vdash e : \langle \tau, p, q, o \rangle$$

This judgment states that in environment Γ

- e has type τ;
- if e evaluates to a non-`false` (i.e. treated as true) value, 'then proposition' p holds;
- if e evaluates to `false`, 'else proposition' q holds;
- e's value corresponds to the symbolic object o.

3.2.1 Syntax

The syntax of terms, types, propositions, and other forms are given in figure 3.2.

Expressions. λ_{RTR} uses a standard set of expressions with explicit pair operations for simplicity (so our presentation may omit polymorphism).

Types. The universal 'top' type `Any` is the type which describes all well typed terms. `Int` is the type of integers, while `True` and `False` are the types of the boolean values `true` and `false`. Pair types are written $\tau \times \sigma$. $(\bigcup \overrightarrow{\tau})$ describes a 'true' (i.e. untagged) union of its components. For convenience we write the boolean type $(\bigcup$ `True False`$)$ as `Bool` and the uninhabited 'bottom' type $(\bigcup)$ as `Empty`. Function types consist of a named argument x, a domain type τ, and range type-result R in which x is bound. $\{x\!:\!\tau \mid p\}$ is a standard refinement type, describing any value x of type τ for which proposition p holds.

Propositions. At our system's core is a propositional logic with domain specific features. $\mathbb{tt}$ and $\mathbb{ff}$ are the trivial and absurd propositions, while $p \wedge q$ and $p \vee q$ represent the conjunction and disjunction of propositions p and q respectively. Type information is expressed by propositions of the form $o \in \tau$ or $o \notin \tau$, which state that symbolic object o

$i ::=$	$1 \mid 2$	**Field Indices**
$c ::=$		**Constants**
	$\mid$ *int*	integer value
	$\mid$ `true`	true value
	$\mid$ `false`	false value
	$\mid$ *uop*	primitive ops
$e ::=$		**Expressions**
	$\mid x, y, z$	variables
	$\mid c$	constant values
	$\mid (\lambda (x \mathbin{:} \tau)\, e)$	abstraction
	$\mid (e\ e)$	application
	$\mid (\texttt{if}\ e\ e\ e)$	conditional
	$\mid (\texttt{let}\ (x\ e)\ e)$	local binding
	$\mid (\texttt{pair}\ e\ e)$	pair
	$\mid (\texttt{proj}\ i\ e)$	field access
$\tau, \sigma ::=$		**Types**
	$\mid$ `Any`	universal type
	$\mid$ `Int`	integer type
	$\mid$ `True`	true types
	$\mid$ `False`	false type
	$\mid \tau \times \tau$	product type
	$\mid (\bigcup \vec{\tau})$	ad-hoc union type
	$\mid (x \mathbin{:} \tau) \rightarrow R$	function type
	$\mid \{x \mathbin{:} \tau \mid p\}$	refinement type

$v ::=$		**Values**
	$\mid c$	constant values
	$\mid (\texttt{pair}\ v\ v)$	pair value
	$\mid [\rho,\ (\lambda (x \mathbin{:} \tau)\, e)]$	closure
$p, q ::=$		**Propositions**
	$\mid \mathbb{tt}$	trivial prop
	$\mid \mathbb{ff}$	absurd prop
	$\mid o \in \tau$	o is of type τ
	$\mid o \notin \tau$	o is not of type τ
	$\mid p \wedge p$	conjunction
	$\mid p \vee p$	disjunction
	$\mid o \equiv o$	object aliasing
	$\mid \mathcal{X}^{\mathcal{T}}$	theory $\mathcal{T}$ prop
$o ::=$		**Symbolic Objects**
	$\mid \top^{o}$	null object
	$\mid x$	variable reference
	$\mid (\mathit{proj}\ i\ o)$	field reference
	$\mid (\mathit{pair}\ o\ o)$	object pair
$R ::=$		**Type-Results**
	$\mid \langle \tau, p, q, o \rangle$	type-result
	$\mid \exists (x \mathbin{:} \tau).\mathrm{R}$	existential
$\Gamma ::=$	$\vec{p}$	**Type Env**
$\rho ::=$	$\overrightarrow{x \mapsto v}$	**Runtime Env**

Figure 3.2: λ_{RTR} Syntax

is or is not of type τ respectively. $o_1 \equiv o_2$ describes an 'alias' between symbolic objects, stating that the object o_1 points to the same value as o_2. Finally, an atomic propositions of the form $\mathcal{X}^{\mathcal{T}}$ represents a statement from a theory $\mathcal{T}$ for which λ_{RTR} has been provided a sound solver. In this way our logic describes an extensible system that can be enriched with various theories according to the needs of the application at hand.

Fields. A field allows us to reference a subcomponent of a structural value. For example, if p is a tree-like structure built using nested pairs, $(proj\ 1\ (proj\ 2\ o))$ would describe the value found by accessing the first field of the result of accessing o's second field. In this model having accessors for pair fields suffices; in general, accessors/descriptors for both built-in and user-defined data types are needed in order to type check more complicated real-world programs. Our vector case study, for example, required an accessor for `vector-length`.

Symbolic Objects. Instead of allowing our types to depend on *arbitrary* program terms (as is done in systems with full dependent types), we define a canonical subset of terms called *symbolic objects* which represent the terms which may be lifted to the type level in our system. These objects act as a conservative 'whitelist' of sorts, allowing our type system to work in a full-scale programming language by only considering obviously safe terms (i.e. excluding mutated fields, potentially non-deterministic functions, etc.).

Initially these objects are only used to describe values bound to variables, field accesses, and pairs of objects, while the null symbolic object $\top^o$ represents terms we do not lift to the type level. These objects (excluding pairs) are what allows standard Typed Racket to type check many dynamic programming idioms. When extending RTR to handle additional theories, the grammar of symbolic objects is extended to include program terms the new theory must reason about.

Finally, when performing standard capture-avoiding substitution we keep symbolic objects in the obvious normal form (e.g. $(proj\ 1\ (pair\ x\ y))$ is reduced to x). Propositions that end up directly referring to $\top^o$, such as $\top^o \in$ `Int`, are treated as equivalent to $\mathbb{tt}$ (i.e. meaningless) and are discarded.

Type-Results. In order to allow our system to easily reason about more than the just the simple type τ of an expression, we assign a well typed expression a type-result. In addition to describing an expression's type, a type-result further informs the system by explicitly

$\boxed{\Delta : uop \to \tau}$

$$
\begin{array}{ll}
\Delta(\texttt{not}) & = (x{:}\texttt{Any}) \to \langle \texttt{Bool}, x \in \texttt{False}, x \notin \texttt{False}, \top^{o} \rangle \\
\Delta(\texttt{zero?}) & = (x{:}\texttt{Int}) \to \langle \texttt{Bool}, \mathbb{tt}, \mathbb{tt}, \top^{o} \rangle \\
\Delta(\texttt{sub1}) & = (x{:}\texttt{Int}) \to \langle \texttt{Int}, \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle \\
\Delta(\texttt{add1}) & = (x{:}\texttt{Int}) \to \langle \texttt{Int}, \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle \\
\Delta(\texttt{int?}) & = (x{:}\texttt{Any}) \to \langle \texttt{Bool}, x \in \texttt{Int}, x \notin \texttt{Int}, \top^{o} \rangle \\
\Delta(\texttt{bool?}) & = (x{:}\texttt{Any}) \to \langle \texttt{Bool}, x \in \texttt{Bool}, x \notin \texttt{Bool}, \top^{o} \rangle \\
\Delta(\texttt{pair?}) & = (x{:}\texttt{Any}) \to \langle \texttt{Bool}, x \in \texttt{Any} \times \texttt{Any}, x \notin \texttt{Any} \times \texttt{Any}, \top^{o} \rangle
\end{array}
$$

Figure 3.3: λ_{RTR} Primitive Types

capturing two additional properties: (1) what is learned when the expression's value is used as the test-expression in a conditional—this is described by the pair of propositions $p_+|p_-$ in the type-result—and (2) which symbolic object o the expression's value corresponds to.

Existentially quantified type-results allow types to depend on terms with no in-scope symbolic object. Our usage of existential quantification resembles the technique introduced by Knowles and Flanagan [33] in many ways, except that our usage is restricted to when substitution is simply not possible (i.e. when the variable's assigned expression has a null object).

Environments. For simplicity in this model we use an environment built entirely of propositions. In a real implementation it is useful to separate the environment into two portions: a traditional mapping of variables to types along with a set of currently known propositions. The latter can then be used to refine the former during type checking.

Runtime Environments. Our runtime environments are standard mappings of variables to closed runtime values, appearing in closures and our big-step reduction semantics.

3.2.2 Typing Rules

The typing judgment is defined in figure 3.4 and an executable PLT Redex [46] model is included in our accompanying artifact. The individual rules are those previously used by Typed Racket with only a few minor modifications to incorporate our new forms (i.e. existential type-results and aliases).

T-Int, T-True, T-False, and T-Prim are used for type checking the respective base

$$\boxed{\Gamma \vdash e : \mathrm{R}}$$

$$\textsc{T-Int}\qquad \Gamma \vdash \mathit{int} : \langle \mathsf{Int}, \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle$$

$$\textsc{T-True}\qquad \Gamma \vdash \mathtt{true} : \langle \mathsf{True}, \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle$$

$$\textsc{T-False}\qquad \Gamma \vdash \mathtt{false} : \langle \mathsf{False}, \mathbb{ff}, \mathbb{tt}, \top^{o} \rangle$$

$$\textsc{T-Prim}\qquad \Gamma \vdash \mathit{uop} : \langle \Delta(\mathit{uop}), \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle$$

$$\textsc{T-Abs}\qquad \frac{\Gamma, x \in \tau \vdash e : \mathrm{R}}{\Gamma \vdash (\lambda(x\!:\!\tau)\, e) : \langle (x\!:\!\tau) \to \mathrm{R}, \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle}$$

$$\textsc{T-Var}\qquad \frac{\Gamma \vdash x \in \tau}{\Gamma \vdash x : \langle \tau, x \notin \mathsf{False}, x \in \mathsf{False}, x \rangle}$$

$$\textsc{T-If}\qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \langle \mathsf{Any}, p_{1+}, p_{1-}, \top^{o} \rangle \\ \Gamma, p_{1+} \vdash e_2 : \mathrm{R} \qquad \Gamma, p_{1-} \vdash e_3 : \mathrm{R}\end{array}}{\Gamma \vdash (\mathtt{if}\ e_1\ e_2\ e_3) : \mathrm{R}}$$

$$\textsc{T-Subsume}\qquad \frac{\begin{array}{c}\Gamma \vdash e : \mathrm{R}' \\ \Gamma \vdash \mathrm{R}' <: \mathrm{R}\end{array}}{\Gamma \vdash e : \mathrm{R}}$$

$$\textsc{T-Let}\qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \langle \tau_1, p_1, q_1, o_1 \rangle \\ p_x = (x \notin \mathsf{False} \wedge p_1) \vee (x \in \mathsf{False} \wedge q_1) \\ \Gamma, x \in \tau_1, x \equiv o_1, p_x \vdash e : \mathrm{R}_2\end{array}}{\Gamma \vdash (\mathtt{let}\,(x\ e_1)\ e_2) : \mathrm{R}_2[x \overset{\tau_1}{\Longmapsto} o_1]}$$

$$\textsc{T-App}\qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \langle (x\!:\!\tau) \to \mathrm{R}, \mathbb{tt}, \mathbb{tt}, \top^{o} \rangle \\ \Gamma \vdash e_2 : \langle \sigma, \mathbb{tt}, \mathbb{tt}, o_2 \rangle \qquad \Gamma \vdash \sigma <: \tau\end{array}}{\Gamma \vdash (e_1\ e_2) : \mathrm{R}[x \overset{\sigma}{\Longmapsto} o_2]}$$

$$\textsc{T-Pair}\qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \langle \tau_1, \mathbb{tt}, \mathbb{tt}, o_1 \rangle \\ \Gamma \vdash e_2 : \langle \tau_2, \mathbb{tt}, \mathbb{tt}, o_2 \rangle \\ R = \langle \tau_1 \times \tau_2, \mathbb{tt}, \mathbb{ff}, (\mathit{pair}\ x_1\ x_2) \rangle\end{array}}{\Gamma \vdash (\mathtt{pair}\ e_1\ e_2) : \mathrm{R}[x_1 \overset{\tau_1}{\Longmapsto} o_1][x_2 \overset{\tau_2}{\Longmapsto} o_2]}$$

$$\textsc{T-Proj}\qquad \frac{\begin{array}{c}\Gamma \vdash e : \langle \tau_1 \times \tau_2, \mathbb{tt}, \mathbb{tt}, o \rangle \\ R = \langle \tau_i, \mathbb{tt}, \mathbb{tt}, (\mathit{proj}\ i\ x) \rangle\end{array}}{\Gamma \vdash (\mathtt{proj}\ i\ e) : \mathrm{R}[x \overset{\tau_i}{\Longmapsto} o]}$$

Figure 3.4: λ_{RTR} Typing Judgment

values, with T-Prim consulting the Δ metafunction described in figure 3.3 for primitive operators. Note that the then- and else-propositions are consistent with their being `false` or non-`false`. Additionally, by default none of these terms will appear in types and propositions, as signified by the null symbolic object $\top^o$.

T-Var may assign any type τ to variable x so long as the system can derive $\Gamma \vdash x \in \tau$. The then- and else-propositions reflect the self evident fact that if x is found to equal `false` then x is of type `False`, otherwise x is not of type `False`. The symbolic object informs the type system that this expression corresponds to the program term x.

T-Abs, the rule for checking lambda abstractions, checks the body of the abstraction in the extended environment which maps x to τ. We use the standard convention of choosing fresh names not currently bound when extending Γ with new bindings. The type-result from checking the body is then used as the range for the function type, and the then- and else-propositions report the non-falseness of the value.

T-App handles function application, first checking that e_1 and e_2 are well-typed individually and then ensuring the type of e_2 is a subtype of the domain of e_1. The overall type-result of the application is the range of the function, R, with the symbolic object of the operand, o_2, lifted and optionally substituted for x. This *lifting substitution* is defined as follows:

$$\boxed{\begin{aligned} R[x \overset{\tau}{\Mapsto} \top^o] &= \exists(x\!:\!\tau).\mathrm{R} \\ R[x \overset{\tau}{\Mapsto} o] &= R[x \mapsto o] \end{aligned}}$$

In essence, if the operand corresponds to a value our type system can reason directly about (i.e. its object is non-null), we perform capture avoiding substitution as expected. Otherwise, an existential quantifier à la Knowles and Flanagan [33] is used to capture the argument expression's *precise* type, even though it's exact identity is unknown; this enables the function's range to depend on its argument regardless of whether the term can soundly be lifted to the type level.

T-If is used for conditionals, describing the important process by which information learned from test-expressions is projected into the respective branches. After ensuring e_1 is well-typed at some type, we make note of the then- and else-propositions p_{1+} and p_{1-}.

We then extend the environment with the appropriate proposition, dependent upon which branch we are checking: p_{1+} is assumed while checking the then-branch and p_{1-} for the else-branch. The type result of a conditional is simply the type result implied by both branches.

T-Let first checks whether the expression e_1 being bound to x is well typed. When checking the body, the environment is extended with the type for x, a proposition describing x's then- and else- propositions, and an alias stating that x refers to o_1 (i.e. the symbolic object of e_1). Since x is unbound outside the body, we perform a lifting substitution of o_1 for x on the result as we do with function application.

In order to omit polymorphism we use explicit pair introduction and elimination rules. T-Cons introduces pairs, first checking the types and symbolic objects for e_1 and e_2. The type-result then includes the product of these individual types, propositions reflecting the non-`false` nature of the value, and a symbolic pair object (all modulo the two lifting substitutions). Pair elimination forms are checked with T-Proj, which ensure its argument is indeed a pair before returning the expected type and a symbolic object describing which field was accessed.

3.2.3 Subtyping and Proof System

The subtyping and proof system use a combination of familiar rules from type theory and formal logic.

Subtyping

Figure 3.5 describes the subtyping relation <: for types, symbolic objects, and type-results.

For objects, the null object $\top^o$ is the top object and objects are sub-objects of any alias-equivalent object. Pair objects are sub-objects in a pointwise fashion.

All types are subtypes of themselves and of the top type `Any`. A type is a subtype of a union if it is a subtype of any element of the union. Unions are only subtypes of a type if *every* member of the union is a subtype of that type. Function subtyping has the standard contra- and co-variance in the domain and range; in order to reason correctly about dependencies when checking the range, the environment is extended to assign x the

$\boxed{\Gamma \vdash o <: o}$

SO-EQUIV
$$\frac{\Gamma \vdash o_1 \equiv o_2}{\Gamma \vdash o_1 <: o_2}$$

SO-NULL
$$\Gamma \vdash o <: \top^o$$

SO-PAIR
$$\frac{\Gamma \vdash o_1 <: o_3 \qquad \Gamma \vdash o_2 <: o_4}{\Gamma \vdash (pair\ o_1\ o_2) <: (pair\ o_3\ o_4)}$$

$\boxed{\Gamma \vdash \tau <: \tau}$

S-REFL
$$\Gamma \vdash \tau <: \tau$$

S-TOP
$$\Gamma \vdash \tau <: \texttt{Any}$$

S-UNION1
$$\frac{\forall \tau \text{ in } \vec{\tau}.\ \Gamma \vdash \tau <: \sigma}{\Gamma \vdash (\bigcup \vec{\tau}) <: \sigma}$$

S-UNION2
$$\frac{\exists \sigma \text{ in } \vec{\sigma}.\ \Gamma \vdash \tau <: \sigma}{\Gamma \vdash \tau <: (\bigcup \vec{\sigma})}$$

S-PAIR
$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 <: \tau_2 \\ \Gamma \vdash \sigma_1 <: \sigma_2\end{array}}{\Gamma \vdash \tau_1 \times \sigma_1 <: \tau_2 \times \sigma_2}$$

S-FUN
$$\frac{\begin{array}{c}x \notin \mathsf{fvs}(\Gamma) \qquad \Gamma \vdash \tau_2 <: \tau_1 \\ \Gamma, x \in \tau_2 \vdash \mathrm{R}_1 <: \mathrm{R}_2\end{array}}{\Gamma \vdash (x{:}\tau_1) \to \mathrm{R}_1 <: (x{:}\tau_2) \to \mathrm{R}_2}$$

S-WEAKEN
$$\frac{\Gamma \vdash \tau <: \sigma}{\Gamma \vdash \{x{:}\tau \mid p\} <: \sigma}$$

S-REFINE1
$$\frac{\begin{array}{c}x \notin \mathsf{fvs}(\Gamma) \\ \Gamma, x \in \tau, p \vdash x \in \sigma\end{array}}{\Gamma \vdash \{x{:}\tau \mid p\} <: \sigma}$$

S-REFINE2
$$\frac{\begin{array}{c}x \notin \mathsf{fvs}(\Gamma) \\ \Gamma \vdash \tau <: \sigma \qquad \Gamma, x \in \tau \vdash p\end{array}}{\Gamma \vdash \tau <: \{x{:}\sigma \mid p\}}$$

$\boxed{\Gamma \vdash \mathrm{R} <: \mathrm{R}}$

SR-RESULT
$$\frac{\begin{array}{cc}\Gamma \vdash \tau_1 <: \tau_2 & \Gamma, p_{1+} \vdash p_{2+} \\ \Gamma \vdash o_1 <: o_2 & \Gamma, p_{1-} \vdash p_{2-}\end{array}}{\Gamma \vdash \langle \tau_1, p_{1+}, p_{1-}, o_1 \rangle <: \langle \tau_2, p_{2+}, p_{2-}, o_2 \rangle}$$

SR-EXISTS
$$\frac{\begin{array}{c}x \notin \mathsf{fvs}(\Gamma) \\ \Gamma, x \in \tau \vdash \mathrm{R}_1 <: \mathrm{R}_2\end{array}}{\Gamma \vdash \exists (x{:}\tau).\mathrm{R} <: \mathrm{R}_2}$$

Figure 3.5: λ_{RTR} Subtyping

$\boxed{\Gamma \vdash p}$

$$\textsc{L-Atom}\quad \Gamma, p \vdash p \qquad \textsc{L-Trivial}\quad \Gamma \vdash \mathbb{tt} \qquad \textsc{L-Absurd}\ \frac{\Gamma \vdash \mathbb{ff}}{\Gamma \vdash p} \qquad \textsc{L-AndI}\ \frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2} \qquad \textsc{L-AndE1}\ \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_1} \qquad \textsc{L-AndE2}\ \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_2}$$

$$\textsc{L-OrI}\ \frac{\Gamma \vdash p_1 \text{ or } \Gamma \vdash p_2}{\Gamma \vdash p_1 \vee p_2} \qquad \textsc{L-OrE}\ \frac{\begin{array}{c}\Gamma \vdash p_1 \vee p_2\\ \Gamma, p_1 \vdash p \qquad \Gamma, p_2 \vdash p\end{array}}{\Gamma \vdash p} \qquad \textsc{L-Sub}\ \frac{\begin{array}{c}\Gamma \vdash o \in \sigma\\ \Gamma \vdash \sigma <: \tau\end{array}}{\Gamma \vdash o \in \tau} \qquad \textsc{L-Not}\ \frac{\Gamma, o \in \tau \vdash \mathbb{ff}}{\Gamma \vdash o \notin \tau} \qquad \textsc{L-Bot}\ \frac{\Gamma \vdash o \in \texttt{Empty}}{\Gamma \vdash p}$$

$$\textsc{L-Refl}\quad \Gamma \vdash o \equiv o \qquad \textsc{L-Sym}\ \frac{\Gamma \vdash o_2 \equiv o_1}{\Gamma \vdash o_1 \equiv o_2} \qquad \textsc{L-Update+}\ \frac{\begin{array}{c}\Gamma \vdash o \in \tau\\ \Gamma \vdash (proj\ \vec{i}\ o) \in \sigma\end{array}}{\Gamma \vdash o \in \mathsf{update}^{+}_{\Gamma}(\tau, \vec{i}, \sigma)} \qquad \textsc{L-Update}{-}\ \frac{\begin{array}{c}\Gamma \vdash o \in \tau\\ \Gamma \vdash (proj\ \vec{i}\ o) \notin \sigma\end{array}}{\Gamma \vdash o \in \mathsf{update}^{-}_{\Gamma}(\tau, \vec{i}, \sigma)}$$

$$\textsc{L-Transport}\ \frac{\Gamma \vdash p(o_1) \qquad \Gamma \vdash o_1 \equiv o_2}{\Gamma \vdash p(o_2)} \qquad \textsc{L-Theory}\ \frac{[\![\Gamma]\!]_{\mathcal{T}} \vdash_{\mathcal{T}} \mathcal{X}^{\mathcal{T}}}{\Gamma \vdash \mathcal{X}^{\mathcal{T}}} \qquad \textsc{L-TypeFork}\ \frac{\Gamma \vdash (pair\ o_1\ o_2) \in \tau_1 \times \tau_2}{\Gamma \vdash o_1 \in \tau_1 \wedge o_2 \in \tau_2}$$

$$\textsc{L-ObjFork}\ \frac{\Gamma \vdash (pair\ o_1\ o_2) \equiv (pair\ o_3\ o_4)}{\Gamma \vdash o_1 \equiv o_3 \wedge o_2 \equiv o_4} \qquad \textsc{L-RefI}\ \frac{\begin{array}{c}\Gamma \vdash o \in \tau\\ \Gamma \vdash p[x \mapsto o]\end{array}}{\Gamma \vdash o \in \{x{:}\tau \mid p\}} \qquad \textsc{L-RefE}\ \frac{\Gamma \vdash o \in \{x{:}\tau \mid p\}}{\Gamma \vdash o \in \tau \wedge p[x \mapsto o]}$$

Figure 3.6: λ_{RTR}-specific Logic Rules

more specific domain type. Pair subtyping is standard.

For refinement types we have three rules: S-Weaken states if τ is a subtype of σ in Γ then so is any refinement of τ; S-Refine1 and S-Refine2 allow subtyping inquiries about refinements to be translated into their equivalent logical inquiries.

The subtyping relation for type-results relies on subtyping for the type and object, and logical implication for the then- and else-propositions. Since existentially quantified type-results are only used as a tool for type checking, there is only one explicit subtyping rule for them: SR-Exists. This rule resembles the standard existential instantiation rule from first order logic, stating an existentially quantified type-result is a subtype of another type result if the subtyping relation holds in the appropriately extended environment.

Proof System

Figure 3.6 describes the type-specific portion of the propositional logic for λ_{RTR}. We omit the introduction and elimination rules for forms from propositional logic, since they are identical to those used by λ_{TR} [9] (i.e. resembling those found in any natural deduction system).

L-Sub says an object o is of type τ when it is a known subtype of τ. L-Not conversely lets us prove object o is not of type τ when assuming the opposite implies a contradiction. L-Bot serves as an '*ex falso quodlibet*' of sorts, allowing us to draw any conclusion since `Empty` is uninhabited.

Object aliasing allows us to reason about the statically known equivalences classes of symbolic objects. L-Refl and L-Sym provide reflexivity and symmetry for aliasing, while L-Transport allows us replace alias-equivalent objects in any derivable proposition (giving us transitivity). L-ObjFork and L-TypeFork provide a means for reducing claims about object pairs to be claims about their fields.

L-Update+ and L-Update– play a key role in our system, allowing positive and negative type statements to refine the known types of objects. Roughly speaking, if we know an object o is of type τ, updating some field $(proj\ i_n\ (...\ (proj\ i_0\ o)))$ within the object (abbreviated $(proj\ \vec{i}\ o)$) with additional information computes the following: if we know $(proj\ \vec{i}\ o) \in \sigma$—that the field *is* of type σ—we update that field's type τ' to be approximately $\tau' \cap \sigma$ (i.e. a conservative 'intersection' of the two types); conversely, updating a field's type τ' with the knowledge that the field *is not* σ updates the field to be approximately $\tau' - \sigma$ (i.e. the 'difference' between the two). A full definition of `update` is given in figure 3.7.

L-RefI and L-RefE construct and eliminate refinement types in the expected ways, essentially saying that the proposition $o \in \{x : \tau \mid p\}$ is equivalent to the compound proposition $o \in \tau \wedge p[x \mapsto o]$.

Finally, a proposition $\mathcal{X}^{\mathcal{T}}$ from theory $\mathcal{T}$ is derived using L-Theory. This rule consults a solver for theory $\mathcal{T}$ with the relevant knowledge from Γ.

$$\boxed{\mathsf{update} : \pm\ \Gamma\ \tau\ \vec{i}\ \tau \to \tau}$$

$$\begin{aligned}
\mathsf{update}^{\pm}_{\Gamma}(\tau_1 \times \tau_2, \vec{i}{::}1, \sigma) &= \mathsf{update}^{\pm}_{\Gamma}(\tau_1, \vec{i}, \sigma) \times \tau_2 \\
\mathsf{update}^{\pm}_{\Gamma}(\tau_1 \times \tau_2, \vec{i}{::}2, \sigma) &= \tau_1 \times \mathsf{update}^{\pm}_{\Gamma}(\tau_2, \vec{i}, \sigma) \\
\mathsf{update}^{+}_{\Gamma}(\tau, \epsilon, \sigma) &= \mathsf{restrict}_{\Gamma}(\tau, \sigma) \\
\mathsf{update}^{-}_{\Gamma}(\tau, \epsilon, \sigma) &= \mathsf{remove}_{\Gamma}(\tau, \sigma) \\
\mathsf{update}^{\pm}_{\Gamma}((\bigcup \vec{\tau}), \vec{i}, \sigma) &= (\bigcup \overrightarrow{\mathsf{update}^{\pm}_{\Gamma}(\tau, \vec{i}, \sigma)})
\end{aligned}$$

$$\boxed{\mathsf{restrict} : \Gamma\ \tau\ \tau \to \tau}$$

$$\begin{aligned}
\mathsf{restrict}_{\Gamma}(\tau, \sigma) &= \texttt{Empty} \quad \text{if } \nexists v. \Gamma \vdash v : \tau \text{ and } \Gamma \vdash v : \sigma \\
\mathsf{restrict}_{\Gamma}((\bigcup \vec{\tau}), \sigma) &= (\bigcup \overrightarrow{\mathsf{restrict}_{\Gamma}(\tau, \sigma)}) \\
\mathsf{restrict}_{\Gamma}(\{x{:}\tau \mid p\}, \sigma) &= \{x{:}\mathsf{restrict}_{\Gamma}(\tau, \sigma) \mid p\} \\
\mathsf{restrict}_{\Gamma}(\tau, \sigma) &= \tau \qquad \text{if } \Gamma \vdash \tau <: \sigma \\
\mathsf{restrict}_{\Gamma}(\tau, \sigma) &= \sigma \qquad \text{otherwise}
\end{aligned}$$

$$\boxed{\mathsf{remove} : \Gamma\ \tau\ \tau \to \tau}$$

$$\begin{aligned}
\mathsf{remove}_{\Gamma}(\tau, \sigma) &= \texttt{Empty} \quad \text{if } \Gamma \vdash \tau <: \sigma \\
\mathsf{remove}_{\Gamma}((\bigcup \vec{\tau}), \sigma) &= (\bigcup \overrightarrow{\mathsf{remove}_{\Gamma}(\tau, \sigma)}) \\
\mathsf{remove}_{\Gamma}(\{x{:}\tau \mid p\}, \sigma) &= \{x{:}\mathsf{remove}_{\Gamma}(\tau, \sigma) \mid p\} \\
\mathsf{remove}_{\Gamma}(\tau, \sigma) &= \tau \qquad \text{otherwise}
\end{aligned}$$

Figure 3.7: λ_{RTR} type-update metafunction

3.2.4 Integrating Additional Theories

Our system is designed in an extensible fashion, allowing an arbitrary external theory to be added so long as a theory-specific solver is provided. To illustrate, we discuss the linear arithmetic extension we implemented in a Typed Racket prototype in order to perform our vector-related case study.

To add this theory, we first must identify the canonical set of program terms which appear in the theory's sentences. For our case study this included integer arithmetic expressions of the form $a_0x_0 + a_1x_1 + ... + a_nx_n$ (i.e. linear combinations over $\mathbb{Z}$) and a field which describes a vector's length. We can extend the grammar of fields and symbolic objects to naturally include these terms:

$$i ::= ... \mid \mathsf{len}$$

$$o ::= ... \mid n \mid n \cdot o \mid o + o$$

Now our type system and logic can reason directly about the terms our theory discusses.

We then identify the theory-relevant *predicates* and extend our grammar of propositions to include them:

$$\chi^{LI} ::= o < o \mid o \leq o$$

Finally, the types of some language primitives must be enriched so these newly added forms are emitted during type checking. For example, we must modify the typing judgment for integer literals to include the precise symbolic object:

$$\textsc{T-Int}$$

$$\Gamma \vdash n : \langle \texttt{Int}, \mathbb{tt}, \mathbb{ff}, n \rangle$$

Similarly, primitive functions which perform arithmetic computation, arithmetic comparison, and report a vector's length must be updated to return the appropriate propositions and symbolic objects (similar to how `int?` and (`proj` i e) are handled in our presentation of λ_{RTR}).

$$\boxed{\rho \vdash e \Downarrow v}$$

$$\frac{}{\rho \vdash v \Downarrow v}\ \textsc{B-Val} \qquad \frac{\rho(x) = v}{\rho \vdash x \Downarrow v}\ \textsc{B-Var} \qquad \frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho[x := v_1] \vdash e_2 \Downarrow v}{\rho \vdash (\texttt{let}\,(x\ e_1)\ e_2) \Downarrow v}\ \textsc{B-Let}$$

$$\frac{}{\rho \vdash (\lambda(x{:}\tau)\, e) \Downarrow [\rho,\ (\lambda(x{:}\tau)\, e)]}\ \textsc{B-Abs}$$

$$\frac{\rho \vdash e \Downarrow (\texttt{pair}\ v_1\ v_2)}{\rho \vdash (\texttt{proj}\ i\ e) \Downarrow v_i}\ \textsc{B-Proj} \qquad \frac{\rho \vdash e_1 \Downarrow v_1 \qquad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash (\texttt{pair}\ e_1\ e_2) \Downarrow (\texttt{pair}\ v_1\ v_2)}\ \textsc{B-Pair}$$

$$\frac{\rho \vdash e_1 \Downarrow [\rho_c, (\lambda(x{:}\tau)\, e)] \quad \rho \vdash e_2 \Downarrow v_2 \quad \rho_c[x := v_2] \vdash e \Downarrow v}{\rho \vdash (e_1\ e_2) \Downarrow v}\ \textsc{B-Beta}$$

$$\frac{\rho \vdash e_1 \Downarrow uop \quad \rho \vdash e_2 \Downarrow v_2 \quad \delta(uop,\ v_2) = v}{\rho \vdash (e_1\ e_2) \Downarrow v}\ \textsc{B-Prim}$$

$$\frac{\rho \vdash e_1 \Downarrow v_1 \quad v_1 \neq \texttt{false} \quad \rho \vdash e_2 \Downarrow v}{\rho \vdash (\texttt{if}\ e_1\ e_2\ e_3) \Downarrow v}\ \textsc{B-IfTrue} \qquad \frac{\rho \vdash e_1 \Downarrow \texttt{false} \quad \rho \vdash e_3 \Downarrow v}{\rho \vdash (\texttt{if}\ e_1\ e_2\ e_3) \Downarrow v}\ \textsc{B-IfFalse}$$

Figure 3.8: λ_{RTR} Big-step Reduction Relation

With these additions in place, a simple function which converts linear integer propositions into solver-compatible assertions allows our system to begin type checking programs with these theory-specific types.

3.2.5 Semantics and Soundness

λ_{RTR} uses the big-step reduction semantics described in figure 3.8, which notably treats all non-`false` values as 'true' for the purposes of conditional test-expressions. The evaluation judgment $\rho \vdash e \Downarrow v$ states that in runtime-environment ρ, expression e evaluates to the value v. A model-theoretic satisfaction relation is used to prove type soundness, just as in prior work on occurrence typing [9].

Models

Because our formalism is described as a type-theory aware logic, it is convenient to examine its soundness using a model-theoretic approach commonly used in proof theory. For λ_{RTR} a model is any runtime-value environment ρ and is said to *satisfy* a proposition p (written $\rho \vDash p$) when its assignment of values to the free variables of p make the proposition a

$$\boxed{\delta : uop\ v \rightarrow v}$$

$$\begin{aligned}
\delta(\texttt{not}, v) &= \begin{cases} \texttt{true} & \text{if } v = \texttt{false} \\ \texttt{false} & \text{otherwise} \end{cases} \\
\delta(\texttt{zero?}, int) &= \begin{cases} \texttt{true} & \text{if } int = 0 \\ \texttt{false} & \text{otherwise} \end{cases} \\
\delta(\texttt{sub1}, int) &= int - 1 \\
\delta(\texttt{add1}, int) &= int + 1 \\
\delta(\texttt{int?}, v) &= \begin{cases} \texttt{true} & \text{if } v \text{ is an integer} \\ \texttt{false} & \text{otherwise} \end{cases} \\
\delta(\texttt{bool?}, v) &= \begin{cases} \texttt{true} & \text{if } v \text{ is a boolean} \\ \texttt{false} & \text{otherwise} \end{cases} \\
\delta(\texttt{pair?}, v) &= \begin{cases} \texttt{true} & \text{if } v \text{ is a pair} \\ \texttt{false} & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3.9: λ_{RTR} Primitive Semantics

$$\boxed{\rho \vDash p}$$

$$\textsc{M-Top}\quad \rho \vDash \mathbb{tt} \qquad \textsc{M-Or}\quad \frac{\rho \vDash p_1 \ \text{ or } \ \rho \vDash p_2}{\rho \vDash p_1 \vee p_2} \qquad \textsc{M-And}\quad \frac{\rho \vDash p_1 \qquad \rho \vDash p_2}{\rho \vDash p_1 \wedge p_2} \qquad \textsc{M-Alias}\quad \frac{\rho(o_1) = \rho(o_2)}{\rho \vDash o_1 \equiv o_2}$$

$$\textsc{M-Refine}\quad \frac{\rho \vDash o \in \tau \qquad \rho \vDash p[x \mapsto o]}{\rho \vDash o \in \{x{:}\tau \mid p\}} \qquad \textsc{M-RefineNot1}\quad \frac{\rho \vDash o \notin \tau}{\rho \vDash o \notin \{x{:}\tau \mid p\}} \qquad \textsc{M-RefineNot2}\quad \frac{\rho \vDash \neg p[x \mapsto o]}{\rho \vDash o \notin \{x{:}\tau \mid p\}}$$

$$\textsc{M-Type}\quad \frac{\vdash \rho(o) : \tau}{\rho \vDash o \in \tau} \qquad \textsc{M-TypeNot}\quad \frac{\vdash \rho(o) : \sigma \qquad \nexists v. \vdash v : \tau \text{ and } \vdash v : \sigma}{\rho \vDash o \notin \tau} \qquad \textsc{M-Theory}\quad \frac{[\![\rho]\!]_{\mathcal{T}} \vDash \mathcal{X}^{\mathcal{T}}}{\rho \vDash \mathcal{X}^{\mathcal{T}}}$$

Figure 3.10: λ_{RTR} Models Relation

tautology. The details of satisfaction are defined in figure 3.10. The satisfaction relation extends to environments in a pointwise manner.

