A Relational Framework for Higher-Order Shape Analysis

Gowtham Kaki Suresh Jagannathan
Purdue University
{gkaki,suresh}@cs.purdue.edu

Abstract

We propose the integration of a relational specification framework within a dependent type system capable of verifying complex invariants over the shapes of algebraic datatypes. Our approach is based on the observation that structural properties of such datatypes can often be naturally expressed as inductively-defined *relations* over the recursive structure evident in their definitions. By interpreting constructor applications (abstractly) in a relational domain, we can define expressive relational abstractions for a variety of complex data structures, whose structural and shape invariants can be automatically verified. Our specification language also allows definitions of *parametric* relations for polymorphic data types that enables highly composable specifications and naturally generalizes to higher-order polymorphic functions.

We describe an algorithm that translates relational specifications into a decidable fragment of first-order logic that can be efficiently discharged by an SMT solver. We have implemented these ideas in a type checker called CATALYST that is incorporated within the MLton SML compiler. Experimental results and case studies indicate that our verification strategy is both practical and effective.

1. Introduction

Dependent types are well-studied vehicles capable of expressing rich program invariants. A prototypical example is the type of a list that is indexed by a natural number denoting its length. Length-indexed lists can be written in several mainstream languages that support some form of dependent typing, including GHC Haskell [2], F* [1**?**], and OCaml [3]. For example, the following Haskell signatures specify how the length of the result list for `append` and `rev` relate to their arguments:

```
append :: List a n -> List a m -> List a (Plus n m)
rev    :: List a n -> List a n
```

While length-indexed lists capture stronger invariants over `append`, and `rev` than possible with just simple types, they still underspecify the intended behavior of these operations. For example, a correctly written `append` function must additionally preserve the order of its input lists; a function that incorrectly produces an output list that is a permutation of its inputs would nonetheless satisfy `append`'s type as written above. Similarly, the identity function would clearly satisfy the type given for `rev`; a type that fully captures `rev`'s behavior would also have to specify that the order of elements in `rev`'s output list is the inverse of the order of its input. Is it possible to ascribe such expressive types to capture these kinds of important shape properties, which can nonetheless be easily stated, and efficiently checked?

One approach is to directly state desired behavior in type refinements, as in the following signature:

`rev : {l : 'a list}` $\longrightarrow$ `{`ν`: 'a list | `ν` = rev'(l)}`

Here, `rev'` represents some reference implementation of `rev`. Checking `rev`'s implementation against this refinement is tantamount to proving the equivalence of `rev` and `rev'`. Given the undecidability of the general problem, expecting these types to be machine checkable would require the definition of `rev'` to closely resemble `rev`'s. For all but the most trivial of definitions, this approach is unlikely to be fruitful. An alternative approach is to define `rev` within a theorem prover, and directly assert and prove properties on it - for example, that `rev` is involutive. Although modern theorem provers support rich theories over datatypes, this strategy nonetheless requires that the program be fully described in logic, and reasoned about by the solver in its entirety. For example, defining `rev` in this way also requires an equational definition of `append`, assuming the former is defined in terms of the latter. For non-trivial programs, this may require equipping provers with arbitrarily complex theories, whose combination may not be decidable. Such a methodology also does not obviously address our original goal of specifying `rev`'s functional correctness, independent of its definition; note that in the case of `rev`, involution does not imply functional correctness. Clearly, the challenges in building suitably typed definitions that let us reason about interesting shape properties of a data structure are substantial.

Nonetheless, the way the length of a list is tracked using its length-indexed type offers a useful hint about how we can reason about its shape. Akin to the `Nat` domain that indexes a list type with a length abstraction, we need an appropriate abstract domain that we can use to help us reason about a list's shape properties. For instance, in the case of list reversal, the abstract domain should allow us to structurally reason about the order of elements in the input and output lists. A useful interpretation of a list order that satisfies this requirement would be one that relates every element in a list with every another element based on an ordering predicate (e.g., *occurs-before* or *occurs-after*). By defining an exhaustive enumeration of the set of all such pairs under this ordering, we can effectively specify the total order of all elements in the list. More precisely, we note that the notion of order can be broken down to the level of a binary relation over elements in the list, with the transitive closure of such a relation effectively serving as a faithful representation.

For example, consider a relation R_{ob} that relates a list to a pair if the first element in the pair *occurs before* the second in the list. For a concrete list `l=[x1,x2,x3]`, its closure R^*_{ob} would be:

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/10.1145/2628136.2628159

$$\{\langle \texttt{l}, \langle \texttt{x1}, \texttt{x2}\rangle\rangle, \langle \texttt{l}, \langle \texttt{x1}, \texttt{x3}\rangle\rangle, \langle \texttt{l}, \langle \texttt{x2}, \texttt{x3}\rangle\rangle\}$$ [1]

Conversely, an *occurs-after* (R_{oa}) relation serves as the semantic inverse of *occurs-before*; given these two relations, we can specify the following type for `rev`:

$$\texttt{rev : \{ l : 'a list\}} \longrightarrow \{\nu\texttt{: 'a list |}\ R^*_{ob}(\texttt{l}) = R^*_{oa}(\nu)\}$$

Since $R^*_{ob}(\texttt{l})$ represents the set of pairs whose elements exhibit the *occurs-before* property in the input list, and $R^*_{oa}(\nu)$ represents the set of pairs whose elements exhibit the *occurs-after* property in the output list, the above specification effectively asserts that for every pair of elements x and y in the input list `l`, if x occurs before y in `l`, then x has to occur after y in the result list ν.

This property succinctly captures the fact that the result list is the same as the original list in reverse order without appealing to the operational definition of how the result list is constructed from the input. By using a relational domain to reason about the shape of the list, we avoid having to construct a statically checkable reference implementation of `rev`.

We refer to operators like R_{ob} and R_{oa} as *structural relations* because they explicitly describe structural properties of a data structure. Such relations can be used as appropriate abstract domains to reason about the shapes of structures generated by constructor applications in algebraic data types. Given that relations naturally translate to sets of tuples, standard set operations such as union and cross-product are typically sufficient to build useful relational abstractions from any concrete domain. This simplicity makes relational specifications highly amenable for automatic verification.

The type of `rev` given above captures its functional behavior by referring to the order of elements in its argument and result lists. However, the notion of order as a relation between elements of the list is not always sufficient. For example, consider the function,

```
dup : 'a list → ('a*'a) list
```

that duplicates the elements in its input list. An invariant that we can expect of any correct implementation is that the order of left components of pairs in the output list is the same as the order of its right components, and both are equal to the order of elements in the input list. Clearly, our definitions of R_{ob} and R_{oa} as relations over elements in a list are insufficient to express the order of individual components of pairs in a list of pairs. How do we construct general definitions that let us capture ordering invariants over different kinds of lists without generating distinct relations for each kind?

We address this issue by allowing structural relations defined over a polymorphic data type to be parameterized by relations over type variables in the data type. For instance, the R_{ob} relation defined over a `'a list` can be parameterized by a polymorphic relation R over `'a`. Instead of directly relating the order of two elements x and y in a polymorphic list, a parametric *occurs-before* relation generically relates the ordering of $R(x)$ and $R(y)$; R's specific instantiation would draw from the set of relations defined over the data type that instantiates the type variable (`'a`). In the case of `dup`, R_{ob} could be instantiated with relations like R_{fst} and R_{snd} that project the first and second elements of the pairs in `dup`'s output list. The ability to parameterize relations in this way allows structural relations to be used seamlessly with higher-order polymorphic functions, and enables composable specifications over defined relations.

[1] Given a relation $R = \{\langle x, y_1\rangle, \langle x, y_2\rangle, \ldots, \langle x, y_n\rangle\}$ where x is an instance of some datatype, and the y_i are tuples that capture some shape property of interest, we write $R(x)$ as shorthand for $\{y_1, y_2, \ldots, y_n\}$. Thus,

$$\texttt{R}^*_{ob}(\texttt{l}) = \{\langle \texttt{x1}, \texttt{x2}\rangle, \langle \texttt{x1}, \texttt{x3}\rangle, \langle \texttt{x2}, \texttt{x3}\rangle\}$$

In this paper, we present an automated verification framework integrated within a refinement type system to express and check specifications of the kind given above. We describe a specification language based on relational algebra to define and compose structural relations for any algebraic data type. These definitions are only as complex as the data type definition itself in the sense that it is possible to construct equivalent relational definitions directly *superimposed* on the data type. Relations thus defined, including their automatically generated inductive variants, can be used to specify shape invariants and other relational properties. Our typechecking procedure verifies specifications by interpreting constructor applications as set operations within these abstract relational domains. Typechecking in our system is decidable, a result which follows from the completeness of encoding our specification language in a decidable logic.

The paper makes the following contributions:

1. We present a rich specification language for expressing refinements that are given in terms of relational expressions and familiar relational algebraic operations. The language is equipped with pattern-matching operations over constructors of algebraic data types, thus allowing the definition of useful shape properties in terms of relational constraints.
2. To allow relational refinements to express shape properties over complex data structures, and to be effective in defining such properties on higher-order programs, we allow the inductive relations found in type refinements to be parameterized over other inductively defined relations. While the semantics of a relationally parametric specification can be understood intuitively in second-order logic, we show that it can be equivalently encoded in a decidable fragment of first-order logic, leading to a practical and efficient type-checking algorithm.
3. We present a formalization of our ideas, including a static semantics, meta-theory that establishes the soundness of well-typed programs, a translation mechanism that maps well-typed relational expressions and refinements to a decidable many-sorted first-order logic, and a decidability result that justifies the translation scheme.
4. We describe an implementation of these ideas in a type checker called CATALYST that is incorporated within the MLton Standard ML compiler, and demonstrate the utility of these ideas through a series of examples, including a detailed case study that automatically verifies the correctness of α-conversion and capture-avoiding substitution operations of the untyped lambda calculus, whose types are expressed using relational expressions.

The remainder of the paper is as follows. In the next section, we present additional motivation and examples for our ideas. Section 3 formalizes the syntax and static semantics of relational refinements in the context of a simply-typed core language. Section 4 extends the formalization to support parametric refinements within a polymorphic core language. Our formalization also presents a translation scheme from relational refinements to a decidable first-order logic. Details about the implementation are given in Section 5. Section 6 presents a case study. Section 8 discusses the expressive power and current limitations of the type system. Sections 7 and 9 present related work and conclusions.

2. Structural Relations

Our specification language is primarily the language of relational expressions composed using familiar relational algebraic operators. This language is additionally equipped with pattern matching over

constructors of algebraic types to define shape properties in terms of these expressions. A number of built-in polymorphic relations are provided, the most important of which are listed below:

$$\begin{array}{lcl} R_{id}\ (\mathtt{x}) & = & \{\langle \mathtt{x}\rangle\} \\ R_{dup}\ (\mathtt{x}) & = & \{\langle \mathtt{x}, \mathtt{x}\rangle\} \\ R_{notEq_k}\ (\mathtt{x}) & = & \{\langle \mathtt{x}\rangle\} - \{\langle \mathtt{k}\rangle\} \\ R_{eq_k}\ (\mathtt{x}) & = & \{\langle \mathtt{x}\rangle\} - (\{\langle \mathtt{x}\rangle\} - \{\langle \mathtt{k}\rangle\}) \end{array}$$

R_{id} is the identity relation, R_{dup} is a relation that associates a value with a pair that duplicates that value, R_{notEq_k} is a relation indexed by a constant `k` (of some base type) that relates `x` to itself, provided `x` is not equal to `k`, and R_{eq_k} is defined similarly, except it relates `x` to itself exactly when `x` is equal to `k`. Apart from the relations defined above, the language also includes the primitive relation $\emptyset$ that denotes the empty set.

To see how new structural relations can be built using relational operators, primitive relations, and pattern-match syntax, consider the specification of the *list-head* relation that relates a list to its head element:

$$\begin{array}{l} \mathtt{relation}\ R_{hd}\ \ (\mathtt{x::xs}) \ = \ \{\langle \mathtt{x}\rangle\} \\ \qquad\qquad \mid R_{hd}\ \ [\,] \qquad = \ \emptyset \end{array}$$

For a concrete list `l`, $R_{hd}(\mathtt{l})$ produces the set of unary tuples whose elements are in the *head* relation with `l`. This set is clearly a singleton when the list is non-empty and empty otherwise. The above definition states that for any list pattern constructed using "::" whose head is represented by pattern variable `x` and whose tail is represented by pattern variable `xs`, (1) $\langle \mathtt{x} :: \mathtt{xs}, \mathtt{x}\rangle \in R_{hd}$, and (2) there does not exist `x'` such that $\mathtt{x}' \neq \mathtt{x}$ and $\langle \mathtt{x} :: \mathtt{xs}, \mathtt{x}'\rangle \in R_{hd}$. The declarative syntax of the kind shown above is the primary means of defining structural relations in our system.

2.1 Relational Composition

Simple structural relations such as R_{hd} have fixed cardinality , i.e., they have a fixed number of tuples regardless of the concrete size of the data structure on which they are defined. However, practical verification problems require relations over algebraic datatypes to have cardinality comparable to the size of the data structure, which may be recursive.

For example, the problem of verifying that an implementation of `rev` reverses the ordering of its input requires specifying a *membership* relation (R_{mem}) that relates a list `l` to every element in `l` (regardless of `l`'s size). This relation would allow us to define an ordering property such as *occurs-before* or *occurs-after* on precisely those elements that comprise `rev`'s input and output lists. A recursive definition of R_{mem} looks like [2]:

$$R_{mem}\ (\mathtt{x} :: \mathtt{xs}) \ = \ \{\langle \mathtt{x}\rangle\} \ \cup \ R_{mem}\ (\mathtt{xs})$$

We can equivalently express R_{mem} as an *inductive extension* of the head relation R_{hd} defined above. Suppose R is a structural relation that relates a list l of type `'a list` with elements v of type `'a`. Then, the inductive extension of R (written R^*) is the least relation that satisfies the following conditions:

- if $\langle l, v\rangle \in R$, then $\langle l, v\rangle \in R^*$
- if $l = x :: xs$ and $\langle xs, v\rangle \in R$ then $\langle l, v\rangle \in R^*$

Thus, $R_{mem} = R^*_{hd}$. We can think of the induction operator as a controlled abstraction for structural recursion. Based on the recursive structure of an algebraic data type, sophisticated inductive definitions can be generated from simple structural relations defined for that data type.