In order to complete our definition of satisfaction, we require a typing rule for closures:

$$\textsc{T-Closure}\quad \frac{\exists\Gamma.\ \rho \vDash \Gamma \qquad \Gamma \vdash (\lambda(x\!:\!\tau)\, e) : \mathrm{R}}{\vdash [\rho,\ (\lambda(x\!:\!\tau)\, e)] : \mathrm{R}}$$

The satisfaction rules are mostly straightforward. $\mathbb{tt}$ is always satisfied, while the logical connectives $\vee$ and $\wedge$ are satisfied in the standard ways. Aliases are satisfied when the objects are equivalent values in ρ.

The satisfaction rules M-Refine, M-RefineNot1, and M-RefineNot2 allow refinement types to be satisfied by satisfying the type and proposition separately. M-Theory consults a decider for the specific theory in order to satisfy sentences in its domain.

From M-Type we see propositions stating an object o is of type τ are satisfied when the value of o in ρ is a subtype of τ. Similarly M-TypeNot tells us if an object o's value in ρ has a type which does not overlap with τ, then the proposition $o \notin \tau$ is satisfied.

Soundness

Our first lemma states that our proof theory respects models.

Lemma 3. *If $\rho \vDash \Gamma$ and $\Gamma \vdash p$ then $\rho \vDash p$.*

Proof. By structural induction on $\Gamma \vdash p$. □

With our proof theory and models in sync and our operational semantics defined, we can state and prove the next key lemma for type soundness which deals with evaluation.

Lemma 4. *If $\Gamma \vdash e : \langle\tau, p_+, p_-, o\rangle$, $\rho \vDash \Gamma$ and $\rho \vdash e \Downarrow v$ then all of the following hold:*

1. *all non-$\top^o$ structural parts of o are equal in ρ to the corresponding parts of v,*
2. *$v \neq$ `false` and $\rho \vDash p_+$, or $v =$ `false` and $\rho \vDash p_-$, and*
3. *$\Gamma \vdash v : \langle\tau, \mathbb{tt}, \mathbb{tt}, \top^o\rangle$*

Proof. By induction on the derivation of $\rho \vdash e \Downarrow v$. $\square$

Now we can state our soundness theorem for λ_{RTR}.

Theorem 2. *(Type Soundness for λ_{RTR}). If $\vdash e : \tau$ and $\vdash e \Downarrow v$ then $\vdash v : \tau$.*

Proof. Corollary of lemma 4. $\square$

Although this model-theoretic proof technique works quite naturally, it includes the standard drawbacks of big-step soundness proofs, saying nothing about diverging or stuck terms. We could address this by adding an `error` value of type `Empty` that is propagated upward during evaluation and modify our soundness claim to show `error` is not derived from evaluating well-typed terms.

3.3 Scaling to a Real Implementation

Although λ_{RTR} describes the essence of our approach, there are additional details to consider when reasoning about a realistic programming language.

3.3.1 Efficient, Algorithmic Subtyping

In order to highlight the essential features of λ_{RTR} we chose a more declarative description of the type system. To make this process efficient and algorithmic several additional steps can be taken.

Hybrid environments. Instead of working with only a set of propositions while type checking, it is helpful to use an environment with two distinct parts: one which resembles a standard type environment—mapping objects to the currently known positive and negative type information—and another which contains only the set of currently known compound propositions (since all atomic type-propositions can be efficiently stored in the former part). With these pieces in place, it is easy to iteratively refine the standard type environment with the `update` metafunction while traversing the abstract syntax tree instead of saving *all* logical reasoning for checking non-trivial terms.

Representative objects. Another valuable simplification which greatly reduced type checking times was the use of representative members from alias-equivalent classes of objects.

By eagerly substituting and using a single representative member in the environment, large complex propositions which conservatively but inefficiently tracked dependencies—such as those arising from local-bindings—can be omitted entirely, resulting in major performance improvements for real world Typed Racket programs.

Propogating existentials. Our typing judgments use subsumption to omit the less interesting details of type checking. Making this system algorithmic would not only require the standard inlining of subtyping throughout many of the judgments, but would also require that existential bindings on the type-results of subterms be propagated upward by the current term's type-result. This ensures all identifiers in the raw results of type checking are still bound and frees us from simplifying every intermediate type-result (as our model with subsumption often requires). This technique is thoroughly described in Knowels and Flanagan's [33] algorithmic type system, which served as an important motivation for this aspect of our approach.

3.3.2 Mutation

We soundly support mutation in our type system in a conservative fashion. First, a preliminary pass identifies which variables and fields may be mutated during program execution. The type checker then proceeds to type check the program, omitting symbolic objects for mutable variables and fields. This way, the initial type of a newly introduced variable will be recorded but no potentially unsound assumptions will be made from runtime tests in the code.

An illustrative example of this approach in action was found during our vector access case study and analysis of the Racket `math` library. It contained a module with a variable `cache-size` of type `Integer`. The type system ensured any updates to the value of `cache-size` were indeed of type `Integer`, but tests on the relative size of the cache—such as `(> cache-size n)`—failed to produce any logical information about the size of `cache-size`. This failure made it impossible to verify accesses whose correctness relied on the result of this test, since a concurrent thread could easily modify the cache and its size between our testing and performing the operation, invalidating any supposed guarantees. Indeed, without much effort we were able to cause a runtime error in the `math` library by exploiting this fact before patching the offending code.

3.3.3 Type Inference and Polymorphism

Typed Racket (and RTR) relies on local type inference [24] to instantiate type variables for polymorphic functions whenever possible. Since type inference is such an essential part of type checking real programs, we were unable to check any interesting examples until we had accommodated refinement types.

The constraint generation algorithm in local type inference, written $\Gamma \vdash^{V}_{\bar{X}} S <: T \Rightarrow C$, takes as input an environment Γ, a set of type variables V, a set of unknown type variables $\bar{X}$, and two types S and T, and produces a constraint set C. Since the implementation of the algorithm already correctly handled when S is a subtype of T, we merely needed to add the natural cases which allow constraint generation to properly recurse into refined types:

CG-Ref

$$\frac{\Gamma, x \in \tau, p_1 \vdash p_2 \qquad \Gamma \vdash^{V}_{\bar{X}} \tau <: \sigma \Rightarrow C}{\Gamma \vdash^{V}_{\bar{X}} \{x{:}\tau \mid p_1\} <: \{x{:}\sigma \mid p_2\} \Rightarrow C}$$

CG-RefLower

$$\frac{\Gamma \vdash^{V}_{\bar{X}} \tau <: \sigma \Rightarrow C}{\Gamma \vdash^{V}_{\bar{X}} \{x{:}\tau \mid p\} <: \sigma \Rightarrow C}$$

CG-RefUpper

$$\frac{\Gamma, x \in \tau \vdash p \qquad \Gamma \vdash^{V}_{\bar{X}} \tau <: \sigma \Rightarrow C}{\Gamma \vdash^{V}_{\bar{X}} \tau <: \{x{:}\sigma \mid p\} \Rightarrow C}$$

This naturally requires maintaining the full environment of propositions throughout the constraint generation process. Although we did not perform a detailed analysis, the annotation burden for polymorphic functions seems unaffected by our changes.

3.3.4 Complex Macros

Racket programmers use a series of `for`-macros for many iteration patterns [47]. This simple dot-product example iterates `i` from `0` to `(sub1 (vector-length A))` to perform the relevant computations:

```
(for/sum ([i (in-range (vector-length A))])
  (* (vector-ref A i)
     (vector-ref B i)))
```

Although initially verifying these vector accesses appears somewhat straightforward, Typed Racket's type checker runs *after* macro expansion on code resembling the following:

```
(letrec ([start 0]
         [end (vector-length A)]
         [step  1]
         [initial 0]
         [loop (λ (pos acc)
                (cond
                  [(< pos end)
                   (define i pos)
                   (loop (+ step pos)
                         (+ acc (* (vector-ref A i)
                                   (vector-ref B i))))]
                  [else acc]))])
 (loop start initial))
```

At this point the obvious nature of the original program may be obfuscated in the sea of primitives that emerge, and the system is left to infer types for the newly introduced identifiers and lambda abstractions.

After expansion of the **`for/sum`** macro, RTR is left to infer types for both the domain and range of the inner `loop` function (note that its arguments were not even annotatable identifiers in the original program). Initially, our local type inference chooses type `Int` for the position argument `pos`. This might be perfectly acceptable in Typed Racket, since `Integer` is a valid argument type for `vector-ref`. However, when attempting to verify the vector access, `Integer` is too permissive: it does not express the loop-invariant that `pos` is always non-negative.

In an effort to effectively reason about these macros we experimented with adding an additional heuristic to our inference for anonymous lambda applications: if a variable is, directly or indirectly, used as a vector index within the function, we try the type `Natural` instead of `Integer`. This type, combined with the upper-bounds check within the loop, is enough to verify the access in `(vector-ref A i)` and `(vector-ref B i)` (assuming they are of equal length). However, the heuristic quickly fails in the reverse iteration case, `(in-range (vector-length A) 0 -1)` (i.e. where `i` steps from `(sub1 (vector-length A))` to `0`) since for the last iteration `pos` is `-1` and not a `Natural`.

More advanced techniques for inferring invariants—such as those used by Liquid Types[20]—will be needed if idiomatic patterns such as Racket's **`for`** are to seamlessly integrate with refinement types.

3.4 Case Study: Safe Vector Access

In order to evaluate our RTR prototype's effectiveness on real programs we examined all unique vector accesses[1] in three large libraries written in Typed Racket, totalling more than 56,000 lines of code:

- The `math` library, a Racket standard library covering operations ranging from number theory to linear algebra. It contains 22,503 lines of code and 301 unique vector operations.
- The `plot` library, also a part of Racket's standard library, which supports both 2- and 3-dimensional plotting. It contains 14,987 lines of code and 655 unique vector operations.
- The `pict3d` library,[2] which defines a performant 3D engine with a purely functional interface, has 19,345 lines of code and 129 unique vector operations.

These libraries were chosen because of their size and frequent use of vector operations. During our analysis we tested whether each vector read and write could be replaced with its equivalent `safe-vector-` counterpart and still type check.

To reason statically about vector bounds and linear integer arithmetic we first enriched Typed Racket's base type environment, modifying the type of 36 functions. This included enriching the types of 7 vector operations, 16 arithmetic operations, 12 arithmetic fixnum operations (i.e. operations that work only on fixed-width integers), and the typing of Racket's `equal?`.

[1]Since we type check programs *after* macro expansion, vector accesses were assessed at this time as well, and accesses in macros were only counted once.

[2]https://github.com/jeapostrophe/pict3d

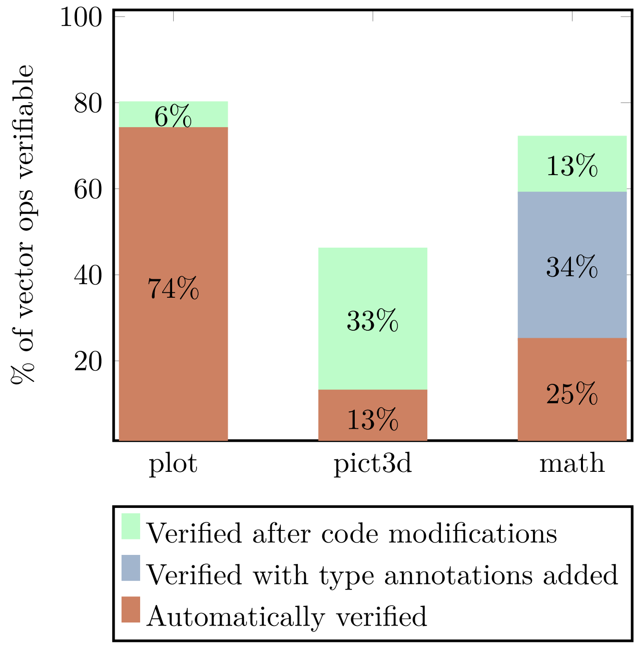


Figure 3.11: `safe-vec-ref` case study

We initially verified over 50% of accesses without the aid of additional annotations to the source code. As figure 3.11 illustrates, our success rate for entirely automatic verification of vector indices was 74% for `plot`, 13% for `pict3d`, and 25% for `math`. We attribute `plot`'s unusually high automatic success rate relative to the other libraries to a few heavily repeated patterns which are guaranteed to produce safe indexing: pattern matching on vectors and loops using a vector's length as an explicit bound were extremely common.

For the remaining vector accesses we performed a preliminary review of the `plot` and `pict3d` libraries and an in depth examination of the `math` library.

3.4.1 Enriching the Math Library

For the `math` library we examined each individual access to determine how many of the failing cases our system might handle with reasonable effort. We identified five general categories that describe these initially unverified vector operations:

Annotations Added. 34% of the failed accesses were unverified until additional (or more specific) type annotations were added to the original program. In this recursive loop snippet taken from our case study, for example, the `Nat` annotation for the index `i` is not specific enough to verify the vector reference:

```
(let loop ([i : Natural (vector-length ds)]
           [res : Natural 1])
  (cond
    [(zero? i) res]
    [else
     (loop (- i 1)
           (* res (safe-vector-ref ds i)))]))
```

Using `(Refine [i : Nat] (≤ i (len ds)))` for the type of `i`, however, allows RTR to verify the vector access immediately. As we discussed in section 3.3.4, a more advanced inference algorithm could potentially help by automatically inferring these types. On the other hand, as code documentation these added annotations often made programs easier to understand and helped us navigate our way through the large, unfamiliar code base.

Code Modified. 13% of the unverified accesses were verifiable after small local modifications were made to the body of the program. In some cases, these modifications moved the code away from particularly complex macros; other programs presented opportunities for a few well-placed dynamic checks to prove the safety of a series of vector operations. An example of the latter can be seen in the function `vector-swap!`:

```
(: vec-swap! (∀ {A} (-> (Vectorof A) Integer Integer Void)))
(define (vector-swap! vs i j)
  (unless (= i j)
    (cond
      [(and (< -1 i (vector-length vs))  ;; added
            (< -1 j (vector-length vs))) ;; added
       (define i-val (safe-vector-ref vs i))
       (define j-val (safe-vector-ref vs j))
       (safe-vector-set! vs i j-val)
       (safe-vector-set! vs j i-val)]
      [else (error "bad index(s)!")])))
```

This function swaps the values at two indices within a vector. Our initial investigation concluded adding constraints to the type was unreasonable for this particular function (i.e. clients could not easily satisfy the more specific types), however we noticed adding two simple tests on the indices in question allowed us to safely verify four separate vector operations without perturbing any client code. This approach seemed like an advantageous tradeoff in this and other situations and worked well in our experience.

Beyond our scope. 22% were unverifiable because, in their current form, their invariants were too complex to describe (i.e. they were outside the scope of our type system and/or linear integer theory). One simple example of this involved determining the maximum dimension `dims` for a list of arrays:

```
(define dims (apply max (map vector-length dss)))
```

Because of the complex higher order nature of these operations, our simple syntactic analysis and linear integer theory was unable to reason about how the integer `dims` related to the vectors in the list `dss`.

Unimplemented features 6% of the unverified accesses involved Racket features we had neglected to support during implementation (e.g. dependent record fields), but which seemed otherwise amenable to our verification techniques.

Unsafe code. As previously mentioned in section 3.3.2, we discovered 2 vector operations which made unsafe assumptions about a mutable cache whose size could shrink and cause errors at runtime. Both of these correctly did not typecheck using our system and were subsequently patched.

Total. In all, 72% of the vector accesses in the math library were verifiable using these approaches without drastically altering any internal algorithms or data representations.[3]

3.5 Adding Refinements to Typed Racket

Since the Racket v6.11 release, refinement types have been available in Typed Racket proper.[4] The process of adding the extension primarily followed the design and lessons learned working with RTR. First, we added a new type scheme `(Refine [x : t] p)` which allows the type `t` to be refined by the proposition `p` (where `x` is in scope for `p`) via the following grammar for propositions:

[3]Our modified math library can be found in our artifact.
[4]Available at https://racket-lang.org/

```
p  ::= Top | Bot | (: o t) | (! o t) | (and p ...) | (or p ...)
     | (when p p) | (unless p p) | (if p p p) | (c o o)
c  ::= < | <= | = | >= | >
n  ::= 0 | 1 | -1 | 2 | -2 | ...
o  ::= n | s | (+ o ...) | (- o ...) | (* n o)
s  ::= i | (f s)
i  ::= x | y | z | ...
f  ::= car | cdr | vector-length
```

While our formalism in section 3.2 talked about supporting arbitrary theories, we chose to initially support type-related propositions and the theory of linear integer arithmetic. The former theory Typed Racket was already well equipped to reason about as it is fundamental to how Typed Racket works; for the latter we use a simple implementation of fourier-motzkin elimination to decide linear inequalities. An more advanced external solver could be used to decide more complex related theories (e.g., non-linear integer arithmetic) but we decided at least initially against making Typed Racket dependent on an external solver for this addition.

As for dependent function types, while Typed Racket did technically already have them to a degree, they did not allow for dependencies between arguments. For this and other subtle implementation-specific reasons, we decided to add a separate internal representation for dependent function types which features a single arrow which allows for argument dependency and preconditions. For example, here is a "safe vector reference" function which uses a refinement on the second argument:

```
(All (A) (-> ([v : (Vectorof A)]
              [n : (v) (Refine [i : Natural]
                               (< i (vector-length v)))])
             A))
```

and here is an equivalent function type which instead uses a precondition:

```
(All (A) (-> ([v : (Vectorof A)]
              [n : Natural])
             #:pre (v n) (< n (vector-length v))
             A))
```

Note that while many function types which feature a refinement on an argument may be expressible via a precondition, not all are amicable to framing in this way. E.g., if an argument is a collection of integers whose values are determined by another argument, that dependency would need to appear in the collection's type directly and could not be expressed as a precondition.

3.5.1 Compiling Dependent Types into Contracts

Because Racket has an advanced contract system which supports dependent contracts [48], we are able to directly compile refinements and dependent function types into contracts when necessary. These contracts are applied when a value with a refined type or dependent function type are imported into an untyped module, allowing the invariants to be checked at runtime. To illustrate, consider the two variants of types for safe vector reference we gave in the previous section. The first variant of the type—which uses an explicit refinement on the second argument—would be compiled into the following dependent contract:

```
(parametric->/c (A)
  (->i ([v (vectorof A)]
        [i (v) (λ (x) (and (exact-integer? x)
                           (<= 0 x (sub1 (vector-length v)))))])
       [_ () A]))
```

And the second variant—which uses a precondition—would be compiled into the following dependent contract:

```
(parametric->/c (A)
  (->i ([v (vectorof A)]
        [i exact-integer?])
       #:pre (v i) (<= 0 i (sub1 (vector-length v)))
       [_ () A]))
```

3.5.2 Pay-as-you-go costs for developers

One concern in adding refinements to Typed Racket was creating type checking overhead for all programmers for a feature many will not use. To mitigate this issue, we introduced a language-level keyword that "turns on" the full spectrum of refinement reasoning on a module-by-module basis:

```
#lang typed/racket #:with-refinements

(require racket/unsafe/ops)
(provide safe-vector-ref)

(: safe-vector-ref
    (All (A) (-> ([v : (Vectorof A)]
                  [n : Natural])
                 #:pre (v n) (< n (vector-length v))
                 A)))
(define safe-vector-ref unsafe-vector-ref)
```

When this module-level **`#:with-refinements`** keyword is provided, the type checker assigns integer literals and many primitive operations more specific refinement types. With this technique for toggling more complex reasoning in the type checker we were able to add refinements and explicit dependent function types without noticeably affecting type checker performance for programs which do not explicitly use these features.

3.5.3 Dealing with Existentials

While our formal model for an occurrence typing calculus used existential quantifiers to refer to terms whose original identifiers were no longer in scope, in the implementation we found it more convenient to simply apply Skolemization and generate fresh "local" identifiers when necessary. To illustrate, consider the following modified rule for checking **`let`**-expressions where the expression whose value is bound to the local variable always has the symbolic object $\top^o$:

T-Let-Skolem

$$\frac{\Gamma \vdash e_1 : \langle \tau_1, p_1, q_1, \top^o \rangle \qquad p_x = (x \notin \texttt{False} \wedge p_1) \vee (x \in \texttt{False} \wedge q_1) \qquad \Gamma, x \in \tau_1, p_x \vdash e : \mathrm{R}_2 \qquad x' \notin (\mathsf{fvs}(\Gamma) \cup \mathsf{fvs}(\tau_1) \cup \mathsf{fvs}(p_1) \cup \mathsf{fvs}(q_1))}{\Gamma \vdash (\texttt{let}\,(x\ e_1)\ e_2) : \mathrm{R}_2[x \mapsto x']}$$

This rule is almost identical to T-Let from figure 3.4 except that x is replaced by the "fresh" local identifier x' instead of bound by an existential quantifier. This skolemization technique naturally accomplishes the same functionality without having to introduce an additional form to the implementation (i.e., existential quantification). Furthermore, it is sound as long as any fresh local identifiers which appear in the codomain of function type are either erased from the type or are "freshened" with each function application. In our implementation we chose erasure for simplicity.

3.6 Related Work

There is a history of using refinements and dependent types to enrich already existing type systems. Dependent ML [41] adds a practical set of dependent types to standard ML to allow for richer specifications and compiler optimizations through simple refinements, using a small custom solver to check constraints. Liquid Haskell [23] extends Haskell's type system with a more general set of refinement types supported by an SMT solver and predicate abstraction. We similarly strive to provide an expressive, practical extension to an existing type system by adding dependent refinements. Our approach, however, seeks to enrich a type system designed *specifically* for dynamically typed languages and therefore is built on a different set of foundational features (e.g. subtyping, 'true' union types, type predicates, etc.).

Some approaches, aiming for more expressive type specifications, have shown how enriching an ML-like typesystem with dependent types and access to theorem proving (automated and manual) provides both expressive and flexible programming tools. ATS, the successor of DML, supports both dependent and linear types as well as a form of interactive theorem proving for more complex obligations [49]. F* [22, 50] adds full dependent types and refinement types (along with other features) to an F_ω-like core while allowing manual and SMT solver-backed discharging of proof obligations. Although our system shares the goal of allowing users to further enrich their typed programs beyond the expressiveness of the core system, we have chosen a simpler, less expressive approach aimed at allowing dynamically typed programs to gradually adopt a simpler set of dependent types.

Chugh et al. [31] explore how extensive use of refinement types and an SMT solver enable type checking for rich dynamically typed languages such as JavaScript [4]. This approach feels similar to ours in terms of features and expressiveness. As seen in our respective metatheories, however, their system requires a much more complicated design and a complex stratified soundness proof; this fact has made it "[difficult to] add extra (basic) typing features to the language" [7]. In contrast, our system uses a well-understood core and does not *require* interaction with an external SMT solver. This allows us to use many common type-theoretic algorithms and techniques—as witnessed by Typed Racket's continued adoption of new features.

Vekris et al. [7] explore how refinements can help reason about complex JavaScript programs utilizing a novel two phase approach. The first phase elaborates the source language into a ML-like target that is checked using standard techniques, at which point the second phase attempts to verify all ill-typed branches are in fact infeasible using refinements in the spirit of Knowles and Flanagan [51] and Rondon et al. [20]. Our single-phase approach, however, does not require elaboration into an ML-like language and allows our system to work more directly with a larger set of types.

Sage's use of a dynamic and static types is similar to our approach for type checking programs. However, their usage of first-class types and arbitrary refinements means their core system is expressive yet undecidable [32]. Our system utilizes a more conservative, decidable core in which only a small set of immutable terms are lifted into types. Because of this, having impure functions and data in the language does not require changes to the type system. Also, our approach only reasons about non-type related theories when they are explicitly added.

Our usage of existential quantification to enable dependent yet abstract reasoning for values no longer in scope strongly resembles the approach described by Knowels and Flanagan [33]. Our design, however, lifts fewer terms into types in general and substitutes terms directly into types when possible. Additionally, our design includes features specifically aimed at dynamic languages instead of refining a more standard type theory.

Ou et al. [34] aim to make the process of working with dependent types more palatable by allowing fine-grained control over the trade-offs between dependent and simple types. This certainly is similar to our system in spirit, but there are several important differences. They choose to automatically insert coercions when dependent fragments and simple types interact, while we do not explicitly distinguish between the two and require explicit code to cast values. Additionally, while they convert their programs from a surface language into an entirely dependently typed language, our programs are translated into dynamically typed Racket code, which is void of any artifacts of our type system. This places us in a more suitable position for supporting sound interoperability between untyped and dependently typed programs.

Manifest contracts [52] are an approach that uses dependent contracts both as a method for ensuring runtime soundness and as a way to provide static typing information. Unlike our system, this method only reasons about explicit casts (i.e. program structure does not inform the type system), and there is no description of how a solver would be utilized to dispatch proof goals.

CHAPTER 4
SEMANTIC SUBTYPING

In this chapter we introduce semantic subtyping: a technique for reasoning soundly and completely about the full spectrum of set-theoretic types. Because the implementation details for such an approach are non-obvious and have rarely been discussed, in this chapter we give a detailed account for how to implement such a system. Later (in chapter 5) we describe a language and give algorithm descriptions that rely on the implementation approaches we describe here.

4.1 Set-theoretic Types

Set-theoretic types are a flexible and natural way for describing sets of values, featuring intuitive "logical combinators" in addition to traditional types for creating detailed specifications. As seen in figure 4.1, languages with set-theoretic types feature (at least some of) the following logical type constructors described below:

- $\tau \cup \sigma$ is the union of types τ and σ, describing values that are of type τ *or* of type σ;
- $\tau \cap \sigma$ is the intersection of types τ and σ, describing values that are of both type τ *and* σ;
- $\neg\tau$ is the complement/negation of type τ, describing values that are *not* of type τ;
- `Any` is the type describing all possible values; and

Base Types
$\iota \quad ::= \texttt{Int} \mid \texttt{Str} \mid \texttt{True} \mid \texttt{False}$
Types
$\tau, \sigma \quad ::= \iota \mid \tau \times \tau \mid \tau \to \tau \mid \tau \cup \tau \mid \tau \cap \tau \mid \neg\tau \mid \texttt{Any} \mid \texttt{Empty}$
Abbreviations
$\texttt{Any}^{\times} \equiv \texttt{Any} \times \texttt{Any} \mid \texttt{Any}^{\to} \equiv \texttt{Empty} \to \texttt{Any} \mid \texttt{Any}^{\iota} \equiv \neg(\texttt{Any}^{\times} \cup \texttt{Any}^{\to})$

Figure 4.1: Set-theoretic Types

- `Empty` is the type describing no values (i.e. $\neg$`Any`).

Additionally, we may specify "specific top type" which denotes all values of that particular kind:

- $\mathtt{Any}^{\times}$ is the type that denotes all pairs,
- $\mathtt{Any}^{\rightarrow}$ is the type that denotes all functions, and
- $\mathtt{Any}^{\iota}$ is the type that denotes all base values (e.g., integers, strings, and booleans).

Set-theoretic types frequently appear in type systems which reason about dynamically typed languages (e.g. TypeScript[3], Flow[25], Typed Racket[14], Typed Clojure[10]), but some statically typed languages have opted to use them as well due to their expressiveness, flexibility, and convenience (e.g. CDuce[53], Pony[54]).

4.1.1 Subtyping

With set-theoretic types, the programmer (and type system) must be able to reason about how various types relate. E.g., even if we know τ is not the same type as σ, is it the case that a value of type τ will necessarrily also be a value of type σ? In other words, does $\tau <: \sigma$ hold (i.e. is τ a subtype of σ)? For example, consider the subtyping question:

$$\boxed{(\mathtt{Int} \cup \mathtt{Str}) \times \mathtt{Str} <: (\mathtt{Int} \times \mathtt{Str}) \cup (\mathtt{Str} \times \mathtt{Str})}$$

Clearly the two types are not syntactically the same... but we can also see that any pair whose first element is either an integer or a string and whose second element is a string (i.e. the type on the left-hand side) is indeed either a pair with an integer and a string or a pair with a string and a string (i.e. the type on the right-hand side). As a programmer then we might reasonably expect that anywhere a $(\mathtt{Int} \times \mathtt{Str}) \cup (\mathtt{Str} \times \mathtt{Str})$ is expected, we could provide a value of type $(\mathtt{Int} \cup \mathtt{Str}) \times \mathtt{Str}$ and things should work just fine. Unfortunately, most systems that feature set-theoretic types use sound but incomplete reasoning to determine subtyping. This is because most type systems reason about subtyping via standard syntactic inference rules:

$$\frac{}{\tau <: \tau} \quad \frac{\tau <: \sigma_1}{\tau <: \sigma_1 \cup \sigma_2} \quad \frac{\tau <: \sigma_2}{\tau <: \sigma_1 \cup \sigma_2} \quad \frac{\tau_1 <: \sigma \quad \tau_2 <: \sigma}{\tau_1 \cup \tau_2 <: \sigma} \quad \frac{\tau_1 <: \sigma_1 \quad \tau_2 <: \sigma_2}{\tau_1 \times \tau_2 <: \sigma_1 \times \sigma_2}$$

These rules allow us to conclude the statement below the line if we can show that the statement(s) above the line hold. Upsides to using a system built from rules like this include (1) the rules can often directly be translated into efficient code and (2) we can generally examine each rule individually and decide if the antecedants necessarily imply the consequent (i.e. determine if the rule valid). The downside is that systems built directly from such rules are almost always incomplete in some way. E.g. with the above rules, we cannot conclude $(\texttt{Int} \cup \texttt{Str}) \times \texttt{Str}$ is a subtype of $(\texttt{Int} \times \texttt{Str}) \cup (\texttt{Str} \times \texttt{Str})$ even though it is true. One way to ensure we arrive at a complete treatment of subtyping for the entire spectrum of set-theoretic types is to adopt a *semantic* (instead of a syntactic) notion of subtyping.[1]

4.1.2 Semantic Subtyping

In the semantic approach to subtyping types will simply denote sets of values in the language in the expected ways:

- `True` denotes singleton set {`true`};
- `False` denotes singleton set {`false`} ;
- `Int` denotes the set of integers;
- `Str` denotes the set of strings;
- $\tau \times \sigma$ denotes the set of pairs whose first element is a value in τ and whose second element is a value in σ (i.e. the cartesian product of τ and σ);
- $\tau \rightarrow \sigma$ denotes the set of functions which can be applied to a value in τ and will return a value from σ (if they return);

[1]At the time of writing this tutorial, CDuce[53] may be the only example of an in-use language with a type system which features the full spectrum of set-theoretic types *and* complete subtyping. This is not surprising since its developers are also the researchers that have pioneered the approaches we will discuss.

$$\begin{array}{rll} \tau <: \sigma & iff & [\![\tau]\!] \subseteq [\![\sigma]\!] \\ & iff & [\![\tau]\!] \setminus [\![\sigma]\!] = \emptyset \\ & iff & [\![\tau]\!] \cap \overline{[\![\sigma]\!]} = \emptyset \\ & iff & [\![\tau]\!] \cap [\![\neg\sigma]\!] = \emptyset \\ & iff & [\![\tau \cap \neg\sigma]\!] = \emptyset \end{array}$$

Figure 4.2: Subtyping/Inhabitation Equivalence

- $\tau \cup \sigma$ denotes the union of the sets denoted by τ and σ;
- $\tau \cap \sigma$ denotes the intersection of the sets denoted by τ and σ;
- $\neg\tau$ denotes the complement of the set denoted by τ;
- `Any` denotes the set of all values; and
- `Empty` denotes the empty set.

Perhaps surprisingly, with our types merely denoting sets of values subtyping can be determined by deciding type inhabitation. As figure 4.2 illustrates, **"is a particular type inhabited" is really the only question we have to be able to answer** since asking $\tau <: \sigma$ is the same as asking if $\tau \cap \neg\sigma$ is uninhabited (i.e. does it denote the empty set?). And while this notion of treating types as sets of values may seem intuitive, the formal justification is quite complex. Systems which wish to reason about types as sets of values and who feature function types can quickly run into a problematic circularity in the metatheory and cardinality issues. Fortunately, these issues have been thoroughly addressed in prior work[28] and we will therefore lean on this fact and focus our efforts on just how one might go about implementing such a system.

4.1.3 Deciding Inhabitation, Normal Forms

To efficiently decide type inhabitation for set-theoretic types we leverage some of the same strategies used to decide the satisfiability of boolean formulas:

- types are kept in disjunctive normal form (DNF), and
- special data structures are used to efficiently represent DNF types.

Types in Disjunctive Normal Form

In addition to using DNF, it will be helpful to impose some additional structure on the normal form of our types. First let us note that any DNF boolean formula F:

$$\boxed{\begin{aligned} F = \quad & (x_3 \land \neg x_7 \land x_{13} \land ...) \\ \lor \quad & (x_{11} \land x_4 \land \neg x_1 \land \neg x_{21} \land ...) \\ \lor \quad & (\neg x_3 \land \neg x_4 \land x_1 \land ...) \end{aligned}}$$

can be reorganized slightly to group the positive and negative atoms in each conjunction:

$$\boxed{\begin{aligned} F = \quad & ((x_3 \land x_{13} \land ...) \land (\neg x_7 \land ...)) \\ \lor \quad & ((x_{11} \land x_4 \land ...) \land (\neg x_1 \land \neg x_{21} \land ...)) \\ \lor \quad & ((x_1 \land ...) \land (\neg x_3 \land \neg x_4 \land ...)) \end{aligned}}$$

We then observe that because F is in DNF, it can easily be described by a set of pairs $\mathsf{dnf}(F) = \{(P_0, N_0), \ldots, (P_n, N_n)\}$, with one pair (P, N) for each conjunctive clause in the overall disjunction, where P is the set of positive atoms in the clause and N is the set of negated atoms in the clause:

$$\boxed{F = \bigcup_{(P,N) \in \mathsf{dnf}(F)} \left(\left(\bigcap_{x \in P} x \right) \land \left(\bigcap_{x \in N} \neg x \right) \right)}$$

Because set-theoretic types have the same logical connectives as boolean formulas, any type τ can also be converted into a DNF $\mathsf{dnf}(\tau) = \{(P_0, N_0), \ldots, (P_n, N_n)\}$ where for each (P, N), P contains the positive atoms and N contains the negated atoms, where an atom (a) is either a base type (ι), a product type ($\tau_1 \times \tau_2$), or function type ($\tau_1 \rightarrow \tau_2$):

$$\boxed{\tau = \bigcup_{(P,N) \in \mathsf{dnf}(\tau)} \left(\left(\bigcap_{\mathrm{a} \in P} \mathrm{a} \right) \land \left(\bigcap_{\mathrm{a} \in N} \neg \mathrm{a} \right) \right)}$$

Partitioning Types

In addition to being able to convert any type into DNF, for any type τ there exists three specialized types τ^{ι}, $\tau^{\times}$, and $\tau^{\rightarrow}$ which contain *only atoms of the same kind* such that:

For any type τ there exists specialized DNF types τ^{ι}, $\tau^{\times}$, and $\tau^{\rightarrow}$ which can be represented as sets of pairs (P, N)—where P and N are sets of atoms of a single kind (i.e. base, product, or arrow)—such that each of the following equivalences hold:

$$\tau = (\mathsf{Any}^{\iota} \cap \tau^{\iota}) \cup (\mathsf{Any}^{\times} \cap \tau^{\times}) \cup (\mathsf{Any}^{\rightarrow} \cap \tau^{\rightarrow})$$

$$\tau^{\iota} = \bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\iota})} \left(\left(\bigcap_{\iota\in P} \iota\right) \wedge \left(\bigcap_{\iota\in N} \neg\iota\right)\right)$$

$$\tau^{\times} = \bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\times})} \left(\left(\bigcap_{(\tau_1\times\tau_2)\in P} \tau_1 \times \tau_2\right) \wedge \left(\bigcap_{(\tau_1\times\tau_2)\in N} \neg(\tau_1 \times \tau_2)\right)\right)$$

$$\tau^{\rightarrow} = \bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\rightarrow})} \left(\left(\bigcap_{(\tau_1\rightarrow\tau_2)\in P} \tau_1 \rightarrow \tau_2\right) \wedge \left(\bigcap_{(\tau_1\rightarrow\tau_2)\in N} \neg(\tau_1 \rightarrow \tau_2)\right)\right)$$

Figure 4.3: Canonical form for representing types

$$\tau = (\mathsf{Any}^{\iota} \cap \tau^{\iota}) \cup (\mathsf{Any}^{\times} \cap \tau^{\times}) \cup (\mathsf{Any}^{\rightarrow} \cap \tau^{\rightarrow})$$

By representing a type in this way, we can efficiently divide types into non-overlapping segments which can each have their own DNF representation.

i.e., τ^{ι} is a type whose atoms are all base types:

$$\tau^{\iota} = \bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\iota})} \left(\left(\bigcap_{\iota\in P} \iota\right) \cap \left(\bigcap_{\iota\in N} \neg\iota\right)\right)$$

$\tau^{\times}$ is a DNF type whose atoms are all arrow types:

$$\tau^{\times} = \bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\times})} \left(\left(\bigcap_{(\tau_1\times\tau_2)\in P} \tau_1 \times \tau_2\right) \cap \left(\bigcap_{(\tau_1\times\tau_2)\in N} \neg(\tau_1 \times \tau_2)\right)\right)$$

and $\tau^{\rightarrow}$ is a DNF type whose atoms are all function types:

$$\tau^{\rightarrow} = \bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\rightarrow})} \left(\left(\bigcap_{(\tau_1\rightarrow\tau_2)\in P} \tau_1 \rightarrow \tau_2\right) \cap \left(\bigcap_{(\tau_1\rightarrow\tau_2)\in N} \neg(\tau_1 \rightarrow \tau_2)\right)\right)$$

To illustrate what this partitioning looks like in practice, here are a few very simple types and their equivalent "partitioned" representation:

$$\begin{aligned}
\texttt{Empty} &\equiv (\texttt{Any}^{\iota} \cap \texttt{Empty}) \cup (\texttt{Any}^{\times} \cap \texttt{Empty}) \cup (\texttt{Any}^{\rightarrow} \cap \texttt{Empty})))\\
\texttt{Any} &\equiv (\texttt{Any}^{\iota} \cap \texttt{Any}) \cup (\texttt{Any}^{\times} \cap \texttt{Any}) \cup (\texttt{Any}^{\rightarrow} \cap \texttt{Any})))\\
\texttt{Int} &\equiv (\texttt{Any}^{\iota} \cap \texttt{Int}) \cup (\texttt{Any}^{\times} \cap \texttt{Empty}) \cup (\texttt{Any}^{\rightarrow} \cap \texttt{Empty})\\
\texttt{Int} \times \texttt{Str} &\equiv (\texttt{Any}^{\iota} \cap \texttt{Empty}) \cup (\texttt{Any}^{\times} \cap (\texttt{Int} \times \texttt{Str})) \cup (\texttt{Any}^{\rightarrow} \cap \texttt{Empty})))\\
\texttt{Int} \rightarrow \texttt{Str} &\equiv (\texttt{Any}^{\iota} \cap \texttt{Empty}) \cup (\texttt{Any}^{\times} \cap \texttt{Empty}) \cup (\texttt{Any}^{\rightarrow} \cap (\texttt{Int} \rightarrow \texttt{Str})))\\
\texttt{Int} \cup (\texttt{Int} \times \texttt{Str}) &\equiv (\texttt{Any}^{\iota} \cap \texttt{Int}) \cup (\texttt{Any}^{\times} \cap (\texttt{Int} \times \texttt{Str})) \cup (\texttt{Any}^{\rightarrow} \cap \texttt{Empty})))
\end{aligned}$$

This technique for partitioning types into separate non-overlapping DNFs—which will inform our strategy for actually representing types as data structures—will make type inhabitation inquiries easier to implement since we're specializing our representation to describe only the interesting, non-trivial clauses in a type. We summarize this discussion's key takeaway in figure 4.3 for reference.