[2] In our examples, we elide the case for the empty list which defaults to the empty set.

Equipped with R_{mem}, we can now precisely define the *occurs-before* relation defined earlier. Because R_{ob} relates a list to a pair whose first element is the head of the list, and whose second element is a member of its tail, it can be expressed in terms of R_{mem} thus:

$$\mathtt{relation}\ R_{ob}\ (\mathtt{x} :: \mathtt{xs}) \ = \ \{\langle \mathtt{x}\rangle\} \times R_{mem}\ (\mathtt{xs})$$

The transitive closure of this relation R^*_{ob} expresses the *occurs-before* property on every element in the list. The *occurs-after* relation can be defined similarly:

$$\mathtt{relation}\ R_{oa}\ (\mathtt{x} :: \mathtt{xs}) \ = \ R_{mem}\ (\mathtt{xs}) \ \times \{\langle \mathtt{x}\rangle\}$$

2.2 Parametric Relations

Consider how we might specify a `zip` function over lists, with the following type:

$$\mathtt{zip} : \mathtt{'a\ list} \rightarrow \mathtt{'b\ list} \rightarrow (\mathtt{'a\ *'b})\ \mathtt{list}$$

Any correct implementation of `zip` must guarantee that the elements of the output list are pairs of elements drawn from both argument lists. The R_{mem} relation defined above provides much of the functionality we require to specify this invariant; intuitively, the specification should indicate that the first (resp. second) element of every pair in the output list is in a membership relation with `zip`'s first (resp. second) argument. Unfortunately, as currently defined, R_{mem} operates directly on the pair elements of the output, not the pair's individual components. What we require is a mechanism that allows R_{mem} to assert the membership property on the pair's components (rather than the pair directly).

To do this, we allow structural relations to be *parameterized* over other relations. In the case of `zip`, the parameterized membership relation can be instantiated with the appropriate relationally-defined projections on a pair type. Concretely, given new parameterized definitions of R_{hd} and R_{mem}, and related auxiliary relations:

$$\begin{array}{l} \mathtt{relation}\ (R_{hd}\ R)\ (\mathtt{x::xs}) \ = \ R\ (\mathtt{x}) \\ \qquad\qquad \mid (R_{hd}\ R)\ [\,] \qquad = \ \emptyset \\ \mathtt{relation}\ (R_{mem}\ R) \ = \ (R_{hd}\ R)^* \\ \mathtt{relation}\ R_{fst}\ (\mathtt{x},\mathtt{y}) = \{\langle \mathtt{x}\rangle\} \\ \mathtt{relation}\ R_{snd}\ (\mathtt{x},\mathtt{y}) = \{\langle \mathtt{y}\rangle\} \end{array}$$

`zip` can now be assigned the following type that faithfully captures the membership relation between its input lists and its output[3]:

$$\begin{array}{l} \mathtt{zip} : \mathtt{l}_1 \rightarrow \mathtt{l}_2 \rightarrow \\ \qquad \{\ \nu \ \mid \ ((R_{mem}\ R_{fst})\ \nu) = ((R_{mem}\ R_{id})\ \mathtt{l}_1\ ) \\ \qquad\quad \wedge\ ((R_{mem}\ R_{snd})\ \nu) = ((R_{mem}\ R_{id})\ \mathtt{l}_2\ )\ \} \end{array}$$

Similarly, we can define parametric versions of R_{ob} and R_{oa}:

$$\begin{array}{l} \mathtt{relation}\ (R_{ob}\ R)\ (\mathtt{x:xs}) = \ R\,(\mathtt{x}) \ \times ((R_{mem}\ R)\ \mathtt{xs}) \\ \mathtt{relation}\ (R_{oa}\ R)\ (\mathtt{x:xs}) = \ ((R_{mem}\ R)\ \mathtt{xs}) \ \times R\,(\mathtt{x}) \end{array}$$

Using R_{ob}, the `dup` function described in the previous section can now be specified thus:

$$\begin{array}{l} \mathtt{dup} : \mathtt{l} \rightarrow \{\ \nu \ \mid \ ((R_{ob}\ R_{fst})^*\ \nu) = ((R_{ob}\ R_{id})^*\ \mathtt{l}) \\ \qquad\qquad\qquad \wedge\ ((R_{ob}\ R_{snd})^*\ \nu) = ((R_{ob}\ R_{id})^*\ \mathtt{l})\ \} \end{array}$$

2.3 Parametric Dependent Types

Our specification language also allows dependent types to be parameterized over relations used in type refinements. In the spirit of type variables, we use relation variables to denote parameterized

[3] We drop ML types from dependent type specifications when obvious from context.

relations in a type. To illustrate why such parameterization is useful, consider the following signature for `foldl`:

```
('R_bm) foldl :
    {l : 'a list} → {b : 'b} →
      ({f: {x :'a} → {acc : 'b} →
           {z : 'b | 'R_bm(z) = {⟨x⟩} ∪ 'R_bm(acc) }}) →
    {ν | 'R_bm(ν) = R_mem (l) ∪ 'R_bm(b)}
```

This type relates membership properties on `foldl`'s input list, expressed in terms of a non-parametric R_{mem} relation, to an abstract notion of membership over its result type (`'b`) captured using a relation variable ($'R_{bm}$). This signature constrains `foldl` to produce a result for which a membership property is a sensible notion. Clearly, if `'b` is instantiated to some base type, the constraints imposed by the structural relations in `foldl`'s signature would not hold, and such an application would be considered ill-typed. On the other hand, if `foldl` were applied to arguments in which `b` was of some list type (e.g., `[]`) because it is used as a list transform operator, then $'R_{bm}$ could be trivially instantiated with R_{mem}. However, allowing types to be parameterized over relation variables allow richer properties to be expressed. For example, consider the function `makeTree` that uses `foldl` to generate a binary tree using function `treeInsert` (not shown):

```
datatype 'a tree = Leaf
                 | Tree of 'a * ('a Tree) * ('a Tree)

relation R_thd Leaf = ∅
       | R_thd (Tree (x,t1,t2)) = {⟨x⟩}
relation R_tmem = R*_thd

makeTree : {l : 'a list} →
           {ν : 'a tree | R_tmem(ν) = R_mem(l)}
val makeTree = fn l =>
      foldl (R_tmem) l Leaf treeInsert
```

Function `makeTree` uses `foldl` by first instantiating the relation variable $'R_{bm}$ in the type of `foldl` to R_{tmem}. The resultant type of `foldl` requires its higher-order argument to construct a tree using members of its tree argument (`acc`), and the list element (`x`) that it is applied to. In return, `foldl` guarantees to produce a tree, which contains all the members of its list argument. It should be noted that a correct implementation of `treeInsert` will have the required type of `foldl`'s higher-order argument, after instantiating $'R_{bm}$ to R_{tmem}. Thus, the application of `foldl` in the above example typechecks, producing the required invariant of `makeTree`.

`Foldl`'s type can also be parameterized over an abstract notion of membership for type variable `'a`, captured by another relation variable ($'R_{am}$) to state a more general membership invariant. Concretely, this requires that the tuple ($\{\langle x \rangle\}$) in the type refinement of higher-order argument (`f`) be replaced with $'R_{am}(x)$), and the non-parametric R_{mem} relation in the result type refinement be substituted with a parametric (R_{mem} $'R_{am}$) relation. In cases when there does not exist any useful notion of membership for types that instantiate `'a` and `'b`, relation variables $'R_{am}$ and R_{bm} can be instantiated with $\emptyset$ to yield tautological type refinements.

An alternative type for `foldl` could relate the order of elements in the argument list to some order of the result. The intuition is as follows: suppose the result type (`'b`) has some notion of order captured by a relation such that the result of `foldl`'s higher-order argument (`f`) has a refinement given in terms of this relation; i.e., it says something about how the order relation of its result (`z`) relates to its arguments (`x` and `acc`). But, `x` comes from the list being folded, and `f` is applied over elements of this list in a pre-defined order. Therefore, we can express invariants that relate the order of the input list to the order of the result type, given that we know the order in which `f` is applied over the list. The type of `foldl` that tries to match the abstract order ($'R_{bo}$) on the result type (`'b`) to an occurs-after order on the input list is shown below. For brevity, we avoid reproducing membership invariants from the type of `foldl` from the previous example, using ellipses in their place:

```
('R_bm,'R_bo) foldl :{l : 'a list} → {b : 'b} →
              ({f : {x : 'a} → {acc : 'b} →
                    {z | 'R_bo(z) = ({⟨x⟩} × 'R_bm(acc)) ∪
                                    'R_bo(acc) ∧ ...}) →
              {ν | 'R_bo(ν) = R*_oa (l) ∪ 'R_bo(b)) ∪
                              ((R_mem (l)) × 'R_bm(b)) ∧ ...}
```

An implementation of `rev` that uses `foldl` is given below:

```
rev : {l : 'a list} → {ν : 'a list | R*_ob(ν) = R*_oa(l)}
val Cons = fn x => fn xs => x::xs
val rev = fn l => foldl (R_mem,R*_ob) l [] Cons
```

Our type checker successfully typechecks the above program, given the standard definition of `foldl`. Note that, due to the difference in the order in which the higher-order argument is applied over the input list, the type of `foldr` will be necessarily different from `foldl`. Consequently, using `foldr` instead of `foldl` in the above program fails type checking, as would be expected.

Calculus λ_R

$x, y, z, \nu \in$ *variables* $\quad n \in$ *integers*

c	::=	Cons | Nil | n	*constants*
v	::=	$x \mid \lambda(x:\tau).e \mid c \mid \mathsf{Cons}\, v \mid \mathsf{Cons}\, v\, v$	*value*
e	::=	$v \mid e\, v \mid$ `let` $x = e$ `in` $e \mid$ `match` v `with` Cons $x\ y \Rightarrow e$ `else` e	*expression*
T	::=	int | intlist	*datatypes*
τ	::=	$\{\nu : T \mid \phi\} \mid x : \tau \to \tau$	*dep. types*

Specification Language

R	$\in$	*relation names*	
r	::=	$R(v) \mid r \cup r \mid r \times r$	*relational exp.*
ϕ	::=	$r = r \mid r \subseteq r \mid \phi \wedge \phi \mid \phi \vee \phi \mid true$	*type refinement*
Δ_R	::=	$\langle R, \tau_R, \mathsf{Cons}\, x\, y \Rightarrow r \mid \mathsf{Nil} \Rightarrow r \rangle$ $\mid \langle R, \tau_R, R^* \rangle$	*relation def.*
θ	::=	$T \mid T * \theta$	*tuple sort*
τ_R	::=	$\mathsf{intlist} :\to \{\theta\} \mid \mathsf{int} :\to \{\theta\}$	*relation sort*

Figure 1: Language

3. Core language

3.1 Syntax

We formalize our ideas using a core calculus (λ_R) shown in Fig. 1, an A-normalized extension of the simply-typed lambda calculus. The language supports a primitive type (int), a recursive data type (intlist), along with dependent base and function types. Because the mechanisms and syntax to define and elaborate recursive data types are kept separate from the core, λ_R is only provided with two constructors, Nil and Cons used to build lists. The language has a standard call-by-value operational semantics, details of which can be found in the appendix[4].

Dependent type refinements (ϕ) in λ_R are assertions over relational expressions (r); these expressions, which are themselves typed, constitute the syntactic class of expressions in our specification language. We refer to the types of relational expressions as *sorts*, in order to distinguish them from λ_R types. We write $r :: s$

[4] Proofs for all lemmas and theorems in the paper are also provided in the appendix

Sort Checking Specification Language $\boxed{\Gamma \vdash r :: \{\theta\}, \quad \Gamma \vdash R :: T :\to \{\theta\}}$

S-REL

$$\frac{R \triangleq \langle \mathsf{Nil} \Rightarrow r_1,\ \mathsf{Cons}\, x\, y \Rightarrow r_2 \rangle \quad \cdot \vdash r1 :: \{\theta\} \quad \cdot, x : \mathsf{int}, y : \mathsf{intlist} \vdash r2 :: \{\theta\}}{\cdot \vdash R :: \mathsf{intlist} :\to \{\theta\}}$$

S-REL-STAR

$$\frac{R_1 \triangleq R_2^* \quad \cdot \vdash R_2 :: \tau_R}{\cdot \vdash R_1 :: \tau_R}$$

S-APP

$$\frac{\|\Gamma\| \Vdash v : T \quad \cdot \vdash R :: T :\to \{\theta\}}{\Gamma \vdash R(v) :: \{\theta\}}$$

S-REL-ID

$$\frac{}{\cdot \vdash R_{id} :: \mathsf{int} :\to \{\mathsf{int}\}}$$

S-UNION

$$\frac{\Gamma \vdash r_1 :: \{\theta\} \quad \Gamma \vdash r_2 :: \{\theta\}}{\Gamma \vdash r_1 \cup r_2 :: \{\theta\}}$$

S-CROSS

$$\frac{\Gamma \vdash r_1 :: \{\theta_1\} \quad \Gamma \vdash r_2 :: \{\theta_2\}}{\Gamma \vdash r_1 \times r_2 :: \{\theta_1 * \theta_2\}}$$

Well-Formedness $\boxed{\Gamma \vdash \phi, \quad \Gamma \vdash \tau}$

WF-RPRED

$$\frac{\odot \in \{=, \subset\} \quad \Gamma \vdash r_1 :: \{\theta\} \quad \Gamma \vdash r_2 :: \{\theta\}}{\Gamma \vdash r_1 \odot r_2}$$

WF-REF

$$\frac{\odot \in \{\wedge, \vee\} \quad \Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \odot \phi_2}$$

WF-BASE

$$\frac{\Gamma, \nu : T \vdash \phi}{\Gamma \vdash \{\nu : T \mid \phi\}}$$

WF-FUN

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma, x : \tau_1 \vdash \tau_2}{\Gamma \vdash x : \tau_1 \to \tau_2}$$

Subtyping $\boxed{\Gamma \vdash \tau_1 <: \tau_2}$

SUBT-BASE

$$\frac{\Gamma \vdash \{\nu : T \mid \phi_1\} \quad \Gamma \vdash \{\nu : T \mid \phi_2\} \quad [\![\Gamma_R]\!] \models [\![\Gamma, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!]}{\Gamma \vdash \{\nu : T \mid \phi_1\} <: \{\nu : T \mid \phi_2\}}$$

SUBT-ARROW

$$\frac{\Gamma \vdash \tau_{21} <: \tau_{11} \quad \Gamma, x : \tau_{21} \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash (x : \tau_{11}) \to \tau_{12} <: (x : \tau_{21}) \to \tau_{22}}$$

Type Checking Expression Language $\boxed{\Gamma \vdash e : \tau}$

T-VAR

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

T-ABS

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).\, e : (x : \tau_1) \to \tau_2}$$

T-MATCH

$$\frac{\begin{array}{c}\Gamma \vdash v : \mathsf{intlist} \quad \Gamma \vdash \mathsf{Nil} : \{\nu : \mathsf{intlist} \mid \phi_n\} \\ \Gamma \vdash \mathsf{Cons} : x : \mathsf{int} \to y : \mathsf{intlist} \to \{\nu : \mathsf{intlist} \mid \phi_c\} \\ \Gamma_c = x : \mathsf{int}, y : \mathsf{intlist}, [v/\nu]\,\phi_c \quad \Gamma_n = [v/\nu]\,\phi_n \\ \Gamma \vdash \tau \quad \Gamma, \Gamma_c \vdash e1 : \tau \quad \Gamma, \Gamma_n \vdash e2 : \tau\end{array}}{\Gamma \vdash \mathtt{match}\ v\ \mathtt{with}\ \mathsf{Cons}\, x\, y \ \Rightarrow\ e_1\ \mathtt{else}\ e_2 : \tau}$$

T-CONST

$$\frac{\cdot \vdash ty(c)}{\Gamma \vdash c : ty(c)}$$

T-SUB

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

T-APP

$$\frac{\Gamma \vdash e : (x : \tau_1) \to \tau_2 \quad \Gamma \vdash v : \tau_1}{\Gamma \vdash e\, v : [v/x]\tau_2}$$

T-LET

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

Figure 3: Static semantics of λ_R

to denote that a relational expression r has sort s. A structural relation is a triple, consisting of a unique relation name, its sort, and its definition as (a) a pattern-match sequence that relates constructors of an algebraic data type to a relation expression, or (b) an inductive extension of an existing relation, captured using the closure operator (*). We write $R \triangleq \delta$ to denote that a relation R has a (pattern-match or inductive) definition δ.

A structural relation maps a value to a set of tuples (θ). We use ":$\to$" to distinguish such maps from the mapping expressed by dependent function types. For example, the notation:

$$R_{ob} :: \mathsf{intlist} :\to \{\mathsf{int} * \mathsf{int}\}$$

indicates that the sort of relation R_{ob} is a map from integer lists to pairs. As reflected by the syntactic class of relation sorts (τ_R), the domain of a λ_R relation is either intlist or int. For the purposes of the formalization, we assume the existence of a single primitive relation R_{id} whose sort is $\mathtt{int} :\to \{\mathtt{int}\}$ that defines an identity relation on integers.

3.2 Sorts, Types and Well-formedness

Fig. 3 defines rules to check sorts of structural relations and relational expressions, establish well-formedness conditions of type refinements, and type-check expressions. The judgments defined by these rules make use of environment Γ, defined as follows:

$$\Gamma \ ::= \ \cdot \ \mid \ \Gamma, x : \tau \ \mid \ \Gamma, \phi$$

Environments are ordered sets of assertions that make up a typing context. Assertions are either (a) type bindings for variables, or (b) type refinements that reflect branch conditions collected from `match` expressions. We assume that any variable is bound only once in Γ.

Structural relations are sort checked under an empty type environment. The rule S-REL type checks a relation definition by ensuring that relational expressions associated with the constructors that comprise the definition all have the same sort. The rule S-REL-STAR captures the fact that an inductive extension of a relation has the same type as the relation itself. The rule S-APP sort checks relation applications by ensuring that the argument to the relation

MSFOL

$$
\begin{array}{llll}
x & \in & \lambda_R \textit{ variable} & \quad i,k,j \in \textit{bound variable} \\
R & \in & \textit{uninterpreted relation} & \quad A \in \textit{uninterpreted sort}
\end{array}
$$

$$
\begin{array}{llll}
\phi^F & ::= & v \mid v = v \mid \phi^F\ \phi^F \mid \phi^F \Leftrightarrow \phi^F & \textit{quantifier-free} \\
 & \mid & \phi^F \Rightarrow \phi^F \mid \phi^F \vee \phi^F \mid \phi^F \wedge \phi^F & \textit{proposition} \\
 & \mid & v : \tau^F & \\
\phi^L & ::= & \forall (k : T^F).\, \phi^L \mid \phi^F \mid \phi^L \wedge \phi^L & \textit{quantified} \\
 & \mid & \phi^L \vee \phi^L & \textit{proposition} \\
v & ::= & x \mid k \mid j \mid R & \textit{variable} \\
T^F & ::= & A \mid \textit{bool} & \textit{sort}
\end{array}
$$

$$\tau^F \quad ::= \quad \textit{bool} \mid T^F \to \tau^F \quad \textit{sort of } \phi^F$$

Auxiliary Definitions

$$
\begin{array}{lcl}
\mathcal{F} & : & T \to A \\
Inst & : & \phi^L \times v \to \phi^L \\
Inst(\forall (k : T^F).\phi^L,\, y) & = & [y/k]\phi^L \\
\eta_{wrap} & : & \phi^F \times \tau^F \to \phi^L \\
\eta_{wrap}(\phi^F, T^F \to \tau^F) & = & \forall (k : T^F).\eta_{wrap}(\phi^F\ k, \tau^F) \\
\eta_{wrap}(\phi^F, bool) & = & \phi^F
\end{array}
$$

Semantics of Relational Expressions $\boxed{[\![r]\!]}$

$$
\begin{array}{lcl}
[\![R(\mathsf{Cons}\, v_1\, v_2)]\!] & = & [\![\Sigma_R(R)(\mathsf{Cons}\, v_1\, v_2)]\!] \\
[\![R(\mathsf{Nil})]\!] & = & [\![\Sigma_R(R)(\mathsf{Nil})]\!] \\
[\![T]\!] & = & \mathcal{F}(T) \\
[\![\{T\}]\!] & = & [\![T]\!] \to \textit{bool} \\
[\![\{T * \theta\}]\!] & = & [\![T]\!] \to [\![\{\theta\}]\!] \\
[\![T :\to \{\theta\}]\!] & = & [\![T]\!] \to [\![\{\theta\}]\!] \\
[\![R_{id}]\!] & = & \forall (j : [\![\,\texttt{int}\,]\!]). \\
 & & \quad \forall (k : [\![\,\texttt{int}\,]\!]).j = k
\end{array}
\qquad
\begin{array}{lcl}
[\![R]\!] & = & \eta_{wrap}(R, [\![\Gamma_R(R)]\!]) \\
[\![R(x)]\!] & = & Inst([\![R]\!],\, x) \\
[\![r_1 \cup r_2]\!] & = & \gamma_{\sqcup}([\![r_1]\!], \vee, [\![r_2]\!]) \\
[\![r_1 \times r_2]\!] & = & \gamma_{\bowtie}([\![r_1]\!], \wedge, [\![r_2]\!]) \\
\gamma_{\sqcup}(\forall (k : T^F).\, e_1, \odot, \forall (k : T^F).\, e_2) & = & \forall (k : T^F).\, \gamma_{\sqcup}(e_1, \odot, e_2) \\
\gamma_{\sqcup}(\phi_1^F, \odot, \phi_2^F) & = & \phi_1^F \odot \phi_2^F \\
\gamma_{\bowtie}(\forall j : T_j^F.\, \phi_1^F, \odot, \forall \overline{k : T_k^F}.\, \phi_1^F) & = & \forall (j : T_j^F).\, \forall (\overline{k : T_k^F}).\, \phi_1^F \odot \phi_2^F
\end{array}
$$

Semantics of Type Refinements $\boxed{[\![\phi]\!]}$

$$
\begin{array}{lcl}
[\![\phi_1 \wedge \phi_2]\!] & = & [\![\phi_1]\!] \wedge [\![\phi_2]\!] \\
[\![\phi_1 \vee \phi_2]\!] & = & [\![\phi_1]\!] \vee [\![\phi_2]\!]
\end{array}
\qquad
\begin{array}{lcl}
[\![r_1 = r_2]\!] & = & \gamma_{\sqcup}([\![r_1]\!], \Leftrightarrow, [\![r_2]\!]) \\
[\![r_1 \subseteq r_2]\!] & = & \gamma_{\sqcup}([\![r_1]\!], \Rightarrow, [\![r_2]\!])
\end{array}
$$

Figure 4: Semantics of Specification Language

has the required simple (non-dependent) type. The rule makes use of a simple typing judgment ($\Vdash$) under a *refinement erased* Γ (denoted $\|\Gamma\|$) for this purpose. Rules for simple typing judgments are straightforward, and are elided here; the full set of rules can be found in the appendix.

Refinement erasure on a dependent base type (τ) sets its type refinement to *true*, effectively erasing the refinement to yield a simple type. For function types, erasure is defined recursively:

$$\|\{\nu : T \mid \phi\}\| \quad = \quad T \qquad \|(x : \tau_1) \to \tau_2\| \quad = \quad \|\tau_1\| \to \|\tau_2\|$$

Refinement erasure for type environments performs erasure over all type bindings within the environment, in addition to erasing all recorded branch conditions. For an empty environment, refinement erasure is an identity.

$$
\begin{array}{lcl}
\|\Gamma,\, x : \tau\| & = & \|\Gamma\|,\, x : \|\tau\| \qquad \|\Gamma,\, \phi\| \quad = \quad \|\Gamma\| \\
\|\cdot\| & = & \cdot
\end{array}
$$

The dependent type checking rules for λ_R expressions are mostly standard, except for T-CONST and T-MATCH. The rule T-CONST makes use of a function ty that maps a constant c to a type ($ty(c)$), which remains its type under any Γ. The function ty is defined below:

$$
\begin{array}{lcl}
\forall i \in \mathbb{Z},\, ty(i) & = & \mathsf{int} \\
ty(\mathsf{Nil}) & = & \{\nu : \mathsf{intlist} \mid \phi_n\} \\
ty(\mathsf{Cons}) & = & x : \mathsf{int} \to y : \mathsf{intlist} \to \{\nu : \mathsf{intlist} \mid \phi_c\}
\end{array}
$$

The type refinements of Nil (ϕ_n) and Cons (ϕ_c) in the T-MATCH rule are conjunctive aggregations of Nil and Cons cases (resp.) of all structural relation definitions. To help us precisely define ϕ_n and ϕ_c, we assume the presence of (a) a globally-defined finite map (Σ_R) that maps relation names to their pattern-match definitions, and (b) a finite ordered map Γ_R that maps relation names to their sorts. We implicitly parameterize our typing judgment over Σ_R (i.e., our $\vdash$ is actually $\vdash_{\langle \Sigma_R, \Gamma_R \rangle}$). Inductive relations defined using the closure operator are assumed to be unfolded to pattern-match definitions before being bound in Σ_R:

$$\frac{R \triangleq R_2^* \quad \Sigma_R(R_2) = \langle \mathsf{Nil} \Rightarrow r_1,\, \mathsf{Cons}\, x\, y \Rightarrow r_2 \rangle}{\Sigma_R(R) = \langle \mathsf{Nil} \Rightarrow r_1,\, \mathsf{Cons}\, x\, y \Rightarrow r_2 \cup R(y) \rangle}$$

For the sake of presentation, we treat the pattern-match definition of a structural relation as a map from constructor patterns to relational expressions. Consequently, when $\Sigma_R(R) = \langle \mathsf{Nil} \Rightarrow r_1,\, \mathsf{Cons}\, x\, y \Rightarrow r_2 \cup R(y) \rangle$, the notation $\Sigma_R(R)(\mathsf{Nil})$ denotes r_1, and $\Sigma_R(R)(\mathsf{Cons}\, x\, y)$ denotes r_2. With help of Σ_R, we now define ϕ_n, and ϕ_c as:

$$
\begin{array}{lcl}
\phi_n & = & \bigwedge_{R \in dom(\Sigma_R)} R(\nu) = \Sigma_R(R)(\mathsf{Nil}) \\
\phi_c & = & \bigwedge_{R \in dom(\Sigma_R)} R(\nu) = \Sigma_R(R)(\mathsf{Cons}\, x\, y)
\end{array}
$$

For instance, consider a case where Σ_R has only one element (R) in its domain:

$$\Sigma_R = [R \mapsto \langle \mathsf{Nil} \Rightarrow R_{id}(0) \mid \mathsf{Cons}\, x\, y \Rightarrow R_{id}(x) \rangle]$$

The type of Nil and Cons in such case is as following:

$$
\begin{array}{lcl}
ty(\mathsf{Nil}) & = & \{\nu : \mathsf{intlist} \mid R(\nu) = R_{id}(0)\} \\
ty(\mathsf{Cons}) & = & x : \mathsf{int} \to y : \mathsf{intlist} \to \{\nu : \mathsf{intlist} \mid R(\nu) = R_{id}(x)\}
\end{array}
$$

The T-MATCH rule type checks each branch of the `match` expression under an environment that records the corresponding branch condition. Additionally, the type environment for the Cons branch is also extended with the types of matched pattern variables (x and y). The branch condition for the Cons (alternatively, Nil) case is obtained by substituting the test value (v) for the bound variable (ν) in the type refinement of Cons (Nil). Intuitively, the branch condition of Cons (alternatively, Nil) captures the fact that the value v was obtained by applying the constructor Cons (Nil); therefore, it should satisfy the invariant of Cons (Nil). For instance, consider the `match` expression:

$$\texttt{match}\ z\ \texttt{with}\ \mathsf{Cons}\ x\ y \Rightarrow e_1\ \texttt{else}\ e_2$$

where Cons has type[5]

$$\mathsf{Cons} : x{:}\mathsf{int} \to y{:}\mathsf{intlist} \to \{\nu{:}\mathsf{intlist} \mid R_{mem}(\nu) = R_{id}(x) \cup R_{mem}(y)\}$$

[5] In our examples, we assign the same names to formal and actual arguments for convenience.