4.2 Type Representation

In section 4.1 we determined that

- many type-related inquiries for set-theoretic types can be reduced to deciding type inhabitation (see section 4.1.2), and that because of this
- a partitioned DNF representation (summarized in figure 4.3) may be useful.

In this section we focus on the latter point—type representation—because it will impact how our algorithms decide type inhabitation. We will introduce several data structures, defining for each the binary operators union (∪), intersection (∩), and difference (\) and the unary operator complement ("¬"); the context of a given operator will determine which metafunction is being referenced.

4.2.1 Types as Data Structures

In figure 4.3 we noted a type can be conveniently deconstructed into three partitions, allowing us to reason separately about the base type (τ^{ι}), product type ($\tau^{\times}$), and function type ($\tau^{\rightarrow}$) portion of a type:

$$\tau = (\mathtt{Any}^{\iota} \cap \tau^{\iota}) \cup (\mathtt{Any}^{\times} \cap \tau^{\times}) \cup (\mathtt{Any}^{\rightarrow} \cap \tau^{\rightarrow})$$

We will use a data structure to represent our types that exactly mirror this structure. As illustrated in figure 4.4, our internal representation of a type is a 3-tuple:

Types (internal representation)
t ::= $\langle\beta,\ \mathrm{b}^{\times},\ \mathrm{b}^{\rightarrow}\rangle$

Figure 4.4: Internal type representation

The subcomponents of this representation correspond to the three specialized segments of a DNF type described in figure 4.3 as follows:

- β (the first field) contains base type information, corresponding to τ^{ι};
- $\mathrm{b}^{\times}$ (the second field) contains product type information, corresponding to $\tau^{\times}$; and
- $\mathrm{b}^{\rightarrow}$ (the third field) contains function type information, corresponding to $\tau^{\rightarrow}$.

The various top types used in figure 4.3 are implicit in the representation, i.e. we know what kind of type-information each field is responsible for so we need not explicitly keep around $\mathtt{Any}^{\iota}$, $\mathtt{Any}^{\times}$, and $\mathtt{Any}^{\rightarrow}$ in our partitioned representation. The grammar and meaning for β is given in section 4.2.2 and for $\mathrm{b}^{\times}$ and $\mathrm{b}^{\rightarrow}$ is given in section 4.2.3.

Top and Bottom Type Representation

The representation of the "top type" `Any`—the type that denotes all values—is written $\top$ and defined in figure 4.5. It places the respective top β, $\mathrm{b}^{\times}$, and $\mathrm{b}^{\rightarrow}$ in each field, mirroring the previous "partitioned" version of `Any` we showed earlier:

$$\mathtt{Any} \equiv (\mathtt{Any}^{\iota} \cap \mathtt{Any}) \cup (\mathtt{Any}^{\times} \cap \mathtt{Any}) \cup (\mathtt{Any}^{\rightarrow} \cap \mathtt{Any})$$

The representation of the "bottom type" `Empty`—the type that denotes no values—is written $\bot$ and also defined in figure 4.5. It similarly places the respective bottom β, $\mathrm{b}^{\times}$, and $\mathrm{b}^{\rightarrow}$ in each field, mirroring the previous "partitioned" version of `Empty` seen previously:

$$\texttt{Empty} \equiv (\texttt{Any}^{\iota} \cap \texttt{Empty}) \cup (\texttt{Any}^{\times} \cap \texttt{Empty}) \cup (\texttt{Any}^{\rightarrow} \cap \texttt{Empty})$$

In sections 4.2.2 and 4.2.3 we describe *why* those are the top and bottom representations for the base and product/arrow subcomponents respectively.

Finally, the representation of the specific top types $\texttt{Any}^{\iota}$, $\texttt{Any}^{\times}$, and $\texttt{Any}^{\rightarrow}$ as data structures $\top^{\iota}$, $\top^{\times}$, and $\top^{\rightarrow}$ (again see figure 4.5) involves placing the appropriate bottom type in each of the fields except for the one currently being represented (that field gets the appropriate top type).

$$\begin{array}{rcll}
\top & \equiv & \langle\langle -, \emptyset\rangle, \mathbb{1}, \mathbb{1}\rangle & \text{top type} \\
\bot & \equiv & \langle\langle +, \emptyset\rangle, \mathbb{0}, \mathbb{0}\rangle & \text{botom type} \\
\top^{\iota} & \equiv & \langle\langle -, \emptyset\rangle, \mathbb{0}, \mathbb{0}\rangle & \text{top base type} \\
\top^{\times} & \equiv & \langle\langle +, \emptyset\rangle, \mathbb{1}, \mathbb{0}\rangle & \text{top product type} \\
\top^{\rightarrow} & \equiv & \langle\langle +, \emptyset\rangle, \mathbb{0}, \mathbb{1}\rangle & \text{top function type}
\end{array}$$

Figure 4.5: Top and bottom type representations

Type Operations

As is seen in figure 4.6, binary operations on types benefit from our partitioned design: each operation is defined pointwise in the natural way across each disjoint partition. We encourage the reader to take a moment to convince themselves this is indeed correct, perhaps by considering intersecting two DNF types and observing what happens to intersections of non-overlapping clauses.

Type complement—also defined in figure 4.6—is simply defined in terms of type difference, subtracting the negated type from the top type.

4.2.2 Base DNF Representation

We now examine how a DNF type with only base type atoms can be efficiently represented (i.e. the base portion τ^{ι} of a type described in figure 4.3 and the β field in our representation of types described in figure 4.4).

$$\boxed{\_\cup\_ : \mathrm{t}\ \mathrm{t} \to \mathrm{t}}$$
$$\langle\beta_1,\, \mathrm{b}_1^{\times},\, \mathrm{b}_1^{\rightarrow}\rangle \cup \langle\beta_2,\, \mathrm{b}_2^{\times},\, \mathrm{b}_2^{\rightarrow}\rangle \quad = \quad \langle\beta_1 \cup \beta_2,\, \mathrm{b}_1^{\times} \cup \mathrm{b}_2^{\times},\, \mathrm{b}_1^{\rightarrow} \cup \mathrm{b}_2^{\rightarrow}\rangle$$

$$\boxed{\_\cap\_ : \mathrm{t}\ \mathrm{t} \to \mathrm{t}}$$
$$\langle\beta_1,\, \mathrm{b}_1^{\times},\, \mathrm{b}_1^{\rightarrow}\rangle \cap \langle\beta_2,\, \mathrm{b}_2^{\times},\, \mathrm{b}_2^{\rightarrow}\rangle \quad = \quad \langle\beta_1 \cap \beta_2,\, \mathrm{b}_1^{\times} \cap \mathrm{b}_2^{\times},\, \mathrm{b}_1^{\rightarrow} \cap \mathrm{b}_2^{\rightarrow}\rangle$$

$$\boxed{\_\setminus\_ : \mathrm{t}\ \mathrm{t} \to \mathrm{t}}$$
$$\langle\beta_1,\, \mathrm{b}_1^{\times},\, \mathrm{b}_1^{\rightarrow}\rangle \setminus \langle\beta_2,\, \mathrm{b}_2^{\times},\, \mathrm{b}_2^{\rightarrow}\rangle \quad = \quad \langle\beta_1 \setminus \beta_2,\, \mathrm{b}_1^{\times} \setminus \mathrm{b}_2^{\times},\, \mathrm{b}_1^{\rightarrow} \setminus \mathrm{b}_2^{\rightarrow}\rangle$$

$$\boxed{\neg : \mathrm{t} \to \mathrm{t}}$$
$$\neg\mathrm{t} \quad = \quad \top \setminus \mathrm{t}$$

Figure 4.6: Internal type operations

Although any type can be represented by some DNF type, in the case of base types things can be simplified even further! Any DNF type τ^{ι} whose atoms are all base types is equivalent to either

- a union of base types, e.g. $\iota_1 \cup \iota_2 \cup ...$
- a negated union of base types, e.g. $\neg(\iota_1 \cup \iota_2 \cup ...)$

To see why this is the case, it may be helpful to recall that (1) each base type is disjoint (i.e. no values inhabit more than one base type), (2) this is obviously true for `Any`, `Empty`, and any a single base type ι or negated base type $\neg\iota$, and (3) examine the details of the base type operations presented in figure 4.8 and note how one of these two representations is always naturally maintained.

Because any DNF of base types can be represented by a set of base types (i.e. the elements in the union) and a polarity (i.e. is the union negated or not), we represent the base portion of a type β using a tuple with these two pieces of information (figure 4.7).

The first field is the polarity flag (using $+$ for a union or $-$ for a negated union) and the second field is the set of base types B in the union. The top base type (i.e. the type which denotes all base type values) is a negated empty set $\langle -,\, \emptyset\rangle$ (indicating that it is *not* the case that this type contains no base values) and the bottom base type (the type which denotes no base type values) is a positive empty set $\langle +,\, \emptyset\rangle$ (indicating that it *is* the case

Base type representation
$\beta \quad ::= \langle \pm, \mathrm{B} \rangle$
Base set polarity
$\pm \quad ::= + \mid -$
Base set
$\mathrm{B} \quad ::= \emptyset \mid \{\iota\} \cup \mathrm{B}$

Figure 4.7: Internal base type representation

$$\boxed{\_ \cup \_ : \beta\ \beta \to \beta}$$
$$\begin{aligned}
\langle +, \mathrm{B}_1 \rangle \cup \langle +, \mathrm{B}_2 \rangle &= \langle +, \mathrm{B}_1 \cup \mathrm{B}_2 \rangle \\
\langle -, \mathrm{B}_1 \rangle \cup \langle -, \mathrm{B}_2 \rangle &= \langle -, \mathrm{B}_1 \cap \mathrm{B}_2 \rangle \\
\langle +, \mathrm{B}_1 \rangle \cup \langle -, \mathrm{B}_2 \rangle &= \langle -, \mathrm{B}_2 \setminus \mathrm{B}_1 \rangle \\
\langle -, \mathrm{B}_1 \rangle \cup \langle +, \mathrm{B}_2 \rangle &= \langle -, \mathrm{B}_1 \setminus \mathrm{B}_2 \rangle
\end{aligned}$$

$$\boxed{\_ \cap \_ : \beta\ \beta \to \beta}$$
$$\begin{aligned}
\langle +, \mathrm{B}_1 \rangle \cap \langle +, \mathrm{B}_2 \rangle &= \langle +, \mathrm{B}_1 \cap \mathrm{B}_2 \rangle \\
\langle -, \mathrm{B}_1 \rangle \cap \langle -, \mathrm{B}_2 \rangle &= \langle -, \mathrm{B}_1 \cup \mathrm{B}_2 \rangle \\
\langle +, \mathrm{B}_1 \rangle \cap \langle -, \mathrm{B}_2 \rangle &= \langle +, \mathrm{B}_1 \setminus \mathrm{B}_2 \rangle \\
\langle -, \mathrm{B}_1 \rangle \cap \langle +, \mathrm{B}_2 \rangle &= \langle +, \mathrm{B}_2 \setminus \mathrm{B}_1 \rangle
\end{aligned}$$

$$\boxed{\_ \setminus \_ : \beta\ \beta \to \beta}$$
$$\begin{aligned}
\langle +, \mathrm{B}_1 \rangle \setminus \langle +, \mathrm{B}_2 \rangle &= \langle +, \mathrm{B}_1 \setminus \mathrm{B}_2 \rangle \\
\langle -, \mathrm{B}_1 \rangle \setminus \langle -, \mathrm{B}_2 \rangle &= \langle +, \mathrm{B}_2 \setminus \mathrm{B}_1 \rangle \\
\langle +, \mathrm{B}_1 \rangle \setminus \langle -, \mathrm{B}_2 \rangle &= \langle +, \mathrm{B}_1 \cap \mathrm{B}_2 \rangle \\
\langle -, \mathrm{B}_1 \rangle \setminus \langle +, \mathrm{B}_2 \rangle &= \langle -, \mathrm{B}_1 \cup \mathrm{B}_2 \rangle
\end{aligned}$$

Figure 4.8: Internal base DNF operations

that this type contains no base values).

Base DNF Operations

Operations on these base type representations boil down to selecting the appropriate set-theoretic operation to combine the sets based on the polarities (figure 4.8).

Base type negation is not shown (because it is not used anywhere in this model), but would simply require "flipping" the polarity flag (i.e. the first field in the tuple).

4.2.3 Product and Function DNFs

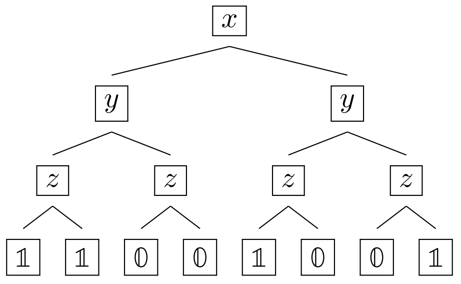
In order to efficiently represent a DNF type with only product or function type atoms (i.e. the $\tau^{\times}$ and $\tau^{\rightarrow}$ portions of a type described in figure 4.3 and the $\mathrm{b}^{\times}$ and $\mathrm{b}^{\rightarrow}$ fields in our type representation described in figure 4.4) we will use a binary decision diagram (BDD). First we include a brief review of how BDDs work, then we discuss how they can be used effectively to represent our product/function DNF types.

Binary Decision Diagrams

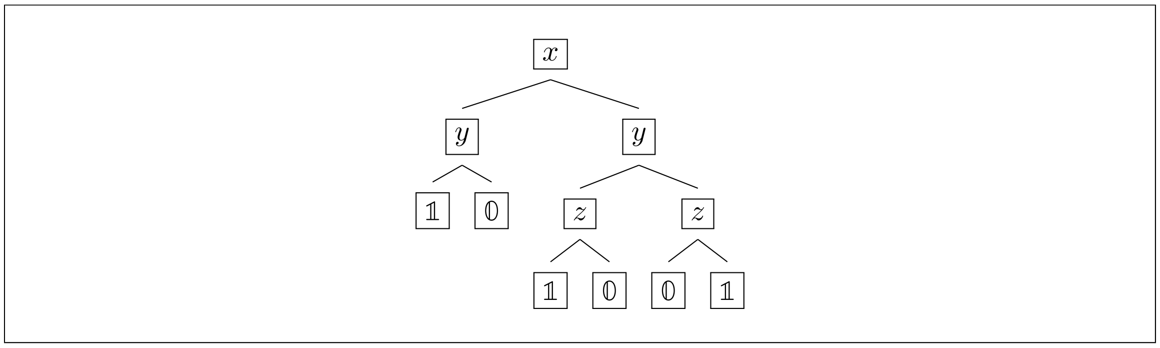
A binary decision diagram (BDD) is a tree-like data structure which provides a convenient way to represent sets or relations. For example, consider the truth table for the boolean formula $(\neg x \wedge \neg y \wedge \neg z) \vee (x \wedge y) \vee (y \wedge z)$:

x	y	z	$(\neg x \wedge \neg y \wedge \neg z) \vee (x \wedge y) \vee (y \wedge z)$
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	1

This formula can also be represented with the following BDD:



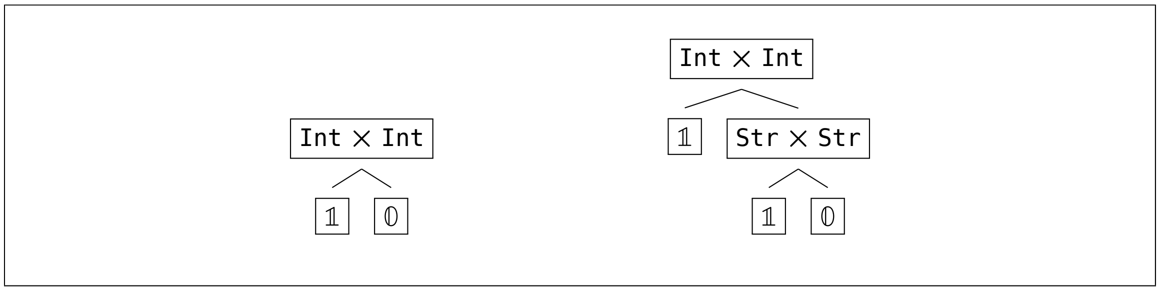
And as it turns out, we can simplify the tree slightly by collapsing a few nodes without losing any information:



In these BDDs, each non-leaf node contains a boolean variable. A node's left subtree describes the "residual formula" for when that variable is true and its right subtree describes the "residual formula" for when that variable is false. We invite the reader to compare the truth table and corresponding BDDs until they are convinced they indeed represent the same boolean formula. It may be useful to observe that the leaves in the unsimplified BDD correspond to the right-most column in the truth table.

Types as BDDs?

BDDs can also naturally encode set-theoretic types (in our case, DNF product or function types). Each node has a function/product type associated with it; henceforth we will call this associated type the *atom* of the node. A node's left sub-tree describes the "residual type" for when the atom is included in the overall type. A node's right sub-tree describes the "residual type" for when the atom's *negation* is included in the overall type. For example, here we have encoded the types $\texttt{Int} \times \texttt{Int}$ (left) and $(\texttt{Int} \times \texttt{Int}) \cup \texttt{Str} \times \texttt{Str}$ (right):



Essentially, each path in the tree represents a conjunction in the overall DNF, so the overall type is the union of all the possibly inhabited paths (i.e. paths that end in $\mathbb{1}$). In other words, for an arbitrary (type) BDD b:

$$\mathrm{b} = \begin{tikzcd}[column sep=small] & \boxed{\tau} \arrow[dl, no head] \arrow[dr, no head] & \\ \boxed{\mathrm{b}_l} & & \boxed{\mathrm{b}_r} \end{tikzcd}$$

we would interpret the *meaning* of b (written $[\![\mathrm{b}]\!]$) as follows:

$$[\![b]\!] = (\tau \cap [\![\mathrm{b}_l]\!]) \cup (\neg\tau \cap [\![\mathrm{b}_r]\!])$$

where $\mathbb{1}$ is interpreted as `Any` and $\mathbb{0}$ as `Empty`. There is, however, a well-known problem with BDDs we will want to avoid: repeatedly unioning trees can lead to significant (i.e. exponential) increases in size. This is particularly frustrating because—as we have previously noted—our primary concern algorithmically is deciding type inhabitation and taking the union of two types will have no *interesting* impact with respect to inhabitation (i.e., the union of τ_1 and τ_2 is empty only when both τ_1 and τ_2 are empty).

Types as Lazy BDDs!

Because there is no interesting impact on inhabitation when computing unions, we can use "lazy" BDDs to represent our function/product DNF types. In a lazy BDD, unions are only fully expand when computing type intersection or difference (i.e. operations that *can* have an interesting impact on inhabitation). Nodes in lazy BDDs have—in addition to the left and right subtrees described before—a "middle" subtree which assumes nothing about its node's atom. In other words, for an arbitrary lazy (type) BDD b:

$$\mathrm{b} = \begin{tikzcd}[column sep=small] & \boxed{\tau} \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\ \boxed{\mathrm{b}_l} & \boxed{\mathrm{b}_m} & \boxed{\mathrm{b}_r} \end{tikzcd}$$

we would interpret the meaning of b (written $[\![\mathrm{b}]\!]$) as follows:

$$[\![b]\!] = (\tau \cap [\![\mathrm{b}_l]\!]) \cup [\![\mathrm{b}_m]\!] \cup (\neg\tau \cap [\![\mathrm{b}_r]\!])$$

again where $\mathbb{1}$ is interpreted as `Any` and $\mathbb{0}$ as `Empty`. Henceforth when we use the term "BDD" we will be in fact referring to these lazy binary decision diagrams, which are the only kind of BDDs our implementation features.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{Lazy Binary Decision Diagram (BDD)}} \\
\quad \mathrm{b} & ::= \mathbb{1} \mid \mathbb{0} \mid \langle \mathrm{a}, \mathrm{b}, \mathrm{b}, \mathrm{b} \rangle \\
\multicolumn{2}{l}{\text{BDD Atom}} \\
\quad \mathrm{a} & ::= \mathrm{t} \times \mathrm{t} \mid \mathrm{t} \rightarrow \mathrm{t} \\
\multicolumn{2}{l}{\text{Product BDD}} \\
\mathrm{b}^{\times} & ::= \mathbb{1} \mid \mathbb{0} \mid \langle \mathrm{t} \times \mathrm{t}, \mathrm{b}^{\times}, \mathrm{b}^{\times}, \mathrm{b}^{\times} \rangle \\
\multicolumn{2}{l}{\text{Function BDD}} \\
\mathrm{b}^{\rightarrow} & ::= \mathbb{1} \mid \mathbb{0} \mid \langle \mathrm{t} \rightarrow \mathrm{t}, \mathrm{b}^{\rightarrow}, \mathrm{b}^{\rightarrow}, \mathrm{b}^{\rightarrow} \rangle
\end{array}
$$

Figure 4.9: Lazy BDDs for type representation

Figure 4.9 describes in detail our representation for the DNF function/product portions of a type as BDDs. Note that

- b describes a BDD of either functions or products and is useful for describing functions that are parametric w.r.t. which kind of atom they contain;
- $\mathbb{1}$ and $\mathbb{0}$ are the leaves in our BDDs, interpreted as `Any` and `Empty` respectively;
- an atom (a) is either a product or a function type—a given BDD will only contain atoms of one kind or the other; and
- $\mathrm{b}^{\times}$ and $\mathrm{b}^{\rightarrow}$ simply allow us to be more specific and describe what kind of atoms a particular BDD contains.

Although not explicit in the grammar, these trees are constructed using a total ordering on atoms (note that this implies types, BDDs, etc all must also have a total ordering defined since these data structures are mutually dependent). Without loss of generality we will assume a simple lexicographic ordering, although any ordering should suffice. The ordering—written $\mathrm{a}_1 < \mathrm{a}_2$ and the like—will be called upon frequently in function definitions for BDDs in the next section. Essentially the ordering allows us to have consistent representations for particular BDDs.

Finally, we use a “smart constructor”—defined in figure 4.10—to perform some obvious simplifications when constructing BDD nodes. We use an implicit syntax for the smart constructor (i.e. it looks identical to constructing a normal node), so whenever we construct a node (except of course on the right-hand side of the definition in figure 4.10 itself) we are

$$\boxed{\langle\_,\ \_,\ \_,\ \_\rangle : \mathrm{a\ b\ b\ b} \to \mathrm{b}}$$
$$\begin{aligned}
\langle \mathrm{a}, \mathrm{b}_l, \mathbb{1}, \mathrm{b}_r\rangle &= \mathbb{1}\\
\langle \mathrm{a}, \mathrm{b}, \mathrm{b}_m, \mathrm{b}\rangle &= \mathrm{b} \cup \mathrm{b}_m\\
\langle \mathrm{a}, \mathrm{b}_l, \mathrm{b}_m, \mathrm{b}_r\rangle &= \langle \mathrm{a}, \mathrm{b}_l, \mathrm{b}_m, \mathrm{b}_r\rangle
\end{aligned}$$

Figure 4.10: BDD node smart constructor

in fact using this smart constructor to simplify away some cases before simply constructing a new BDD node.

(Lazy) BDD Operations

The operations on BDDs can be understood by again considering how we interpret BDDs:

$$\begin{aligned}
[\![\mathbb{1}]\!] &= \texttt{Any}\\
[\![\mathbb{0}]\!] &= \texttt{Empty}\\
[\![\langle \mathrm{a}, \mathrm{b}_l, \mathrm{b}_m, \mathrm{b}_r\rangle]\!] &= (\mathrm{a} \cap [\![\mathrm{b}_l]\!]) \cup [\![\mathrm{b}_m]\!] \cup (\neg\mathrm{a} \cap [\![\mathrm{b}_r]\!])
\end{aligned}$$

Also, recall that BDD binary operations will only ever be used on two BDDs whose atoms are all of the same kind (either all product or all function arrows). With that in mind, we invite the reader to peruse figures 4.11 and 4.12 for the detailed descriptions of BDD union, intersection, difference, and negation. We will not enumerate justification for every line, but invite the reader to examine some of the details to gain intuition for the operations and how they relate to the underlying logical combinators. Suffice it to say here that each definition begins with a series of trivial cases before describing how to handle non-trivial BDD node arguments more generally in a manner that is semantically correct and maintains the ordering of atoms in the resulting BDD.

4.2.4 Parsing and Example Types

Figure 4.13 defines a function that converts the more readable surface-level types defined in figure 4.1 into the internal representation we have just finished describing. Examining the results of parsing functions can be helpful in better understanding the representation:

$$
\boxed{\_ \setminus \_ : \mathrm{b}\ \mathrm{b} \to \mathrm{b}}
$$

$$
\begin{array}{ll}
\mathrm{b}_1 \setminus \mathrm{b}_2 & = \mathbb{0} \quad \text{if } \mathrm{b}_1 = \mathrm{b}_2 \\
\mathrm{b}_1 \setminus \mathbb{1} & = \mathbb{0} \\
\mathbb{1} \setminus \mathrm{b}_2 & = \neg \mathrm{b}_2 \\
\mathrm{b}_1 \setminus \mathbb{0} & = \mathrm{b}_1 \\
\mathbb{0} \setminus \mathrm{b}_2 & = \mathbb{0} \\
\mathrm{b}_1 \setminus \mathrm{b}_2 & = \begin{cases} \langle \mathrm{a}_1,\ (\mathrm{b}_{l1} \cup \mathrm{b}_{m1}) \setminus \mathrm{b}_2,\ \mathbb{0},\ (\mathrm{b}_{r1} \cup \mathrm{b}_{m1}) \setminus \mathrm{b}_2 \rangle & \text{if } \mathrm{a}_1 < \mathrm{a}_2 \\ \langle \mathrm{a}_2,\ \mathrm{b}_1 \setminus (\mathrm{b}_{l2} \cup \mathrm{b}_{m2}),\ \mathbb{0},\ \mathrm{b}_1 \setminus (\mathrm{b}_{r2} \cup \mathrm{b}_{m2}) \rangle & \text{if } \mathrm{a}_1 > \mathrm{a}_2 \\ \langle \mathrm{a}_1,\ \mathrm{b}_{l1} \setminus \mathrm{b}_2,\ \mathrm{b}_{m1} \setminus \mathrm{b}_2,\ \mathrm{b}_{r1} \setminus \mathrm{b}_2 \rangle & \text{if } \mathrm{a}_1 = \mathrm{a}_2 \end{cases} \\
& \quad \text{where } \langle \mathrm{a}_1,\ \mathrm{b}_{l1},\ \mathrm{b}_{m1},\ \mathrm{b}_{r1} \rangle = \mathrm{b}_1 \\
& \quad \text{where } \langle \mathrm{a}_2,\ \mathrm{b}_{l2},\ \mathrm{b}_{m2},\ \mathrm{b}_{r2} \rangle = \mathrm{b}_2
\end{array}
$$

$$
\boxed{\neg : \mathrm{b} \to \mathrm{b}}
$$

$$
\begin{array}{ll}
\neg \mathbb{1} & = \mathbb{0} \\
\neg \mathbb{0} & = \mathbb{1} \\
\neg \langle \mathrm{a},\ \mathrm{b}_1,\ \mathrm{b}_2,\ \mathbb{0} \rangle & = \langle \mathrm{a},\ \mathbb{0},\ \neg(\mathrm{b}_2 \cup \mathrm{b}_1),\ \neg \mathrm{b}_2 \rangle \\
\neg \langle \mathrm{a},\ \mathbb{0},\ \mathrm{b}_2,\ \mathrm{b}_3 \rangle & = \langle \mathrm{a},\ \neg \mathrm{b}_2,\ \neg(\mathrm{b}_2 \cup \mathrm{b}_3),\ \mathbb{0} \rangle \\
\neg \langle \mathrm{a},\ \mathrm{b}_1,\ \mathbb{0},\ \mathrm{b}_3 \rangle & = \langle \mathrm{a},\ \neg \mathrm{b}_1,\ \neg(\mathrm{b}_1 \cup \mathrm{b}_3),\ \neg \mathrm{b}_3 \rangle \\
\neg \langle \mathrm{a},\ \mathrm{b}_1,\ \mathrm{b}_2,\ \mathrm{b}_3 \rangle & = \langle \mathrm{a},\ \neg(\mathrm{b}_1 \cup \mathrm{b}_2),\ \mathbb{0},\ \neg(\mathrm{b}_3 \cup \mathrm{b}_2) \rangle
\end{array}
$$

Figure 4.11: BDD difference and negation

$$\mathsf{parse}(\mathsf{Int}) \equiv \langle \langle +,\ \{\mathsf{Int}\} \rangle,\ \mathbb{0},\ \mathbb{0} \rangle$$

$$\mathsf{parse}(\neg \mathsf{Str}) \equiv \langle \langle -,\ \{\mathsf{Str}\} \rangle,\ \mathbb{0},\ \mathbb{0} \rangle$$

$$\mathsf{parse}(\mathsf{Int} \cup \mathsf{Str}) \equiv \langle \langle +,\ \{\mathsf{Int},\ \mathsf{Str}\} \rangle,\ \mathbb{0},\ \mathbb{0} \rangle$$

$$\mathsf{parse}(\mathsf{Int} \times \mathsf{Str}) \equiv \langle \langle +,\ \emptyset \rangle,\ \langle \langle +,\ \{\mathsf{Int}\} \rangle,\ \mathbb{0},\ \mathbb{0} \rangle \times \langle \langle +,\ \{\mathsf{Int}\} \rangle,\ \mathbb{0},\ \mathbb{0} \rangle,\ \mathbb{0} \rangle$$

$$\mathsf{parse}(\mathsf{Str} \to \mathsf{Str}) \equiv \langle \langle +,\ \emptyset \rangle,\ \mathbb{0},\ \langle \langle +,\ \{\mathsf{Str}\} \rangle,\ \mathbb{0},\ \mathbb{0} \rangle \to \langle \langle +,\ \{\mathsf{Str}\} \rangle,\ \mathbb{0},\ \mathbb{0} \rangle \rangle$$

4.3 Type Inhabitation

Because we are working with set-theoretic types, we are free to define subtyping purely in terms of type inhabitation (see the initial justification for this in figure 4.2), which is precisely what we do in figure 4.3. In the remainder of this section we examine how to decide type inhabitation using the data structures introduced in section 4.2.

$$\boxed{\_\cup\_ : \mathrm{b}\ \mathrm{b} \to \mathrm{b}}$$

$$\begin{array}{ll}
\mathrm{b}_1 \cup \mathrm{b}_2 & = \mathrm{b} \quad \text{if } \mathrm{b}_1 = \mathrm{b}_2 \\
\mathrm{b}_1 \cup \mathbb{1} & = \mathbb{1} \\
\mathbb{1} \cup \mathrm{b}_2 & = \mathbb{1} \\
\mathrm{b}_1 \cup \mathbb{0} & = \mathrm{b}_1 \\
\mathbb{0} \cup \mathrm{b}_2 & = \mathrm{b}_2 \\
\mathrm{b}_1 \cup \mathrm{b}_2 & = \begin{cases} \langle \mathrm{a}_1,\, \mathrm{b}_{l1},\, \mathrm{b}_{m1} \cup \mathrm{b}_2,\, \mathrm{b}_{r1} \rangle & \text{if } \mathrm{a}_1 < \mathrm{a}_2 \\ \langle \mathrm{a}_2,\, \mathrm{b}_{l2},\, \mathrm{b}_{m2} \cup \mathrm{b}_1,\, \mathrm{b}_{r2} \rangle & \text{if } \mathrm{a}_1 > \mathrm{a}_2 \\ \langle \mathrm{a}_1,\, \mathrm{b}_{l1} \cup \mathrm{b}_{l2},\, \mathrm{b}_{m1} \cup \mathrm{b}_{m2},\, \mathrm{b}_{r1} \cup \mathrm{b}_{r2} \rangle & \text{if } \mathrm{a}_1 = \mathrm{a}_2 \end{cases} \\
& \quad \text{where } \langle \mathrm{a}_1,\, \mathrm{b}_{l1},\, \mathrm{b}_{m1},\, \mathrm{b}_{r1} \rangle = \mathrm{b}_1 \\
& \quad \text{where } \langle \mathrm{a}_2,\, \mathrm{b}_{l2},\, \mathrm{b}_{m2},\, \mathrm{b}_{r2} \rangle = \mathrm{b}_2
\end{array}$$

$$\boxed{\_\cap\_ : \mathrm{b}\ \mathrm{b} \to \mathrm{b}}$$

$$\begin{array}{ll}
\mathrm{b}_1 \cap \mathrm{b}_2 & = \mathrm{b} \quad \text{if } \mathrm{b}_1 = \mathrm{b}_2 \\
\mathrm{b}_1 \cap \mathbb{1} & = \mathrm{b}_1 \\
\mathbb{1} \cap \mathrm{b}_2 & = \mathrm{b}_2 \\
\mathrm{b}_1 \cap \mathbb{0} & = \mathbb{0} \\
\mathbb{0} \cap \mathrm{b}_2 & = \mathbb{0} \\
\mathrm{b}_1 \cap \mathrm{b}_2 & = \begin{cases} \langle \mathrm{a}_1,\, \mathrm{b}_{l1} \cap \mathrm{b}_2,\, \mathrm{b}_{m1} \cap \mathrm{b}_2,\, \mathrm{b}_{r1} \cap \mathrm{b}_2 \rangle & \text{if } \mathrm{a}_1 < \mathrm{a}_2 \\ \langle \mathrm{a}_2,\, \mathrm{b}_1 \cap \mathrm{b}_{l2},\, \mathrm{b}_1 \cap \mathrm{b}_{m2},\, \mathrm{b}_1 \cap \mathrm{b}_{r2} \rangle & \text{if } \mathrm{a}_1 > \mathrm{a}_2 \\ \langle \mathrm{a}_1,\, (\mathrm{b}_{l1} \cup \mathrm{b}_{m1}) \cap (\mathrm{b}_{l2} \cup \mathrm{b}_{m2}),\, \mathbb{0},\, (\mathrm{b}_{r1} \cup \mathrm{b}_{m1}) \cap (\mathrm{b}_{r2} \cup \mathrm{b}_{m2}) \rangle & \text{if } \mathrm{a}_1 = \mathrm{a}_2 \end{cases} \\
& \quad \text{where } \langle \mathrm{a}_1,\, \mathrm{b}_{l1},\, \mathrm{b}_{m1},\, \mathrm{b}_{r1} \rangle = \mathrm{b}_1 \\
& \quad \text{where } \langle \mathrm{a}_2,\, \mathrm{b}_{l2},\, \mathrm{b}_{m2},\, \mathrm{b}_{r2} \rangle = \mathrm{b}_2
\end{array}$$

Figure 4.12: BDD union and intersection

4.3.1 Deciding Type Inhabitation

A DNF type is uninhabited exactly when *each clause in the overall disjunction is uninhabited.* With our DNF types partitioned into base, product, and function parts (see figure 4.3):

$$\tau = (\mathtt{Any}^{\iota} \cap \tau^{\iota}) \cup (\mathtt{Any}^{\times} \cap \tau^{\times}) \cup (\mathtt{Any}^{\rightarrow} \cap \tau^{\rightarrow})$$

we simply need ways to check if the base component (τ^{ι}), product component ($\tau^{\times}$), and function component ($\tau^{\rightarrow}$) are each empty. As figure 4.15 suggests, the representation of the base type portion is simple enough that we can pattern match on it directly to check if it is empty (recall that $\langle +, \emptyset \rangle$ is the bottom/empty β).