Expression e_1 is type-checked under the extended environment:

$$\Gamma,\, x{:}\{\nu{:}\mathsf{int} \mid true\},\, xs{:}\{\nu{:}\mathsf{intlist} \mid true\},\\ R_{mem}(z) = R_{id}(x) \cup R_{mem}(y)$$

The subtyping rules allow us to propagate dependent type information, and relate the subtype judgment to a notion of semantic entailment ($\models$) in logic. The cornerstone of subtyping is the subtyping judgment between base dependent types defined by the rule SUBT-BASE. The rule refers to the map Γ_R that provides sorts for relations occurring free in type refinements. Intuitively, the rule asserts dependent type τ_1 to be a subtype of τ_2, if:

- Their base types match, and,
- Given a logical system L, and interpretations of type environment ($\Gamma,\ \nu\ :\ T$) and the type refinement ϕ_1 (of τ_1) in L, the following implication holds in L:

  $$[\![\Gamma, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!]$$

  The context under which the implication has to be valid ($[\![\Gamma_R]\!]$, is the interpretation of sort bindings of relations in L.

The soundness of λ_R's type system is defined with respect to a reduction relation ($\longrightarrow$) that specifies the langauge's operational semantics:

THEOREM 3.1. (**Type Safety**) *if* $\cdot \vdash e : \tau$, *then either* e *is a value, or there exists an* e' *such that* $e \longrightarrow e'$ *and* $\cdot \vdash e' : \tau$.

3.3 Semantics of the Specification Language

The semantics of our specification language is defined via a translation from well-typed relational expressions and well-formed type refinements to propositions of many-sorted first-order logic (MSFOL).

Many-sorted first-order logic extends first-order logic (FOL) with sorts (types) for variables. For our purpose, we only consider the extension with booleans and uninterpreted sorts, i.e., sorts that, unlike `int`, do not have an attached interpretation. Ground terms, or quantifier-free formulas, of MSFOL are drawn from propositional logic with equality and n-ary uninterpreted functions.

Our MSFOL semantics make use of the Σ_R map defined previously. For perspicuity, we introduce the following syntactic sugar:

$$\Sigma_R(R)(\mathsf{Cons}\, v_1\, v_2) = [v_2/y]\,[v_1/x]\,\Sigma_R(R)(\mathsf{Cons}\, x\, y)$$

Further, we also assume a finite ordered map Γ_R that maps structural relations to their sorts. That is, for all R such that $\cdot \vdash R :: \tau_R$, we have that $\Gamma_R(R) = \tau_R$.

Figure 4 describes the MSFOL semantics of λ_R's specification language. The semantics is operational in the sense that it describes an algorithm to compile assertions in λ_R type refinements to formulas in MSFOL. Our semantics are parameterized over an auxiliary function ($\mathcal{F}$) that maps λ_R datatypes to uninterpreted sorts in MSFOL. The specific uninterpreted sorts types map to are not relevant here. However, $\mathcal{F}$ has to be a total function over λ_R datatypes. Note that despite treating interpreted types (eg: intlist and int) as uninterpreted sorts in the underlying logic, the exercise of ascribing a semantics to the type refinement language is complete. This is because the interpretation of any type is the collection of operations allowed on that type, and our type refinement language does not contain operations that are specific to values of any specific type.

Relations translate to uninterpreted functions with a Boolean co-domain in MSFOL. We choose to curry sorts of uninterpreted functions representing relations (R) to simplify the semantics. The auxiliary function η_{wrap} wraps an uninterpreted function under a quantified formula; this can be construed as an eta-equivalent abstraction of an uninterpreted function in quantified logic. As an example, suppose we have

$$R :: \mathsf{intlist} :\rightarrow \{\mathsf{int}\}$$

That is, Γ_R maps R to $\mathsf{intlist} :\rightarrow \{\mathsf{int}\}$. Assume that: $[\![\mathsf{int}]\!] = A_0$ and $[\![\mathsf{intlist}]\!] = A_1$. Now,

$$\begin{aligned}
[\![R]\!] \;=\;& \eta_{wrap}(R, \Gamma_R(R))\\
& \eta_{wrap}(R, [\![\mathsf{intlist} :\rightarrow \{\mathsf{int}\}]\!])\\
& \eta_{wrap}(R, [\![\mathsf{intlist}]\!] \rightarrow [\![\{\mathsf{int}\}]\!])\\
& \eta_{wrap}(R, A_1 \rightarrow A_0 \rightarrow \mathit{bool})\\
& \forall(k : A_1).\eta_{wrap}(R\,k, A_0 \rightarrow \mathit{bool})\\
& \forall(k : A_1).\forall(j : A_0).\eta_{wrap}(R\,k\,j, \mathit{bool})\\
& \forall(k : A_1).\forall(j : A_0).R\,k\,j
\end{aligned}$$

Auxiliary function *Inst* instantiates a prenex-quantified formula. We employ the standard interpretation of set union and cross product operations, when sets are represented using prenex-quantified propositions:

$$\forall\overline{x}.\phi_1 \;\cup\; \forall\overline{x}.\phi_2 \;= \forall\overline{x}.(\phi_1 \vee \phi_2)$$
$$\forall\overline{x}.\phi_1 \;\times\; \forall\overline{y}.\phi_2 \;= \forall\overline{x}.\forall\overline{y}.(\phi_1 \wedge \phi_2)$$

Our semantics use syntactic rewrite functions - $\gamma_{\sqcup}$ and $\gamma_{\bowtie}$, to perform this translation, and to move quantification to prenex position when composing quantified formulas using logical connectives.

To demonstrate the compilation process, we consider the following λ_R assertion involving the occurs-before relation:

$$(R_{ob} :: \texttt{intlist} :\rightarrow \{\texttt{int*int}\}),$$

and membership relation

$$(R_{mem} :: \texttt{intlist} :\rightarrow \{\mathsf{int}\})$$

for integer lists:

$$R_{ob}\texttt{(l)} = R_{id}\texttt{(x)} \times R_{mem}\texttt{(xs)}$$

The series of steps that compile the assertion to an MSFOL formula, which captures the semantics of the assertion, are shown in Fig. 5.[6] The example assumes that $\mathcal{F}$ maps `int` to sort $\mathtt{A_0}$, and `intlist` to sort $\mathtt{A_1}$.

The semantics of types and type refinements given Fig. 4 can be lifted in a straightforward way to the level of type environments (Γ):

$$\begin{array}{lcl}
[\![\Gamma,\, x : \{\nu : T \mid \phi\}]\!] & = & [\![\Gamma]\!] \Rightarrow x : [\![T]\!] \Rightarrow [\![x/\nu]\phi]\!]\\
[\![\Gamma,\, \phi]\!] & = & [\![\Gamma]\!] \Rightarrow [\![\phi]\!]\\
[\![\cdot]\!] & = & \mathit{true}
\end{array}$$

The interpretation of relation sort environment (Γ_R) is a set of assertions over MSFOL sorts of uninterpreted relations:

$$\begin{array}{lcl}
[\![\Gamma_R,\, R :: \tau_R]\!] & = & [\![\Gamma_R]\!] \cup \{R : [\![\tau_R]\!]\}\\
[\![\cdot]\!] & = & \{\}
\end{array}$$

The following lemma states that the translation to MSFOL is complete for a well-formed type refinement:

LEMMA 3.2. (Completeness of semantics) *Forall* ϕ, Γ, *if* $\Gamma \vdash \phi$, *then there exists an MSFOL proposition* ϕ^L *such that* $[\![\phi]\!] = \phi^L$.

3.4 Decidability of λ_R Type Checking

The subtyping judgment in our core language (λ_R) relies on the semantic entailment judgment of MSFOL. The premise of SUBT-BASE contains the following:

$$[\![\Gamma_R]\!] \models [\![\Gamma, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!]$$

[6] We focus only on the underlined part of the assertion as compilation stack increases. We switch back to showing complete assertion when all sub-parts are reduced. The digit before the dot in a step number indicates this switch.

$$[\![\mathtt{R}_{ob}(\mathtt{l}) = \mathtt{R}_{id}(\mathtt{x}) \times \mathtt{R}_{mem}(\mathtt{xs})]\!] \quad (1.1)$$

$$\gamma_{\sqcup}([\![\underline{\mathtt{R}_{ob}(\mathtt{l})}]\!], \Leftrightarrow, [\![\mathtt{R}_{id}(\mathtt{x}) \times \mathtt{R}_{mem}(\mathtt{xs})]\!]) \quad (1.2)$$

$$Inst [\![\mathtt{R}_{ob}]\!]\ l \quad (2.1)$$

$$Inst(\ \forall(\mathtt{i} : [\![\mathtt{intlist}]\!]).\forall(\mathtt{j} : [\![\mathtt{int}]\!])\ . \quad (2.2)$$
$$\forall(\mathtt{k} : [\![\mathtt{int}]\!]).(\mathtt{Rob\ i\ j\ k})\ )\ \mathtt{x}$$

$$Inst\ (\forall(\mathtt{i} : \mathtt{A}_1).\forall(\mathtt{j} : \mathtt{A}_0).\forall(\mathtt{k} : \mathtt{A}_0).(\mathtt{Rob\ i\ j\ k}))\ \mathtt{x} \quad (2.3)$$

$$(\forall(\mathtt{j} : \mathtt{A}_0).\forall(\mathtt{k} : \mathtt{A}_0).(\mathtt{Rob\ x\ j\ k})) \quad (2.4)$$

$$\gamma_{\sqcup}([\![\mathtt{R}_{ob}(\mathtt{l})]\!], \Leftrightarrow, [\![\underline{\mathtt{R}_{id}(\mathtt{x}) \times \mathtt{R}_{mem}(\mathtt{xs})}]\!]) \quad (1.2)$$

$$\gamma_{\bowtie}([\![\underline{\mathtt{R}_{id}(\mathtt{x})}]\!], \wedge, [\![\mathtt{R}_{mem}(\mathtt{xs})]\!]) \quad (3.1)$$

$$Inst\ (\forall(\mathtt{i} : [\![\mathtt{int}]\!].\forall(\mathtt{j} : [\![\mathtt{int}]\!]).(\mathtt{i} = \mathtt{j}))\ \mathtt{x} \quad (4.1)$$

$$\forall(\mathtt{j} : \mathtt{A}_0).(\mathtt{x} = \mathtt{j}) \quad (4.2)$$

$$\gamma_{\bowtie}([\![\mathtt{R}_{id}(\mathtt{x})]\!], \wedge, [\![\underline{\mathtt{R}_{mem}(\mathtt{xs})}]\!]) \quad (3.1)$$

$$(\forall(\mathtt{k} : \mathtt{A}_0)(\mathtt{Rmem\ xs\ k})) \quad (5.1)$$

$$\gamma_{\bowtie}(\ \forall(\mathtt{j} : \mathtt{A}_0).(\mathtt{x} = \mathtt{j})\ ,\ \wedge,\ (\forall(\mathtt{k} : \mathtt{A}_0)(\mathtt{Rmem\ xs\ k}))\ ) \quad (3.2)$$

$$\forall(\mathtt{j} : \mathtt{A}_0).\forall(\mathtt{k} : \mathtt{A}_0).(\mathtt{x} = \mathtt{j}) \wedge (\mathtt{Rmem\ xs\ k}) \quad (3.3)$$

$$\gamma_{\sqcup}(\ (\forall(\mathtt{j} : \mathtt{A}_0).\forall(\mathtt{k} : \mathtt{A}_0).(\mathtt{Rob\ x\ j\ k}))\ ,\ \Leftrightarrow, \quad (1.3)$$
$$\forall(\mathtt{j} : \mathtt{A}_0).\forall(\mathtt{k} : \mathtt{A}_0).(\mathtt{x} = \mathtt{j}) \wedge (\mathtt{Rmem\ xs\ k})\ )$$

$$\forall(\mathtt{j} : \mathtt{A}_0).\forall(\mathtt{k} : \mathtt{A}_0).(\mathtt{Rob\ l\ j\ k}) \Leftrightarrow (\mathtt{x} = \mathtt{j}) \wedge (\mathtt{Rmem\ xs\ k}) \quad (1.4)$$

Figure 5: Compiling a λ_R assertion to MSFOL

Consequently, decidability of type checking in λ_R reduces to decidability of semantic entailment in MSFOL. Although semantic entailment is undecidable for full first-order logic, our subset of MSFOL is a carefully chosen decidable fragment. This fragment, known as Effectively Propositional (EPR) first-order logic, or Bernay-Schönfinkel-Ramsey (BSR) logic, consists of prenex quantified propositions with uninterpreted relations and equality. Off-the-shelf SMT solvers (e.g., Z3) are equipped with efficient decision procedures for EPR logic [18], making type checking in λ_R a practical exercise.

THEOREM 3.3. (**Decidability**) *Type checking in λ_R is decidable.*

Proof Follows from Lemma 3.2 and decidability proof of EPR logic. ∎

4. Parametricity

4.1 Syntax

We now extend our core language (λ_R) with parametric polymorphism, and the specification language with parametric relations - relations parameterized over other relations . We refer to the extended calculus as $\lambda_{\forall R}$. Figure 6 shows the type and specification language of $\lambda_{\forall R}$. We have elided $\lambda_{\forall R}$'s expression language in the interest of space. Unmodified syntactic forms of λ_R are also elided.

The only algebraic data type in $\lambda_{\forall R}$ is a polymorphic list, which is the domain for structural relations. Consequently, structural relations have sort schemes (σ_R), akin to type schemes (σ) of the term language. For example, the non-parametric head relation (R_{hd}) from Section 2, when defined over a polymorphic `'a list` will have sort scheme, $\forall$ `'a. 'a list` $:\!\!\rightarrow$ `'a`. The specification language also contains an expression ($\mathcal{R}\,T$) to instantiate a generalized type variable in parametric relation sorts.

A parametric relation generalizes a structural relation, just as a polymorphic list generalizes a monomorphic one. Our syntax and semantics for parametric relations are based on this correspondence. Since the `list` type constructor takes only one type argument, structural relations in $\lambda_{\forall R}$ are parameterized over one relational parameter. The domain of a relational parameter to a structural relation over a `'a list` should be `'a`. When `'a` in `'a list` is instantiated with, e.g., `'b list`, the parameter of a parametric relation over `'a list` can be instantiated with a structural relation over `'b list`. For instance, the relational parameter R in the parametric membership relation (R_{mem} R), defined in Section 2, can be instantiated with the non-parametric head relation, R_{hd}[7], after instantiating `'a` in its sort scheme with a `'b list`. The resulting relation can now be applied to a list of lists (i.e., a `'b list list`) to denote the set of head elements in the constituent lists.

Calculus $\lambda_{\forall R}$

t	$\in$	*tuple − sort variables*	x, y, k	$\in$	*variables*
`'a`, `'b`	$\in$	*type variables*			

T	$::=$	`'a` $\mid$ `'a list` $\mid$ int	*datatypes*
τ	$::=$	$\{\nu : T \mid \Phi\}$ $\mid$ $(x : \tau) \rightarrow \tau$	*dependent type*
δ	$::=$	$\forall t.\forall(R :: $ `'a` $:\!\!\rightarrow t).\ \delta \mid \tau$	*parametric dep. type*
σ	$::=$	$\forall$ `'a`$.\ \sigma \mid \delta$	*type scheme*

Specification Language

Φ	$::=$	$\rho = \rho \mid \rho \subseteq \rho \mid \Phi \wedge \Phi \mid true$	*type refinement*
ρ	$::=$	$\mathcal{R}(x) \mid \rho \cup \rho \mid \rho \times \rho$	*rel. expression*
$\mathcal{R}$	$::=$	$\mathcal{R}\,T \mid \mathcal{R}\,\theta\,\mathcal{R} \mid R$	*instantiation*
θ	$::=$	$t \mid t * \theta \mid T * \theta \mid T$	*tuple sort*
τ_R	$::=$	$\forall t.$ (`'a` $:\!\!\rightarrow t) :\!\!\rightarrow$ (`'a list` $:\!\!\rightarrow \theta$)	*relation sort*
	$\mid$	`'a list` $:\!\!\rightarrow \theta$	
σ_R	$::=$	$\forall$ `'a`$.\ \tau_R \mid \tau_R$	*sort scheme*
Σ_R	$::=$	$\langle R,\ R_p,\ \sigma_R,\ \mathsf{Cons}\,x\,y \Rightarrow r \mid \mathsf{Nil} \Rightarrow r \rangle$	*rel. definition*
	$\mid$	$\langle R,\ R_p,\ \sigma_R,\ \mathcal{R}^* \rangle$	

Figure 6: $\lambda_{\forall R}$ - Language with parametric relations

The definition (Σ_R) of a parametric relation is a tuple containing its name (R), the name of its relational parameter (R_p), its sort scheme (σ_R), and its definition. A parametric relation definition very often does not place constraints over the co-domain of its relational parameter. For instance, consider the parametric R_{hd} relation over `'a list` reproduced from Section 2:

```
relation (R_hd R)  (x::xs)  =  R(x)
       | (R_hd R)  []       =  ∅
```

R_{hd} requires that the domain of its parameter be `'a`, but it places no restriction on the co-domain of R. In order to have a truly parametric definition of R_{hd}, it is essential that we let the relational parameter have an unrestricted co-domain. Therefore, we let tuple-sort variables (t) be used in tuple sorts (θ). Such a variable can be instantiated with a tuple sort, such as int*int.

In order to use a parametric relation in a type refinement, its relational parameter has to be instantiated. Polymorphism in $\lambda_{\forall R}$ is predicative so, parameterization over relations in $\lambda_{\forall R}$ is also predicative. An *instantiated* parametric relation is equivalent to a non-parametric relation; it can be *applied* to a variable of the term language, and can also be used to instantiate other parametric relations.

To extend the generality of parametric relations to dependent types of the term language, we lift the parameterization over relations from the level of type refinements to the level of types. We refer to dependent types parameterized over relations as *parametric dependent types* (δ). An example of a parametric dependent type is the type of `foldl` from Section 2. Another example is the type of map shown below:

[7] A note on notation: We use (R_{mem} R) and (R_{hd} R) to denote parametric membership and head relations, resp. We continue to use R_{mem} and R_{hd} to denote their non-parametric versions. We use qualifiers "parametric" and "non-parametric" to disambiguate.

Semantics of Bind Equations $\boxed{[\![\psi]\!]}$

$$
\begin{array}{rcl}
[\![R_2 = \lambda(x:T_1).\ \texttt{bind}\,(R_1(x), \lambda(\overline{k:T_2}).\, r)]\!] & = & \forall(x:[\![T_1]\!]).\ \gamma_{\Rightarrow}([\![R_1(x)]\!],\ \forall(\overline{(k:[\![T_2]\!])}.[\![r]\!],\ [\![R_2(x)]\!]) \\
& \wedge & \forall(x:[\![T_1]\!]).\ \gamma_{\Leftarrow}([\![R_1(x)]\!],\ \forall(\overline{k:[\![T_2]\!]}).[\![r]\!],\ [\![R_2(x)]\!]) \\
\gamma_{\Rightarrow}(\forall(\overline{k:T_1^F}).\phi_1^F,\ \forall(\overline{k:T_1^F}).\forall(\overline{j:T_2^F}).\phi_2^F,\ \nu^F) & = & \forall(\overline{k:T_1^F}).\forall(\overline{j:T_2^F}).\ \phi_1^F \wedge \phi_2^F \Rightarrow \nu^F\ \overline{j} \\
\gamma_{\Leftarrow}(\forall(\overline{k:T_1^F}).\phi_1^F,\ \forall(\overline{k:T_1^F}).\forall(\overline{j:T_2^F}).\phi_2^F,\ \nu^F) & = & \forall(\overline{j:T_2^F}).\exists(\overline{k:T_1^F}).\ \nu^F\ \overline{j} \Rightarrow \phi_1^F \wedge \phi_2^F
\end{array}
$$

Figure 7: Semantics of bind equations for parametric relations in $\lambda_{\forall R}$

$$
\begin{array}{lll}
r & ::= & R(x) \mid r \times r \\
F_R & ::= & \lambda(\overline{x:T}).\, r \qquad\qquad \textit{transformer} \\
e_b & ::= & \texttt{bind}\,(R(x), F_R) \qquad \textit{bind expression} \\
E_b & ::= & \lambda(x:T).\ \texttt{bind}\,(R(x), F_R) \qquad \textit{bind abstraction} \\
\psi & ::= & R = E_b \qquad \textit{bind equation} \\
\Sigma_R^b & ::= & \lambda R.\, E_b \qquad \textit{bind definition}
\end{array}
$$

Figure 8: Bind Syntax

```
('R1, 'R2) map :
      l → (f : x → {ν | 'R2(ν) = 'R1(x)})  →
      {ν | ((R_ob 'R2)* ν) = ((R_ob 'R1)* l)}
```

4.2 Sort and Type Checking

Rules to check sorts of relational expressions and well-formedness of type refinements (Φ) in $\lambda_{\forall R}$ are straightforward extensions of similar rules for λ_R and are omitted here. Sort-checking a parametric relation definition reduces to sort-checking a non-parametric relation definition under an environment extended with the sort of its relational parameter. Checking the sort of a relation instantiation is the same as checking the sort of a function application in other typed calculi, such as System F, as are rules to type-check generalization and instantiation expressions.

4.3 Semantics of Parametric Relations

Before we describe our semantics for parametric relations, we present a few auxiliary definitions:

Ground Relations. A ground relation of a parametric relation (R) is a non-parametric relation obtained by instantiating the relational parameter with the identity R_{id} relation in its definition. Since we require the co-domain of the relational parameter to be a tuple-sort variable (t), an instantiation of the parameter with R_{id} is always sort-safe. Therefore, there exists a ground relation for every parametric relation in $\lambda_{\forall R}$.

Transformer Expression. A transformer expression (F_R) is a λ_R relational expression under a binder that binds a tuple of variables. A transformer expression is expected to transform the tuple to a set of tuples through a cross-product combination of relation applications. The sort of a transformer application is a map (under ':→') from tuple-sort (θ_1) to a set sort ($\{\theta_2\}$). An example of a transformer expression of sort `'a` :→ { `'a*'a` } is the *reflexive transformer*:

$$\lambda x.\, R_{id}(x) \times R_{id}(x)$$

Bind Expressions. Consider an operator that accepts a relation application and a transformer expression (F_R), applies F_R over every tuple in the set representing a relation application, and subsequently folds the resulting set of sets using set union. Such an operator has following sort:

$$\forall t_1, t_2.\{t_1\} :\rightarrow (t_1 :\rightarrow \{t_2\}) :\rightarrow \{t_2\}$$

We name the operator `bind`, after set monadic bind. The syntax of bind expressions is given in Fig. 8.

By *binding* a relation application with a transformer expression, a bind expression effectively creates a new relation. For instance, given a list `l` with type `'a list`, the bind expression that binds $R_{mem}(l)$ with a reflexive transformer is as following:

$$\texttt{bind}(R_{mem}\texttt{(l)},\ \lambda\texttt{x}.R_{id}\texttt{(x)} \times\ R_{id}\texttt{(x)})$$

The result of evaluating this expression is the set of reflexive pairs of elements in the list, which is equivalent to instantiating R_{mem} with R_{dup}:

$$(R_{mem}\ R_{dup})\texttt{(l)} = \texttt{bind}(R_{mem}\texttt{(l)},\lambda\,\texttt{x}.R_{id}\texttt{(x)} \times\ R_{id}\texttt{(x)})$$

Here, equality is interpreted as equality of sets on both sides. Since the semantics of a relation application is the set of tuples, the above equation defines the semantics of $(R_{mem}\ R_{dup})$ in terms of its ground relation R_{mem}. Indeed, a parameteric R_{mem} relation (call it R^{π}_{mem}) can be defined equivalently in terms of its non-parameteric variant as:

$$R^{\pi}_{mem} \equiv \lambda\, R.\lambda\,\texttt{l}.\ \texttt{bind}(R_{mem}\texttt{(l)},\lambda\texttt{x}.R\texttt{(x)})$$

We refer to the above definition as the *bind definition* of R_{mem}. Every well-sorted parametric structural relation definition in $\lambda_{\forall R}$ can be transformed to a bind definition that is extensionally equal, i.e., both produce the same set of tuples for every instantiation, and subsequent application. Therefore, the pattern-match syntax used to define parametric relations is simply syntactic sugar over its underlying bind definition.

4.3.1 Elaboration to Bind Definition

Elaborating a parametric relation definition to a bind definition requires that we construct its ground relation, and a transformer expression (F_R). A ground relation definition is derived by instantiating its parametric definition with R_{id}, as stated previously. Constructing a transformer expression is equally simple - one only needs to examine the co-domain tuple sort of the parametric relation, which is also the co-domain tuple sort of the transformer expression (from the type of `bind`). A sort variable in the tuple sort is interpreted as application of its parameter relation, an asterisk in the sort translates to a cross-product, and a $\lambda_{\forall R}$ type in the tuple sort translates to application of R_{id}. For instance, consider a hypothetical parametric relation R_x with the following sort:

$$R_x\ \texttt{:: } \forall\texttt{t. (int :→ \{t\}) :→ (int list :→ \{int*t*t\})}$$

Its transformer expression is:

$$\lambda\texttt{(x,y,z)}.R_{id}\texttt{(x)} \times\ R\texttt{(y)} \times R\texttt{(z)}$$

Renaming the ground relation of R_x as $R_{x'}$, we derive the following bind definition of R_x:

$$\lambda R.\lambda\texttt{l.bind}(R_{x'}\texttt{(l)},\lambda\texttt{(x,y,z)}.R_{id}\texttt{(x)} \times\ R\texttt{(y)} \times R\texttt{(z)})$$

```
datatype color = R | B
datatype 'a tree = E | T of color * 'a tree
                          * 'a * 'a tree
fun balance (t:'a tree) : 'a tree = case t of
    T (B,T (R,T (R,a,x,b),y,c),z,d) =>
              T (R,T (B,a,x,b),y,T (B,c,z,d))
  | T (B,T (R,a,x,T (R,b,y,c)),z,d) =>
              T (R,T (B,a,x,b),y,T (B,c,z,d))
  | T (B,a,x,T (R,T (R,b,y,c),z,d)) =>
              T (R,T (B,a,x,b),y,T (B,c,z,d))
  | T (B,a,x,T (R,b,y,T (R,c,z,d))) =>
              T (R,T (B,a,x,b),y,T (B,c,z,d))
  | _ => t
```

(a) `balance`

```
(* Tree head (root) relation *)
relation R_thd (T(c,l,n,r)) = {(n)};
(* Tree membership relation *)
relation R_tmem = R*_thd;
(* Total-order relation among tree members *)
relation R_to (T (c,l,n,r)) = R_tmem(l) × {(n)}
                            ∪ {(n)} × R_tmem(r)
                            ∪ R_tmem(l) × R_tmem(r);
(*
 * "balance" preserves the total-order among members
 * of the tree
 *)
balance : t → {t' | R*_to(t') = R*_to(t)};
```

(b) Relational specification of `balance`

Figure 9: Red-Black Tree Example

4.3.2 Bind Equations

By substituting parametric relations with their bind definitions, every instantiation of a parametric relation can be reduced to a bind abstraction (E_b in Figure 8), which, like any non-parametric structural relation in $\lambda_{\forall R}$, is a map from a `'a list` to a set of tuples. Therefore, an instantiated parametric relation can be treated as a new non-parametric relation that is defined using `bind`. For example, $(R_{mem}\,R_{dup})$ can be treated as a new non-parametric relation R_1, defined in terms of `bind`:

$$R_1 = \lambda \mathtt{l}.\mathtt{bind}(R_{mem}(\mathtt{l}),\lambda \mathtt{x}.R_{id}(\mathtt{x}) \times R_{id}(\mathtt{x}))$$

By rigorously defining the semantics of *bind equations* as above, we can effectively capture the semantics of any instantiation of a parametric relation in terms of its ground relation. This is the insight that allows us to use parametric relations seamlessly in type refinements. For instance, the bind semantics for $(R_{mem}\ R_{dup})$ lets us prove the following implication, which could potentially arise during subtype checking:

$$((R_{mem}\ R_{dup})\ \mathtt{l}_1) = ((R_{mem}\ R_{dup})\ \mathtt{l}_2) \Rightarrow R_{mem}(\mathtt{l}_1) = R_{mem}(\mathtt{l}_2)$$

The formal semantics of bind equations, which also define an algorithm to compile bind equations to MSFOL formulas, is described in Fig. 7. Under our semantics, the bind equation for $(R_{mem}\ R_{dup})$ is interpreted as a conjunction of following first-order formulas (elaborated for clarity):

- If $\langle x \rangle \in R_{mem}(l)$, and $\langle y \rangle \in R_{id}(x) \times R_{id}(x)$, then $\langle y \rangle \in R_1(l)$.
- If $\langle y \rangle \in R_1(l)$, then there must exist x such that $\langle x \rangle \in R_{mem}(l)$ and $\langle y \rangle \in R_{id}(x) \times R_{id}(x)$.