For deciding if the product and function components—which are represented with lazy

$$\begin{array}{ll}
\boxed{\mathsf{parse} : \tau \to \mathrm{t}} & \\
\mathsf{parse}(\iota) & = \langle\langle +,\ \{\iota\}\rangle,\ \mathbb{0},\ \mathbb{0}\rangle \\
\mathsf{parse}(\tau \times \sigma) & = \langle\langle +,\ \emptyset\rangle,\ \langle \mathrm{t} \times \mathrm{s},\ \mathbb{1},\ \mathbb{0},\ \mathbb{0}\rangle,\ \mathbb{0}\rangle \\
& \quad \text{where } \mathrm{t} = \mathsf{parse}(\tau),\ \mathrm{s} = \mathsf{parse}(\sigma) \\
\mathsf{parse}(\tau \to \sigma) & = \langle\langle +,\ \emptyset\rangle,\ \mathbb{0},\ \langle \mathrm{t} \to \mathrm{s},\ \mathbb{1},\ \mathbb{0},\ \mathbb{0}\rangle\rangle \\
& \quad \text{where } \mathrm{t} = \mathsf{parse}(\tau),\ \mathrm{s} = \mathsf{parse}(\sigma) \\
\mathsf{parse}(\tau \cup \sigma) & = \mathsf{parse}(\tau) \cup \mathsf{parse}(\sigma) \\
\mathsf{parse}(\tau \cap \sigma) & = \mathsf{parse}(\tau) \cap \mathsf{parse}(\sigma) \\
\mathsf{parse}(\neg\tau) & = \neg\mathsf{parse}(\tau) \\
\mathsf{parse}(\mathtt{Any}) & = \top \\
\mathsf{parse}(\mathtt{Empty}) & = \bot
\end{array}$$

Figure 4.13: Type parsing

$$\begin{array}{l}
\boxed{<:\ :\ \mathrm{t}\ \mathrm{t} \to \mathrm{bool}} \\
\mathrm{t}\ <:\ \mathrm{s} = \mathsf{empty}(\mathrm{s} \setminus \mathrm{t})
\end{array}$$

Figure 4.14: Semantic subtyping, defined in terms of type emptiness

BDDs (see previous discussion in section 4.2.3)—are empty, we rely on helper functions $\mathsf{empty}^{\times}$ and $\mathsf{empty}^{\to}$ which are defined later in this section. In these sections, we will use non-terminals P and N to represent a collection of atoms (see figure 4.16) with the intuition that when we are using P, it is a set of "positive" type information, and when we are using N it is a set of "negative" type information (even though no explicit negations are present).

Product Type Inhabitation

To decide if the product portion of a type is uninhabited, we recall (from section 4.1.3) that it is a union of conjunctive clauses, each of which can be described with a pair of two sets (P,N), where P contains the positive product types and N the negated product types:

$$\begin{array}{ll}
\boxed{\mathsf{empty} : \mathrm{t} \to \mathrm{bool}} & \\
\mathsf{empty}(\langle\langle +,\ \emptyset\rangle,\ \mathrm{b}^{\times},\ \mathrm{b}^{\to}\rangle) & = \mathsf{empty}^{\times}(\mathrm{b}^{\times}, \top, \top, \emptyset) \text{ and } \mathsf{empty}^{\to}(\mathrm{b}^{\to}, \bot, \emptyset, \emptyset) \\
\mathsf{empty}(\_) & = \text{false}
\end{array}$$

Figure 4.15: Type emptiness predicate

$$
\begin{array}{ll}
\text{Atom sets} & \\
P, N & = \emptyset \mid \{\text{a}\} \cup P
\end{array}
$$

Figure 4.16: Sets of atoms

$$\tau^{\times} = \bigcup_{(P,N) \in \mathsf{dnf}(\tau^{\times})} \left(\left(\bigcap_{(\tau_1 \times \tau_2) \in P} \tau_1 \times \tau_2 \right) \cap \left(\bigcap_{(\tau_1 \times \tau_2) \in N} \neg(\tau_1 \times \tau_2) \right) \right)$$

For $\tau^{\times}$ to be uninhabited, *each* (P, N) clause must be uninhabited. Checking that a given (P, N) clause is uninhabited occurs in two steps:

1. accumulate the positive type information in P into a single product $s_1 \times s_2$ (i.e. fold over the product types in P, accumulating their pairwise intersection into a single product type), and

2. check that for each $N' \subseteq N$ the following holds:

$$\left(\sigma_1 \; <: \bigcup_{(\tau_1 \times \tau_2) \in N'} \tau_1 \right) \text{ or } \left(\sigma_2 \; <: \bigcup_{(\tau_1 \times \tau_2) \in N \setminus N'} \tau_2 \right)$$

The first step is justified because pairs are covariant in their fields, i.e. if something is of type $\sigma_1 \times \sigma_2$ *and* of type $\sigma_1' \times \sigma_2'$ then it is also of type $(\sigma_1 \cap \sigma_1') \times (\sigma_2 \cap \sigma_2')$. The second step is more complicated. To understand, let us first note that a product type is uninhabited if *either* subcomponent is uninhabited. Next, observe that if we know something is of type `Any` $\times$ `Any` (i.e., it is a product of some sort) and also that it is of type $\neg(\tau_1 \times \tau_2)$, then it is either of type $\neg\tau_1 \times$ `Any` *or* of type `Any` $\times \neg\tau_2$; this is essentially the same as applying DeMorgan's law to a negated conjunction in logic: one of the conjuncts must be false for their conjunction to be false. And so for a (P, N) clause to be uninhabited where $(\tau_1 \times \tau_2) \in N$, it must be uninhabited for both possibilities implied by that negated product. By exploring each subset $N' \subseteq N$ and verifying that either in N' the left-hand side of the product is empty (i.e. the union of the negated types for the left-hand side are a supertype of σ_1) *or* the in $N \setminus N'$ the right-hand side is empty (i.e. the union of the negated types

$$\boxed{\mathsf{empty}^{\times} : \mathrm{b}^{\times}\ \mathrm{s}\ \mathrm{s}\ N \to \mathrm{bool}}$$

$$\begin{aligned}
\mathsf{empty}^{\times}(\mathbb{0}, \mathrm{s}_1, \mathrm{s}_2, N) &= \mathrm{true} \\
\mathsf{empty}^{\times}(\mathbb{1}, \mathrm{s}_1, \mathrm{s}_2, N) &= \mathsf{empty}(\mathrm{s}_1) \text{ or } \mathsf{empty}(\mathrm{s}_2) \text{ or } \theta^{\times}(\mathrm{s}_1, \mathrm{s}_2, N) \\
\mathsf{empty}^{\times}(\langle\!\langle \mathrm{t}_1 \times \mathrm{t}_2,\, \mathrm{b}_l^{\times},\, \mathrm{b}_m^{\times},\, \mathrm{b}_r^{\times} \rangle\!\rangle, \mathrm{s}_1, \mathrm{s}_2, N) &= \mathsf{empty}^{\times}(\mathrm{b}_l^{\times}, \mathrm{s}_1 \cap \mathrm{t}_1, \mathrm{s}_2 \cap \mathrm{t}_2, N) \\
&\quad \text{and } \mathsf{empty}^{\times}(\mathrm{b}_m^{\times}, \mathrm{s}_1, \mathrm{s}_2, N) \\
&\quad \text{and } \mathsf{empty}^{\times}(\mathrm{b}_r^{\times}, \mathrm{s}_1, \mathrm{s}_2, \{\mathrm{t}_1 \times \mathrm{t}_2\} \cup N)
\end{aligned}$$

$$\boxed{\theta^{\times} : \mathrm{s}\ \mathrm{s}\ N \to \mathrm{bool}}$$

$$\begin{aligned}
\theta^{\times}(\mathrm{s}_1, \mathrm{s}_2, \emptyset) &= \mathrm{false} \\
\theta^{\times}(\mathrm{s}_1, \mathrm{s}_2, \{\mathrm{t}_1 \times \mathrm{t}_2\} \cup N) &= (\mathrm{s}_1 <: \mathrm{t}_1 \text{ or } \theta^{\times}(\mathrm{s}_1 \setminus \mathrm{t}_1, \mathrm{s}_2, N)) \\
&\quad \text{and } (\mathrm{s}_2 <: \mathrm{t}_2 \text{ or } \theta^{\times}(\mathrm{s}_1, \mathrm{s}_2 \setminus \mathrm{t}_2, N))
\end{aligned}$$

Figure 4.17: Product BDD inhabitation functions

for the right-hand side is a supertype of σ_2), we are exploring all possible combinations of negated first and negated second fields from the negated products in N and thus ensuring all possible combinations are uninhabited.

We describe an algorithm to perform these computations in figure 4.17. The function $\mathsf{empty}^{\times}$ walks over each path in the product BDD accumulating the positive field information in s_1 and s_2 and the negative information in the set N. Then at each non-trivial leaf in the BDD, we call the helper function $\theta^{\times}$ which searches the space of possible negation combinations ensuring that for each possibility the product ends up being uninhabited.

Note that $\theta^{\times}$ is designed with a "short-circuiting" behavior, i.e. as we are exploring each possible combination of negations, if a negated field we are considering would negate the corresponding positive field (i.e. $\mathrm{s}_1 <: \mathrm{t}_1$ or $\mathrm{s}_2 <: \mathrm{t}_2$) then we can stop searching for emptiness on that side, otherwise we subtract that negated type from the corresponding field and we keep searching the remaining possible negations checking for emptiness. If we reach the base case when N is the empty set, then we have failed to show the product is empty and we return false. Note that $\mathsf{empty}^{\times}$ checks for emptiness before calling $\theta^{\times}$ to avoid unnecessary searching. If it did not, $\theta^{\times}$'s base case would need to check s_1 and s_2 for emptiness as well.

Function Type Inhabitation

Just like with products, to show that the function portion of a type is uninhabited we show that each (P, N) clause in the DNF—

$$\boxed{\tau^{\rightarrow} = \bigcup_{(P,N)\in \mathsf{dnf}(\tau^{\rightarrow})} \left(\left(\bigcap_{(\tau_1\rightarrow\tau_2)\in P} \tau_1 \rightarrow \tau_2\right) \cap \left(\bigcap_{(\tau_1\rightarrow\tau_2)\in N} \neg(\tau_1 \rightarrow \tau_2)\right)\right)}$$

—represents an uninhabited function type. To do this, we show that for each clause (P, N) there exists a $(\mathrm{t}_1 \rightarrow \mathrm{t}_2) \in N$ such that

$$\boxed{\left(\mathrm{t}_1 <: \bigcup_{(\mathrm{s}_1\rightarrow\mathrm{s}_2)\in P} \mathrm{s}_1\right)}$$

(i.e. t_1 is in the domain of the function represented by this (P, N) clause) and that for each possible combination of arrows $P' \subseteq P$,

$$\boxed{\left(\mathrm{t}_1 <: \bigcup_{(\mathrm{s}_1\rightarrow\mathrm{s}_2)\in P\setminus P'} \mathrm{s}_1\right) \text{ or } \left(\bigcap_{(\mathrm{s}_1\rightarrow\mathrm{s}_2)\in P'} \mathrm{s}_2 <: \mathrm{t}_2\right)}$$

You can roughly think of this as verifying that for each possible set of arrows P' which *must* handle a value of type t_1 (i.e. the left-hand check fails), those arrows together would map the value to t_2 (the right-hand check), which would be a contradiction since we know this function is *not* of type $\mathrm{t}_1 \rightarrow \mathrm{t}_2$.

We implement this algorithm with the function $\mathsf{empty}^{\rightarrow}$ defined in figure 4.18. It walks each path in a function BDD accumulating the domain along the way and collecting the negated function types in the variable N. At the non-trivial leaves of the BDD, it calls $\theta^{\rightarrow}$ with each function type $(\mathrm{t}_1 \rightarrow \mathrm{t}_2) \in N$ until it finds a contradiction (i.e. an arrow that satisfies the above described equation) or runs out of negated function types.

$\theta^{\rightarrow}$ is the function which explores each set of arrows $P' \subseteq P$ checking that one of the two clauses in the above noted disjunction is true. Note that in the initial call from $\mathsf{empty}^{\rightarrow}$ we negate the original t_2: this is because although we are interested in checking for $\mathrm{s}_2 <: \mathrm{t}_2$ as we accumulate the codomains in s_2, the equivalent "contrapositive" statement $\neg\mathrm{t}_2 <: \neg\mathrm{s}_2$ is more convenient to check as we iterate through the function types in P.

$$\boxed{\mathsf{empty}^{\rightarrow} : \mathrm{b}^{\rightarrow}\ \mathrm{s}\ P\ N \rightarrow \mathrm{bool}}$$

$$\begin{aligned}
\mathsf{empty}^{\rightarrow}(\mathbb{0}, \mathrm{s}, P, N) &= \text{true} \\
\mathsf{empty}^{\rightarrow}(\mathbb{1}, \mathrm{s}, P, N) &= \text{if } \exists(\mathrm{t}_1 \rightarrow \mathrm{t}_2) \in N.\, (\mathrm{t}_1 <: \mathrm{s} \text{ and } \theta^{\rightarrow}(\mathrm{t}_1, \neg\mathrm{t}_2, P)) \\
&\ \ \text{then true} \\
&\ \ \text{else false} \\
\mathsf{empty}^{\rightarrow}(\langle \mathrm{s}_1 \rightarrow \mathrm{s}_2,\, \mathrm{b}_l^{\rightarrow},\, \mathrm{b}_m^{\rightarrow},\, \mathrm{b}_r^{\rightarrow} \rangle, \mathrm{s}, P, N) &= \mathsf{empty}^{\rightarrow}(\mathrm{b}_l^{\rightarrow}, \mathrm{s}, \{\mathrm{s}_1 \rightarrow \mathrm{s}_2\} \cup P, N) \\
&\ \ \text{and } \mathsf{empty}^{\rightarrow}(\mathrm{b}_m^{\rightarrow}, \mathrm{s}, P, N) \\
&\ \ \text{and } \mathsf{empty}^{\rightarrow}(\mathrm{b}_m^{\rightarrow}, \mathrm{s}, P, \{\mathrm{s}_1 \rightarrow \mathrm{s}_2\} \cup N)
\end{aligned}$$

$$\boxed{\theta^{\rightarrow} : \mathrm{t}\ \mathrm{t}\ P \rightarrow \mathrm{bool}}$$

$$\begin{aligned}
\theta^{\rightarrow}(\mathrm{t}_1, \mathrm{t}_2, \emptyset) &= \mathsf{empty}(\mathrm{t}_1) \text{ or } \mathsf{empty}(\mathrm{t}_2) \\
\theta^{\rightarrow}(\mathrm{t}_1, \mathrm{t}_2, \{\mathrm{s}_1 \rightarrow \mathrm{s}_2\} \cup P) &= (\mathrm{t}_1 <: \mathrm{s}_1 \text{ or } \theta^{\rightarrow}(\mathrm{t}_1 \setminus \mathrm{s}_1, \mathrm{t}_2, P)) \\
&\ \ \text{and } (\mathrm{t}_2 <: \neg\mathrm{s}_2 \text{ or } \theta^{\rightarrow}(\mathrm{t}_1, \mathrm{t}_2 \cap \mathrm{s}_2, P))
\end{aligned}$$

Figure 4.18: Functions for checking if a function BDD is uninhabited

In the base case of $\theta^{\rightarrow}$ when P has been exhausted, the function checks that either the arrows not in P' could have handled the value of (the *original*) type t_1 (i.e. is t_1 now empty), otherwise it checks if the value we mapped the input to must be a subtype of (the *original*) type t_2 (i.e. is t_2 now empty).

In the case where P has not been exhausted, we examine the first arrow $(\mathrm{s}_1 \rightarrow \mathrm{s}_2)$ in P and check two cases: one for when that arrow is not in P' (i.e. when it is in $P \setminus P'$) and one for when it is in P'. The first clause in the conjunction of the non-empty P case is for when $\mathrm{s}_1 \rightarrow \mathrm{s}_2$ is not in P'. It first checks if the set of arrows we're not considering (i.e. $P \setminus P'$) would handle a value of type t_1 (i.e. $\mathrm{t}_1 <: \mathrm{s}_1$), and if not it remembers that $(\mathrm{s}_1 \rightarrow \mathrm{s}_2)$ is not in P' by subtracting s_1 from t_1 for the recursive call which keeps searching. The second clause in the conjunction is for when $\mathrm{s}_1 \rightarrow \mathrm{s}_2$ is in P'. As we noted, instead of checking $\mathrm{s}_2 <: \mathrm{t}_2$ (resembling the original mathematical description above), it turns out to be more convenient to check the contrapositive statement $\mathrm{t}_2 <: \neg\mathrm{s}_2$ (recall that t_2 was actually negated originally when $\theta^{\rightarrow}$ was called). First we check if having $(\mathrm{s}_1 \rightarrow \mathrm{s}_2)$ in P' means we would indeed map a value of type t_1 to a value of type t_2 (i.e. the $\mathrm{t}_2 <: \neg\mathrm{s}_2$ check). If so we are done, otherwise we recur while remembering that $(\mathrm{s}_1 \rightarrow \mathrm{s}_2)$ is in P' by adding s_2 to t_2 (i.e. "subtracting" negated s_2 from the negated t_2 we are accumulating by using intersection).

4.4 Other Type-level Metafunctions

In addition to being able to decide type inhabitation, we need to be able to semantically calculate precise types for the following situations:

1. projection from a product,
2. a function's domain, and
3. the result of function application.

4.4.1 Product Projection

In a language with syntactic types, calculating the type of the first or second projection of a pair simply involves matching on the product type and extracting the first or second field's type. In a language with semantic types, however, we cannot simply pattern match because we could be dealing with an arbitrarily complex pair type constructed using many set-theoretic type connectives. Instead, we must reason semantically about the types of the fields.

To begin, first note that if a type is a subtype of $\texttt{Any}^{\times}$ (i.e. it is indeed a pair), we can focus on the product portion of the type:

$$\boxed{\tau^{\times} = \bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\times})} \left(\left(\bigcap_{(\tau_1\times\tau_2)\in P} \tau_1 \times \tau_2 \right) \cap \left(\bigcap_{(\tau_1\times\tau_2)\in N} \neg(\tau_1 \times \tau_2) \right) \right)}$$

Projecting the field i from $\tau^{\times}$ (where $i \in \{1,2\}$) involves unioning each positive type for field i in the DNF intersected with each possible combination of negations for that field:

$$\boxed{\bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\times})} \left(\left(\bigcap_{(\tau_1\times\tau_2)\in P} \tau_i \right) \cap \bigcup_{N'\subseteq N} \left(\bigcap_{(\tau_1\times\tau_2)\in N'} \neg\tau_i \right) \right)}$$

This follows the same line of reasoning we used for deciding product type inhabitation in section 4.3.1), i.e. although we can intersect all of the positive information due to the covariance of product fields, the negative product information must be considered by considering all possible combinations of negations.

Actually that equation is sound but a little too coarse: it *only* considers the type of field i and thus may include some impossible cases where the *other* field would have been uninhabited (and thus the whole product in that case would be uninhabited). In other words, if j is an index and $j \neq i$ (i.e. j is the index of the other field), then as we're calculating the projection of i, we'll want to "skip" any N' cases where the following is true:

$$\boxed{\bigcap_{(\tau_1 \times \tau_2) \in P} \tau_j <: \bigcup_{(\tau_1 \times \tau_2) \in N \setminus N'} \tau_j}$$

i.e. cases where the other field is uninhabited. If we incorporate that subtlety, our inner loop will end up containing a conditional statement:

$$\boxed{\bigcup_{(P,N) \in \mathsf{dnf}(\tau^{\times})} \left(\left(\bigcap_{(\tau_1 \times \tau_2) \in P} \tau_i \right) \cap \bigcup_{N' \subseteq N} \left(\begin{array}{ll} \text{if} & \bigcap_{(\tau_1 \times \tau_2) \in P} \tau_j <: \bigcup_{(\tau_1 \times \tau_2) \in N \setminus N'} \tau_j \\ \text{then} & \texttt{Empty} \\ \text{else} & \bigcap_{(\tau_1 \times \tau_2) \in N'} \neg \tau_i \end{array} \right) \right)}$$

Implementing Product Projection

As was suggested by our use of index variables i and j in the previous section's discussion, we implement product projection as a single function indexed by some $i \in \{1, 2\}$ to return the appropriate type in non-empty clauses. Because projection can fail, we have the function $\mathsf{proj}^?$ as the "public interface" to projection. $\mathsf{proj}^?$ performs important preliminary checks (i.e. is this type actually a product type?) before extracting the product portion of the type and passing it to the "internal" function proj where the real work begins.

proj walks the BDD, accumulating for each path (i.e. each clause in the DNF) the positive type information for each field in variables s_1 and s_2 respectively. Along the way, if either s_1 or s_2 are empty we can ignore that path. Otherwise at non-trivial leaves we call the helper function $\phi^{\times}$ which traverses the possible combinations of negations, calculating and unioning the type of field i for each possibility.

$\boxed{\mathsf{proj}^? : i \ \mathrm{t} \rightarrow \mathrm{t} \text{ or false}}$

$$\begin{array}{rcl}
\mathsf{proj}_i(\mathrm{t}) & = & \text{false} \quad \text{if } \mathrm{t} \not<: \top^{\times} \\
\mathsf{proj}_i(\langle\!\langle \_, \mathrm{b}^{\times}, \_ \rangle\!\rangle) & = & \phi_i^{\times}(\mathrm{b}^{\times}, \top, \top)
\end{array}$$

$\boxed{\mathsf{proj} : i \ \mathrm{b}^{\times} \ \mathrm{s} \ \mathrm{s} \ N \rightarrow \mathrm{t}}$

$$\begin{array}{rcl}
\mathsf{proj}_i(\mathbb{0}, \mathrm{s}_1, \mathrm{s}_2, N) & = & \bot \\
\mathsf{proj}_i(\mathrm{b}^{\times}, \mathrm{s}_1, \mathrm{s}_2, N) & = & \bot \quad \text{if } \mathsf{empty}(\mathrm{s}_1) \text{ or } \mathsf{empty}(\mathrm{s}_2) \\
\mathsf{proj}_i(\mathbb{1}, \mathrm{s}_1, \mathrm{s}_2, N) & = & \phi_i^{\times}(\mathrm{s}_1, \mathrm{s}_2, N) \\
\mathsf{proj}_i(\langle\!\langle \mathrm{t}_1 \times \mathrm{t}_2, \mathrm{b}_l^{\times}, \mathrm{b}_m^{\times}, \mathrm{b}_r^{\times} \rangle\!\rangle, \mathrm{s}_1, \mathrm{s}_2, N) & = & \mathrm{t}_l \cup \mathrm{t}_m \cup \mathrm{t}_r \\
 & & \text{where } \mathrm{t}_l = \mathsf{proj}_i(\mathrm{b}_l^{\times}, \mathrm{s}_1 \cap \mathrm{t}_1, \mathrm{s}_2 \cap \mathrm{t}_2, N) \\
 & & \qquad \mathrm{t}_m = \mathsf{proj}_i(\mathrm{b}_m^{\times}, \mathrm{s}_1, \mathrm{s}_2, N) \\
 & & \qquad \mathrm{t}_r = \mathsf{proj}_i(\mathrm{b}_r^{\times}, \mathrm{s}_1, \mathrm{s}_2, \{\mathrm{t}_1 \times \mathrm{t}_2\} \cup N)
\end{array}$$

$\boxed{\phi^{\times} : i \ \mathrm{s} \ \mathrm{s} \ N \rightarrow \mathrm{t}}$

$$\begin{array}{rcl}
\phi_i^{\times}(\mathrm{s}_1, \mathrm{s}_2, N) & = & \bot \quad \text{if } \mathsf{empty}(\mathrm{s}_1) \text{ or } \mathsf{empty}(\mathrm{s}_2) \\
\phi_i^{\times}(\mathrm{s}_1, \mathrm{s}_2, \emptyset) & = & \mathrm{s}_i \\
\phi_i^{\times}(\mathrm{s}_1, \mathrm{s}_2, \{\mathrm{t}_1 \times \mathrm{t}_2\} \cup N) & = & \phi_i^{\times}(\mathrm{s}_1 \setminus \mathrm{t}_1, \mathrm{s}_2, N) \cup \phi_i^{\times}(\mathrm{s}_1, \mathrm{s}_2 \setminus \mathrm{t}_2, N)
\end{array}$$

Figure 4.19: Functions for projecting from a product type

4.4.2 Function Domain

Similar to product projection, deciding the domain of a function in a language with set-theoretic types cannot be done using simple pattern matching; we must reason about the domain of a function type potentially constructed with intersections and/or unions. To do this, first note that for an intersection of arrows, the domain is equivalent to the union of each of the domains (i.e. the function can accept any value any of the various arrows can collectively accept):

$$\boxed{\mathit{domain}((\sigma_1 \rightarrow \tau_1) \cap \ldots \cap (\sigma_n \rightarrow \tau_n)) = \sigma_1 \cup \ldots \cup \sigma_n}$$

Second, note that for a union of arrows, the domain is equivalent to the intersection of each of the domains (i.e. the function can only accept values that each of the arrows can accept since we're not sure which arrow actually describes the value):

$$\boxed{\mathit{domain}((\sigma_1 \rightarrow \tau_1) \cup \ldots \cup (\sigma_n \rightarrow \tau_n)) = \sigma_1 \cap \ldots \cap \sigma_n}$$

With those two points in mind, we can deduce the domain of an *arbitrary* function type

$$\boxed{\mathsf{dom}^? : \mathrm{t} \to \mathrm{t}\ \text{or false}}$$

$$\begin{aligned}
\mathsf{dom}^?(\mathrm{t}) &= \text{false} \ \text{ if } \ \mathrm{t} <: \bot \\
\mathsf{dom}^?(\langle\_,\ \_,\ \mathrm{b}^{\to}\rangle) &= \mathsf{dom}(\bot, \mathrm{b}^{\to})
\end{aligned}$$

$$\boxed{\mathsf{dom} : \mathrm{t}\ \mathrm{b}^{\to} \to \mathrm{t}}$$

$$\begin{aligned}
\mathsf{dom}(\mathrm{t}, \mathbb{1}) &= \mathrm{t} \\
\mathsf{dom}(\mathrm{t}, \mathbb{0}) &= \top \\
\mathsf{dom}(\mathrm{t}, \langle \mathrm{s}_1 \to \mathrm{s}_2,\ \mathrm{b}_l^{\to},\ \mathrm{b}_m^{\to},\ \mathrm{b}_r^{\to}\rangle) &= \mathrm{t}_l \cup \mathrm{t}_m \cup \mathrm{t}_r \\
& \quad \text{where } \mathrm{t}_l = \mathsf{dom}(\mathrm{t} \cup \mathrm{s}_1, \mathrm{b}_l^{\to}) \\
& \quad \phantom{\text{where }} \mathrm{t}_m = \mathsf{dom}(\mathrm{t}, \mathrm{b}_m^{\to}) \\
& \quad \phantom{\text{where }} \mathrm{t}_r = \mathsf{dom}(\mathrm{t}, \mathrm{b}_r^{\to})
\end{aligned}$$

Figure 4.20: Domain calculation for function types

$$\tau^{\to} = \bigcup_{(P,N)\in\mathsf{dnf}(\tau^{\to})} \left(\left(\bigcap_{(\tau_1\to\tau_2)\in P} \tau_1 \to \tau_2 \right) \cap \left(\bigcap_{(\tau_1\to\tau_2)\in N} \neg(\tau_1 \to \tau_2) \right) \right)$$

is the following intersection of unions:

$$\bigcap_{(P,N)\in\mathsf{dnf}(\tau^{\to})} \left(\bigcup_{(\tau_1\to\tau_2)\in P} \tau_1 \right)$$

Implementing Function Domain

We perform those domain calculations with the functions defined in figure 4.20. `dom`$^?$ first checks if the type is indeed a function (i.e. is it a subtype of `Any`$^{\to}$), if so it then calls `dom` with the function portion of the type ($\mathrm{b}^{\to}$) to begin traversing the BDD calculating the intersection of the union of the respective domains.

4.4.3 Function Application

When applying an arbitrary function to a value, we must be able to determine the type of the result. If the application is simple, e.g. a function of type `Int` $\to$ `Str` applied to an argument of type `Int`, calculating the result (`Str`) is trivial. However, when we are dealing with an arbitrarily complicated function type which could contain set-theoretic connectives, deciding the return type is a little more complicated. As we did in the previous section, let us again reason separately about how we might apply intersections and unions of function types to guide our intuition.

In order to apply a union of arrow types $(\sigma_1 \to \tau_1) \cup \ldots \cup (\sigma_n \to \tau_n)$, the argument type σ of course would have to be in the domain of each arrow, i.e. $\sigma <: \sigma_1 \cap \ldots \cap \sigma_n$ (see the discussion in the previous section). The result type of the application would then be the union of the ranges:

$$\boxed{\begin{array}{c} \mathit{apply}((\sigma_1 \to \tau_1) \cup \ldots \cup (\sigma_n \to \tau_n), \sigma) = \tau_1 \cup \ldots \cup \tau_n \\ \text{where } \sigma <: \sigma_1 \cap \ldots \cap \sigma_n \end{array}}$$

This corresponds to the logical observation that if we know that *either* P implies Q *or* R implies S, and we know that *both* P *and* R hold, then we can conclude that either Q or S holds. Now consider the problem of applying an intersection of arrow types. If for some intersection of arrows, there is a subset $(\sigma_1 \to \tau_1) \cap \ldots \cap (\sigma_n \to \tau_n)$ which all can be applied to the argument type σ (i.e., if $\sigma <: \sigma_1 \cap \ldots \cap \sigma_n$) then we get the following as the result:

$$\boxed{\begin{array}{c} \mathit{apply}((\sigma_1 \to \tau_1) \cap \ldots \cap (\sigma_n \to \tau_n), \sigma) = \tau_1 \cap \ldots \cap \tau_n \\ \text{where } \sigma <: \sigma_1 \cap \ldots \cap \sigma_n \end{array}}$$

This more or less corresponds to the logical observation that if we know that *both* P implies Q *and* R implies S, and we know that *both* P *and* R hold, then we can conclude that *both* Q *and* S hold. Finally, sometimes for a given intersection of arrows there may not be a single arrow that can handle a particular argument type, but some collection of those arrows certainly could together. E.g., consider an argument type σ, an intersection of arrows $(\sigma_1 \to \tau_1) \cap \ldots \cap (\sigma_n \to \tau_n)$, and the assumption that only collectively these arrows can cover the argument type (i.e., $\sigma <: \sigma_1 \cup \ldots \cup \sigma_n$). In this case, the resulting type of an application would look like the following:

$$\boxed{\begin{array}{c} apply((\sigma_1 \to \tau_1) \cap \ldots \cap (\sigma_n \to \tau_n),\, \sigma) = \tau_1 \cup \ldots \cup \tau_n \\ \text{where } \sigma <: \sigma_1 \cup \ldots \cup \sigma_n \end{array}}$$

This can be seen as corresponding to the logical observation that if we know that *both* P implies Q and R implies S, and we know that *either* P or R holds, then we can conclude that *either* Q *or* S hold.

By combining all of these lines of reasoning we can deduce that when considering an arbitrary function type

$$\boxed{\tau^{\to} = \bigcup_{(P,N) \in \mathsf{dnf}(\tau^{\to})} \left(\left(\bigcap_{(\tau_1 \to \tau_2) \in P} \tau_1 \to \tau_2 \right) \wedge \left(\bigcap_{(\tau_1 \to \tau_2) \in N} \neg(\tau_1 \to \tau_2) \right) \right)}$$

being applied to an argument of type σ, we first verify that σ is in the domain of $\tau^{\to}$ (i.e. using $\mathsf{dom}^?$ for example) and then calculate the result type of the application as follows:

$$\boxed{\bigcup_{(P,N) \in \mathsf{dnf}(\tau^{\to})} \left(\bigcup_{P' \subseteq P} \left(\begin{array}{ll} \text{if} & \sigma <: \left(\bigcup_{(\tau_1 \to \tau_2) \in P \setminus P'} \tau_1 \right) \\ \text{then} & \mathtt{Empty} \\ \text{else} & \bigcap_{(\tau_1 \to \tau_2) \in P'} \tau_2 \end{array} \right) \right)}$$

Basically, we traverse each clause in the DNF of the function type (i.e. each pair (P, N)) unioning the results. In each clause (P, N), we consider each possible set of arrows P' in P and only consider those which would necessarily have to handle a value of type σ (i.e. when it is not the case that the arrows in $P \setminus P'$ could handle the argument). For those sets P' that would necessarily handle the input, we intersect their arrows' codomains (otherwise we ignore the set by returning `Empty` for that clause). This reasoning resembles that which was required to decide function type inhabitation (see section 4.3.1), i.e. both are considering which combinations of arrows necessarily need to be considered to perform the relevant calculation.

Implementing Function Application

Figure 4.21 describes the functions which calculate the result type for function application. $\mathsf{apply}^?$ first ensures that the alleged function type is indeed a function with the appropriate

$$\boxed{\mathsf{apply}^{?} : \mathrm{t}\ \mathrm{t} \rightarrow \mathrm{t} \text{ or false}}$$

$$\begin{array}{rcl}
\mathsf{apply}^{?}(\tau_f, \tau_a) & = & \text{false} \quad \text{if } \tau_a \not<: \mathsf{dom}^{?}(\tau_f) \\
\mathsf{apply}^{?}(\langle\!\langle \_, \_, \mathrm{b}^{\rightarrow} \rangle\!\rangle, \tau_a) & = & \mathsf{apply}(\tau, \mathbb{1}, \mathrm{b}^{\rightarrow})
\end{array}$$

$$\boxed{\mathsf{apply} : \mathrm{t}\ \mathrm{t}\ \mathrm{b}^{\rightarrow} \rightarrow \mathrm{t} \text{ or false}}$$

$$\begin{array}{rcl}
\mathsf{apply}(\mathrm{t}_a, \mathrm{t}, \mathbb{0}) & = & \bot \\
\mathsf{apply}(\mathrm{t}_a, \mathrm{t}, \mathrm{b}^{\rightarrow}) & = & \bot \quad \text{if } \mathsf{empty}(\mathrm{t}_a) \text{ or } \mathsf{empty}(\mathrm{t}) \\
\mathsf{apply}(\mathrm{t}_a, \mathrm{t}, \mathbb{1}) & = & \mathrm{t} \\
\mathsf{apply}(\mathrm{t}_a, \mathrm{t}, \langle\!\langle \mathrm{s}_1 \rightarrow \mathrm{s}_2, \mathrm{b}_l^{\times}, \mathrm{b}_m^{\times}, \mathrm{b}_r^{\times} \rangle\!\rangle) & = & \mathrm{t}_{l1} \cup \mathrm{t}_{l2} \cup \mathrm{t}_m \cup \mathrm{t}_r \\
 & & \text{where } \mathrm{t}_{l1} = \mathsf{apply}(\mathrm{t}_a, \mathrm{t} \cap \mathrm{s}_2, \mathrm{b}_l^{\times}) \\
 & & \qquad\quad \mathrm{t}_{l2} = \mathsf{apply}(\mathrm{t}_a \setminus \mathrm{s}_1, \mathrm{t}, \mathrm{b}_l^{\times}) \\
 & & \qquad\quad \mathrm{t}_m = \mathsf{apply}(\mathrm{t}_a, \mathrm{t}, \mathrm{b}_m^{\times}) \\
 & & \qquad\quad \mathrm{t}_r = \mathsf{apply}(\mathrm{t}_a, \mathrm{t}, \mathrm{b}_r^{\times})
\end{array}$$

Figure 4.21: Function application result type calculations

domain before calling apply to calculate the result type of the application. apply then traverses the BDD combining recursive results via union. As it traverses down a BDD node's left edge (i.e. when a function type *is* a member of a set P) it makes two recursive calls: one for when that arrow *is* in P' (where we intersect the arrow's range s_2 with the result type accumulator t) and one for when it *is not* in P' (where we subtract s_1 from the argument type parameter t_a to track if the arrows in $P \setminus P'$ can handle the argument type). At non-trivial leaves where t_a is not empty (i.e. when we're considering a set of arrows P which necessarily *would* need to handle the argument) we return the accumulated range type (t) for that set of arrows. Note that we can "short-circuit" the calculation when either of the accumulators (t or t_a) are empty, which can be important for large function types since it frequently greatly reduces the search space.

4.5 Strategies for Testing

For testing an implementation of the data structures and algorithms described in this tutorial there are some convenient properties we can leverage:

1. any type generated by the grammar of types in figure 4.1 is a valid type;

2. since these types logically correspond to sets, we can create tests based on the many

well-known properties about sets to help ensure our types behave correctly; and

3. we have “naive”, inefficient mathematical descriptions of many of the algorithms in addition to more efficient algorithms which purport to perform the same calculation.

With these properties in mind, in addition to creating simple hand-written “unit tests”, we can easily use a tool such as QuickCheck [55] to generate random types and verify our implementation behaves properly. Additionally, we can write two implementations of each algorithm that has both a naive (i.e. more mathematical) and efficient description and feed them random input while checking that their outputs are always equivalent. This approach helped us discover several subtle bugs in our initial implementation at various points that simpler hand-written unit tests had not yet exposed and helped us be confident that there were not typos in the key mathematical equations we were basing our reasoning on.

4.6 Related Work

In this section we discuss other works which may be useful when implementing a system with semantic subtyping, systems which have unique implementations of semantic subtyping for first-order languages, features which have been explored in the context of semantic subtyping but go beyond the simple language we describe in thsi chapter, and approaches in syntactic subtyping which increase the completeness of subtyping with set-theoretic types.

4.6.1 Other Tutorials and Overviews

This chapter was partially written to help the authors better understand the implementation details of semantic subtyping and partially because of the relative paucity of “boots on the ground” accounts of working with such systems. ℂDuce—as far as we are aware—is the only programming language to date to feature sound and complete subtyping for the spectrum of types given in figure 4.1[53]. As impressive as that system is, its implementation is a nontrivial library of highly optimized OCaml code and thus perhaps not the best instructional resource for recreating the features or understanding why they work. Alain Frisch’s dissertation[56] is said to include detailed accounts of many of the lessons learned while working on ℂDuce, however this work is written in French and the authors of this work ne comprend pas le français. Luckily for us, Giuseppe Castagna has carefully extracted a significant amount of the implementation knowledge from these experiences and included it in an extremely helpful unpublished manuscript[57]. As we mentioned previously, the vast majority of the implementation techniques we discuss in this chapter came from our reading that manuscript and attempting to put the ideas into practice. We felt documenting our understanding and including extremely specific implementation details might be another useful point of reference for future researchers or enthusiasts wishing to implement such a system.

4.6.2 First-order or incomplete semantic subtyping

There has been a history of semantic subtyping work prior to the 2008 article by Frisch et al. [28] which has involved languages with semantic subtyping *without* first-class functions and the like. We will not review all of those works here (see the related works section in Frisch et al.'s journal article for a thorough summary); instead we will mention a few subsequent works whose implementation details are of possible interest.

Bierman et al. [21] show how semantic subtyping for a first-order language with refinement types can be achieved by deciding subtyping via an external SMT solver. In this approach, deciding whether τ is a subtype of σ involves deciding whether the first-order formula interpretation of τ implies the first-order formula interpretation of σ.

The Whiley programming language [16] features intersection, union, and negation types in a flow sensitive type system and boasts sound and complete reasoning about these connectives. Function types, however, are not included in the system and thus many functional idioms are impossible to express in this context. Nevertheless, it may be edifying to peruse implementation insights and developments in that space. Recently, for example, the developers have explored how declarative rewriting can, for the most part, capture the type semantics of the original system[58]. They admit it is not immediately clear, however, if this declarative rewriting approach could be extended to reason about function types.

4.6.3 Semantic subtyping with additional features

In this chapter we have discussed the most basic features necessary for implementing semantic subtyping for a functional language. Many additional features, however, have also been studied in this space.

The work by Frisch et al. [28] includes recursive types, which is easy to miss on a first glance since they define types coinductively rather than including an explicit recursive type constructor. Implementation wise, recursive types can be added by simply introducing cycles in the type data structures themselves and then keeping track of which types have already been "seen" while performing emptiness checks. If a previously seen type is encountered then the check simply halts and true is returned.

The remaining features we mention we have not implemented ourselves but are certainly worth noting: mutable state [59] and polymorphism[60, 61] have both been described in the literature and are supported today in the ℂDuce language; polymorphic variants (a la OCaml) have also been successfully combined with set-theoretic types and semantic subtyping to create a more expressive and intuitive system for programmers[27]; Castagna and Lanvin have explored how gradual typing might be combined with union and intersection types for a functional language[62]; Ancona et al. describe an approach for adapting semantic subtyping to work in languages with non-strict evaluation strategies[63]; and the π-calculus has also been studied in the context of semantic subtyping[64].

4.6.4 Expressive Syntactic Subtyping

There have been some advances in syntactic subtyping that help bring syntactic systems with set-theoretic types closer to what semantic subtyping offers in terms of subtyping expressiveness. In particular, Muehlboeck and Tate [65] describe an approach for empowering "textbook" algorithmic subtyping implementations with rich extensions called "integrated subtyping". This approach allows for certain subtyping properties–such as the distributivity of intersection over unions—to be clearly expressed and incorporated into the subtyping algorithm. While this approach may be a tractable path down the completeness spectrum for many languages, it is unclear how it would handle and/or scale with more advanced features such as negation types (which can play a key role in a language intent on reasoning set-theoretically). If we were to scale this approach to a level of completeness comparable with semantic subtyping, the lessons learned from implementing semantic subtyping may be necessary to keep the system efficient and tractable.

CHAPTER 5
A SET-THEORETIC FOUNDATION FOR OCCURRENCE TYPING

In section 2.2.8 we described the various techniques which have been used to support occurrence typing, which included simple syntactic reasoning, dependent types, and untagged union normalization. In this chapter, we describe a new approach—which we call *function application inversion*—based on the following observation: set-theoretic types are expressive enough to describe type predicates. For example, note that a standard type predicate can be *completely* described using set-theoretic type connectives:

$$(\tau \rightarrow \texttt{True}) \cap (\neg\tau \rightarrow \texttt{False}) \tag{5.1}$$

However, instead of merely pattern matching on types with the above schema, we explore a generic technique (function application inversion) for determining what type the input to a function must have been based on its observed output.

The remainder of this chapter is as follows: in section 5.1 we describe function application inversion intuitively and then mathematically, proving that it is both sound and complete; in section 5.2 introduce a calculus (λ_{SO}) which demonstrates how this technique, when coupled with with the full spectrum of set-theoretic types and semantic subtyping, can serve as a reasonable foundation for occurrence typing and is capable of type checking all of the occurrence typing examples listed in section 2.1; in section 5.3 we further examine this novel approach by discussing how a system like λ_{SO} handles the complex interfaces and idioms needed for typing Racket's numeric tower; in section 5.4 we discuss expressiveness tradeoffs between λ_{OT} and λ_{SO} like approaches to occurrence typing; finally in section 5.5 we discuss relevant related work.

5.1 Logical Inversion

Before outlining our general method for reasoning about predicate-like functions, it will be useful to review the fundamentally related "principle of inversion" from logic. Consider a propositional logic with the following introduction rule for $\wedge$ (i.e. conjunction):

$$\boxed{\dfrac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2}\ \wedge\text{–}\textsc{Intro}}$$

It stands to reason that if we can derive $\Gamma \vdash p_1 \wedge p_2$ and if $\wedge$–Intro is the only way to introduce a $\wedge$-proposition, then clearly there must exist derivations for $\Gamma \vdash p_1$ and $\Gamma \vdash p_2$ (how else could we have used $\wedge$–Intro to construct the conjunction?). In other words, given that we have a full accounting for how $\wedge$ is introduced, we can use "backwards reasoning"—i.e. determining what must have been true to arrive at a certain proof—to derive the following valid inference rule:

$$\boxed{\dfrac{\Gamma \vdash p_1 \wedge p_2 \qquad i \in \{1, 2\}}{\Gamma \vdash p_i}\ \wedge\text{–}\textsc{Elim}}$$

More generally speaking, given a set of rules $\{\mathcal{R}_1, ..., \mathcal{R}_n\}$ such that a certain formula φ can only be introduced by those rules

$$\boxed{\dfrac{\Gamma \vdash \mathcal{C}_1^1 \qquad \cdots \qquad \Gamma \vdash \mathcal{C}_1^j}{\Gamma \vdash \varphi}\ \mathcal{R}_1 \qquad \cdots \qquad \dfrac{\Gamma \vdash \mathcal{C}_n^1 \qquad \cdots \qquad \Gamma \vdash \mathcal{C}_n^k}{\Gamma \vdash \varphi}\ \mathcal{R}_n}$$

(where $\mathcal{C}_b^a$ is the a^{th} premise for rule $\mathcal{R}_b$) then we can admit this "*meta*-inference rule":

$$\boxed{\dfrac{\Gamma \vdash \varphi \qquad \Gamma, \mathcal{C}_1^1, \ldots, \mathcal{C}_1^j \vdash p \qquad \ldots \qquad \Gamma, \mathcal{C}_n^1, \ldots, \mathcal{C}_n^k \vdash p}{\Gamma \vdash p}}$$

This strategy of backward reasoning—which highlights the natural link between introduction and elimination rules—is known as the *inversion principle*. It was first introduced by Paul Lorenzen in 1950 [66] and has been used in a variety of ways since that time [67]. In the next section we introduce yet another use for this principle which seeks to reason backwards from a specific result type of a function application to determine what must have been true of the input to that function.

5.1.1 Function Application Inversion

As it turns out, we can effectively solve the problem of identifying "predicate-like" function types by being able to answer the following inversion-like question:

If applying a function of type τ_f to an argument produces a value of type σ_{out}, what type σ_{in} must the argument have been? (5.2)

In fact, we want to calculate the smallest such σ_{in}, since this question can always be trivially answered by returning the less-than-helpful type $domain(\tau_f)$ for σ_{in}. Equipped with such a function—which we will write $inv(\tau_f, \sigma_{out}) = \sigma_{in}$—we could determine what type a function is a predicate for (if any) by calculating $inv(\tau_f, \neg\texttt{False})$[1] and $inv(\tau_f, \texttt{False})$.

When considering how to calculate the inversion of a function application, first recall generally that to use the *inversion principle* we must know the set of rules $\{\mathcal{R}_1, ..., \mathcal{R}_n\}$ which could possibly produce the result we are interested in reasoning backwards from. In this case, the function type τ_f itself contains the rules that soundly describe how values of type $domain(\tau_f)$ can produce values of type σ_{out}. In other words, all of the arrows in the DNF of τ_f can be thought of as the various introduction forms which describe how input values in $domain(\tau_f)$ can produce output values.

One concern when thinking of function types as sets of rules may be that for a given function value f of type τ_f, there may exist a more precise function type τ'_f which gives a *more detailed* accounting for where values of type $domain(\tau_f)$ are mapped (i.e., $\tau'_f <: \tau_f$). Does this mean that reasoning based on the rules in τ_f will be flawed? Fortunately not. Our reasoning based on τ_f will still be valid (as it is still a proper type for f), but it may be the case that reasoning based on τ'_f would produce more precise predictions. For this reason it is always helpful to have precise function types when reasoning about what can be learned from the result of a function's application.

Another way of thinking about the question posed in (5.2) is to see function application

[1]We use $\neg$`False` instead of `True` because in the language we're imagining any non-false value is considered "truthy".

Function	τ_f	$inv(\tau_f, \neg\texttt{False})$	$inv(\tau_f, \texttt{False})$
`boolean?`	$\texttt{Bool} \rightarrow \texttt{True}$ $\cap \quad \neg\texttt{Bool} \rightarrow \texttt{False}$	$\texttt{Bool}$	$\neg\texttt{Bool}$
`file-stream-port?`	$\texttt{Port} \rightarrow \texttt{Bool}$ $\cap \quad \neg\texttt{Port} \rightarrow \texttt{False}$	$\texttt{Port}$	$\texttt{Any}$
`path-string?`	$\neg(\texttt{Path} \cup \texttt{Str}) \rightarrow \texttt{False}$ $\cap \quad \texttt{Path} \rightarrow \texttt{True}$ $\cap \quad \texttt{Str} \rightarrow \texttt{Bool}$	$\texttt{Path} \cup \texttt{Str}$	$\neg\texttt{Path}$

Figure 5.1: Function Application Inversion Examples

inversion as recovering implicit type-flow information from predicate-like function types of varying complexity, as demonstrated by the examples in figure 5.1. Inversion on the type of `boolean?` reaffirms it is a predicate on boolean values. Inversion on the type of `file-stream-port?` shows it returns true only for a subset of ports (not all ports are file streams). More complicated still, inversion on `path-string?`'s type shows it returns true for all Paths but only for a subset of strings (some strings do not correspond to valid OS file paths).

Given this intuition for how function application should behave, we proceed to describe how it can be calculated.

5.1.2 Algorithm Intuition

First, we recall—as shown in section 4.1.3—that any function type τ_f can be viewed as a union of intersections of arrows (i.e., it can be placed in DNF). Next, we must examine how each "rule" (i.e. each arrow $(\tau_1 \rightarrow \tau_2)$ in the DNF of τ_f) could contribute to producing a value of type σ_{out}. First, note that for a given intersection, if multiple arrows' domains handle a particular argument, then the result will be the intersection of those arrows' codomains:

$$\frac{\Gamma \vdash f : (\tau_1 \rightarrow \sigma_1) \cap \ldots \cap (\tau_n \rightarrow \sigma_n) \qquad \Gamma \vdash x : \tau_1 \cap \ldots \cap \tau_n}{\Gamma \vdash (f\ x) : \sigma_1 \cap \ldots \cap \sigma_n}$$

In order to determine the inversion of a function application we essentially examine each possible combination of arrows $(\tau_1 \rightarrow \sigma_1) \cap \ldots \cap (\tau_n \rightarrow \sigma_n)$ in τ_f, noting which combinations *could possibly* produce a value of type σ_{out} if each arrow applied to an argument (i.e. where $\sigma_1 \cap \ldots \cap \sigma_n \cap \sigma_{out} \not<:$ `Empty`) and which *could not possibly* produce a value of type σ_{out} if each arrow applied to an argument (i.e. where $\sigma_1 \cap \ldots \cap \sigma_n \cap \sigma_{out} <:$ `Empty`); we then subtract the domains of the latter group from the former. Stated differently, we start with the function's overall domain and subtract all input types which cannot possibly produce values of type σ_{out}. (Note that a slightly more efficient implementation would start with the particular *argument type* in question instead of the domain of the function when possible; this would allow us to ignore combinations of arrows which are provably irrelevant, trimming the search space and improving performance for large function types.)

5.1.3 Algorithm

With some intuition in place—i.e. that we are looking to subtract those input types which could not possibly produce values of type σ_{out} from the function's domain—we now look at how to perform this computation. Again, recall from section 4.1.3 that our function type τ_f can be structured as follows:

$$\boxed{\tau_f = (\mathtt{Any}^{\iota} \cap \tau_f^{\iota}) \cup (\mathtt{Any}^{\times} \cap \tau_f^{\times}) \cup (\mathtt{Any}^{\rightarrow} \cap \tau_f^{\rightarrow})}$$

We are only concerned with the function portion of the type $\tau_f^{\rightarrow}$ since for τ_f to be a function type, τ_f^{ι} and $\tau_f^{\times}$ must be equivalent to the empty type.

Since $\tau_f^{\rightarrow}$ is in DNF, we subtract from the domain of τ_f for each intersection of arrows P and for each possible non-empty subset of arrows $P' \subseteq P$ the intersection of the arrows' domains whose associated intersection of codomains does not overlap with σ_{out}. This gives us the following algorithm for computing function application inversion:

$$\boxed{inv : \tau\ \tau \to \tau}$$

$$inv(\tau_f, \sigma_{out}) = domain(\tau_f) \setminus \tau_a$$

$$\text{where } \tau_a = \bigcup_{(P,N)\in \mathsf{dnf}(\tau_f^{\rightarrow})} \left(\bigcup_{\emptyset \subsetneq P' \subseteq P} \left(\begin{array}{ll} \text{if} & \left(\bigcap_{(\tau_i\rightarrow\sigma_i)\in P'} \sigma_i\right) \cap \sigma_{out} <: \texttt{Empty} \\ \text{then} & \bigcap_{(\tau_i\rightarrow\sigma_i)\in P'} \tau_i \\ \text{else} & \texttt{Empty} \end{array} \right) \right)$$

Figure 5.2: Function Application Inversion Algorithm

Note that this algorithm assumes $\tau_f <: \texttt{Any}^{\rightarrow}$ (i.e. that the given function type is indeed a function type); a more general version would obviously need to first check this assumption. In the following section (5.1.4) we discuss the formal correctness properties for this algorithm; section 5.1.5 defines an efficient implementation of function application inversion ($\mathsf{inv}^?$) based on the implementation techniques discussed throughout chapter 4.

5.1.4 Soundness and Completeness

To prove our algorithm for function application inversion from figure 5.2 is correct, we define a mathematical relation $\mathbb{INV}$ which describes precisely what we intend function application inversion to mean:

Definition 1 (Function Application Inversion Relation)**.** *The function application inversion relation* $\mathbb{INV}$ *is a ternary relation on types, defined as the set of 3-tuples* $\langle \tau_f, \sigma_{out}, \sigma_{in} \rangle$ *such that for any values* v_f*,* v_a*, and* v*, if*

- v_f *is of type* τ_f*,*
- $\tau_f <: \texttt{Any}^{\rightarrow}$*,*
- v_a *is of type* $domain(\tau_f)$*,*
- $(v_f\ v_a)$ *reduces to* v*, and*
- v *is of type* σ_{out}*,*

then v_a is of type σ_{in}.

We can now state our soundness property for function application inversion, i.e. that all answers reported by the algorithm are valid predictions about the argument:

Theorem 3 (Function Application Inversion Soundness)**.** *For any types τ_f, σ_{out}, and σ_{in}, if τ_f <: $\mathsf{Any}^{\rightarrow}$ and $inversion(\tau_f, \sigma_{out}) = \sigma_{in}$, then $\langle \tau_f, \sigma_{out}, \sigma_{in} \rangle \in \mathbb{INV}$.*

Proof. By nested inductions on the DNF of τ_f. $\square$

The other property we are interested in is completeness, which in this context can be understood to mean that the type produced by function application inversion is the smallest such type and thus the most specific prediction about the argument possible:

Theorem 4 (Function Application Inversion Completeness)**.** *For any types τ_f, σ_{out}, and σ_{in}, if $\langle \tau_f, \sigma_{out}, \sigma_{in} \rangle \in \mathbb{INV}$, then $inversion(\tau_f, \sigma_{out})$ <: σ_{in}.*

Proof. By nested inductions on the DNF of τ_f. $\square$

These theorems have accompanying mechanized proofs, the Coq source code for which is found in appendix A.

5.1.5 Efficient Implementation

In figure 5.2 we describe how to calculate function application inversion. Here—in figure 5.3—we define a function $\mathsf{inv}^?$ to be an *efficient* implementation of function application inversion, after the manner of functions defined throughout chapter 4 based on the type representation presented in figure 4.9.

First, $\mathsf{inv}^?$ checks if its “function argument” t_f is indeed a function (returning false if not) before calling inv with the specified output type s_o, function domain t_d,[2] function portion of the type $\mathrm{b}^{\rightarrow}$, and an initially empty accumulator for the set of positive arrow types seen along a particular path in $\mathrm{b}^{\rightarrow}$.

inv then simply traverses the BDD $\mathrm{b}^{\rightarrow}$ which effectively considers each intersection of arrows in the type’s DNF: for absurd paths in the BDD we return the uninhabited type;

[2]As noted previously, passing the argument type instead of the domain here results in a more precise result and reduces the search space.

$$\boxed{\mathsf{inv}^{?} : \mathrm{t}\ \mathrm{s} \to \mathrm{t} \text{ or false}}$$

$$\begin{array}{rcl}
\mathsf{inv}^{?}(\mathrm{t}_f,\, \mathrm{s}_o) & = & \text{false} \quad \text{if } \mathrm{t}_f \not<: \top^{\rightarrow} \\
\mathsf{inv}^{?}(\langle\!\langle \_,\, \_,\, \mathrm{b}^{\rightarrow} \rangle\!\rangle,\, \mathrm{s}_o) & = & \mathsf{inv}^{\mathrm{s}_o}_{\mathrm{t}_d}(\mathrm{b}^{\rightarrow},\, \emptyset) \\
 & & \text{where } \mathrm{t}_d = \mathsf{dom}(\bot, \mathrm{b}^{\rightarrow})
\end{array}$$

$$\boxed{\mathsf{inv} : \mathrm{s}\ \mathrm{t}\ \mathrm{b}^{\times}\ P \to \mathrm{t}}$$

$$\begin{array}{rcl}
\mathsf{inv}^{\mathrm{s}_o}_{\mathrm{t}_d}(\mathbb{0},\, P) & = & \bot \\
\mathsf{inv}^{\mathrm{s}_o}_{\mathrm{t}_d}(\mathbb{1},\, P) & = & \mathrm{t}_d \setminus \gamma(\mathrm{t}_d,\, \mathrm{s}_o,\, P) \\
\mathsf{inv}^{\mathrm{s}_o}_{\mathrm{t}_d}(\langle\!\langle \mathrm{t}_1 \to \mathrm{t}_2,\, \mathrm{b}^{\rightarrow}_l,\, \mathrm{b}^{\rightarrow}_m,\, \mathrm{b}^{\rightarrow}_r \rangle\!\rangle,\, P) & = & \mathrm{t}_l \cup \mathrm{t}_m \cup \mathrm{t}_r \\
 & & \text{where } \mathrm{t}_l = \mathsf{inv}^{\mathrm{s}_o}_{\mathrm{t}_d}(\mathrm{b}^{\rightarrow}_l,\, \{\mathrm{t}_1 \to \mathrm{t}_2\} \cup P) \\
 & & \qquad\quad \mathrm{t}_m = \mathsf{inv}^{\mathrm{s}_o}_{\mathrm{t}_d}(\mathrm{b}^{\rightarrow}_m,\, P) \\
 & & \qquad\quad \mathrm{t}_r = \mathsf{inv}^{\mathrm{s}_o}_{\mathrm{t}_d}(\mathrm{b}^{\rightarrow}_r,\, P)
\end{array}$$

$$\boxed{\gamma : \mathrm{t}\ \mathrm{t}\ P \to \mathrm{t}}$$

$$\begin{array}{rcl}
\gamma(\mathrm{t}_d,\, \mathrm{t}_c,\, \emptyset) & = & \mathrm{t}_d \text{ if } \mathrm{t}_c <: \bot \\
\gamma(\mathrm{t}_d,\, \mathrm{t}_c,\, \emptyset) & = & \bot \\
\gamma(\mathrm{t}_d,\, \mathrm{t}_c,\, \{\mathrm{t}_1 \to \mathrm{t}_2\} \cup P) & = & \mathrm{s}_1 \cup \mathrm{s}_2 \\
 & & \text{where } \mathrm{t}'_d = \mathrm{t}_1 \cap \mathrm{t}_d \\
 & & \qquad\quad \mathrm{t}'_c = \mathrm{t}_2 \cap \mathrm{t}_c \\
 & & \qquad\quad \mathrm{s}_1 = \begin{cases} \bot & \text{if } \mathrm{t}'_d <: \bot \\ \mathrm{t}'_d & \text{if } \mathrm{t}'_c <: \bot \\ \gamma(\mathrm{t}'_d,\, \mathrm{t}'_c,\, P) & \text{otherwise} \end{cases} \\
 & & \qquad\quad \mathrm{s}_2 = \gamma(\mathrm{t}_d,\, \mathrm{t}_c,\, P)
\end{array}$$

Figure 5.3: Efficient algorithm for function application inversion.

for non-trivial paths we subtract from the function's domain the result of calling the helper γ with the gathered set of arrows P along this particular path; and at non-leaf nodes we recurse into each subtree to consider each possible path and we union the results.

Auxiliary function γ takes the set of arrow types (the third argument) and traverses each possible subset of arrows, tracking the intersection of the domains in the first accumulator argument t_d and the intersection of the desired output type (i.e. the initial second argument value) with the codomains in the second accumulator argument t_c (progressively computing the intersections in figure 5.2). The computation for a particular combination of arrows is halted and the appropriate result is returned if either accumulator becomes empty along the way (i.e. the initial check for if t_c is empty and the checks for $\perp$ when defining s_1).

5.1.6 Conservative Function Application Inversion

One interesting question is whether function application inversion could be used without semantic subtyping (i.e. if we have set-theoretic types but incomplete subtyping). Let us, for example, consider a language which reasons soundly but incompletely about its types and which features unions and simple intersections (i.e., an interface $\mathcal{I} = \{\tau_1 \rightarrow \sigma_1, \ldots, \tau_n \rightarrow \sigma_n\}$) for describing function types. In this language, let us say that when a function with interface $\mathcal{I}$ is applied to an argument of type τ, the return type will be some σ_i where $(\tau_i \rightarrow \sigma_i) \in \mathcal{I}$ and $\tau <: \tau_i$. What would a less complete/conservative function application inversion function *cinv* look like in this context?

In order to perform function application inversion in such a context, we would need two sound (possibly incomplete) operations on types:

- $\mathsf{overlap} : \tau\ \tau \rightarrow \mathrm{bool}$, a binary relation on types which returns true if there *might* exist some value which has both those types and false if there is certainly no such value; and

- $\mathsf{diff} : \tau\ \tau \rightarrow \tau$, a binary operator on types which computes the difference between two types (i.e. subtracting the second argument from the first).

Note that in some sense these functions are essentially helping us deal with the lack of intersection and negation types. With two such functions, we could essentially use the *same*

$$\boxed{cinv : \mathcal{I}\ \tau \rightarrow \tau}$$

$$cinv(\mathcal{I}, \sigma_{out}) = \mathsf{diff}(domain(\mathcal{I}),\ \tau_a)$$

$$\text{where } \tau_a = \bigcup_{(\tau_i \rightarrow \sigma_i) \in \mathcal{I}} \left(\begin{array}{ll} \text{if} & \mathsf{overlap}(\sigma_i,\ \sigma_{out}) = \text{false} \\ \text{then} & \tau_i \\ \text{else} & \texttt{Empty} \end{array} \right)$$

Figure 5.4: Conservative Function Application Inversion Algorithm

algorithm from figure 5.2 except simplified to suit our needs: instead of considering each possible combinations of arrows as we did in the semantic context, we can simply consider each arrow individually as does our hypothetical language during function application.

The *cinv* algorithm in figure 5.4 really is just a conservative approximation of the original algorithm: we subtract potentially less type information from the domain type than the original algorithm does since we're considering a subset of original algorithm's cases and our overlap and diff may themselves be conservative approximations. Whether or not this conservative algorithm would be expressive enough in practice is difficult to tell, but it seems there is at least one obstacle worth noting: a system would need negation types to effectively describe type predicates in a function interface and few systems at the time of writing this document include them. If negation types were present then *cinv* would be able to identify standard predicate types.

5.2 Formal Language Model

In this section we introduce a calculus (λ_{SO}) which demonstrates how set-theoretic types, semantic subtyping, and function application inversion can serve as an expressive, powerful foundation for occurrence typing. λ_{SO} essentially combines the logical foundations of λ_{OT} from section 2.2—i.e. using a logical environment and propositions to inform the type system in a control flow sensitive way—with the full spectrum of set-theoretic types, semantic subtyping (discussed in chapter 4), and function application inversion (discussed in section 5.1.1). λ_{SO} avoids much of the underlying complexity inherited by systems based on occurrence typing [10, 68] by leveraging semantic subtyping to completely reason about the intersection and negation of types.

$v ::=$		**Values**	$c ::=$		**Constants**
	$\mid c$	constant value		$\mid \mathit{int}$	integer value
	$\mid (\lambda\{\mathcal{I}\}(x)\,e)$	function value		$\mid$ `true`	true value
$e ::=$		**Expressions**		$\mid$ `false`	false value
	$\mid x, y, z$	variables		$\mid \mathit{str}$	string value
	$\mid v$	values		$\mid \mathit{uop}$	primitive ops
	$\mid (e\ e)$	application	$\pi ::=$		**Paths**
	$\mid$ (`if` $e\ e\ e$)	conditional		$\mid v$	value path
$\tau, \sigma ::=$		**Types**		$\mid x$	variable path
	$\mid$ `Any`	universal type	$p, q ::=$		**Propositions**
	$\mid$ `Empty`	uninhabited type		$\mid \mathbb{tt}$	trivial prop
	$\mid$ `Int`	integer type		$\mid \mathbb{ff}$	absurd prop
	$\mid$ `Str`	string type		$\mid \pi \in \tau$	π is of type τ
	$\mid$ `True`	true types		$\mid p \wedge p$	conjunction
	$\mid$ `False`	false type		$\mid p \vee p$	disjunction
	$\mid \tau \rightarrow \tau$	arrow type	$o ::=$		**Symbolic Objects**
	$\mid \tau \cup \tau$	type union		$\mid \top^{o}$	null object
	$\mid \tau \cap \tau$	type intersection		$\mid \pi$	path object
	$\mid \neg\tau$	type negation	$R ::=$	$\langle \tau, p, q, o \rangle$	**Type-Results**
$\mathcal{I} ::=$	$\overrightarrow{\tau \rightarrow \sigma}$	**Function Interfaces**	$\Gamma ::=$	$\overrightarrow{p}$	**Type Env**

Figure 5.5: λ_{SO} Syntax

Like previous occurrence typing calculi, the λ_{SO} typing judgment assigns *type-results* to well-typed expressions instead of merely types:

$$\Gamma \vdash e : \langle \tau, p, q, o \rangle$$

This judgment states that in environment Γ

- e has type τ;
- if e evaluates to a non-`false` (i.e. treated as true) value, "then-proposition" p holds;
- if e evaluates to `false`, "else-proposition" q holds;
- e's value corresponds to the symbolic object o.

5.2.1 λ_{SO} Syntax

The syntax of terms, types, propositions, and other forms are given in figure 5.5.

Constants (c) are integers, strings, booleans, or unary primitive operations. The individual primitive operators are enumerated later in the specification of the semantics (figure 5.9).

Expressions (e) **and values** (v) describe a simple lambda calculus, consisting of constants, variables, functions, function application, and conditionals. λ-abstractions are annotated with an "interface" $\mathcal{I}$ which consists a series of function arrows that describe the function's behavior. A function's type is the intersection of the arrows in its interface.

Types (τ, σ) include the full spectrum of set-theoretic types described previously in figure 4.1 (sans pairs for simplicity). Again, since we are interpreting types semantically, they can simply be thought of as denoting the sets of values described in section 4.1.2.

Paths (π) describe the pure terms we want our logic to be able to make type-related statements about, i.e. the terms whose type we want to refine based on the control flow of our program. λ_{SO}—unlike similar previous occurrence typing calculi—includes values as valid paths. As we discuss later in section 5.2.4, this choice was made to prove soundness via the standard small-step approach.

Propositions (p, q) **and Environments** (Γ) are at the core of how we describe the types of program terms. An environment Γ is simply a set of logical propositions. A proposition p can be a standard propositional atom (i.e. the trivial and absurd propositions $\mathbb{tt}$ and $\mathbb{ff}$), a conjunction or disjunction, or a type-related proposition $\pi \in \tau$ which states that path π as type τ. Note that whereas previous calculi [9, 10, 68] used two propositional forms to describe the types of terms—one stating a path has some type and one stating a path does not have some type—λ_{SO} requires only one kind of type proposition since negative type information can be encoded directly in the type itself. I.e., $\pi \notin \tau$ can instead be written as $\pi \in \neg\tau$.

Symbolic Objects (o) allow us to state that an expression either corresponds to a particular path (i.e. a value who's type we wish to keep track of) or it does not and thus corresponds to the trivial symbolic object $\top^o$.

A **type-result** (R)—as suggested at the beginning of this section—is a 4-tuple $\langle \tau, p, q, o \rangle$ which lets the typing judgment state more information about well typed terms: they describe not only a terms type τ, but also their positive and negative propositions p and q

$$\boxed{\Gamma \vdash e : \mathrm{R}}$$

$$\textsc{T-Const}\quad \frac{}{\Gamma \vdash c : \Delta^{\mathrm{R}}(c)} \qquad \textsc{T-Var}\quad \frac{\Gamma \vdash x \in \tau}{\Gamma \vdash x : \langle \tau, x \in \neg\mathtt{False}, x \in \mathtt{False}, x \rangle}$$

$$\textsc{T-Abs}\quad \frac{\begin{array}{c} \forall(\sigma \to \sigma') \in \mathcal{I}.\ \Gamma, x \in \sigma \vdash e : \sigma' \\ \tau = \left(\bigcap_{(\sigma \to \sigma') \in \mathcal{I}} (\sigma \to \sigma') \right) \cap \neg\tau' \\ x \notin \mathsf{fvs}(\Gamma) \qquad \tau \not<: \mathtt{Empty} \end{array}}{\Gamma \vdash (\lambda\{\mathcal{I}\}(x)\, e) : \langle \tau, \mathbb{tt}, \mathbb{ff}, (\lambda\{\mathcal{I}\}(x)\, e) \rangle}$$

$$\textsc{T-App}\quad \frac{\begin{array}{cc} \Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \langle \tau_2, \mathbb{tt}, \mathbb{tt}, o_2 \rangle \\ \tau_1 <: \tau_2 \to \tau & (\sigma_+, \sigma_-) = \mathsf{pred}(\tau_1, \tau_2) \end{array}}{\Gamma \vdash (e_1\ e_2) : \langle \tau, \mathsf{is}(o_2, \sigma_+), \mathsf{is}(o_2, \sigma_-), \top^{o} \rangle}$$

$$\textsc{T-If}\quad \frac{\Gamma \vdash e_1 : \langle \tau_1, p_1, q_1, o_1 \rangle \qquad p_1, \Gamma \vdash e_2 : \mathrm{R} \qquad q_1, \Gamma \vdash e_3 : \mathrm{R}}{\Gamma \vdash (\mathtt{if}\ e_1\ e_2\ e_3) : \mathrm{R}} \qquad \textsc{T-Sub}\quad \frac{\Gamma \vdash e : \mathrm{R}' \qquad \Gamma \vdash \mathrm{R}' <: \mathrm{R}}{\Gamma \vdash e : \mathrm{R}}$$

Figure 5.6: λ_{SO} Typing Judgment

and symbolic object o.

5.2.2 λ_{SO} Type System

The type system for λ_{SO} is described in figure 5.6.

T-Const type checks constants, consulting the Δ^{R} metafunction described in figure 5.7. In addition to assigning a type, this metafunction returns then- and else-propositions that are consistent with whether the constant is `false`. All constants have themselves as their symbolic object.

T-Var may assign any type τ to variable x so long as the $x \in \tau$ is provable in Γ. The then- and else-propositions reflect the self evident fact that if x evaluates to a non-`false` value then x is not of type `False`, otherwise it is of type `False`. The symbolic object informs the type system that this expression corresponds to the path x.

T-Abs, the rule for checking lambda abstractions, checks the body of the abstraction once for each arrow type $(\sigma \to \sigma')$ in $\mathcal{I}$. In each case, the environment is extended by assigning x the domain type σ and the body is checked to be well-typed at the codomain type σ'. The overall type of the abstraction τ is then the intersection of all of the arrows in $\mathcal{I}$ along with any desired negated type information τ', so long as τ is still an inhabited type;

this ability to add arbitrary negative type information to a lambda is standard practice in semantic subtyping calculi [28], as it ensures that all values are either in a type or its negation. We use the standard convention of choosing fresh names not currently bound when extending Γ with new bindings. The overall type-result for the lambda includes its type τ, then- and else-propositions stating the value is non-`false`, and the lambda value itself as the symbolic object. (Note that only well-typed lambdas are lifted into the space of symbolic objects during type checking, so we need not type check the bodies of lambda value paths—inspecting their interface will suffice.)

T-App handles function application, first checking that e_1 and e_2 are well-typed individually at some types τ_1 and τ_2. Then, it checks if τ_1 is a subtype of some function type $\tau_2 \to \tau$ (i.e., a function whose domain covers the type of e_2); the codomain (τ) of this function type will be the overall type of the application expression. The pred metafunction—described in figure 5.7—is used to determine what (if anything) is learned about the argument based on whether the result is non-`false` (σ_+) or `false` (σ_-). pred relies on the *inv* metafunction described in section 5.1.1 which performs function application inversion to determine this information. The then- and else-propositions describe in terms of the argument's symbolic object o_2 what is learned from the result (if anything), leveraging the is metafunction (also defined in figure 5.7) to deal sensibly with when $o_2 = \top^o$. The symbolic object for the application is simply $\top^o$ since we choose not to reason about complex expressions such as the result of arbitrary function applications.

T-If is used for conditionals, describing the important process by which information learned from evaluating test-expressions is projected into the respective branches. After ensuring e_1 is well-typed at some type, we make note of the then- and else-propositions p_1 and q_1. We then extend the environment with the appropriate proposition, dependent upon which branch we are checking: p_1 is assumed for checking the then-branch and q_1 for the else-branch. The type-result of a conditional is simply the type-result implied by both branches (which can be determined by subsuming their results via type-result subtyping).

T-Sub allows us to naturally abstract or refine a type-result R via subtyping and based on information in Γ; the details of the type-result subtyping relation are found in figure 5.8.

Well Formedness. For any judgment $\Gamma \vdash e : \mathrm{R}$, we require that the free variables in e

$$\boxed{\mathsf{pred} : \tau\ \tau \to \tau}$$

$$\mathsf{pred}(\tau_f, \tau_a) = (\sigma_+ \cap \tau_a,\ \sigma_- \cap \tau_a)$$
$$\text{where}\quad \sigma_+ = inv(\tau_f, \neg\texttt{False}) \qquad \sigma_- = inv(\tau_f, \texttt{False})$$

$$\boxed{\mathsf{is} : o\ \tau \to p}$$

$$\mathsf{is}(\pi,\ \tau) \qquad \pi \in \tau$$
$$\mathsf{is}(\top^o,\ \tau) = \begin{cases} \mathbb{ff} & \text{if } \tau <: \texttt{Empty} \\ \mathbb{tt} & \text{otherwise} \end{cases}$$

$$\boxed{\Delta^{\mathrm{R}} : \mathrm{R} \to \tau}$$

$$\begin{aligned} \Delta^{\mathrm{R}}(\texttt{false}) &= \langle \texttt{False}, \mathbb{ff}, \mathbb{tt}, \texttt{false} \rangle \\ \Delta^{\mathrm{R}}(c) &= \langle \Delta^{\tau}(c), \mathbb{tt}, \mathbb{ff}, c \rangle \qquad \text{if } c \neq \texttt{false} \end{aligned}$$

$$\boxed{\Delta^{\tau} : c \to \tau}$$

$$\begin{aligned} \Delta^{\tau}(int) &= \texttt{Int} \\ \Delta^{\tau}(str) &= \texttt{Str} \\ \Delta^{\tau}(\texttt{true}) &= \texttt{True} \\ \Delta^{\tau}(\texttt{false}) &= \texttt{False} \\ \Delta^{\tau}(\texttt{add1}) &= \texttt{Int} \to \texttt{Int} \\ \Delta^{\tau}(\texttt{sub1}) &= \texttt{Int} \to \texttt{Int} \\ \Delta^{\tau}(\texttt{strlen}) &= \texttt{Str} \to \texttt{Int} \\ \Delta^{\tau}(\texttt{not}) &= (\texttt{False} \to \texttt{True}) \cap (\neg\texttt{False} \to \texttt{False}) \\ \Delta^{\tau}(\texttt{int?}) &= (\texttt{Int} \to \texttt{True}) \cap (\neg\texttt{Int} \to \texttt{False}) \\ \Delta^{\tau}(\texttt{str?}) &= (\texttt{Str} \to \texttt{True}) \cap (\neg\texttt{Str} \to \texttt{False}) \\ \Delta^{\tau}(\texttt{zero?}) &= \texttt{Int} \to \texttt{Bool} \end{aligned}$$

Figure 5.7: λ_{SO} Type Metafunctions

and R be a subset of those found in Γ.