Since sets have no other notion associated with them other than membership, the above first-order assertions *completely* describe $((R_{mem}\ R_{dup})\ l)$ in terms of $(R_{mem}\ l)$.

4.4 Decidability of Type Checking

Type refinements (Φ) in $\lambda_{\forall R}$ can be elaborated to a conjunction of bind equations representing semantics of instantiated relations, and a λ_R type refinement (ϕ). Consequently, we have the following result:

THEOREM 4.1. (**Decidability**) *Type checking in $\lambda_{\forall R}$ is decidable.*

Proof Follows from the decidability proof of EPR logic, to which bind equations are compiled to, and decidability result (Theorem 3.3) for λ_R. ∎

5. Implementation

We have implemented our specification language and verification procedure as an extended type-checking pass (called CATALYST) in MLton, a whole-program optimizing compiler for Standard ML (SML)[8] The input to our system is CoreML, an A-normalized intermediate representation with pattern-matching, but with all SML module constructs elaborated and removed. SML programs are annotated with relational specifications, defined in terms of relational dependent types that decorate function signatures, along with definitions of parameterized structural relations over the program's datatypes. The type system is a conservative extension of SML's, so all programs that are well-typed under CATALYST are well-typed SML programs. Our type-checking and verification process closely follows the description given in the previous sections. Verification conditions, representing the consequent of the SUBT-BASE type-checking rule (Fig. 3) are compiled to a first-order formula, as described in Sections 3 and 4, and checked for validity (satisfiability of its negation) using the Z3 SMT solver.

To be practically useful, our implementation extends the formal system described thus far in three important ways:

Primitive Relations. We provide a general framework to add new primitive relations that allows the class of relational expressions to be extended by permitting relational expressions to be abstracted in prenex form. The framework only needs to be seeded with the single primitive relation R_{id}. For example, R_{notEq_k} can be defined as the following primitive relation:

$$R_{notEq} = \lambda\ \mathtt{k}.\lambda\ \mathtt{x}.\ R_{id}(\mathtt{x}) - R_{id}(\mathtt{k})$$

Similarly, R_{eq_k} can be defined as:

$$R_{eq} = \lambda\ \mathtt{k}.\lambda\ \mathtt{x}.\ R_{id}(\mathtt{x}) - (R_{id}(\mathtt{x}) - R_{id}(\mathtt{k}))$$

Both R_{notEq} and R_{eq} can be ascribed colon-arrow sorts, similar to structural relations. Once defined, a primitive relation can be used freely in type refinements. For example, the relation yielded by evaluating $(R_{notEq}\ \mathtt{c})$ can be used to instantiate the parametric R_{mem} relation to define the set of all elements in a list that are not equal to some constant `c`.

Base Predicates: Consider the obvious relation refinement for the polymorphic identity function:

$$\mathtt{id} : \mathtt{x} \rightarrow \{\mathtt{v} \mid R_{id}(\mathtt{v}) = R_{id}(\mathtt{x})\ \}$$

The type refinement used here is an unintuitive way of expressing the simple fact that `id` returns its argument. To avoid such needless verbosity, we admit non-relational assertions (called *base*

[8] The source code for the implementation is available online from: *https://github.com/tycon/catalyst*.

```
fun exists l f = case l of
  [] => false
| x::xs =>
  let
    val v1 = exists xs f
    val v2 = f x
  in
    v1 orelse v2
  end
```

(a) `exists`

```
fun filter f l = case l of
  [] => []
| x::xs =>
  let
    val xs' = filter f xs
  in
    if f x then x::xs'
           else xs'
  end
```

(b) `filter`

```
fun contains l str =
  let
    val isStr = fn x => x=str
    (* Instantiate the implicit
     * relational parameter in type
     * of "exists" with (REq str) *)
    val hasStr = exists (REq str) l
        isStr
  in
    hasStr
  end
```

(c) `contains`

Figure 10: Examples

predicates), drawn from propositional logic with equality, to our specification language; these predicates may be freely composed in type refinements using logical connectives.

Inference and Annotation Burden: Our implementation infers sorts for structural relations, and relational parameters in dependent types. Our term language and specification language have distinct sort instantiation expressions. We also infer appropriate tuple-sort instantiations by unification. Therefore, neither the ML program, nor the specification needs to be annotated with sorts.

The type checking algorithm performs bi-directional type checking [17], and needs annotations only for recursive function definitions. For all other expressions, CATALYST synthesizes a suitable dependent type. For example, types from different branches of ML `case` expressions are unified using a logical disjunction. Generating a suitable type for a `let` expression requires that we use an existential quantifier in type refinements, which is skolemized while encoding the VC in MSFOL. Notably, we do not expose any quantifiers in our specification language.

For non-recursive function applications, although it is possible to infer instantiation annotations for parametric relations with the help of an expensive fixpoint computation that generates an exhaustive list of all possible instantiations, CATALYST relies on manual annotations for parameter instantiations to avoid this cost.

5.1 Experiments

We have used CATALYST to verify shape invariants, often to the extent of full functional correctness, on:

- List library functions, such as as `concat`, `rev`, `revAppend`, `foldl`, `foldr`, `zip`, `unzip` etc., and
- Okasaki's red-black tree [16] library functions, such as `balance`, multiple order traversal functions, and `mirrorImage`.

Excluding the time take by MLton compiler to elaborate and type check these Standard ML programs, none of our benchmarks took more than 0.2s to verify; this time includes A-Normalization, specification elaboration, VC generation, and SMT solving through Z3.

The specification of red-black tree `balance` function, shown in Fig. 9b, illustrates the kind of specifications that were automatically verified by CATALYST in our experiments. The specification asserts that the `balance` function on red-black trees (Fig. 9a) preserves the *total-order* among members of the tree. The non-inductive *total-order* relation (R_{to} in Fig. 9b) is defined in terms of tree *membership* relation (R_{tmem}), and relates (a) elements in the left sub-tree to the root element, (b) root to the elements in the right sub-tree, and (c) elements in the left sub-tree to those in right. Inductive *total-order* relation (R^*_{to}) on a red-black tree, obtained by closing the R_{to} relation over the tree, relates every pair of elements in the tree that are *in-order*. Consequently, the specification on `balance` function effectively asserts that in-order traversal on an unbalanced red-black tree, and in-order traversal on its balanced version, return same sequence of elements.

CATALYST can verify full functional correctness of standard tree traversal functions that return a list of elements. Relational specifications for such functions essentially relate different order relations on the input tree to *occurs-before* order of the result list. For instance, a function `inOrder` that performs in-order traversal on a red-black tree (t) returns a list (l) such that its inductive *occurs-before* relation is same as that of t's inductive *total-order* relation:

$$\texttt{inOrder} : \texttt{t} \rightarrow \{\texttt{l} \mid R^*_{ob}(\texttt{l}) = R^*_{to}(\texttt{t})\}$$

The specifications verified by CATALYST for higher-order `foldl` and `map` were shown in sections 2 and 4, respectively.

6. Case Study

An SML implementation of the untyped lambda calculus is shown in Fig. 11. The implementation makes use of auxiliary functions, such as `filter` and `contains`, directly, and `exists` through `contains`. By the virtue of being compositional, our verification process relies on expressive relational types of these auxiliary functions, which can nevertheless be verified by CATALYST. We present them below:

exists. Consider the higher-order `exists` function over lists shown in Fig. 10a; dependent type signatures are elided for brevity. A type that captures the semantics of `exists`, irrespective of its implementation, should assert that `exists` returns `true` if and only if its higher-order argument returns `true` for some member of the list. We express the invariant as the following type:

$$\begin{array}{l}(\texttt{'}R\ \texttt{exists}) : \\ \quad \texttt{l} \rightarrow \texttt{f} : \texttt{x} \rightarrow \{\ \nu \mid \nu = \texttt{true} \Leftrightarrow \texttt{'}R(\texttt{x}) \neq \emptyset)\} \rightarrow \\ \quad \{\nu \mid \nu = \texttt{true} \Leftrightarrow ((R_{mem}\ \texttt{'}R)\ \ \nu) \neq \emptyset\ \}\end{array}$$

The interpretation of the type is as follows: Let there be a relation $\texttt{'}R$ such that `f` returns `true` if and only if relation $\texttt{'}R(\texttt{x})$ is not the empty set for `f`'s argument `x`. Then, `exists` returns `true` if and only if relation R is not the empty set for some element in list.

filter. A parametric dependent type for `filter`, shown in Fig. 10b is given below:

$$\begin{array}{l}(\texttt{'}R\ \texttt{filter}) : \\ \quad \texttt{l} \rightarrow \texttt{f} : \texttt{x} \rightarrow \{\ \nu \mid \nu = \texttt{false} \Rightarrow \texttt{'}R(\texttt{x}) = \emptyset \\ \qquad\qquad\qquad\qquad \wedge\ \nu = \texttt{true} \Rightarrow \texttt{'}R(\texttt{x}) = R_{id}(\texttt{x})\ \} \rightarrow \\ \quad \{\nu \mid ((R_{mem}\ \texttt{'}R)\ \ \nu) = R_{mem}(\texttt{l})\ \}\end{array}$$

The intuition behind this type is same as that of `exists`. Filter retains only those elements for which its higher-order argument returns `true`. Consequently, `f`'s refinement requires $\texttt{'}R(\texttt{x})$ to be equal to $\emptyset$ when $\nu = \texttt{false}$.

ML Program

```
1  datatype exp =    Var of string
2                  | App of exp*exp
3                  | Abs of string*exp
4
5  fun freeVars e = case e of
6      Var id => [id]
7    | App (e1,e2) =>
8       concat [freeVars e1, freeVars e2]
9    | Abs (id,e') => filter (RNeq id)
10       (fn fv => not (fv = id)) (freeVars e')
11
12 fun alphaConvert e = case e of
13     Abs (id,e') =>
14     let
15       val fv_e' = freeVars e'
16       val id' = createNewName fv_e' id
17     in
18       Abs(id',subst(Var id',id,e'))
19     end
20   | _ => raise Error
21 and subst e1 id e2  = case e2 of
22     Var id' => if id = id'
23       then e1 else e2
24   | App(e21,e22) =>
25     let
26       val e21' = subst e1 id e21,
27       val e22' = subst e1 id e22
28     in
29       App (e21',e22')
30     end
31   | Abs(id',e2') => if id' = id then e2 else
32     let
33       val fv_e1 = freeVars e1
34     in
35       if contains fv_e1 id'
36       then subst e1 id (alphaConvert e2)
37       else Abs(id',subst e1 id e2')
38     end
```

Relational Specification

```
relation R_fv (Var x) = {(x)}
       | R_fv (App (e1,e2)) = R_fv(e1) ∪ R_fv(e2)
       | R_fv (Abs (id,e)) = R_fv(e) - {(id)};

createNewName : fvs → id → {v | not (v = id) ∧ not ({(v)} ⊆ Rmem(fvs))};
freeVars : e → {l | Rmem(l) = R_fv(e)};
alphaConvert : e → {ex | R_fv(ex) = R_fv(e)};
subst : e1 → id → e2 →
        {ex | if ({(id)} ⊆ R_fv(e2)) then R_fv(ex) = (R_fv(e2) - {(id)}) ∪ R_fv(e1) else R_fv(ex) = R_fv(e2)};
```

Figure 11: SML implementation and specification of the untyped lambda calculus.

contains. Consider the definition of the `contains` function shown in Fig. 10c that uses `exists` to check for the existence of a constant string `str` in a list `l`. Since the higher-order function passed to `exists` is:

$$\texttt{val isStr = fn x => x=str}$$

the relational dependent type of `isStr` is:

$$\texttt{isStr : x} \rightarrow \{\nu \mid R_{eq_{str}}(\nu) \neq \emptyset\}$$

This clearly suggests that the relational parameter of `exists` has to be instantiated with ($R_{eq_{str}}$ str). Having made this observation, we stress that no type annotation is required for `isStr`, as it is a non-recursive function.