The **logic** for λ_{SO} (see figure 5.8) is a straightforward propositional logic with a few (highlighted) rules for reasoning about the types of terms. We will only describe the type-related rules since the others are entirely standard.

P-Sub allows us to use subtyping in the expected way when proving some path π has a particular type. I.e., since $\Gamma \vdash \pi \in \tau$ and τ is a subtype of σ, then π also is a σ.

P-Empty is similar to the principle of explosion (*ex falso quodlibet*), allowing us to prove anything if some path is of the uninhabited type.

P-Combine says that if we can prove a path π has both type τ and σ, then it must have type $\tau \cap \sigma$. This essentially lets us lift logical "and" into the type space. Furthermore, with proper intersections and negations, this simple rule replaces the several metafunctions used in previous work [9].

P-Val allows us to prove a value v has a type τ if the type system claims it does. In practice the only values we consider here are ones which have already been found to be

$\boxed{\Gamma \vdash p}$

$$\overset{\textsc{P-Atom}}{\dfrac{}{\Gamma, p \vdash p}} \qquad \overset{\textsc{P-Trivial}}{\dfrac{}{\Gamma \vdash \mathbb{tt}}} \qquad \overset{\textsc{P-Sub}}{\dfrac{\Gamma \vdash \pi \in \tau \qquad \tau <: \sigma}{\Gamma \vdash \pi \in \sigma}} \qquad \overset{\textsc{P-Empty}}{\dfrac{\Gamma \vdash \pi \in \texttt{Empty}}{\Gamma \vdash p}}$$

$$\overset{\textsc{P-Combine}}{\dfrac{\Gamma \vdash \pi \in \tau \qquad \Gamma \vdash \pi \in \sigma}{\Gamma \vdash \pi \in \tau \cap \sigma}} \qquad \overset{\textsc{P-Val}}{\dfrac{\vdash v : \tau}{\Gamma \vdash v \in \tau}} \qquad \overset{\textsc{P-ExFalso}}{\dfrac{\Gamma \vdash \mathbb{ff}}{\Gamma \vdash p}} \qquad \overset{\textsc{P-AndE}}{\dfrac{\Gamma \vdash p_1 \wedge p_2 \qquad i \in \{1, 2\}}{\Gamma \vdash p_i}}$$

$$\overset{\textsc{P-AndI}}{\dfrac{\Gamma \vdash p \qquad \Gamma \vdash q}{\Gamma \vdash p \wedge q}} \qquad \overset{\textsc{P-OrE}}{\dfrac{\Gamma \vdash p_1 \vee p_2 \qquad \Gamma, p_1 \vdash q \qquad \Gamma, p_2 \vdash q}{\Gamma \vdash q}} \qquad \overset{\textsc{P-OrI}}{\dfrac{\Gamma \vdash p \qquad \text{or} \qquad \Gamma \vdash q}{\Gamma \vdash p \vee q}}$$

$\boxed{\tau <: \tau}$

$\tau <: \sigma \quad \text{iff} \quad [\![\tau]\!] \subseteq [\![\sigma]\!]$

$\boxed{o <: o}$

$o_1 <: o_2 \quad \text{iff} \quad o_1 = o_2 \quad \text{or} \quad o_2 = \top^o$

$\boxed{\Gamma \vdash \mathrm{R} <: \mathrm{R}}$

$$\dfrac{\begin{array}{c}\tau_1 <: \tau_2 \qquad o_1 <: o_2 \\ \mathsf{is}(o_1,\ \tau_1 \cap \neg\texttt{False}), p_1, \Gamma \vdash p_2 \\ \mathsf{is}(o_1,\ \tau_1 \cap \texttt{False}), q_1, \Gamma \vdash q_2\end{array}}{\Gamma \vdash \langle \tau_1, p_1, q_1, o_1 \rangle <: \langle \tau_2, p_2, q_2, o_2 \rangle}$$

Figure 5.8: λ_{SO} Logic and Subtyping

well-typed, and thus the checks are either trivial (for constants) or surface level (i.e. we can just inspect the interface of a well-typed lambda to determine its type).

Subtyping in λ_{SO} utilizes the standard semantic approach: τ is a subtype of σ if and only if the set of values τ denotes is a subset of the values σ denotes. Readers can find intuition and implementation details for this approach in the contents of chapter 4; previous work by Frisch et al. [28] lays out rigorous mathematical justifications. Suffice it here to recall that we can decide subtyping in a sound and complete manner for the entire spectrum of set-theoretic types. Object subtyping has $\top^o$ as the top object and is also reflexive.

The type-result subtyping allows us to adjust the type and symbolic object an expression has via subtyping and further allows us to refine the then- and else-propositions with the propositions in Γ and the knowledge that o_1 is of type τ_1.

5.2.3 λ_{SO} Semantics

The dynamic semantics of λ_{SO} are presented in figure 5.9 as a series of standard syntactic small-step reduction rules. $e \longrightarrow e'$ says that e steps to e' in a single reduction. Notably any non-`false` value is treated as "true" in conditional test expressions, SS-Beta uses standard capture-avoiding substitution to implement function application, and the partial metafunction δ (also defined in figure 5.9) is used by SS-UOp to describe how the language's unary primitives operate. We write $e \longrightarrow^* e'$ to mean the reflexive, transitive closure of the single-step relation, meaning e steps to e' in zero or more steps.

5.2.4 λ_{SO} Soundness

We prove soundness for λ_{SO} via the standard progress and preservation lemmas [69].

The statement of progress is straightforward: any well-typed term is either a value or can take a single step of evaluation.

Theorem 5 (λ_{SO} Progress)**.** *If* $\vdash e : \mathrm{R}$ *then either*

- $\exists v.\, e = v$*, or*
- $\exists e'.\, e \longrightarrow e'$*.*

$\boxed{e \longrightarrow e}$

SS-App1
$$\frac{e_1 \longrightarrow e_1'}{(e_1\ e_2) \longrightarrow (e_1'\ e_2)}$$

SS-App2
$$\frac{e_2 \longrightarrow e_2'}{(e_1\ e_2) \longrightarrow (e_1\ e_2')}$$

SS-UOp
$$\frac{v' = \delta(uop,\ v)}{(uop\ v) \longrightarrow v'}$$

SS-Beta
$$\frac{}{((\lambda\{\mathcal{I}\}(x)\, e)\ v) \longrightarrow e[x \mapsto v]}$$

SS-If
$$\frac{e_1 \longrightarrow e_1'}{(\texttt{if}\ e_1\ e_2\ e_3) \longrightarrow (\texttt{if}\ e_1'\ e_2\ e_3)}$$

SS-If-False
$$\frac{}{(\texttt{if}\ \texttt{false}\ e_2\ e_3) \longrightarrow e_3}$$

SS-If-NonFalse
$$\frac{v \neq \texttt{false}}{(\texttt{if}\ v\ e_2\ e_3) \longrightarrow e_2}$$

$\boxed{\delta : uop\ v \rightarrow v}$

$$\begin{aligned}
\delta(\texttt{add1},\ int) &= int + 1\\
\delta(\texttt{sub1},\ int) &= int - 1\\
\delta(\texttt{strlen},\ str) &= |str|\\
\delta(\texttt{not},\ v) &= \begin{cases}\texttt{true} & \text{if } v = \texttt{false}\\ \texttt{false} & \text{otherwise}\end{cases}\\
\delta(\texttt{int}?,\ v) &= \begin{cases}\texttt{true} & \text{if } \exists int.\, v = int\\ \texttt{false} & \text{otherwise}\end{cases}\\
\delta(\texttt{str}?,\ v) &= \begin{cases}\texttt{true} & \text{if } \exists str.\, v = str\\ \texttt{false} & \text{otherwise}\end{cases}\\
\delta(\texttt{zero}?,\ int) &= \begin{cases}\texttt{true} & \text{if } int = 0\\ \texttt{false} & \text{otherwise}\end{cases}
\end{aligned}$$

Figure 5.9: λ_{SO} Small-step Reduction Semantics

Proof. By induction on the typing derivation for e. $\square$

Our substitution lemma is a little more complicated. We first describe what it says and then discuss why this particular framing is needed: it states that if an expression e is well typed at R in a context Γ', and if that context is implied by the context $\Gamma, z \in \tau_1$ (where z does not occur free in Γ), then in context Γ the expression e with z consistently replaced by v will type check at R' which is just as or more precise than R when all its free occurrences of z are replaced by v.

Lemma 5 (λ_{SO} Substitution)**.** *Assuming*

- $\Gamma' \vdash e : \mathrm{R}$*,*
- $\vdash v : \tau_1$*,*
- $z \notin \mathsf{fvs}(\Gamma)$*, and*
- $\Gamma, z \in \tau_1 \vdash \Gamma'$*,*

then there exists an R' *such that*

- $\Gamma \vdash e[z \mapsto v] : \mathrm{R}'$ *and*
- $\Gamma \vdash \mathrm{R}' <: \mathrm{R}[z \mapsto v]$*.*

Proof. By induction on the typing derivation for e. $\square$

The reason for this particular phrasing stems from the fact that both our environments and type-results contain logical propositions which can discuss any in-scope identifiers. So with the environments, we are essentially saying that whatever Γ' says about z, these statements are true exactly when they are true for v since v is of type τ_1 (and recall that we can easily decide whether a well-typed value is in a particular type) and whatever Γ' says about terms that do not contain z is implied by Γ. Therefore the substitution $\mathrm{R}[z \mapsto v]$ is essentially converting all statements made by the type system which mention z into the appropriate atomic proposition—$\mathbb{tt}$ or $\mathbb{ff}$—based on whether or not they hold for v. This corresponds exactly to what is happening when we type check a term prior to performing the substitution mapping z to v: the type system can only prove claims about the expression which are provable in the environment without z or with the knowledge that z is of type τ_1, and so by replacing z by v the derivation on $e[z \mapsto v]$ remains valid in Γ with respect to that R'.

With substitution handled, preservation becomes straightforward, stating that evaluation preserves type, proposition, and object information.

Theorem 6 (λ_{SO} Preservation). *If* $\vdash e : \mathrm{R}$ *and* $e \longrightarrow e'$ *then there exists a* R' *such that* $\vdash e' : \mathrm{R}'$ *and* $\vdash \mathrm{R}' <: \mathrm{R}$.

Proof. By induction on the typing derivation for e. $\square$

If we allowed evaluation in arbitrary contexts further work would be required. I.e., because we are only considering the empty context at the top level, R and R′ contain no free variables, and thus the claims they make in the propositions and symbolic object are trivial.

Proof techniques comparison: λ_{SO} vs λ_{OT}

Previous proofs for occurrence typing calculi such as λ_{OT} [9] have used a unique model theoretic proof technique. In this approach runtime environments serve as the model in which satisfaction of a particular proposition or logical environment can be decided. The soundness theorem then states that values produced by the big-step evaluation semantics will have the correct type *and* any propositional claims made by the type system will be satisfied by the runtime environment. Although this technique is convenient and works well with the model-theoretic nature of the type system itself, it features the standard drawbacks of big-step soundness proofs by saying nothing of diverging or stuck terms. However, because type-results can contain free variables—in particular they may mention those which are eliminated by certain reduction steps—the more common small-step proof technique seemed ill fitted to reason about the soundness of such calculi. I.e., some reductions in these calculi would produce expressions which no longer contained terms the previous type-result made claims about. For these reasons, the less precise model theoretic big-step technique was favored.

In proving soundness for λ_{SO}, however, we discovered a compromise of sorts between these two proof techniques. Instead of using a big-step semantics and a model theoretic notion of satisfaction to decide soundness—where all free variables are replaced at once by their associated values from the runtime environment and the truth of all propositions becomes trivially decidable—we use a small-step semantics and replace the free variables (in both the program and in type system's claims) incrementally as we perform reduction steps. Essentially, by allowing propositions to talk about *values* in addition to variables, we end up more-or-less incrementally deciding whether or not the propositions in the type-result would be satisfied by evaluation one substitution at a time. This allows us to effectively reason about a calculus whose design is somewhat inspired by model theory with the standard tools from semantics engineering, thereby allowing us to make more precise claims about the soundness of the language.

5.2.5 Additional Language Features

Our presentation of λ_{SO} focused on the smallest set of features necessary to demonstrate how semantic subtyping and function application inversion could support occurrence typing. In particular, we omitted features like local binding and pairs, which were present in previous occurrence typing calculi. To support these features, in addition to extending the grammar of values and expressions in the natural ways, typing rules roughly equivalent to T-Let, T-Cons, T-First, and T-Second from Kent et al. [68] would need to be added along with paths describing product projections (e.g. a path $(proj\ i\ \pi)$ describing the i^{th} field of path π). Furthermore, since combining type information for a particular path is performed entirely within the semantics of the types themselves (i.e. there is no need for a metafunction similar to `update` from λ_{OT} which syntactically combines type information, since we simply intersect the types with P-Combine from figure 5.8) we would need to add a few logical rules such as the following to convert propositions into a normal form so they mention the same paths whenever possible:

$$\frac{\Gamma \vdash (proj\ 1\ \pi) \in \tau}{\Gamma \vdash \pi \in \tau \times \texttt{Any}}\ \textsc{P-Fst} \qquad \frac{\Gamma \vdash (proj\ 2\ \pi) \in \tau}{\Gamma \vdash \pi \in \texttt{Any} \times \tau}\ \textsc{P-Snd}$$

I.e., when we have a proposition about the projection of a path's field we can convert that into a proposition about the underlying path by pushing the type into the appropriate product field.

5.3 Semantic Numeric Tower

The motivation for exploring function application inversion stemmed from the observation that set-theoretic types appear well-suited for precisely describing standard type predicates. However, standard type predicates are not the only kind of functions which can inform a type checker in a flow sensitive way. For example, Typed Racket has made significant progress in uniquely leveraging occurrence typing and set-theoretic types to precisely describe Racket's rich numeric tower[70]. In Racket, many numeric types contain many different distinct kinds of numeric values, e.g. a `Number` can be an integer, an exact rational, and inexact IEEE floating point number, etc. Figure 5.10 gives a rough outline for how this numeric tower of types is constructed:

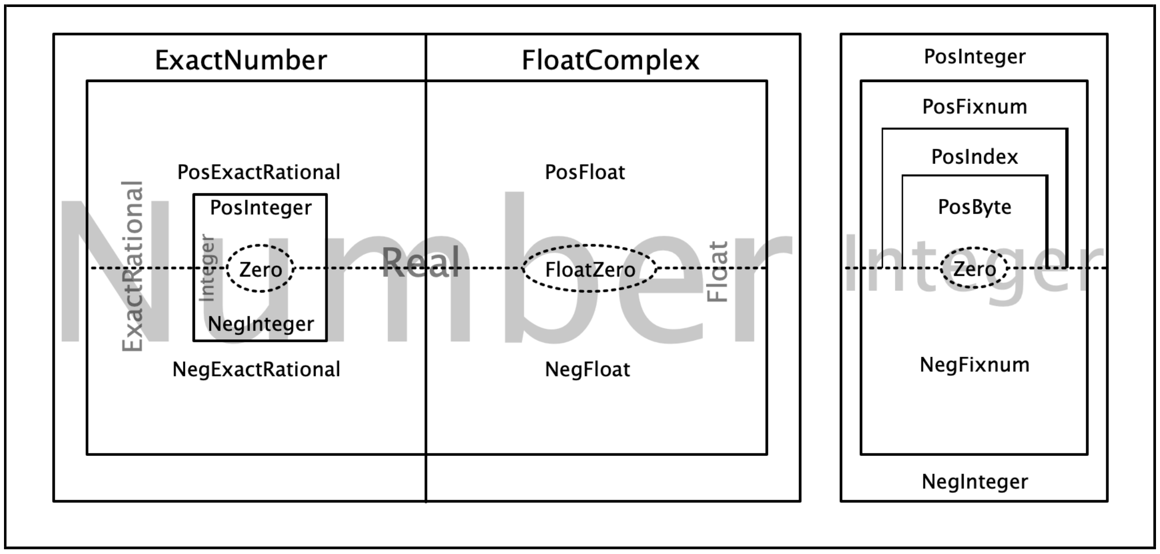


Figure 5.10: Numeric Tower Overview for `Number` (left) and `Integer` (right)

Essentially, the set of numeric values in Racket is partitioned into disjoint sets described by "numeric base types": `Zero` which covers the set {`0`}, `One` which covers {`1`}, `ByteLargerThanOne` which covers {`2`, ... , `255`}, etc. Then, unions of these base types are defined to describe natural sets of numbers one might want to reason about: `PosByte` is the union of `One` and `ByteLargerThanOne`, `Byte` is the union of `Zero` and `PosByte`, etc. The entire tower is given in appendix B for reference.

This tower of types lets Typed Racket precisely describe the behavior of many primitive numeric operations. For example, here is one of the types for the `<` operator:

$$\texttt{<} \in (x{:}\texttt{Zero}\ y{:}\texttt{Real} \rightarrow \langle \texttt{Bool}, y \in \texttt{PosReal}, y \in \texttt{NonnegReal}, \top^{o} \rangle)$$

This type says that whenever the first argument to less-than is `0` the result witnesses whether or not the second argument is positive. The full type for `<` is an intersection of *many* such arrows, describing what can be learned when comparing various subsets of the real numbers. With these sorts of precise types and occurrence typing, numeric programs can more accurately describe their semantics at the type level. For example, while many typed languages describe the absolute value function as a unary operator on a particular numeric type, in Typed Racket we can give a type which includes the expected sign information:

```
(: absolute-value (-> Real Nonnegative-Real))
(define (absolute-value x)
  (if (< 0 x)
      x
      (- 0 x)))
```

Here, the type system learns from the test expression `(< 0 x)` that `x` is positive in the then-branch and nonpositive in the else-branch, which allows the type checker to guarantee that each branch produces a nonnegative real number.

5.3.1 Semantic Types for Comparison Operators

Now we examine how function application inversion can also express the aforementioned unique numeric occurrence typing found in Typed Racket. Recall that in Typed Racket this is achievable because an intersection of arrows can include cases for specific argument combinations of interest, to which it assigns latent propositions stating what would be

$$
\begin{array}{ll}
 & (x : \texttt{Integer})(y : \texttt{One}) \to \langle \texttt{Bool}, x \in \texttt{NonposInteger}, x \in \texttt{PosInteger}, \top^o \rangle \\
\cap & (x : \texttt{Real})(y : \texttt{Zero}) \to \langle \texttt{Bool}, x \in \texttt{NegReal}, x \in \texttt{NonnegReal}, \top^o \rangle \\
\cap & (x : \texttt{Zero})(y : \texttt{Real}) \to \langle \texttt{Bool}, x \in \texttt{PosReal}, x \in \texttt{NonposReal}, \top^o \rangle \\
\cap & (x : \texttt{Real})(y : \texttt{RealZero}) \to \langle \texttt{Bool}, x \in \texttt{NegReal}, \mathbb{tt}, \top^o \rangle \\
\cap & (x : \texttt{RealZero})(y : \texttt{Real}) \to \langle \texttt{Bool}, x \in \texttt{PosReal}, \mathbb{tt}, \top^o \rangle \\
\cap & (x : \texttt{Byte})(y : \texttt{PosByte}) \to \langle \texttt{Bool}, \mathbb{tt}, x \in \texttt{PosByte}, \top^o \rangle \\
\cap & (x : \texttt{Byte})(y : \texttt{Byte}) \to \langle \texttt{Bool}, y \in \texttt{PosByte}, \mathbb{tt}, \top^o \rangle \\
\cap & \ldots
\end{array}
$$

Figure 5.11: (Partial) Syntactic Type of `<` (7 of 88 arrows shown)

learned about the arguments depending on whether the result was `false`. In figure 5.11 we list 7 of the 88 arrows Typed Racket uses to describe the binary cases for the `<` operator to get a better sense of what this looks like.

With function application inversion, however, we do not have latent propositions to describe what can be learned in certain cases. Instead we need to *directly* encode the functional details we want the type system to know about. We do this—as we did with simple type predicates—by clearly indicating which values will be mapped to which result. E.g., for `(< x y)`, if `x` is negative and `y` is nonnegative, then we know the result will be `true`. Or if we know either is `+nan.0` (i.e. the IEEE floating point "not a number" value), then the result will be **`false`**. When we do this for all the relevant combinations of arguments we get a function type with 23 arrows shown in figure 5.12.

The semantic type for `<` in figure 5.12—when used in the expressive context of semantic subtyping and function application inversion—is just as expressive as the Typed Racket version with 88 arrows. I.e., given any combination of arguments with numeric tower types the semantic approach always produces a prediction about those arguments that is just as or more precise than the prediction given by the syntactic type Typed Racket uses.

$$\begin{array}{ll}
 & \texttt{NaN} \times \texttt{Real} \longrightarrow \texttt{False}^{1} \\
\cap & \texttt{Real} \times \texttt{NaN} \longrightarrow \texttt{False}^{1} \\
\cap & (\texttt{NonposReal} \cap \neg\texttt{NaN}) \times (\texttt{PosReal} \cap \neg\texttt{NaN}) \longrightarrow \texttt{True}^{2} \\
\cap & \texttt{PosReal} \times \texttt{NonposReal} \longrightarrow \texttt{False}^{2} \\
\cap & (\texttt{NegReal} \cap \neg\texttt{NaN}) \times \texttt{RealZeroNoNaN} \longrightarrow \texttt{True}^{2} \\
\cap & \texttt{RealZero} \times \texttt{NegReal} \longrightarrow \texttt{False}^{2} \\
\cap & \texttt{NegInfinity} \times (\texttt{Real} \cap \neg\texttt{NaN} \cap \neg\texttt{NegInfinity}) \longrightarrow \texttt{True}^{3} \\
\cap & \texttt{Real} \times \texttt{NegInfinity} \longrightarrow \texttt{False}^{3} \\
\cap & (\texttt{Real} \cap \neg\texttt{NaN} \cap \neg\texttt{PosInfinity}) \times \texttt{PosInfinity} \longrightarrow \texttt{True}^{3} \\
\cap & \texttt{PosInfinity} \times \texttt{Real} \longrightarrow \texttt{False}^{3} \\
\cap & \texttt{NegIntegerNotFixnum} \times (\texttt{Integer} \cap \neg\texttt{NegIntegerNotFixnum}) \longrightarrow \texttt{True}^{4} \\
\cap & (\texttt{Integer} \cap \neg\texttt{NegIntegerNotFixnum}) \times \texttt{NegIntegerNotFixnum} \longrightarrow \texttt{False}^{4} \\
\cap & \texttt{RealZero} \times \texttt{RealZero} \longrightarrow \texttt{False}^{4} \\
\cap & \texttt{One} \times \texttt{One} \longrightarrow \texttt{False}^{4} \\
\cap & \texttt{One} \times (\texttt{PosInteger} \cap \neg\texttt{One}) \longrightarrow \texttt{True}^{4} \\
\cap & (\texttt{PosInteger} \cap \neg\texttt{One}) \times \texttt{One} \longrightarrow \texttt{False}^{4} \\
\cap & \texttt{Byte} \times (\texttt{PosInteger} \cap \neg\texttt{Byte}) \longrightarrow \texttt{True}^{4} \\
\cap & (\texttt{PosInteger} \cap \neg\texttt{Byte}) \times \texttt{Byte} \longrightarrow \texttt{False}^{4} \\
\cap & \texttt{Index} \times (\texttt{PosInteger} \cap \neg\texttt{Index}) \longrightarrow \texttt{True}^{4} \\
\cap & (\texttt{PosInteger} \cap \neg\texttt{Index}) \times \texttt{Index} \longrightarrow \texttt{False}^{4} \\
\cap & \texttt{Fixnum} \times \texttt{PosIntegerNotFixnum} \longrightarrow \texttt{True}^{4} \\
\cap & \texttt{PosIntegerNotFixnum} \times \texttt{Fixnum} \longrightarrow \texttt{False}^{4} \\
\cap & \texttt{Real} \times \texttt{Real} \longrightarrow \texttt{Bool}^{5}
\end{array}$$

Key NaN^{1} sign^{2} $\pm$ infinity^{3} finite bounds^{4} cumulative type^{5}

Figure 5.12: Semantic Type of `<`

5.3.2 Semantic Types for Other Numeric Operators

In addition to examining how a system like λ_{SO} could handle non-standard predicates such as numeric comparisons, we also examined how the types of some simple mathematical operators would be expressed in a context with semantic subtyping. For example, the Racket `add1` function simply adds `1` to the value of its argument, but the kind of numeric value that is returned will depend on the input. Figure 5.13 gives the syntactic (i.e. Typed Racket) type for `add1`. It contains 23 individual arrows describing where various numeric values are mapped. Figure 5.14 contains the semantic type for `add1`, which consists of 14 arrows. In appendix B we give the full syntactic and semantic types for the `+` operator, which is interesting to compare but whose size make them ill-suited to include here.

$$
\begin{aligned}
& \texttt{Zero} \longrightarrow \texttt{One} \\
\cap\ & \texttt{One} \longrightarrow \texttt{PosByte} \\
\cap\ & \texttt{Byte} \longrightarrow \texttt{PosIndex} \\
\cap\ & \texttt{Index} \longrightarrow \texttt{PosFixnum} \\
\cap\ & \texttt{NegFixnum} \longrightarrow \texttt{NonposFixnum} \\
\cap\ & \texttt{NonposFixnum} \longrightarrow \texttt{Fixnum} \\
\cap\ & \texttt{NonnegInteger} \longrightarrow \texttt{PosInteger} \\
\cap\ & \texttt{NegInteger} \longrightarrow \texttt{NonposInteger} \\
\cap\ & \texttt{Integer} \longrightarrow \texttt{Integer} \\
\cap\ & \texttt{NonnegRational} \longrightarrow \texttt{PosRational} \\
\cap\ & \texttt{Rational} \longrightarrow \texttt{Rational} \\
\cap\ & \texttt{NonnegFloat} \longrightarrow \texttt{PosFloat} \\
\cap\ & \texttt{Float} \longrightarrow \texttt{Float} \\
\cap\ & \texttt{NonnegSingleFloat} \longrightarrow \texttt{PosSingleFloat} \\
\cap\ & \texttt{SingleFloat} \longrightarrow \texttt{SingleFloat} \\
\cap\ & \texttt{NonnegInexactReal} \longrightarrow \texttt{PosInexactReal} \\
\cap\ & \texttt{InexactReal} \longrightarrow \texttt{InexactReal} \\
\cap\ & \texttt{NonnegReal} \longrightarrow \texttt{PosReal} \\
\cap\ & \texttt{Real} \longrightarrow \texttt{Real} \\
\cap\ & \texttt{FloatComplex} \longrightarrow \texttt{FloatComplex} \\
\cap\ & \texttt{SingleFloatComplex} \longrightarrow \texttt{SingleFloatComplex} \\
\cap\ & \texttt{InexactComplex} \longrightarrow \texttt{InexactComplex} \\
\cap\ & \texttt{Number} \longrightarrow \texttt{Number}
\end{aligned}
$$

Figure 5.13: Syntactic Type of `add1`

$$
\begin{array}{ll}
 & \texttt{Zero} \longrightarrow \texttt{One}^{1} \\
\cap & \texttt{One} \longrightarrow \texttt{Byte}^{1} \\
\cap & \texttt{Byte} \longrightarrow \texttt{Index}^{1} \\
\cap & \texttt{Index} \longrightarrow \texttt{Fixnum}^{1} \\
\cap & \texttt{NonposFixnum} \longrightarrow \texttt{Fixnum}^{1} \\
\cap & \texttt{NegInteger} \longrightarrow \texttt{NonposInteger}^{2} \\
\cap & \texttt{NonnegReal} \longrightarrow \texttt{PosReal}^{2} \\
\cap & \texttt{Integer} \longrightarrow \texttt{Integer}^{3} \\
\cap & \texttt{Rational} \longrightarrow \texttt{Rational}^{3} \\
\cap & \texttt{Float} \longrightarrow \texttt{Float}^{3} \\
\cap & \texttt{SingleFloat} \longrightarrow \texttt{SingleFloat}^{3} \\
\cap & \texttt{FloatComplex} \longrightarrow \texttt{FloatComplex}^{3} \\
\cap & \texttt{SingleFloatComplex} \longrightarrow \texttt{SingleFloatComplex}^{3} \\
\cap & \texttt{Number} \longrightarrow \texttt{Number}^{4}
\end{array}
$$

Key finite integer range^{1} sign^{2} compound^{3} cumulative type^{4}

Figure 5.14: Semantic Type of `add1`

5.3.3 Semantic/Syntactic Function Type Comparison

After confirming semantic subtyping and function application inversion were expressive enough to handle the unique idioms that have emerged in Typed Racket's numeric tower, we began investigating how the sizes of numeric function types compared more generally in these two settings. We examined 17 numeric operators in total: 6 unary numeric functions, 9 binary numeric functions, and 3 numeric comparison functions. In each case, the semantic version was able to produce equivalent or more precise result types for applicable numeric tower input types and required significantly fewer arrows. We summarize these results in figure 5.15.

The reason for this disparity in the number of arrows largely stems from the fact that Typed Racket (the "syntactic" system) uses a simple linear algorithm for computing the result type of function application: when given argument types, Typed Racket scans the list of arrows in order until an arrow whose domain covers the arguments is found and that arrow's codomain is used. The semantic approach, on the other hand, considers the given argument types and all possible combinations of arrows which might cover them. This means fewer, more precise arrows are needed to fully specify the behavior, but the computation can have exponential complexity. In our small scale testing the overhead did not seem prohibitive, but a larger scale study in an actual type checker for a reasonable language would be required to see if the numeric tower's types are reasonably handled in a semantic fashion in practice.

Function	Arrow Count		Ratio
	Syntactic	Semantic	$\frac{\text{Sem. \#}}{\text{Sem.\# + Syn.\#}}$
`add1`[1]	23	14	0.61
`sub1`[1]	23	13	0.57
`abs`[1]	17	6	0.35
`sqr`[1]	17	12	0.71
`sqrt`[1]	24	11	0.46
`expt`[2]	37	23	0.62
`modulo`[2]	13	9	0.69
`quotient`[2]	24	10	0.42
`+`[2]	85	26	0.31
`-`[2]	65	24	0.37
`*`[2]	58	22	0.38
`/`[2]	46	20	0.44
`max`[2]	82	20	0.24
`min`[2]	89	20	0.23
`<`[3]	88	23	0.26
`<=`[3]	89	22	0.25
`=`[3]	57	22	0.39

Key unary function[1] binary function[2] binary comparison function[3]

Figure 5.15: Size of certain Racket math operations (syntactically vs semantically)

5.3.4 Challenges and Future Work

While these initial results are promising—we can express both simple and non-standard type predicates through semantic subtyping and function application inversion—there remains work to be done if we want to claim this technique is capable of being a true "replacement" for a system as large and complex as Typed Racket. In particular, exploring how these features perform in the wild (i.e. are type checking times tolerable) and how they work with other important features such as polymorphism remains future work.

For an initial glimpse into how types such as those found in Typed Racket's numeric tower might work in a full-scale system with semantic subtyping, we encoded the Typed Racket numeric tower's types in ℂDuce[53]—a language with both semantic subtyping and polymorphism—to see if things would "just work". Unfortunately with the size of the function types we were interested in (i.e. the semantic function types in figure 5.15 which feature almost 30 arrows in some cases) ℂDuce was unable to perform type inference for even a single polymorphic function application in a reasonable amount of type (we stopped the experiment after 15 minutes). We contacted the creators of ℂDuce and they confirmed the example we presented was problematic and likely due to an "explosion in the normalization process"[71]. It is unclear whether further engineering would make these large numeric types more compatible with ℂDuce's type inference algorithm or if further fundamental research on this problem is required. We have included our encoding of the numeric tower's types into ℂDuce in appendix C.

5.4 Expressiveness

With λ_{OT} and λ_{SO} presenting fundamentally different foundations for the same general approach to occurrence typing, it is worth examining how these differences affect the expressiveness of the type system. On the one hand, λ_{SO}—which features semantic subtyping—is better able to completely reason about set-theoretic types and how they relate to one another. This means that idioms which rely on advanced set-theoretic features—such as intersection and negation types—can more easily be described in a system like λ_{SO} (e.g., see `hash-ref` from section 1.2). λ_{OT} on the other hand—with its dependent function types—is better suited to express relationships between different syntactic portions of a program. For example, λ_{OT} can—because of its dependent function types—have abstractions whose results tell us something about non-argument in-scope identifiers:

```
(let ([is-y-an-number? (λ (x) (number? y))])
    (if (is-y-a-number? #f)
        (add1 y)
        0))
```

In this example, the function `is-y-a-number?` ignores its argument and tells us if `y` is a number. I.e., it would be assigned the following type by λ_{OT}'s type system:

$$(x\!:\!\texttt{Any}) \longrightarrow \langle \texttt{Bool}, y \in \texttt{Num}, y \notin \texttt{Num}, \top^{o} \rangle$$

Because of this, the overall program would type check in λ_{OT} but not in λ_{SO}. This is because function application inversion (which λ_{SO} relies on) can only uncover type information about the actual arguments to a function. We might think such programs are few and far between: surely no programmer would manually write such a block of code, right? Perhaps not. There is, however, a long standing tradition of writing extraordinarily complex macros in languages such as Scheme and Racket. Some of these macros generate code which has the appearance of rubbish upon first glance, but whose semantics exactly capture the needs of the syntactic abstraction. Some of these macros—such as the widely used **`for`**-loops in Racket—can generate seemingly odd abstractions and conditional tests which resemble the `is-y-a-number?` example. To illustrate, consider the following Typed Racket program:

```
(: vector-ormap (All (X) (-> (-> X Boolean)
                             (Vectorof X)
                             Boolean)))
(define (vector-ormap f xs)
  (for/or ([x (in-vector xs)])
    (f x)))
```

`vector-ormap` simply iterates over all the elements of `xs`, returning the disjunction of the predicate `f` applied to each element. In other words, it is roughly equivalent to the following expression:

```
  (and (> (vector-length xs) 0)
       (or (f (vector-ref xs 0))
           ...
           (f (vector-ref xs (sub1 (vector-length xs))))))
```

As we noted, however, the various **for**-loops in Racket are not primitives in the language: they are macros. The body of `vector-ormap` expands into roughly the following program:

```
((letrec ([for-loop
           (λ (acc pos)
             (cond
               [(>= pos (vector-length xs)) acc]
               [else
                (define x (vector-ref xs pos))
                (define f-of-x (f x))
                (if (not ((λ args f-of-x) x))
                    (for-loop f-of-x (add1 pos))
                    f-of-x)]))])
   for-loop)
 #f 0)
```

Typed Racket (as of version 7.2) looks at this program and "guesses"—through some undocumented adhoc type inference heuristics which have evolved since its inception—that the recursive `for-loop` function should have type `(-> False Integer Boolean)`. This happens to work for Typed Racket in this case (i.e. the program type checks) in part because Typed Racket can tell that the result of the application `((λ args f-of-x) x)` actually gives us the value of `f-of-x` (i.e. the lambda's type depends on an identifier that is not one of its arguments).[3]

If, however, we limit Typed Racket so dependent function types can only mention their arguments in their codomain, we get the following error:

```
Type Checker: type mismatch
 expected: False
 given: Boolean
 in: f-of-x
```

This initially seems to suggest that there are macros in use today which function application inversion would fail to type check, since their correctness appears to depend on functions whose codomains mention non-arguments. However, we cannot ignore the role Typed Racket's adhoc type inference plays in this failure. It seems, for example, that if Typed Racket's inference determined that the first argument to `for-loop` could be any

[3]Generally speaking, however, these type inference heuristics are a well-known pain point, being insufficient for type checking many usages of **for**-macros in the wild and being difficult to predict for programmers.

`Boolean` and not just `False`, this program *would* type check in a type system where function types cannot describe how non-arguments affect their behavior (i.e. like λ_{SO}). Indeed, if we merely annotate the initial first argument to the recursive function to be `(ann #f Boolean)` the program *does* type check even with the aforementioned limitation on dependent functions. This indicates that a Typed Racket-like system built on semantic subtyping and function application inversion would be able to handle such programs given that its type inference was stronger or it featured a different set of inference heuristics. Regarding stronger inference, work has already been done showing how full-program type inference and polymorphism for semantic subtyping can be achieved[60, 61]. So, while we certainly can come up with programs which λ_{OT} can type check and λ_{SO} cannot, in practice it appears most idioms are well-suited for type checking via λ_{SO}'s type system.