Observe that the call to `exists` from `contains` includes explicit parameter instantiation. The resultant type of `hasStr` is:

$$\texttt{hasStr} : \{\nu \mid \nu\texttt{=true} \Leftrightarrow ((R_{mem}\ R_{eq_{str}})\ \texttt{l}) \neq \emptyset\}$$

The type refinement for `hasStr` indicates that `hasStr` is `true` if and only if the set of all elements of list `l` that are equal to `str` is not empty. Due to the equivalence of its first-order encoding to that of the following assertion:

$$\{\nu\texttt{=true} \Leftrightarrow R_{id}(\texttt{s}) \subseteq R_{mem}(\texttt{l})\},$$

the implementation of `contains` type-checks against the type:

$$\texttt{l} \rightarrow \texttt{str} \rightarrow \{\nu \mid \nu\texttt{=true} \Leftrightarrow R_{id}(\texttt{str}) \subseteq R_{mem}(\texttt{l})\}$$

6.1 α-conversion

The substitution operation (`subst`) substitutes a free variable (`id`) in an expression (`e2`) with another expression (`e1`). Function `alphaConvert` consistently renames occurrences of the bound variable in an abstraction expression. Observe that `subst` and `alphaConvert` are mutually recursive definitions. Both functions make use of `freeVars`, which returns a list of an expression's free variables.

It is widely agreed that substitution and α-conversion operations on lambda calculus terms are quite tricky to define correctly [22**?**]. Some of the behaviors exhibited by incorrect implementations include (a) α-conversion renames a free variable, or fails to rename a bound variable; (b) substitution fails to substitute free occurrences of the variable (`id`), or substitutes a bound occurrence of the variable; or (c) substitution is not capture-avoiding, i.e., substituting `e1` for `id` in `e2` captures variables of `e1`, which are otherwise free.

The relational specification of substitution and α conversion is given in the bottom-half of Fig. 11.[9] Note that one need not expose notions of capture-avoidance, or other such intricacies, to write down the specification, which is given in terms of a new structural relation R_{fv} that relates an expression of the calculus to its free variables. Function `freeVars` returns a list, whose members are free variables of its input expression. Its type represents this fact.

CATALYST successfully verifies the implementation against its specification. Alternate (incorrect) implementations such as those that fail to perform the capture-avoiding check on line 35, or the free variable check on line 31 trigger a type error. Conversely, note that, despite enforcing strong invariants, the relational specifications for `subst` and `alphaConvert` do not constrain how these functions are realized in ML. For instance, an implementation of `subst` that proactively renames bound variables in `e2` before substitution is successfully verified against the same specification.

[9] We introduce some syntactic sugar in defining type refinements. For example, the branch expression (`if` ϕ `then` ϕ_1 `else` ϕ_2) in a type refinement translates to $((\phi \wedge \phi_1) \vee (\neg\phi \wedge \phi_2))$.

7. Related Work

Type systems of mainstream functional languages, such as GHC Haskell and OCaml, support a basic form of dependent typing [13, 14] using GADTs [23]. At a high level, a structural relation of a data type is similar to a GADT insofar as it corresponds to an index that tracks an inductively definable relation over the data type. However, unlike the indexed type systems of Haskell and OCaml, where types are kept separate from terms, ours is a dependent type system. In this sense, our type system is similar to the refinement based dependent type system of F* [**?**]. Type refinements in F* are drawn from full first-order logic extended with theories that an SMT solver can reason with, whereas our specification language for ML programs is an abstraction over first-order logic that was tailor-made for equational and relational reasoning. The expressivity of using full-first order logic in F* comes at the cost of decidability of type checking. Further, even with access to full first-order logic in type refinements, a relationally parametric type cannot be directly expressed in F*; second-order quantification is required.

Structural relations, in their operational manifestation, can be compared to the structurally recursive *measures* of liquid types [12, 21] where the co-domain is always a set. Parametric structural relations may be viewed as generalizing such measures to higher-order measures. Relationally parametric dependent types can be compared to liquid types with abstract refinements [21], which let liquid types parameterize over type refinements (Boolean predicates). Once applied to a value, an abstract refinement becomes a concrete refinement, which can only be used to refine a type. On the other hand, a relational parameter can be treated just as any other relation in our type refinements, including being passed as an argument to other parametric relations. We require this generality to reason about shape invariants of higher-order catamorphisms such as `map` and `foldr`. For example, using only abstract refinements, it is not possible to verify that projecting a list of pairs using `map` and `fst` preserves ordering, or that an implementation of list `append` that uses `foldr` is correct.

Measures are an example of structurally recursive abstraction functions that map an algebraic data type to an abstract domain, such as natural numbers or sets. Suter *et al.* [20] describe decision procedures for the theory of algebraic data types extended with abstraction functions to decidable abstract domains. Our encoding does not require such extensions since a structural relation directly translates to an uninterpreted relation in first-order logic. Our encoding also supports parametric relations, which would otherwise require higher-order abstraction functions.

Imperative shape analyses have previously used relations to capture some inductive properties [8], and to describe memory configurations [11]. However, their applicability has been limited owing to destructive updates and pointer manipulations in imperative programs. In [15], Might describes a shape analysis of closures in higher-order programs. Our type system is capable of describing some notion of control flow for higher-order functions; e.g., the order in which the higher-order argument of `foldl` is applied over the list. However, inductive relations are conspicuous by their absence in functional program analysis, despite the fact that such programs are highly amenable for inductive reasoning. To the best of our knowledge, our type system is the first to use inductive relations for performing shape analysis on functional programs.

Logical relations have been used extensively to reason about contextual equivalence [4, 9]. Whereas a logical relation relates two terms of a (possibly recursive) type, a structural relation relates a term of an algebraic type to its constituent values. Parametric logical relations have also been used to reason about contextual equivalence for effectful programs [5–7]. In these efforts, a binary logical relation that relates effectful expressions is parametrized by a relation that relates their states. In contrast, a parametric structural relation is a structural relation over a polymorphic data type, that is parametrized by relations over type variables in the data type. While the primary purpose of structural relations is to enable specification and static verification, there is a possibility of sufficiently equipping our framework to reason about invariance of arbitrary relations, which is the key to reasoning about contextual equivalence. This is a possible avenue for future research.

Henglein [10] describes a domain-specific language to define ordering relations for composite data types such as lists and trees. However, the notion of order explored is the domain order used to compare two elements of same domain, such as a lexicographic order. In contrast, the order relation in our system describes relative ordering of elements in a composite data type.

8. Limitations and Future Work

Due to the undecidability of program equivalence in general, it is impossible for any specification language that is based on a decidable logic to be completely specify functional correctness of all possible ML programs. The expressivity of our specification language is inherently bound by the limits imposed by our choice of the underlying decidable first-order logic. Confinement to relational and equational theory means that it is not possible to express properties that rely on specific theories, such as arithmetic. For instance, it is not possible to write a relational specification that asserts that the result of folding over a list of integers with `(op +)` is the sum of all integers in the list. Further, we restrict ourselves to (parametric) structural relations over (polymorphic) inductive datatypes in this work. With this restriction, it may not be possible to express shape related properties over arbitrary non-inductive datatypes. For example, it is currently not possible to assert that in a random access array, an element at a smaller index *occurs-before* an element at a larger index. Nevertheless, these drawbacks can be mitigated by (a) admitting relations without requiring their equational definitions, and (b) extending our specification language with theory-specific artifacts (especially, from the theory of arithmetic) in a way such that the combination remains decidable. We intend to explore both these extensions as part of future work.

Another noticeable limitation is the lack of a general type inference mechanism. Given that relational specifications that make use of parametric relations to express rich invariants are non-trivial, and can be quite verbose, writing such specifications requires considerable manual effort. While providing higher level abstractions in the specification language can mitigate the problem by enabling the programmer to reason directly at the level of properties, rather than at the level of relations, the approach can be substantiated with a lightweight type inference mechanism based on refinement templates [19] to reduce the burden of manual annotation. The integration of such mechanisms within CATALYST is another avenue we anticipate pursuing.

9. Conclusions

This paper presents a relational specification language integrated with a dependent type system that is expressive enough to state structural invariants on functions over algebraic data types, often to the extent of full-functional correctness. We describe how parametric relations can be used to enable compositional verification in the presence of parametric polymorphism and higher-order functions. We additionally provide a translation mechanism to a decidable fragment of first-order logic that enables practical type checking. Experimental results based on an implementation (CATALYST) of these ideas justify the applicability of our approach.

Acknowledgments

We thank Matt Might, Ranjit Jhala, Niki Vazou, as well as the anonymous reviewers for their detailed comments and suggestions. This work is supported by the National Science Foundation under grants CCF-1216613.

References

[1] F*. `http://rise4fun.com/FStar/tutorial/guide`.

[2] The Glasgow Haskell Compiler. `https://www.haskell.org/ghc/`.

[3] Objective Caml. `http://ocaml.org/`.

[4] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP'06*, pages 69–83, 2006.

[5] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, 2005.

[6] N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations: Towards extensional semantics for effect analyses. In *APLAS*, 2006.

[7] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: Higher-order store. In *PPDP*, 2009.

[8] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.

[9] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. In *LICS'09*, pages 71–80, 2009.

[10] F. Henglein. Generic top-down discrimination for sorting and partitioning in linear time*. *J. Funct. Program.*

[11] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32(2), Feb. 2010.

[12] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, 2009.

[13] S. Lindley and C. McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *Haskell Symposium*, 2013.

[14] C. McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5), July 2002.

[15] M. Might. Shape analysis in the absence of pointers and structure. In *VMCAI*, 2010.

[16] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[17] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1), Jan. 2000.

[18] R. Piskac, L. de Moura, and N. Bjørner. Deciding effectively propositional logic with equality. Technical Report MSR-TR-2008-181.

[19] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *PLDI*, 2008.

[20] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, 2010.

[21] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.

[22] S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *ICFP*, 2011.

[23] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL*, 2003.

A. Appendix

A.1 Definitions

Our meta-theory relies on several definitions, which are stated below:

Definition *Simply Typed* λ_R We define simply typed λ_R whose type rules are same as those of simply typed lambda calculus. The rules are reproduced in Fig. 12 for the sake of completeness. The rules reuse Γ to denote simple type environment (against dependent type environment in the dependent type rules of λ_R) that maps variables to their (unrefined) types. Recall that the domain of a λ_R relation is a simple type. The S-APP rule, which sort checks relation applications, uses simple typing judgment to type check the argument.

The relationship between dependent typing judgment and simple typing judgment is established in Lemma A.8. Since typing judgment under a context Γ relies on well-sortedness of relation applications under same Γ, via well-formedness judgment, using simple typing judgment instead of dependent typing judgment avoids any circular reasoning.

Definition ***(Primitive Types of Constants)*** Just as the dependent typing judgment makes use of a function ty that maps constants to their dependent types, simple typing judgment makes use of the function pty that maps constants to their primitive types. It is defined as following:

$$\begin{array}{lcl} \forall i \in \mathbb{Z},\, pty(i) & = & \mathsf{int} \\ pty(\mathsf{Nil}) & = & \mathsf{intlist} \\ pty(\mathsf{Cons}) & = & \mathsf{int} \to \mathsf{intlist} \to \mathsf{intlist} \end{array}$$

Definition ***(VC Prelude)*** Sec. 3.3 of the paper introduces Γ_R as an ordered map from structural relations to their colon-arrow sorts. MSFOL translation of Γ_R (i.e., $[\![\Gamma_R]\!]$) is a set of assertions that assert sorts of uninterpreted relations in MSFOL. Since this set forms the context for verification conditions generated by subtype judgment in λ_R, we call the set as VC prelude.

Formally, Δ is the smallest set of MSFOL formulas such that forall R, $\cdot \vdash R :: T :\to \{\theta\}$,

$$R : [\![T :\to \{\theta\}]\!] \in \Delta$$

Remark ***(Entailment)*** In our proofs, we use $\models$ to denote semantic entailment in first-order logic. We write $\phi_1 \models \phi_2$ to denote that ϕ_1 semantically entails ϕ_2. Since several deductive systems, such as sequent calculus and natural deduction, are complete for first-order logic, we abuse $\models$ notation to also denote logical consequence. However, instead of using a set of hypotheses to the left of $\models$, we use a sequence (i.e., ordered set Γ) of hypotheses. We adapt few standard theorems of first-order deductive systems to our setting:

- Deduction Theorem: $\Gamma,\, \phi_1 \models \phi_2$ is equivalent to $\Gamma \models \phi_1 \Rightarrow \phi_2$.
- Monotonicity of Entailment (or Thinning): if $\Gamma \models \phi$, then forall Γ', $\Gamma',\, \Gamma \models \phi$
- Cut Elimination: If $\Gamma_1 \models \phi_1$, and $\Gamma_1,\, \phi_1,\, \Gamma_2 \models \phi_2$, then $\Gamma_1,\, \Gamma_2 \models \phi_2$

Definition *Free Variables* We define a function $freevars$ that returns a set of free variables in expressions (e) and type refinements (ϕ) of λ_R. For expressions, the definition of $freevars$ is straightforward, and follows that of simply typed lambda calculus. For a type refinement ϕ, $freevars(\phi)$ returns the union of $freevars$ of all λ_R values (v) that occur as arguments to relations in type refinements.