5.5 Related Work

Much of what should be said about work related to λ_{SO} has already been discussed in section 2.2.8 in our overview of occurrence typing and in section 4.6 in our discussion of work related to semantic subtyping. Essentially, while λ_{SO} borrows the logical techniques from prior occurrence typing work [9] and the rich power of semantic subtyping [28], it introduces a truly novel approach for identifying predicate-like functions. Instead of examining a programs syntax, having dependent types, or trying to reason about all possible programs union types may be flattened into, λ_{SO} uses the unique *function application inversion* algorithm introduced in section 5.1 to support occurrence typing. The resulting system seems roughly equivalent in expressiveness to previous work in occurrence typing as it is able to type check the various examples appearing in previous work[14, 70]. And while some occurrence typing languages will be able to express more complex program dependencies in types due to their inclusion of dependent types[9, 30, 31, 4, 23, 22, 32, 20, 33, 34, 35], λ_{SO} seems well suited to handle the majority of idioms occurring in practice due to its building on the rich foundation of semantic subtyping[28].

Appendices

APPENDIX A
FUNCTION APPLICATION INVERSION PROOFS

This appendix contains the mechanized proofs for the theorems in section 5.1.4. The proofs basically assume the following:

- there exists a type and programming language framework in which the entire spectrum of set-theoretic types are available and we can soundly and completely decide their subtyping and inhabitation;
- types can be interpreted as sets of values and we can identify when a value is in the set corresponding to some type;
- any function type can be treated as if being in DNF without loss of generality; and
- for any function type there is a function value which inhabits that type.

The first four items roughly correspond to results from foundational work in semantic subtyping [28] and the last item is based on the assumption that if a type in this language is inhabited then we can find some element of that type and the fact that the underlying language supports nontermination. We conduct our proofs using as few definitions and as little boiler-plate as possible given these assumptions (i.e. we do not include a language definition, a type system, reduction rules, etc).

Function application inversion soundness (see **`Theorem`**s `i_inv_sound` and `d_inv_sound`) and completeness (see **`Theorem`**s `i_inv_minimal` and `d_inv_minimal`) are then proved in that order, with each first proving the property for an intersection of arrows (the **`Inductive`** `interface` datatype) and then for a union of such intersections (the **`Inductive`** `dnf` datatype).

The proofs were compiled with Coq version 8.8.2. The library `CpdtTactics` is the only library which is required but not included in Coq's standard library; it is required for the helpful `crush` tactic and can be found online accompanying Adam Chlipala's textbook *Certified Programming with Dependent Types* [72].

```
Require Import Coq.Sets.Ensembles.
Require Import Coq.Sets.Classical_sets.
Require Import Coq.Sets.Image.
Require Import CpdtTactics.

Set Implicit Arguments.

(* * * * * * * * * * * * * * * * * * * * * * * * * *)
(*             A few useful tactics                *)
(* * * * * * * * * * * * * * * * * * * * * * * * * *)

Ltac ifcase :=
  match goal with
  | [ |- context[if ?X then _ else _] ] => destruct X
  end.

Ltac ifcaseH :=
  match goal with
  | [ H : context[if ?X then _ else _] |- _ ] => destruct X
  end.

Ltac matchcase :=
  match goal with
  | [ |- context[match ?term with
                 | _ => _
                 end] ] => destruct term
  end.

Ltac matchcaseH :=
  match goal with
  | [ H: context[match ?term with
                 | _ => _
                 end] |- _ ] => destruct term
  end.


Ltac applyH :=
  match goal with
  | [H : _ -> _ |- _] => progress (apply H)
  end.

Ltac applyHinH :=
  match goal with
  | [H1 : _ -> _ , H2 : _ |- _] => apply H1 in H2
  end.


(* * * * * * * * * * * * * * * * * * * * * * * * * *)
(*           Value/Type Definitions                *)
(* * * * * * * * * * * * * * * * * * * * * * * * * *)

Axiom V : Type.
Notation Ty := (Ensemble V).

Notation "'0'" := (Empty_set V).
Notation "'1'" := (Full_set V).
Notation "T1 '∩' T2" :=
  (Intersection V T1 T2) (at level 52, right associativity).
Notation "T1 '∪' T2" :=
    (Union V T1 T2) (at level 53, right associativity).
Notation "T1 '\' T2" :=
  (Setminus V T1 T2) (at level 54, right associativity).
Notation "'¬' T" :=
  (1 \ T) (at level 51, right associativity).

Notation "T1 '≠' T2" :=
  (T1 = T2 -> False) (at level 55, right associativity).
Axiom empty_dec : forall (t: Ty), {t = 0} + {t ≠ 0}.
Notation "x '∈' T" :=
  (In V T x) (at level 55, right associativity).
Notation "x '∉' T" :=
  (~ In V T x) (at level 55, right associativity).
Axiom in_dec : forall (v:V) (t: Ty), {v ∈ t} + {v ∉ t}.
Notation "T1 '<:' T2" :=
  (Included V T1 T2) (at level 55, right associativity).

Hint Unfold Included Setminus.
Hint Constructors Union Intersection Inhabited.


(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*         Basic Type Lemmas/Tactics             *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)


Lemma nonempty_inhab : forall t,
    t ≠ 0 -> exists x, x ∈ t.
Proof.
  intros t Hneq.
  apply not_empty_Inhabited in Hneq.
  destruct Hneq as [x H].
  exists x; auto.
Qed.

Lemma empty_uninhab : forall t,
    t = 0 -> forall x, x ∉ t.
Proof.
  intros t Hneq x Hnot.
  rewrite Hneq in Hnot. inversion Hnot.
Qed.


Lemma no_empty_val : forall v P,
    v ∈ 0 -> P.
Proof.
  intros v P Hmt. inversion Hmt.
Qed.

Lemma union_empty_l : forall t,
    0 ∪ t = t.
Proof. crush. Qed.

Lemma union_empty_r : forall t,
    t ∪ 0 = t.
Proof.
  intros. rewrite Union_commutative.
  crush.
Qed.
```

```
Lemma intersection_empty_l : forall t,
    0 ∩ t = 0.
Proof. crush. Qed.

Lemma intersection_empty_r : forall t,
    t ∩ 0 = 0.
Proof. crush. Qed.

Lemma intersection_assoc : forall T1 T2 T3,
    T1 ∩ (T2 ∩ T3) = (T1 ∩ T2) ∩ T3.
Proof.
  intros.
  apply Extensionality_Ensembles; constructor; intros x Hx.
  destruct Hx as [x Hx1 Hx2]. destruct Hx2 as [x Hx2 Hx3].
  crush.
  destruct Hx as [x Hx1 Hx2]. destruct Hx1 as [x Hx1 Hx3].
  crush.
Qed.

Lemma intersection_comm : forall T1 T2,
    T1 ∩ T2 = T2 ∩ T1.
Proof.
  intros T1 T2.
  apply Extensionality_Ensembles; constructor; intros x Hx;
    inversion Hx; crush.
Qed.

Lemma demorgan : forall x T1 T2,
    x ∉ (T1 ∪ T2) ->
    x ∉ T1 /\ x ∉ T2.
Proof. crush. Qed.


Hint Rewrite
     union_empty_l
     union_empty_r
     intersection_empty_l
     intersection_empty_r.

Hint Extern 1 =>
match goal with
| [H : ?x ∈ 0 |- ?P] =>
  apply (no_empty_val P H)
| [H : ?x ∈ ?T, H' : ?T = 0 |- ?P] =>
  rewrite H' in H; apply (no_empty_val P H)
| [H : ?x ∈ ?T, H' : ?T = 0 |- ?P] =>
  symmetry in H'; rewrite H' in H; apply (no_empty_val P H)
end.

Hint Extern 1 (_ ∈ _) =>
match goal with
| [H : ?x ∈ (?T1 ∩ ?T2) |- ?x ∈ ?T1]
  => destruct H; assumption
| [H : ?x ∈ (?T1 ∩ ?T2) |- ?x ∈ ?T2]
  => destruct H; assumption
| [H1 : ?x ∈ ?T1, H2 : ?x ∈ ?T2 |- ?x ∈ (?T1 ∩ ?T2)]
  => constructor; assumption
| [H1 : ?x ∈ ?T1 |- ?x ∈ (?T1 ∪ _)]
```

```
  => left; exact H1
| [H2 : ?x ∈ ?T2 |- ?x ∈ (_ ∪ ?T2)]
  => left; exact H2
end.

Ltac inv_in_intersection :=
  match goal with
  | [H : _ ∈ (_ ∩ _) |- _] => destruct H
  end.

Ltac inv_in_union :=
  match goal with
  | [H : _ ∈ (_ ∪ _) |- _] => destruct H
  end.

Ltac inv_exists :=
  match goal with
  | [H : exists x, _ |- _] => destruct H
  end.


(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*       Function Related Definitions            *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

(* the result of function application *)
Inductive res : Type :=
| Err : res (* invalid argument/type error *)
| Bot : res (* non-termination *)
| Res : V -> res. (* a value result *)
Hint Constructors res.


(* We use a shallow embedding in Gallina's functions
   to model the target language functions. *)
Definition fn := (V -> res).



(* An `interface` is the set of arrows that describe a
   function (i.e. an intersection of 1 or more arrows). We
   use a pair for each arrow, where the fst is the domain
   and the snd is the codomain. *)
Inductive interface : Type :=
| IBase: (Ty * Ty) -> interface
| ICons : (Ty * Ty) -> interface -> interface.
Hint Constructors interface.



(* The domain for an interface is the union of each
   individual arrow's domain. *)
Fixpoint i_dom (i : interface) : Ty :=
  match i with
  | IBase (T1,_) => T1
  | ICons (T1,_) i' => T1 ∪ (i_dom i')
  end.
Hint Unfold i_dom.
```

```
(* Calculates the result type of calling a function which
   has the arrow type `a` on value `v`. *)
Fixpoint a_result (a : (Ty * Ty)) (v : V) : option Ty :=
  if in_dec v (fst a)
  then Some (snd a)
  else None.
Hint Unfold a_result.


(* Function Arrow *)
(* I.e., what it means for a function `f` to conform to the
   description given by arrow `a`. *)
Definition FnA (f : fn) (a : (Ty * Ty)) : Prop :=
forall x T,
  a_result a x = Some T ->
  (f x = Bot \/ exists y, f x = Res y /\ y ∈ T).
Hint Unfold FnA.

(* Calculates the result type of calling a function which
   has the interface type `i` on value `v`. *)
Fixpoint i_result (i : interface) (v : V) : option Ty :=
  match i with
  | IBase a => a_result a v
  | ICons a i' => match a_result a v, i_result i' v with
                  | None, None => None
                  | Some T, None => Some T
                  | None, Some T => Some T
                  | Some T, Some T' => Some (T ∩ T')
                  end
  end.
Hint Unfold i_result.

(* Function Interface *)
(* I.e., what it means for a function `f` to conform to the
   description given by interface `i`. *)
Definition FnI (f : fn) (i : interface) : Prop :=
  forall x T,
    i_result i x = Some T ->
    f x = Bot \/ (exists y, (f x = Res y /\ y ∈ T)).
Hint Unfold FnI.


(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*      Function Related Lemmas/Tactics          *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

Lemma FnI_base : forall f a,
    FnI f (IBase a) ->
    FnA f a.
Proof.
  unfold FnI. unfold FnA.
  intros f [T1 T2] H x T Har.
  simpl in *.
  specialize H with x T2.
  ifcaseH; inversion Har; subst; auto.
Qed.
```

```
Ltac same_Res :=
  match goal with
  | [H1 : ?f ?v = Res ?x , H2 : ?f ?v = Res ?y |- _] =>
    rewrite H1 in H2; inversion H2; subst; clear H2
  end.


Lemma FnI_first : forall f a i,
    FnI f (ICons a i) ->
    FnA f a.
Proof.
  unfold FnI. unfold FnA.
  intros f [T1 T2] i H x T Har.
  specialize (H x).
  unfold a_result in *. simpl in *.
  ifcaseH; matchcaseH; crush.
  specialize (H (T ∩ e)); crush.
  inv_in_intersection; crush; eauto.
Qed.

Lemma FnI_rest : forall f a i,
    FnI f (ICons a i) ->
    FnI f i.
Proof.
  intros f [T1 T2] i Hfi.
  unfold FnI in *.
  intros x T Hres.
  specialize (Hfi x).
  simpl in *.
  ifcaseH; matchcaseH; inversion Hres; subst; eauto.
  specialize (Hfi (T2 ∩ T));
    intuition; crush; inv_in_intersection; eauto.
Qed.

Lemma FnI_cons : forall f a i,
    FnA f a ->
    FnI f i ->
    FnI f (ICons a i).
Proof.
  intros f [T1 T2] i Ha Hi.
  unfold FnI in *. unfold FnA in *.
  intros x T Hres.
  specialize (Ha x). specialize (Hi x).
  simpl in *.
  destruct (in_dec x T1) as [Hx1 | Hx1].
  remember (i_result i x) as Hxr.
  destruct Hxr as [T'|]; inversion Hres; subst.
  destruct (Ha T2 eq_refl). left; assumption.
  destruct (Hi T' eq_refl). left; assumption.
  repeat inv_exists. crush.
  same_Res.
  right.
  match goal with
  | [H : f x = Res ?y |- _] => exists y
  end; crush.
  destruct (Ha T eq_refl); crush; eauto.
  remember (i_result i x) as Hxr.
  destruct Hxr as [T'|]; inversion Hres; subst.
  apply Hi; auto.
```

```
Qed.


Ltac inv_FnI :=
  match goal with
  | [H : FnI _ _ |- _] => inversion H; subst
  end.


(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*         Function Inversion Algorithm          *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

(* Consider function `f` of type `a`. This function
   calculates what type an argument `x` must _not_
   have had  if `(f x) ↝ v` and `v ∈ outT` *)
Fixpoint a_neg (a : (Ty * Ty)) (outT : Ty) : Ty :=
  if empty_dec ((snd a) ∩ outT)
  then (fst a)
  else 0.

(* Consider function `f` of type `i`. This function
   calculates what type an argument `x` must _not_
   have had if `(f x) ↝ v` and `v ∈ outT` *)
Fixpoint i_neg (i : interface) (outT : Ty) : Ty :=
  match i with
  | IBase a => a_neg a outT
  | ICons (S1,S2) i' =>
    let T1 := a_neg (S1,S2) outT in
    let T2 := i_neg i' outT in
    let T3 := S1 ∩ (i_neg i' (S2 ∩ outT)) in
    T1 ∪ T2 ∪ T3
  end.

(* Consider function `f` of type `i`. This function
   calculates what type an argument `x` must _have_
   had if `(f x) ↝ v` and `v ∈ outT` *)
Definition i_inv (i : interface) (outT : Ty) : Ty :=
  (i_dom i) \ (i_neg i outT).


(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*      Function Inversion Lemmas/Tactics        *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

Lemma FnA_res_ty : forall T1 T2 f x y,
    x ∈ T1 ->
    f x = Res y ->
    FnA f (T1,T2) ->
    y ∈ T2.
Proof.
  intros T1 T2 f x y Hx Hfx Hfa.
  assert (f x = Bot \/ (exists y, (f x = Res y /\ y ∈ T2)))
    as Hex.
  eapply Hfa; crush.
  ifcase; crush.
  crush.
Qed.
```

```
Hint Extern 1 (_ ∈ _) =>
match goal with
| [Hx : ?x ∈ ?T1,
        Hfy : ?f ?x = Res ?y,
              HFnA : FnA ?f (?T1,?T2)
   |- ?y ∈ ?T2]
  => apply (FnA_res_ty x Hx Hfy HFnA)
| [Hx : ?x ∈ ?T1,
        Hfy : ?f ?x = Res ?y,
              HFnA : FnI ?f (IBase (?T1,?T2))
   |- ?y ∈ ?T2]
  => apply (FnA_res_ty x Hx Hfy (FnI_first HFnA))
| [Hx : ?x ∈ ?T1,
        Hfy : ?f ?x = Res ?y,
              HFnA : FnI ?f (ICons (?T1,?T2) _)
   |- ?y ∈ ?T2]
  => apply (FnA_res_ty x Hx Hfy (FnI_first HFnA))
  end.


Lemma i_neg_sub : forall i T1 T2,
    T2 <: T1 ->
    (i_neg i T1) <: (i_neg i T2).
Proof with auto.
  intros i. induction i as [[T1 T2] | [T1 T2] i' IH].
  {
    intros T T' Hsub x Hx. simpl in *.
    destruct (empty_dec (T2 ∩ T'))
      as [Hmt' | Hnmt']...
    destruct (empty_dec (T2 ∩ T))
      as [Hmt | Hnmt]...
    destruct (empty_dec (T2 ∩ T))
      as [Hmt | Hnmt]...
    apply nonempty_inhab in Hnmt'.
    destruct Hnmt' as [v Hv].
    assert (v ∈ T2 ∩ T)...
  }
  {
    intros T T' Hsub x Hx.
    simpl in *.
    destruct (empty_dec (T2 ∩ T)) as [Hmt | Hnmt].
    {
      destruct (empty_dec (T2 ∩ T')) as [Hmt' | Hnmt'].
      {
        destruct Hx as [x Hx | x Hx]...
        destruct Hx as [x Hx | x Hx]...
        right; left; eapply IH; eauto.
      }
      {
        apply nonempty_inhab in Hnmt'.
        destruct Hnmt' as [v Hv].
        assert (v ∈ T2 ∩ T)...
      }
    }
    {
      destruct (empty_dec (T2 ∩ T')) as [Hmt' | Hnmt'].
      {
        right.
        destruct Hx as [x Hx | x Hx]...
        destruct Hx as [x Hx | x Hx]...
        {
          left; eapply IH; eauto.
        }
        {
          destruct Hx...
          right; split...
          apply (IH (T2 ∩ T) (T2 ∩ T'))...
        }
      }
      {
        destruct Hx as [x Hx | x Hx]...
        destruct Hx as [x Hx | x Hx]...
        right. left. eapply IH; eauto.
        right. right; split...
        destruct Hx as [x Hx' Hx'']...
        apply (IH (T2 ∩ T) (T2 ∩ T'))...
      }
    }
  }
Qed.

Ltac apply_fun :=
  match goal with
  | [H1 : ?x ∈ ?T1,
          Hf : FnI ?f (IBase (?T1,?T2)),
               Hres : ?f ?x = Res ?y
     |- _] =>
    assert (y ∈ T2)
      by (exact (FnA_res_ty x H1 Hres (FnI_base Hf)))
  | [H1 : ?x ∈ ?T1,
          Hf : FnA ?f (?T1,?T2),
               Hres : ?f ?x = Res ?y
     |- _] =>
    assert (y ∈ T2)
      by (exact (FnA_res_ty x H1 Hres (FnI_base Hf)))
  end.

Lemma in_i_neg : forall i v v' f T,
    FnI f i ->
    v ∈ (i_neg i T) ->
    f v = Res v' ->
    v' ∉ T.
Proof with auto.
  intros i.
  induction i as [[T1 T2] | [T1 T2] i' IH];
    intros v v' f T Hfi Hv Hfv Hcontra.
  (* IBase (Arrow T1 T2) *)
  {
    simpl in *.
    ifcaseH; crush.
    apply_fun...
    assert (v' ∈ (T2 ∩ T)) as impossible...
  }
  (* ICons (Arrow T1 T2) i' *)
  {
    simpl in *.
    assert (FnA f (T1,T2)) as Hfa by (eapply FnI_first; eauto).
    assert (FnI f i') as Hfi' by (eapply FnI_rest; eauto).
    destruct (empty_dec (T2 ∩ T)) as [Hmt | Hnmt]...
    {
      inv_in_union.
      {
        apply_fun...
        assert (v' ∈ (T2 ∩ T)) as impossible...
      }
      {
        inv_in_union.
        {
          eapply IH; eauto.
        }
        {
          inv_in_intersection.
          apply_fun...
          assert (v' ∈ (T2 ∩ T)) as impossible...
        }
      }
    }
    {
      rewrite union_empty_l in *.
      destruct Hv as [v Hv | v Hv].
      {
        eapply IH; eauto.
      }
      {
        eapply IH; eauto.
        destruct Hv as [v Hv1 Hv2].
        eapply IH; eauto.
      }
    }
  }
Qed.

Lemma not_in_i_neg : forall i v v' f T,
    FnI f i ->
    f v = Res v' ->
    v' ∈ T ->
    v ∉ (i_neg i T).
Proof.
  intros i v v' f T Hfi Hfv Hv' Hcontra.
  eapply in_i_neg; eauto.
Qed.


(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*          Inversion Definition                 *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)


Definition Inv (i : interface) (outT inT: Ty) : Prop :=
  forall (f:fn),
    FnI f i ->
    forall (v v':V),
      v ∈ (i_dom i) ->
      f v = Res v' ->
      v' ∈ outT ->
      v ∈ inT.
```

```
(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*               i_inv soundness                  *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

(* Interface Inversion Soundness
   i.e. the input type we predict is correct *)
Theorem i_inv_sound : forall i outT,
    Inv i outT (i_inv i outT).
Proof with crush.
  intros i outT f Hint v v' Hv Hf Hv'.
  unfold i_inv in *.
  constructor; auto.
  intros Hcontra.
  eapply in_i_neg; eauto.
Qed.

(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*         Axioms for proving minimality          *)
(*                                                *)
(* i.e. basically we assume if a codomain is      *)
(* inhabited, then there exists a function which *)
(* will map inputs to those codomain values.      *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)


Definition MapsTo (f : fn) (i : interface) : Prop :=
  forall v T,
    i_result i v = Some T ->
    T ≠ 0 ->
    exists v', f v = Res v' /\ v' ∈ T.


Definition MapsToTarget (f : fn) (i : interface) (tgt : Ty) : Prop :=
  forall v T,
    i_result i v = Some T ->
    (T ∩ tgt) ≠ 0 ->
    exists v', f v = Res v'
               /\ v' ∈ (T ∩ tgt).

Axiom exists_fn : forall i,
    exists f, FnI f i /\ MapsTo f i.

Axiom exists_target_fn : forall i outT,
    exists f, FnI f i /\ MapsToTarget f i outT.



(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*         Lemmas related to i_result             *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

Lemma i_result_None : forall i x outT,
    x ∈ (i_dom i) ->
    i_result i x = None ->
    x ∈ (i_neg i outT).
Proof with auto.
  intros i; induction i as [[T1 T2] | [T1 T2] i' IH].
  {
    intros x outT Hdom Hires.
    simpl in *.
    destruct (in_dec x T1) as [Hx | Hx]; crush.
  }
  {
    intros x outT Hx Hires.
    simpl in *.
    destruct (in_dec x T1) as [Hx1 | Hx1].
    {
      destruct (empty_dec (T2 ∩ outT)) as [Hmt | Hnmt].
      {
        left. assumption.
      }
      {
        right.
        remember (i_result i' x) as ires'.
        destruct ires' as [T' |].
        inversion Hires. inversion Hires.
      }
    }
    {
      destruct Hx as [x Hx | x Hx]; try solve[contradiction].
      right. left.
      apply IH... matchcaseH; crush.
    }
  }
Qed.


Lemma i_result_Some : forall i x T outT,
    i_result i x = Some T ->
    (T ∩ outT) = 0 ->
    x ∈ (i_neg i outT).
Proof with auto.
  intros i x; induction i as [[T1 T2] | [T1 T2] i' IH].
  {
    intros T outT Hires Hmt.
    simpl in *.
    destruct (in_dec x T1) as [Hx1 | Hx1]; crush.
    destruct (empty_dec (T ∩ outT)) as [Hmt' | Hmt']...
    ifcase... contradiction. contradiction.
  }
  {
    intros T outT Hires Hmt.
    simpl in *.
    destruct (in_dec x T1) as [Hx1 | Hx1].
    {
      destruct (empty_dec (T2 ∩ outT)) as [Hmt2 | Hnmt2].
      {
        left...
      }
      {
        right.
        destruct (i_result i' x) as [S |]; try solve[crush].
        specialize (IH S).
        inversion Hires; subst. clear Hires.
        right. split...
        eapply IH...
        rewrite intersection_assoc.
        rewrite (intersection_comm S T2)...
      }
    }
    {
      destruct (i_result i' x) as [S |]; try solve[crush].
      inversion Hires; subst.
      specialize (IH T outT eq_refl Hmt)...
    }
  }
Qed.


(* Interface Inversion Minimality
   i.e. the input type we predict is minimal *)
Lemma i_inv_exists_fn : forall i outT x,
    x ∈ (i_inv i outT) ->
    exists f y, FnI f i /\ f x = Res y /\ y ∈ outT.
Proof with auto.
  intros i outT x Hx.
  unfold i_inv in Hx.
  destruct Hx as [HxIs HxNot].
  remember (i_result i x) as xres.
  destruct xres as [S |].
  {
    symmetry in Heqxres.
    destruct (empty_dec (S ∩ outT)) as [Hmt | Hnmt].
    {
      assert (x ∈ (i_neg i outT)) as impossible.
      {
        eapply i_result_Some; eauto.
      }
      contradiction.
    }
    {
      destruct (exists_target_fn i outT)
        as [f [Hfi Hmaps]].
      unfold MapsToTarget in Hmaps.
      destruct (Hmaps x S Heqxres Hnmt) as [y [Hfx Hy]].
      exists f. exists y...
    }
  }
  {
    assert (x ∈ (i_neg i outT)) as impossible.
    {
      eapply i_result_None; eauto.
    }
    contradiction.
  }
Qed.

(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*             i_inv minimality                  *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)


Theorem i_inv_minimal : forall i outT inT,
    Inv i outT inT ->
    (i_inv i outT) <: inT.
Proof with auto.
  intros i outT inT Hinv x Hx.
  unfold Inv in Hinv.
  destruct (i_inv_exists_fn Hx) as [f [y [Hf [Hres Hy]]]].
  specialize (Hinv f Hf x y). eapply Hinv; eauto.
  destruct Hx...
Qed.




(* * * * * * * * * * * * * * * * * * * * * * * * * *)
(*           DNF Function Definitions             *)
(* * * * * * * * * * * * * * * * * * * * * * * * * *)


(* A `dnf` is a union of interfaces, at least one of which
  describes a function (i.e. an DNF with 1 or more
  clauses). *)
Inductive dnf : Type :=
| DBase : interface -> dnf
| DCons : interface -> dnf -> dnf.
Hint Constructors dnf.

(* The domain for a dnf is the intersection of each
   individual interface's domain. *)
Fixpoint d_dom (d : dnf) : Ty :=
  match d with
  | DBase i => (i_dom i)
  | DCons i d' => (i_dom i) ∩ (d_dom d')
  end.
Hint Unfold d_dom.

(* Disjunction of Function Arrows *)
(* I.e., what it means for a function `f` to conform to the
   description given by arrow `a`. *)
Fixpoint FnD (f : fn) (d : dnf) : Prop :=
  match d with
  | DBase i => FnI f i
  | DCons i d' => FnI f i \/ FnD f d'
  end.
Hint Unfold FnD.


(* * * * * * * * * * * * * * * * * * * * * * * * * *)
(*                  DNF Lemmas                    *)
(* * * * * * * * * * * * * * * * * * * * * * * * * *)

Lemma FnD_base : forall f i,
    FnD f (DBase i) ->
    FnI f i.
Proof.
  intros f i H.
  crush.
Qed.

Lemma FnD_Cons_i : forall i d f,
    FnI f i ->
    FnD f (DCons i d).
Proof. crush. Qed.
```

```
Lemma FnD_Cons_d : forall i d f,
    FnD f d ->
    FnD f (DCons i d).
Proof. crush. Qed.



(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*      DNF Function Inversion Algorithm         *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

Fixpoint d_inv_aux (d : dnf) (outT : Ty) : Ty :=
  match d with
  | DBase i => i_inv i outT
  | DCons i d' => (i_inv i outT) ∪ (d_inv_aux d' outT)
  end.

(* Calculates the result type of calling a function which
   has the interface type `i` on value `v`. *)
Definition d_inv (d : dnf) (outT : Ty) : Ty :=
 (d_dom d) ∩ (d_inv_aux d outT).
Hint Unfold d_inv d_inv_aux.


(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*          DNF Inversion Definition             *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)


Definition InvD (d : dnf) (outT inT: Ty) : Prop :=
  forall (f:fn),
    FnD f d ->
    forall (v v':V),
      v ∈ (d_dom d) ->
      f v = Res v' ->
      v' ∈ outT ->
      v ∈ inT.
Hint Unfold InvD.

(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*                 Soundness                     *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

(* Interface Inversion Soundness
   i.e. the input type we predict is correct *)
Theorem d_inv_sound : forall d outT,
    InvD d outT (d_inv d outT).
Proof with auto.
  intros d.
  induction d as [i | i d' IH].
  {
    unfold InvD.
    intros outT f Hfd v v' Hv Hf Hv'. simpl in *.
    split...
    eapply i_inv_sound; eauto.
  }
  {
    unfold InvD.
    intros outT f Hfd v v' Hv Hfv Hv'.
    simpl in *.
    destruct Hfd as [Hfi | Hfd].
    {
      split...
      left; eapply i_inv_sound; eauto.
    }
    {
      split...
      assert (v ∈ (d_inv d' outT)) by (eapply IH; eauto).
      right... unfold d_inv in *...
    }
  }
Qed.

(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*            Lemma for Minimality               *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)

Lemma d_inv_exists_fn : forall d outT x,
    x ∈ (d_inv d outT) ->
    exists f y, FnD f d /\ f x = Res y /\ y ∈ outT.
Proof with auto.
  intros d.
  induction d as [i | i d' IH];
    intros outT x Hx.
  {
    unfold d_inv in Hx.
    simpl in *. eapply i_inv_exists_fn...
  }
  {
    unfold d_inv in Hx.
    simpl in *.
    destruct Hx as [x Hx1 Hx2].
    destruct Hx2 as [x Hx2 | x Hx2].
    {
      assert (x ∈ (i_inv i outT)) as Hx by auto.
      destruct (i_inv_exists_fn Hx) as [f [y [H]]].
      exists f. exists y...
    }
    {
      unfold d_inv in Hx1.
      assert (x ∈ (d_inv d' outT)) as Hx.
      unfold d_inv...
      destruct (IH outT x Hx) as [f [y [H1 [H2 H3]]]].
      exists f. exists y...
    }
  }
Qed.


(* * * * * * * * * * * * * * * * * * * * * * * * *)
(*                  Minimality                   *)
(* * * * * * * * * * * * * * * * * * * * * * * * *)


Theorem d_inv_minimal : forall d outT inT,
    InvD d outT inT ->
    (d_inv d outT) <: inT.
Proof with auto.
```

```
  intros d outT inT Hinv x Hx.
  destruct (d_inv_exists_fn Hx) as [f [y [H1 [H2 H3]]]].
  unfold d_inv in *. destruct Hx as [x Hx1 Hx2].
  unfold InvD in Hinv.
  specialize (Hinv f H1 x y Hx1 H2 H3)...
Qed.
```

APPENDIX B
NUMERIC TOWER TYPES

To demonstrate that function application inversion can scale to handle the Typed Racket numeric tower (see section 5.3), we use a model of the numeric tower which is described precisely in this appendix. It more-or-less identical to the types Typed Racket uses (as of version 7.2) to model the numeric tower (the differences are not interesting, e.g. we use abbreviated camel-case instead of hyphens, etc). In this model, the set of numeric values in Racket are partitioned into disjoint sets described by the following "numeric base types":

`Zero`	`One`	`ByteLargerThanOne`
`PosIndexNotByte`	`PosFixnumNotIndex`	`NegFixnum`
`PosIntegerNotFixnum`	`NegIntegerNotFixnum`	`PosRationalNotInteger`
`NegRationalNotInteger`	`FloatNaN`	`FloatPosZero`
`FloatNegZero`	`PosFloatNumber`	`PosFloatInfinity`
`NegFloatNumber`	`NegFloatInfinity`	`SingleFloatNaN`
`SingleFloatPosZero`	`SingleFloatNegZero`	`PosSingleFloatNumber`
`PosSingleFloatInfinity`	`NegSingleFloatNumber`	`NegSingleFloatInfinity`
`ExactImaginary`	`ExactComplex`	`FloatImaginary`
`SingleFloatImaginary`	`FloatComplex`	`SingleFloatComplex`

E.g., the type `Zero` covers the set $\{0\}$, `One` covers $\{1\}$, `ByteLargerThanOne` covers $\{2, \ldots, 255\}$, etc. From here, unions of types are defined to describe the natural subsets of the numeric tower. In the following tables, the left column names a numeric union which is defined to be the union of all of the types in the right column of the same row.

Named Union	Union Members
NaN	SingleFloatNaN FloatNaN
PosByte	One ByteLargerThanOne
Byte	Zero PosByte
PosIndex	One ByteLargerThanOne PosIndexNotByte
Index	Zero PosIndex
PosFixnum	PosFixnumNotIndex PosIndex
NonnegFixnum	PosFixnum Zero
NonposFixnum	NegFixnum Zero
Fixnum	NegFixnum Zero PosFixnum
IntegerNotFixnum	NegIntegerNotFixnum PosIntegerNotFixnum
FixnumNotIndex	NegFixnum PosFixnumNotIndex
PosInteger	PosIntegerNotFixnum PosFixnum
NonnegInteger	Zero PosInteger
NegInteger	NegFixnum NegIntegerNotFixnum
NonposInteger	NegInteger Zero
Integer	NegInteger Zero PosInteger
PosRational	PosRationalNotInteger PosInteger
NonnegRational	Zero PosRational
NegRational	NegRationalNotInteger NegInteger
NonposRational	NegRational Zero
RationalNotInteger	NegRationalNotInteger PosRationalNotInteger

Named Union	Union Members
`Rational`	`NegRational` `Zero` `PosRational`
`FloatZero`	`FloatPosZero` `FloatNegZero` `FloatNaN`
`PosFloat`	`PosFloatNumber` `PosFloatInfinity` `FloatNaN`
`NonnegFloat`	`PosFloat` `FloatZero`
`NegFloat`	`NegFloatNumber` `NegFloatInfinity` `FloatNaN`
`NonposFloat`	`NegFloat` `FloatZero`
`Float`	`NegFloatNumber` `NegFloatInfinity` `FloatNegZero` `FloatPosZero` `PosFloatNumber` `PosFloatInfinity` `FloatNaN`
`SingleFloatZero`	`SingleFloatPosZero` `SingleFloatNegZero` `SingleFloatNaN`
`InexactRealNaN`	`FloatNaN` `SingleFloatNaN`
`InexactRealPosZero`	`SingleFloatPosZero` `FloatPosZero`
`InexactRealNegZero`	`SingleFloatNegZero` `FloatNegZero`
`InexactRealZero`	`InexactRealPosZero` `InexactRealNegZero` `InexactRealNaN`
`PosSingleFloat`	`PosSingleFloatNumber` `PosSingleFloatInfinity` `SingleFloatNaN`
`PosInexactReal`	`PosSingleFloat` `PosFloat`
`NonnegSingleFloat`	`PosSingleFloat` `SingleFloatZero`
`NonnegInexactReal`	`PosInexactReal` `InexactRealZero`

Named Union	Union Members
`NegSingleFloat`	`NegSingleFloatNumber` `NegSingleFloatInfinity` `SingleFloatNaN`
`NegInexactReal`	`NegSingleFloat` `NegFloat`
`NonposSingleFloat`	`NegSingleFloat` `SingleFloatZero`
`NonposInexactReal`	`NegInexactReal` `InexactRealZero`
`SingleFloat`	`NegSingleFloat` `SingleFloatNegZero` `SingleFloatPosZero` `PosSingleFloat` `SingleFloatNaN`
`InexactReal`	`SingleFloat` `Float`
`PosInfinity`	`PosFloatInfinity` `PosSingleFloatInfinity`
`NegInfinity`	`NegFloatInfinity` `NegSingleFloatInfinity`
`RealZero`	`Zero` `InexactRealZero`
`RealZeroNoNaN`	`Zero` `InexactRealPosZero` `InexactRealNegZero`
`PosReal`	`PosRational` `PosInexactReal`
`NonnegReal`	`NonnegRational` `NonnegInexactReal`
`NegReal`	`NegRational` `NegInexactReal`
`NonposReal`	`NonposRational` `NonposInexactReal`
`Real`	`Rational` `InexactReal`
`ExactNumber`	`ExactImaginary` `ExactComplex` `Rational`
`InexactImaginary`	`FloatImaginary` `SingleFloatImaginary`
`Imaginary`	`ExactImaginary` `InexactImaginary`
`InexactComplex`	`FloatComplex` `SingleFloatComplex`
`Number`	`Real` `Imaginary` `ExactComplex` `InexactComplex`

Next for comparison in figures B.1 through B.4 we give the full Typed Racket type for + (the binary cases) which has 85 arrows, followed by the semantic type (i.e. for a language like λ_{SO}) in figure B.5 which has 26 arrows.

$$
\begin{array}{ll}
 & \texttt{PosByte} \times \texttt{PosByte} \longrightarrow \texttt{PosIndex} \\
\cap & \texttt{Byte} \times \texttt{Byte} \longrightarrow \texttt{Index} \\
\cap & \texttt{PosByte} \times \texttt{PosByte} \longrightarrow \texttt{PosIndex} \\
\cap & \texttt{PosIndex} \times \texttt{Index} \longrightarrow \texttt{PosFixnum} \\
\cap & \texttt{Index} \times \texttt{PosIndex} \longrightarrow \texttt{PosFixnum} \\
\cap & \texttt{Index} \times \texttt{Index} \longrightarrow \texttt{NonnegFixnum} \\
\cap & \texttt{NegFixnum} \times \texttt{One} \longrightarrow \texttt{NonposFixnum} \\
\cap & \texttt{One} \times \texttt{NegFixnum} \longrightarrow \texttt{NonposFixnum} \\
\cap & \texttt{NonposFixnum} \times \texttt{NonnegFixnum} \longrightarrow \texttt{Fixnum} \\
\cap & \texttt{NonnegFixnum} \times \texttt{NonposFixnum} \longrightarrow \texttt{Fixnum} \\
\cap & \texttt{PosInteger} \times \texttt{NonnegInteger} \longrightarrow \texttt{PosInteger} \\
\cap & \texttt{NonnegInteger} \times \texttt{PosInteger} \longrightarrow \texttt{PosInteger} \\
\cap & \texttt{NegInteger} \times \texttt{NonposInteger} \longrightarrow \texttt{NegInteger} \\
\cap & \texttt{NonposInteger} \times \texttt{NegInteger} \longrightarrow \texttt{NegInteger} \\
\cap & \texttt{NonnegInteger} \times \texttt{NonnegInteger} \longrightarrow \texttt{NonnegInteger} \\
\cap & \texttt{NonposInteger} \times \texttt{NonposInteger} \longrightarrow \texttt{NonposInteger} \\
\cap & \texttt{Integer} \times \texttt{Integer} \longrightarrow \texttt{Integer} \\
\cap & \texttt{PosRational} \times \texttt{NonnegRational} \longrightarrow \texttt{PosRational} \\
\cap & \texttt{NonnegRational} \times \texttt{PosRational} \longrightarrow \texttt{PosRational} \\
\cap & \texttt{NegRational} \times \texttt{NonposRational} \longrightarrow \texttt{NegRational} \\
\cap & \texttt{NonposRational} \times \texttt{NegRational} \longrightarrow \texttt{NegRational} \\
\cap & \texttt{NonnegRational} \times \texttt{NonnegRational} \longrightarrow \texttt{NonnegRational} \\
\cap & \texttt{NonposRational} \times \texttt{NonposRational} \longrightarrow \texttt{NonposRational} \\
\cap & \texttt{Rational} \times \texttt{Rational} \longrightarrow \texttt{Rational}
\end{array}
$$

Figure B.1: Syntactic Type of + (1 of 4)

$\cap$ `PosFloat` $\times$ `NonnegReal` $\longrightarrow$ `PosFloat`
$\cap$ `NonnegReal` $\times$ `PosFloat` $\longrightarrow$ `PosFloat`
$\cap$ `PosReal` $\times$ `NonnegFloat` $\longrightarrow$ `PosFloat`
$\cap$ `NonnegFloat` $\times$ `PosReal` $\longrightarrow$ `PosFloat`
$\cap$ `NegFloat` $\times$ `NonposReal` $\longrightarrow$ `NegFloat`
$\cap$ `NonposReal` $\times$ `NegFloat` $\longrightarrow$ `NegFloat`
$\cap$ `NegReal` $\times$ `NonposFloat` $\longrightarrow$ `NegFloat`
$\cap$ `NonposFloat` $\times$ `NegReal` $\longrightarrow$ `NegFloat`
$\cap$ `NonnegFloat` $\times$ `NonnegReal` $\longrightarrow$ `NonnegFloat`
$\cap$ `NonnegReal` $\times$ `NonnegFloat` $\longrightarrow$ `NonnegFloat`
$\cap$ `NonposFloat` $\times$ `NonposReal` $\longrightarrow$ `NonposFloat`
$\cap$ `NonposReal` $\times$ `NonposFloat` $\longrightarrow$ `NonposFloat`
$\cap$ `Float` $\times$ `Real` $\longrightarrow$ `Float`
$\cap$ `Real` $\times$ `Float` $\longrightarrow$ `Float`
$\cap$ `Float` $\times$ `Float` $\longrightarrow$ `Float`
$\cap$ `PosSingleFloat` $\times$ (`NonnegRational` $\cup$ `NonnegSingleFloat`) $\longrightarrow$ `PosSingleFloat`
$\cap$ (`NonnegRational` $\cup$ `NonnegSingleFloat`) $\times$ `PosSingleFloat` $\longrightarrow$ `PosSingleFloat`
$\cap$ (`PosRational` $\cup$ `PosSingleFloat`) $\times$ `NonnegSingleFloat` $\longrightarrow$ `PosSingleFloat`
$\cap$ `NonnegSingleFloat` $\times$ (`PosRational` $\cup$ `PosSingleFloat`) $\longrightarrow$ `PosSingleFloat`
$\cap$ `NegSingleFloat` $\times$ (`NonposRational` $\cup$ `NonposSingleFloat`) $\longrightarrow$ `NegSingleFloat`
$\cap$ (`NonposRational` $\cup$ `NonposSingleFloat`) $\times$ `NegSingleFloat` $\longrightarrow$ `NegSingleFloat`
$\cap$ (`NegRational` $\cup$ `NegSingleFloat`) $\times$ `NonposSingleFloat` $\longrightarrow$ `NegSingleFloat`
$\cap$ `NonposSingleFloat` $\times$ (`NegRational` $\cup$ `NegSingleFloat`) $\longrightarrow$ `NegSingleFloat`
$\cap$ `NonnegSingleFloat` $\times$ (`NonnegRational` $\cup$ `NonnegSingleFloat`)
$\longrightarrow$ `NonnegSingleFloat`
$\cap$ (`NonnegRational` $\cup$ `NonnegSingleFloat`) $\times$ `NonnegSingleFloat`
$\longrightarrow$ `NonnegSingleFloat`

Figure B.2: Syntactic Type of `+` (2 of 4)

$$
\begin{array}{ll}
\cap & \texttt{NonposSingleFloat} \times (\texttt{NonposRational} \cup \texttt{NonposSingleFloat}) \\
 & \longrightarrow \texttt{NonposSingleFloat} \\
\cap & (\texttt{NonposRational} \cup \texttt{NonposSingleFloat}) \times \texttt{NonposSingleFloat} \\
 & \longrightarrow \texttt{NonposSingleFloat} \\
\cap & \texttt{SingleFloat} \times (\texttt{Rational} \cup \texttt{SingleFloat}) \longrightarrow \texttt{SingleFloat} \\
\cap & (\texttt{Rational} \cup \texttt{SingleFloat}) \times \texttt{SingleFloat} \longrightarrow \texttt{SingleFloat} \\
\cap & \texttt{SingleFloat} \times \texttt{SingleFloat} \longrightarrow \texttt{SingleFloat} \\
\cap & \texttt{PosInexactReal} \times \texttt{NonnegReal} \longrightarrow \texttt{PosInexactReal} \\
\cap & \texttt{NonnegReal} \times \texttt{PosInexactReal} \longrightarrow \texttt{PosInexactReal} \\
\cap & \texttt{PosReal} \times \texttt{NonnegInexactReal} \longrightarrow \texttt{PosInexactReal} \\
\cap & \texttt{NonnegInexactReal} \times \texttt{PosReal} \longrightarrow \texttt{PosInexactReal} \\
\cap & \texttt{NegInexactReal} \times \texttt{NonposReal} \longrightarrow \texttt{NegInexactReal} \\
\cap & \texttt{NonposReal} \times \texttt{NegInexactReal} \longrightarrow \texttt{NegInexactReal} \\
\cap & \texttt{NegReal} \times \texttt{NonposInexactReal} \longrightarrow \texttt{NegInexactReal} \\
\cap & \texttt{NonposInexactReal} \times \texttt{NegReal} \longrightarrow \texttt{NegInexactReal} \\
\cap & \texttt{NonnegInexactReal} \times \texttt{NonnegReal} \longrightarrow \texttt{NonnegInexactReal} \\
\cap & \texttt{NonnegReal} \times \texttt{NonnegInexactReal} \longrightarrow \texttt{NonnegInexactReal} \\
\cap & \texttt{NonposInexactReal} \times \texttt{NonposReal} \longrightarrow \texttt{NonposInexactReal} \\
\cap & \texttt{NonposReal} \times \texttt{NonposInexactReal} \longrightarrow \texttt{NonposInexactReal} \\
\cap & \texttt{InexactReal} \times \texttt{Real} \longrightarrow \texttt{InexactReal} \\
\cap & \texttt{Real} \times \texttt{InexactReal} \longrightarrow \texttt{InexactReal} \\
\cap & \texttt{PosReal} \times \texttt{NonnegReal} \longrightarrow \texttt{PosReal} \\
\cap & \texttt{NonnegReal} \times \texttt{PosReal} \longrightarrow \texttt{PosReal} \\
\cap & \texttt{NegReal} \times \texttt{NonposReal} \longrightarrow \texttt{NegReal} \\
\cap & \texttt{NonposReal} \times \texttt{NegReal} \longrightarrow \texttt{NegReal}
\end{array}
$$

Figure B.3: Syntactic Type of + (3 of 4)

$$
\begin{array}{ll}
\cap & \texttt{NonnegReal} \times \texttt{NonnegReal} \to \texttt{NonnegReal} \\
\cap & \texttt{NonposReal} \times \texttt{NonposReal} \to \texttt{NonposReal} \\
\cap & \texttt{Real} \times \texttt{Real} \to \texttt{Real} \\
\cap & \texttt{ExactNumber} \times \texttt{ExactNumber} \to \texttt{ExactNumber} \\
\cap & \texttt{FloatComplex} \times \texttt{Number} \to \texttt{FloatComplex} \\
\cap & \texttt{Number} \times \texttt{FloatComplex} \to \texttt{FloatComplex} \\
\cap & \texttt{Float} \times \texttt{InexactComplex} \to \texttt{FloatComplex} \\
\cap & \texttt{InexactComplex} \times \texttt{Float} \to \texttt{FloatComplex} \\
\cap & \texttt{SingleFloatComplex} \times (\texttt{Rational} \cup \texttt{SingleFloat} \cup \texttt{SingleFloatComplex}) \\
 & \to \texttt{SingleFloatComplex} \\
\cap & (\texttt{Rational} \cup \texttt{SingleFloat} \cup \texttt{SingleFloatComplex}) \times \texttt{SingleFloatComplex} \\
 & \to \texttt{SingleFloatComplex} \\
\cap & \texttt{InexactComplex} \times (\texttt{Rational} \cup \texttt{InexactReal} \cup \texttt{InexactComplex}) \\
 & \to \texttt{InexactComplex} \\
\cap & (\texttt{Rational} \cup \texttt{InexactReal} \cup \texttt{InexactComplex}) \times \texttt{InexactComplex} \\
 & \to \texttt{InexactComplex} \\
\cap & \texttt{Number} \times \texttt{Number} \to \texttt{Number}
\end{array}
$$

Figure B.4: Syntactic Type of + (4 of 4)

$$
\begin{array}{ll}
 & \texttt{Byte} \times \texttt{Byte} \longrightarrow \texttt{Index}^{1} \\
\cap & \texttt{Index} \times \texttt{Index} \longrightarrow \texttt{NonnegFixnum}^{1} \\
\cap & \texttt{NegFixnum} \times \texttt{One} \longrightarrow \texttt{NonposFixnum}^{1} \\
\cap & \texttt{One} \times \texttt{NegFixnum} \longrightarrow \texttt{NonposFixnum}^{1} \\
\cap & \texttt{NonposFixnum} \times \texttt{NonnegFixnum} \longrightarrow \texttt{Fixnum}^{1} \\
\cap & \texttt{NonnegFixnum} \times \texttt{NonposFixnum} \longrightarrow \texttt{Fixnum}^{1} \\
\cap & \texttt{PosReal} \times \texttt{NonnegReal} \longrightarrow \texttt{PosReal}^{2} \\
\cap & \texttt{NonnegReal} \times \texttt{PosReal} \longrightarrow \texttt{PosReal}^{2} \\
\cap & \texttt{NegReal} \times \texttt{NonposReal} \longrightarrow \texttt{NegReal}^{2} \\
\cap & \texttt{NonposReal} \times \texttt{NegReal} \longrightarrow \texttt{NegReal}^{2} \\
\cap & \texttt{NonnegReal} \times \texttt{NonnegReal} \longrightarrow \texttt{NonnegReal}^{2} \\
\cap & \texttt{NonposReal} \times \texttt{NonposReal} \longrightarrow \texttt{NonposReal}^{2} \\
\cap & \texttt{Integer} \times \texttt{Integer} \longrightarrow \texttt{Integer}^{3} \\
\cap & \texttt{Float} \times \texttt{Real} \longrightarrow \texttt{Float}^{3} \\
\cap & \texttt{Real} \times \texttt{Float} \longrightarrow \texttt{Float}^{3} \\
\cap & \texttt{SingleFloat} \times (\texttt{Rational} \cup \texttt{SingleFloat}) \longrightarrow \texttt{SingleFloat}^{3} \\
\cap & (\texttt{Rational} \cup \texttt{SingleFloat}) \times \texttt{SingleFloat} \longrightarrow \texttt{SingleFloat}^{3} \\
\cap & \texttt{Real} \times \texttt{Real} \longrightarrow \texttt{Real}^{3} \\
\cap & \texttt{ExactNumber} \times \texttt{ExactNumber} \longrightarrow \texttt{ExactNumber}^{3} \\
\cap & \texttt{FloatComplex} \times \texttt{Number} \longrightarrow \texttt{FloatComplex}^{3} \\
\cap & \texttt{Number} \times \texttt{FloatComplex} \longrightarrow \texttt{FloatComplex}^{3} \\
\cap & \texttt{Float} \times \texttt{InexactComplex} \longrightarrow \texttt{FloatComplex}^{3} \\
\cap & \texttt{InexactComplex} \times \texttt{Float} \longrightarrow \texttt{FloatComplex}^{3} \\
\cap & \texttt{SingleFloatComplex} \times (\texttt{Rational} \cup \texttt{SingleFloat} \cup \texttt{SingleFloatComplex}) \\
 & \longrightarrow \texttt{SingleFloatComplex}^{3} \\
\cap & (\texttt{Rational} \cup \texttt{SingleFloat} \cup \texttt{SingleFloatComplex}) \times \texttt{SingleFloatComplex} \\
 & \longrightarrow \texttt{SingleFloatComplex}^{3} \\
\cap & \texttt{Number} \times \texttt{Number} \longrightarrow \texttt{Number}^{4}
\end{array}
$$

Key finite integer range[1] sign[2] compound category[3] cumulative type[4]

Figure B.5: Semantic Type of +

APPENDIX C

CDUCE NUMERIC TOWER

```
(* This file defines type synonymns meant to mimic the *)
(* Racket numeric tower. These are roughly the same base types *)
(* and unions Typed Racket uses today. However, Typed Racket *)
(* does not attempt to reason _completely_ about the types in *)
(* general (e.g., function application for an intersection of *)
(* arrows simply picks the first applicable arrow instead of *)
(* reasoning about all applicable arrows). *)

(* base numeric types *)
type zero = `Zero
type one = `One
type byteLargerThanOne = `ByteLargerThanOne
type posIndexNotByte = `PosIndexNotByte
type posFixnumNotIndex = `PosFixnumNotIndex
type negFixnum = `NegFixnum
type posIntegerNotFixnum = `PosIntegerNotFixnum
type negIntegerNotFixnum = `NegIntegerNotFixnum
type posRationalNotInteger = `PosRationalNotInteger
type negRationalNotInteger = `NegRationalNotInteger
type floatNaN = `FloatNaN
type floatPosZero = `FloatPosZero
type floatNegZero = `FloatNegZero
type posFloatNumber = `PosFloatNumber
type posFloatInfinity = `PosFloatInfinity
type negFloatNumber = `NegFloatNumber
type negFloatInfinity = `NegFloatInfinity
type singleFloatNaN = `SingleFloatNaN
type singleFloatPosZero = `SingleFloatPosZero
type singleFloatNegZero = `SingleFloatNegZero
type posSingleFloatNumber = `PosSingleFloatNumber
type posSingleFloatInfinity = `PosSingleFloatInfinity
type negSingleFloatNumber = `NegSingleFloatNumber
type negSingleFloatInfinity = `NegSingleFloatInfinity
type exactImaginary = `ExactImaginary
type exactComplex = `ExactComplex
type floatImaginary = `FloatImaginary
type singleFloatImaginary = `SingleFloatImaginary
type floatComplex = `FloatComplex
type singleFloatComplex = `SingleFloatComplex


(* compound numeric types *)
type posByte = (one | byteLargerThanOne)
type byte = (zero | posByte)
type posIndex = (one | byteLargerThanOne | posIndexNotByte)
type index = (zero | posIndex)
type posFixnum = (posFixnumNotIndex | posIndex)
type nonnegFixnum = (posFixnum | zero)
type nonposFixnum = (negFixnum | zero)
type fixnum = (negFixnum | zero | posFixnum)
type posInteger = (posIntegerNotFixnum | posFixnum)
type nonnegInteger = (zero | posInteger)
type negInteger = (negFixnum | negIntegerNotFixnum)
type nonposInteger = (negInteger | zero)
type integer = (negInteger | zero | posInteger)
type posRational = (posRationalNotInteger | posInteger)
type nonnegRational = (zero | posRational)
type negRational = (negRationalNotInteger | negInteger)
type nonposRational = (negRational | zero)
type rational = (negRational | zero | posRational)
type floatZero = (floatPosZero | floatNegZero | floatNaN)
type posFloat = (posFloatNumber | posFloatInfinity | floatNaN)
type nonnegFloat = (posFloat | floatZero)
type negFloat = (negFloatNumber | negFloatInfinity | floatNaN)
type nonposFloat = (negFloat | floatZero)
type float = (negFloatNumber | negFloatInfinity | floatNegZero
             | floatPosZero | posFloatNumber | posFloatInfinity
             | floatNaN)
type singleFloatZero = (singleFloatPosZero | singleFloatNegZero
                        | singleFloatNaN)
type inexactRealNaN = (floatNaN | singleFloatNaN)
type inexactRealPosZero = (singleFloatPosZero | floatPosZero)
type inexactRealNegZero = (singleFloatNegZero | floatNegZero)
type inexactRealZero = (inexactRealPosZero | inexactRealNegZero
                       | inexactRealNaN)
type posSingleFloat = (posSingleFloatNumber | posSingleFloatInfinity
                      | singleFloatNaN)
type posInexactReal = (posSingleFloat | posFloat)
type nonnegSingleFloat = (posSingleFloat | singleFloatZero)
type nonnegInexactReal = (posInexactReal | inexactRealZero)
type negSingleFloat = (negSingleFloatNumber | negSingleFloatInfinity
                      | singleFloatNaN)
type negInexactReal = (negSingleFloat | negFloat)
type nonposSingleFloat = (negSingleFloat | singleFloatZero)
type nonposInexactReal = (negInexactReal | inexactRealZero)
type singleFloat = (negSingleFloat | singleFloatNegZero | singleFloatPosZero
                   | posSingleFloat | singleFloatNaN)
type inexactReal = (singleFloat | float)
type posInfinity = (posFloatInfinity | posSingleFloatInfinity)
type negInfinity = (negFloatInfinity | negSingleFloatInfinity)
type realZero = (zero | inexactRealZero)
type realZeroNoNaN = (zero | inexactRealPosZero | inexactRealNegZero)
type posReal = (posRational | posInexactReal)
type nonnegReal = (nonnegRational | nonnegInexactReal)
type negReal = (negRational | negInexactReal)
type nonposReal = (nonposRational | nonposInexactReal)
type real = (rational | inexactReal)
type exactNumber = (exactImaginary | exactComplex | rational)
type inexactImaginary = (floatImaginary | singleFloatImaginary)
type imaginary = (exactImaginary | inexactImaginary)
type inexactComplex = (floatComplex | singleFloatComplex)
type number = (real | imaginary | exactComplex | inexactComplex)


(* Typed Racket's type for plus has about 125 arrows in its type. *)
(* However, Typed Racket reasons simply about function application; *)
(* Typed Racket simply picks the first applicable arrow and uses that. *)
(* The type below for plus is a condensed version that is just *)
(* as expressive as Typed Racket (for the 2 argument case) *)
(* when a more complete/precise calculation is used for function *)
(* application. *)
let plus ( (byte, byte) -> index
         ; (index, index) -> nonnegFixnum
         ; (negFixnum, one) -> nonposFixnum
         ; (one, negFixnum) -> nonposFixnum
         ; (nonposFixnum, nonnegFixnum) -> fixnum
         ; (nonnegFixnum, nonposFixnum) -> fixnum
         ; (integer, integer) -> integer
         ; (float, real) -> float
         ; (real, float) -> float
         ; (singleFloat, rational | singleFloat) -> singleFloat
         ; (rational | singleFloat, singleFloat) -> singleFloat
         ; (posReal, nonnegReal) -> posReal
         ; (nonnegReal, posReal) -> posReal
         ; (negReal, nonposReal) -> negReal
         ; (nonposReal, negReal) -> negReal
         ; (nonnegReal, nonnegReal) -> nonnegReal
         ; (nonposReal, nonposReal) -> nonposReal
         ; (real, real) -> real
         ; (exactNumber, exactNumber) -> exactNumber
         ; (floatComplex, number) -> floatComplex
         ; (number, floatComplex) -> floatComplex
         ; (float, inexactComplex) -> floatComplex
         ; (inexactComplex, float) -> floatComplex
         ; (singleFloatComplex, rational | singleFloat | singleFloatComplex)
           -> singleFloatComplex
         ; (rational | singleFloat | singleFloatComplex, singleFloatComplex)
           -> singleFloatComplex
         ; (number, number) -> number
         )
  | (a, b) -> raise (a,b)
;;

let applyToPair (f : ('a , 'b) -> 'c) (p : ('a , 'b)) : 'c = f p;;

(* This line takes an extremely long time to type check (> 15 min): *)
let addPosBytes (b1 : posByte) (b2 : posByte) : posIndex =
    applyToPair plus (b1, b2);;
```

REFERENCES

[1] Robert Cartwright. “User-defined data types as an aid to verifying LISP programs”. In: ICALP. 1976.

[2] Gilad Bracha and David Griswold. “Strongtalk: typechecking Smalltalk in a production environment.” In: OOPSLA. 1993.

[3] Microsoft Co. *TypeScript Language Specification.* http://www.typescriptlang.org. 2014.

[4] Ravi Chugh, David Herman, and Ranjit Jhala. “Dependent Types for JavaScript”. In: OOPSLA. 2012.

[5] Benjamin S. Lerner et al. “TeJaS: Retrofitting Type Systems for JavaScript”. In: *Proceedings of the 9th Symposium on Dynamic Languages*. DLS. 2013.

[6] Aseem Rastogi et al. “Safe & Efficient Gradual Typing for TypeScript”. In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’15. Mumbai, India: ACM, 2015, pp. 167–180. ISBN: 978-1-4503-3300-9.

[7] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. “Trust, but Verify: Two-Phase Typing for Dynamic Languages”. In: ECOOP. 2015.

[8] Avik Chaudhuri et al. “Fast and Precise Type Checking for JavaScript”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 48:1–48:30.

[9] Sam Tobin-Hochstadt and Matthias Felleisen. “Logical Types for Untyped Languages”. In: *Proc. ACM Intl. Conf. on Functional Programming.* ICFP. 2010.

[10] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. “Practical Optional Types for Clojure”. In: ESOP 2016. 2016.

[11] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. “Typed Lua: An Optional Type System for Lua”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2014).

[12] Facebook Inc. *Hack*. http://hacklang.org. 2014.

[13] Michael M. Vitousek et al. “Design and Evaluation of Gradual Typing for Python”. In: DLS. 2014.

[14] Sam Tobin-Hochstadt and Matthias Felleisen. “The Design and Implementation of Typed Scheme”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: ACM, 2008, pp. 395–406. ISBN: 978-1-59593-689-9.

[15] Ceylon Project. *The Ceylon Language*. https://ceylon-lang.org/documentation/1.3/spec/. Visited on 2019-02-14.

[16] David J. Pearce. “Sound and Complete Flow Typing with Unions, Intersections and Negations”. In: VMCAI. 2013.

[17] JetBrains. *Kotlin*. http://kotlinlang.org/docs/reference/. Visited on 2019-02-14.

[18] Apache Software Foundation. *Groovy*. http://groovy-lang.org. Visited on 2019-02-14.

[19] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. “The essence of JavaScript”. In: ECOOP. 2010.

[20] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. “Liquid Types”. In: *Proc. ACM Conf. on Programming Language Design and Implementation*. PLDI. 2008.

[21] Gavin M. Bierman et al. “Semantic Subtyping with an SMT Solver”. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’10. Baltimore, Maryland, USA: ACM, 2010, pp. 105–116. ISBN: 978-1-60558-794-3.

[22] Nikhil Swamy et al. “Secure Distributed Programming with Value-dependent Types”. In: *Proc. ACM Intl. Conf. on Functional Programming*. ICFP. 2011.

[23] Niki Vazou et al. “Refinement Types for Haskell”. In: *Proc. ACM Intl. Conf. on Functional Programming*. ICFP. 2014.

[24] Benjamin C. Pierce and David N. Turner. “Local Type Inference”. In: *ACM Trans. Program. Lang. Syst.* (2000).

[25] Facebook Inc. *Flow: A static type checker for JavaScript*. http://flowtype.org. 2014.

[26] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. “Typing Local Control and State Using Flow Analysis”. In: *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*. ESOP. 2011.

[27] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. “Set-Theoretic Types for Polymorphic Variants”. In: ICFP. 2016.

[28] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. “Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types”. In: *J. ACM* 55.4 (Sept. 2008), 19:1–19:64.

[29] Simon Peyton Jones et al. “Simple Unification-based Type Inference for GADTs”. In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP ’06. Portland, Oregon, USA: ACM, 2006, pp. 50–61. ISBN: 1-59593-309-3.

[30] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. “Practical Optional Types for Clojure”. In: *Thiemann P. (eds) Programming Languages and Systems*. ESOP ’06. Springer, 2006, pp. 68–94.

[31] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. “Nested Refinements: A Logic for Duck Typing”. In: *Proc. ACM Sym. on Principles of Programming Languages*. POPL. 2012.

[32] Jessica Gronski et al. “Sage: Hybrid Checking for Flexible Specifications”. In: *Proc. Wksp. on Scheme and Functional Programming*. 2006.

[33] Kenneth Knowles and Cormac Flanagan. “Compositional Reasoning and Decidable Checking for Dependent Contract Types”. In: PLPV. 2009.

[34] Xinming Ou et al. “Dynamic Typing with Dependent Types”. In: *IFIP Intl. Conf. on Theoretical Computer Science* (2004).

[35] Benjamin Cosman and Ranjit Jhala. “Local Refinement Typing”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), 26:1–26:27.

[36] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. "Refinement Types for TypeScript". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: ACM, 2016, pp. 310–325. ISBN: 978-1-4503-4261-2.

[37] Sam Tobin-Hochstadt and Matthias Felleisen. "Interlanguage Migration: From Scripts to Programs". In: DLS. 2006.

[38] Esteban Allende et al. "Gradual Typing for Smalltalk". In: *Science of Computer Programming* (2014).

[39] Matthew Fluet and Riccardo Pucella. "Practical Datatype Specializations with Phantom Types and Recursion Schemes". In: *Electronic Notes in Theoretical Computer Science* (2006).

[40] Stephanie Weirich. "Depending on Types". In: *Proc. ACM Intl. Conf. on Functional Programming*. ICFP. 2014.

[41] Hongwei Xi and Frank Pfenning. "Eliminating Array Bound Checking Through Dependent Types". In: *Proc. ACM Conf. on Programming Language Design and Implementation*. PLDI. 1998.

[42] George B. Dantzig and B. Curtis Eaves. "Fourier-Motzkin Elimination and Its Dual". In: *J. Combinatorial Theory Series A* (1973).

[43] Hongwei Xi. "Dependent ML: An Approach to Practical Programming with Dependent Types". In: *J. Functional Programming* (2007).

[44] Leonardo De Moura and Nikolaj Bjorner. "Z3: An Efficient SMT Solver". In: TACAS. 2008.

[45] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. 2009.

[46] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[47] Matthew Flatt and PLT. *Reference: Racket*. Tech. rep. PLT-TR-2010-1. https://racket-lang.org/tr1. PLT Design Inc., 2010.

[48] Christos Dimoulas et al. “Correct Blame for Contracts: No More Scapegoating”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 215–226. ISBN: 978-1-4503-0490-0.

[49] Chiyan Chen and Hongwei Xi. “Combining Programming with Theorem Proving”. In: *Proc. ACM Intl. Conf. on Functional Programming*. ICFP. 2005.

[50] Nikhil Swamy et al. “Dependent Types and Multi-monadic Effects in F*”. In: *Proc. ACM Sym. on Principles of Programming Languages*. POPL. 2016.

[51] Kenneth Knowles and Cormac Flanagan. “Hybrid Type Checking”. In: *ACM Trans. Program. Lang. Syst.* (2010).

[52] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. “Contracts Made Manifest”. In: *Proc. ACM Sym. on Principles of Programming Languages*. POPL. 2010.

[53] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. “CDuce: An XML-centric General-purpose Language”. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’03. Uppsala, Sweden: ACM, 2003, pp. 51–63. ISBN: 1-58113-756-7.

[54] The Pony Developers. *Pony*. https://www.ponylang.io/. Visited on 2019-02-18.

[55] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’00. New York, NY, USA: ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6.

[56] Alain Frisch. *Théorie, conception et réalisation d’un langage adapté à XML*. Ph.D thesis (in French). 2004.

[57] Giuseppe Castagna. “Covariance and Controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers)”. In: *CoRR* abs/1809.01427 (2018). arXiv: `1809.01427`.

[58] David J. Pearce. "Rewriting for Sound and Complete Union, Intersection and Negation Types". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* GPCE 2017. Vancouver, BC, Canada: ACM, 2017, pp. 117–130. ISBN: 978-1-4503-5524-7.

[59] Giuseppe Castagna and Alain Frisch. "A Gentle Introduction to Semantic Subtyping". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming.* PPDP '05. Lisbon, Portugal: ACM, 2005, pp. 198–199. ISBN: 1-59593-090-6.

[60] Giuseppe Castagna et al. "Polymorphic Functions with Set-theoretic Types: Part 1: Syntax, Semantics, and Evaluation". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '14. San Diego, California, USA: ACM, 2014, pp. 5–17. ISBN: 978-1-4503-2544-8.

[61] Giuseppe Castagna et al. "Polymorphic Functions with Set-Theoretic Types: Part 2: Local Type Inference and Type Reconstruction". In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '15. Mumbai, India: ACM, 2015, pp. 289–302. ISBN: 978-1-4503-3300-9.

[62] Giuseppe Castagna and Victor Lanvin. "Gradual Typing with Union and Intersection Types". In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), 41:1–41:28.

[63] Davide Ancona et al. "Semantic subtyping for non-strict languages". In: *24th International Conference on Types for Proofs and Programs (TYPES 2018).* 2018.

[64] G. Castagna, R. De Nicola, and D. Varacca. "Semantic subtyping for the π-calculus". In: *Theoretical Computer Science* 398.1-3 (2008). Essays in honour of Mario Coppo, Mariangiola Dezani-Ciancaglini and Simona Ronchi della Rocca, pp. 217–242.

[65] Fabian Muehlboeck and Ross Tate. "Empowering Union and Intersection Types with Integrated Subtyping". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 112:1–112:29.

[66] Paul Lorenzen. "Konstruktive Begründung der mathematik". In: *Mathematische Zeitschrift* 53 (1950).

[67] Enrico Moriconi and Laura Tesconi. “On Inversion Principles”. In: *History and Philosophy of Logic* 29 (May 2008), pp. 103–113.

[68] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. “Occurrence Typing Modulo Theories”. In: *Proc. ACM Conf. on Programming Language Design and Implementation*. PLDI. 2016.

[69] Andrew K. Wright and Matthias Felleisen. “A Syntactic Approach to Type Soundness”. In: *Information and Computation* 115.1 (Nov. 1994), pp. 38–94.

[70] Vincent St-Amour et al. “Typing the Numeric Tower”. In: *Proceedings of the 14th International Conference on Practical Aspects of Declarative Languages*. PADL’12. Philadelphia, PA: Springer-Verlag, 2012, pp. 289–303. ISBN: 978-3-642-27693-4.

[71] Giuseppe Castagna. Private communication. 2018-09-06.

[72] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN: 0262026651, 9780262026659.

Andrew M. Kent

Researcher · Software Engineer · Programming Language Specialist

pnwamk@gmail.com | pnwamk.github.io | pnwamk

Education

Indiana University *Bloomington, Indiana*

Ph.D. in Computer Science *May 2014 - Oct. 2019*

- Dissertation topic: Advanced Logical Type Systems for Untyped Languages.
- Advised by Sam Tobin-Hochstadt.

Indiana University *Bloomington, Indiana*

M.S. in Computer Science *May 2014 - May 2017*

Brigham Young University *Provo, Utah*

B.S. in Computer Science *Aug. 2010 - Aug. 2013*

- Graduated magna cum laude.

Technical Skills

Programming	Racket, Haskell, C/C++, Java, Scala, Python, unix tools, LaTeX, etc
Verification	Coq

Experience

Galois, Inc. *Portland, Oregon*

Research Engineer *Apr. 2019 - Present*

- Investigate and develop of research software and technologies leveraging programming language and verification techniques.

Indiana University *Bloomington, Indiana*

Graduate Research Assistant *May 2014 - Mar. 2019*

- Developed novel technique for adding refinement types to a type system for untyped languages (i.e. Typed Racket) which is included in releases of the Racket programming language since v6.11.
- Modeled and mechanically verified a novel technique for combining semantic subtyping with a type system for untyped languages.
- Advised by Sam Tobin-Hochstadt.

Instructor (CSCI-B 490/629 Dependent Types) *Spring 2018*

- Taught introductory dependent types course based on Friedman and Christiansen's "The Little Typer".

Microsoft Research Ltd. *Cambridge, UK*

Research Intern *May 2017 - Jul. 2017*

- Developed and prototyped unique solutions to trusted computing problems in the cloud leveraging TPM and SGX/Enclave technologies. (U.S. patent application 20190163898.)
- Advised by Sylvan Clebsch.

Brigham Young University *Provo, Utah*

Graduate Research Assistant *Aug. 2013 - Apr. 2014*

- Investigated the formalization of security protocol analysis techniques (Strand Spaces) utilizing the Coq proof assistant.
- Advised by Dr. Jay McCarthy.

Microsoft Corporation *Redmond, Washington*

Software Development Engineer Intern *May 2012 - Aug. 2012*

- Explored optimizations and improvements for Microsoft OneNote during a summer internship.

Brigham Young University *Provo, Utah*

Undergraduate Research Assistant *May 2011 - Sep. 2011*

- Developed method for automatically generating historical social networks from source documents.
- Advised by Dr. William Berret and Dr. Tom Sederberg.

United States Marine Corps *Camp Pendleton, California*

Signals Intelligence Analyst *Nov. 2005 - Aug. 2010*

- Provided SIGINT analysis and reporting in support of military operations during deployments in 2008 and 2009; led and trained team of five SIGINT analysts during 2009 deployment.
- Received honorable discharge at the rank of Sergeant.

Publications

Migratory Typing: Ten Years Later *SNAPL*

S. Tobin-Hochstadt, M. Felleisen, R. B. Findler, M. Flatt, B. Greenman, **A. M. Kent**, V. St-Amour, , T. S. Strickland, A. Takikawa. *Proc. 2nd Summit on Advances in Programming Languages.* *2017*

Occurrence Typing Modulo Theories *PLDI*

A. M. Kent, D. Kempe, S. Tobin-Hochstadt. *Proc. 37th ACM Conf. on Programming Language Design and Implementation.* *2016*

Design and Evaluation of Gradual Typing for Python *DLS*

M.M. Vitousek, **A. M. Kent**, J.G. Siek, J. Baker. *Proc. 10th ACM Symposium on Dynamic Languages.* *2014*

Linking the Past: Discovering Historical Social Networks from Documents and Linking to a Genealogical Database *HIP*

D.J. Kennard, **A. M. Kent**, W.A. Barret. *Proc. 1st Workshop on Historical Document Imaging and Processing.* *2011*

Open Source and Community Involvement

Typed Racket *github.com/racket/typed-racket*

Core Contributor *Dec. 2014 - Mar. 2019*

- Performed significant refactorings to improve performance and code maintainability.
- Added refinement types (v6.11) and other features/enhancements to the type system.
- Increased coverage of manual and random test suites.
- Helped adopt (Rust-inspired) RFC process to better coordinate contributions to the type system.

Racket *github.com/racket/racket*

Contributor *Oct. 2016 - Mar. 2019*

- Contributed occasional bug fixes, features, and improvements to the core Racket language.

Interfaith Winter Shelter Volunteer *Bloomington, Indiana*

Volunteer *Jan. 2016 - Mar. 2018*

- Semi-regular night-shift volunteer at shelter for the homeless during winter months.

Honors, Awards, Etc

2017	**People's Choice Runner-up**, MSR Cambridge Hackathon	*Cambridge, UK*
2013	**Fellowship**, NASA Space Grant Consortium	*Provo, Utah*
2007	**1st in class**, Intermediate Comm. Signals Analysis Course, Naval Center for Information Warfare Training	*Pensacola, Florida*
2006	**TS/SCI Clearance**, expired approximately 2011	
2002	**Eagle Scout**, Boy Scouts of America	*Camas, Washington*