Definition *Substitution* Substitution operation substitutes a λ_R value (v) for a variable (z) in a λ_R expression (e), or a λ_R type refinement (ϕ). The definition of capture avoiding substitution for λ_R expressions is standard. The only caveat is that the substitution should also be performed on the type annotations occurring within expressions:

$$\begin{array}{lcll} [v/z]\,\lambda(x:\tau).\,e & = & \lambda(x:[v/z]\,\tau).\,e & \text{if } z = x \\ [v/z]\,\lambda(x:\tau).\,e & = & [v/z]\,alphaConvert(\lambda(x:\tau).\,e) & \text{if } x \in freevars(v) \\ [v/z]\,\lambda(x:\tau).\,e & = & \lambda(x:[v/z]\,\tau).\,[v/z]\,e & \textit{otherwise} \end{array}$$

Substitution operation for types is defined in terms of substitution operation for type refinements. For function types, capture avoidance property needs to be explicitly ensured:

$$\begin{array}{lcll} [v/z]\,\{\nu:T \mid \phi\} & = & \{\nu:T \mid [v/z]\,\phi\} & \\ [v/z]\,(x:\tau_1) \to \tau_2 & = & (x:[v/z]\,\tau_1) \to \tau_2 & \text{if } z = x \\ [v/z]\,(x:\tau_1) \to \tau_2 & = & [v/z]\,alphaConvert((x:\tau_1) \to \tau_2) & \text{if } x \in freevars(v) \\ [v/z]\,(x:\tau_1) \to \tau_2 & = & (x:[v/z]\,\tau_1) \to [v/z]\,\tau_2 & \text{otherwise} \end{array}$$

We assume a function $alphaConvert$ that performs alpha renaming of bound variable for abstraction expressions and function types. Substitution operation for type refinements is recursively defined in terms of relational predicates, and relational expressions. For relation application expressions, substitution is performed on the value (v') to which the relation is being applied:

$$[v/z]\,R(v') \quad = \quad R([v/z]\,v')$$

Definition ***(Basic Axioms)*** Basic axioms assert sorts, and validity of type refinements of constants in MSFOL. This accounts for theT-CONST rule of λ_R type system, which seeds the typing judgment with assumptions on types of constants. The axioms are stated below:

- Type of Integers: Forall integer constants c, $\models c : [\![\mathsf{int}]\!]$
- Type of Nil: $\models \mathsf{Nil} : [\![\mathsf{intlist}]\!]$
- Type of Cons: $x : [\![\mathsf{int}]\!],\, y : [\![\mathsf{intlist}]\!] \models \mathsf{Cons}\ x\ y : [\![\mathsf{intlist}]\!]$
- Validity of $\phi_n : \Delta \models [\mathsf{Nil}/\nu]\,\phi_n$
- Validity of $\phi_c : \Delta,\, x : [\![\mathsf{int}]\!],\, y : [\![\mathsf{intlist}]\!] \models [\mathsf{Cons}\ x\ y/\nu]\,\phi_c$

ST-VAR

$$\frac{(x : T) \in \Gamma}{\Gamma \Vdash x : T}$$

ST-APP

$$\frac{\Gamma \Vdash e : T_1 \to T_2 \quad \Gamma \Vdash v : T_1}{\Gamma \Vdash e\ v : T_2}$$

ST-ABS

$$\frac{T_1 = \|\tau_1\| \quad \Gamma, x : T_1 \Vdash e : T_2}{\Gamma \Vdash \lambda(x : \tau_1).\, e : T_1 \to T_2}$$

ST-CONST

$$\frac{}{\Gamma \Vdash c : pty(c)}$$

ST-LET

$$\frac{\Gamma \Vdash e_1 : T_1 \quad \Gamma, x : T_1 \Vdash e_2 : T_2}{\Gamma \Vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2}$$

ST-MATCH

$$\frac{\Gamma \Vdash v : \mathsf{intlist} \quad \Gamma_c = x : \mathsf{int}, y : \mathsf{intlist} \quad \Gamma, \Gamma_c \Vdash e1 : T \quad \Gamma \Vdash e2 : T}{\Gamma \Vdash \texttt{match}\ v\ \texttt{with}\ \mathsf{Cons}\ x\ y\ \Rightarrow\ e_1\ \texttt{else}\ e_2 : T}$$

Figure 12: Type Rules for simply typed λ_R

Evaluation Rules $\boxed{\text{e1} \longrightarrow \text{e2}}$

$$(\lambda(x : \tau).\, e)\, \nu \longrightarrow [\nu/x]\, e \qquad \text{E-}\beta$$

$$\texttt{let}\ x = \nu\ \texttt{in}\ e \longrightarrow [\nu/x]\, e \qquad \text{E-LET}$$

$$(\texttt{match}\ \mathsf{Cons}\ \nu_1\ \nu_2\ \texttt{with}\ \mathsf{Cons}\ x\ y\ \Rightarrow\ e_1\ \texttt{else}\ e_2) \longrightarrow [\nu_2/y][\nu_1/x]\, e_1 \qquad \text{E-MCONS}$$

$$(\texttt{match}\ \mathsf{Nil}\ \texttt{with}\ \mathsf{Cons}\ x\ y\ \Rightarrow\ e_1\ \texttt{else}\ e_2) \longrightarrow e_2 \qquad \text{E-MNIL}$$

$$\frac{e_1 \longrightarrow e_2}{E[e_1] \longrightarrow E[e_2]} \qquad \text{E-CO}$$

Evaluation Context $\boxed{\text{E}}$

$$E \quad ::= \quad \bullet \mid \bullet\ \nu \mid \texttt{let}\ x = \bullet\ \texttt{in}\ e$$

Figure 13: Operational Semantics of Core Calculus (λ_R)

A.2 Type Safety

We now prove[10] the type safety of λ_R's type system by proving its progress and preservation properties. Call-by-value operational semantics of λ_R are given in Fig. 13.

THEOREM A.1. (**Progress**) *If $\cdot \vdash e : \tau$, then either e is a value or there exists an e' such that $e \longrightarrow e'$.*

Proof By induction on type derivation. Cases:

- Case T-VAR: e is a variable x. By inversion on $\cdot \vdash x : \tau$ we get $x : \tau \in \cdot$, which is absurd. Proof follows from *ex falso quodlibet*.
- Case T-CONST: e is a constant c, which is a value.
- Case T-APP: e is of form $e_1\ v$, where $\cdot \vdash e_1 : (x : \tau_1) \to \tau_2$, and $\cdot \vdash v : \tau_1$. By IH, either e_1 can take a step or e_1 is a value.
  - $e_1 \longrightarrow e_2$. As per our definition of evaluation contexts (Fig. 13), $e_1\ v \longrightarrow e_2\ v$; So, $e' = e_2\ v$.
  - e_1 is a value v_1. By inversion on v_1, followed by eliminating cases using the assumption $\cdot \vdash v_1 : (x : \tau_1) \to \tau_2$, we are left with following cases:
    - v_1 is of form $\lambda x : \tau.\, e_3$, for some e_3: Consequently, $e = \lambda x : \tau.\, e_3\ v$, which reduces by E-$\beta$ to $[v/x]e_3$
    - v_1 is Cons: $\mathsf{Cons}\ v$ is a value.
    - v_1 is $\mathsf{Cons}\ v_2$, for some v_2: $\mathsf{Cons}\ v_2\ v$ is a value.
- Case T-ABS: $e = \lambda x : \tau.\, e_1$, which is a value.
- Case T-LET: $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$, where $\cdot \vdash e_1 : \tau_1$, and $\cdot, x : \tau_1 \vdash e_2 : \tau_2$. Similar to T-APP, we have two cases:
  - $e_1 \longrightarrow e_1'$, in which case $e \longrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2$ (as per our definition of evaluation contexts).
  - e_1 is a value v_1, in which case e reduces by rule E-LET to $[v_1/x]e_2$.
- Case T-SUB: $\cdot \vdash e : \tau'$, where $\tau' <: \tau$. Proof follows from IH.
- Case T-MATCH: $e = \texttt{match}\ v\ \texttt{with}\ \mathsf{Cons}\ x\ y\ \Rightarrow\ e_1\ \texttt{else}\ e_2$, where $\cdot \vdash v : \{\nu : \mathsf{intlist} \mid \phi\}$. By inversion on v and eliminating absurd cases, we are left with two cases:
  - $v = \mathsf{Cons}\ v_1\ v_2$, in which case e reduces by E-MCONS to $[v_1/x][v_2/y]e_1$.
  - $v = \mathsf{Nil}$, in which case e reduces by M-ENIL to e_2.

LEMMA A.2. (**Context Invariance for Well-Formedness**) *If $\Gamma \vdash \tau$, then forall Γ' such that $\|\Gamma'\| = \|\Gamma\|$, $\Gamma' \vdash \tau$*

Proof Since well-formedness of a type directly derives from well-sortedness of relational expressions occurring in its type refinement, It suffices to prove that:

$$\text{forall } r, \text{ if } \Gamma \vdash r :: \{\theta\}, \text{ then forall } \Gamma' \text{ such that } \|\Gamma'\| = \|\Gamma\|, \Gamma' \vdash r :: \{\theta\}.$$

[10] A note on the notation adapted in writing proofs: Most proofs are by induction or inversion, leading to cases. Hypotheses and inductive hypotheses are named as per $H[0-9]+$ and $IH[0-9]+$ grammars, respectively. Names refer to a different hypotheses in different cases. Coq tactic names are used used to convey proof strategy, wherever applicable.

We prove this by induction on $\Gamma \vdash r :: \{\theta\}$. Cases S-UNION and S-CROSS follow directly from inductive hypotheses. The only interesting case is S-APP, where $r = R(v)$, for some R and v. The proof is by inversion on $\Gamma \vdash R(v) :: \{\theta\}$. Hypotheses:

$$\begin{array}{rl} \cdot \vdash R :: T :\to \{\theta\} & (H1) \\ \|\Gamma\| \Vdash v : T & (H2) \end{array}$$

Rewriting $H2$ using $\|\Gamma\| = \|\Gamma'\|$:

$$\|\Gamma'\| \Vdash v : T \quad (H3)$$

Applying S-APP rule over $H1$ and $H3$ proves the goal. ∎

LEMMA A.3. (**Cut Elimination**) *Forall x, e, τ, ϕ, and Γ, If $\Delta \models [\![\phi]\!]$, and ϕ, $\Gamma \vdash e : \tau$, then $\Gamma \vdash e : \tau$*

Proof by induction on ϕ, $\Gamma \vdash e : \tau$. Cases:

- Case T-VAR : e is a variable y. Inversion of ϕ, $\Gamma \vdash y : \tau$ produces $y : \tau \in \phi$, Γ. From the definition of type environment, it follows that $y : \tau \in \Gamma$. Applying T-VAR produces proof.
- Case T-CONST: e is a constant c. Proof trivial, as constants have same type irrespective of the context.
- Case T-SUB: Hypotheses:

  $$\begin{array}{rl} \tau = \tau_2 & (H0) \\ \Delta \models [\![\phi]\!] & (H1) \\ \phi, \Gamma \vdash \tau_1 <: \tau_2 & (H3) \end{array}$$

  Inductive hypothesis is:

  $$\Gamma \vdash e : \tau_1 \quad (IH0)$$

  It remains to show that $\Gamma \vdash \tau_1 <: \tau$, which we prove by induction on subtype derivation in $H3$. Cases:
  - SubCase SUBT-BASE: Hypotheses:

    $$\begin{array}{rl} \tau_1 = \{\nu : T \mid \phi_1\} & (H4) \\ \tau_2 = \{\nu : T \mid \phi_2\} & (H5) \\ \phi, \Gamma \vdash \{\nu : T \mid \phi_1\} & (H6) \\ \phi, \Gamma \vdash \{\nu : T \mid \phi_2\} & (H7) \\ \Delta \models [\![\phi]\!] \Rightarrow [\![\Gamma]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!] & (H8) \end{array}$$

    Since $\|\phi, \Gamma\| = \|\Gamma\|$, we can apply Lemma A.2 to derive the following from $H6 - 7$:

    $$\begin{array}{rl} \Gamma \vdash \{\nu : T \mid \phi_1\} & (H8) \\ \Gamma \vdash \{\nu : T \mid \phi_2\} & (H9) \end{array}$$

    From $H8$, using deduction theorem of first order logic, we obtain:

    $$\Delta, [\![\phi]\!], [\![\Gamma]\!], [\![\phi_1]\!] \models [\![\phi_2]\!] \quad (H10)$$

    We apply cut elimination theorem of logical consequence in first-order logic to $H1$ and $H10$ and derive:

    $$\Delta, [\![\Gamma]\!], [\![\phi_1]\!] \models [\![\phi_2]\!] \quad (H11)$$

    Using the the deduction, $H11$ is equivalent to:

    $$\Delta \models [\![\Gamma]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!] \quad (H12)$$

    Finally, applying SUBT-BASE rule to $H8$,$H9$, and $H12$ leads us to conclude that $\Gamma \vdash \tau_1 <: \tau_2$
  - Case SUBT-ARROW: One can derive proof for this case by simply applying SUBT-ARROW to inductive hypotheses.
- Cases T-APP, T-ABS, T-LET, T-MATCH: Proofs for these cases follow straightforwardly from respective inductive hypotheses.

LEMMA A.4. (λ_R **Type System is Conservative**) *forall Γ, if $\Gamma \vdash e : \tau$ then $\|\Gamma\| \Vdash e : \|\tau\|$.*

Proof By induction on $\Gamma \vdash e : \tau$. Cases:

- Case T-VAR: e is a variable x. Hypotheses:

  $$(x : \tau) \in \Gamma$$

  From the definition of $\|\Gamma\|$, we know that $(x : \|\tau\|) \in \|\Gamma\|$. Applying ST-VAR gives the proof.
- Case T-CONST:e is a constant c. Case analyzing c:
  - c is an integer constant: Observing that $\|\mathsf{int}\| = \mathsf{int}$, and $ty(c) = pty(c)$, gives us proof.
  - c is Nil: Observing that $ty(\mathsf{Nil}) = \{\nu : \mathsf{intlist} \mid \phi_n\}$, and $\|ty(\mathsf{Nil})\| = \mathsf{intlist} = pty(\mathsf{Nil})$, gives us proof
  - c is Cons: Proof similar to the Nil case.
- Case T-APP: $e = e_1\ v$, where:

  $$\begin{array}{rl} \Gamma \vdash e_1 : (x : \tau_1) \to \tau_2 & (H0) \\ \Gamma \vdash v_1 : \tau_1 & (H1) \end{array}$$

  Inductive hypotheses (after simplifying $\|(x : \tau_1) \to \tau_2\|$):

  $$\begin{array}{rl} \|\Gamma\| \Vdash e_1 : \|\tau_1\|) \to \|\tau_2\| & (IH0) \\ \|\Gamma\| \Vdash v_1 : \|\tau_1\| & (IH1) \end{array}$$

  Applying ST-APP on $IH0 - 1$ gives proof.

- Case T-SUB: Hypotheses:

$$\Gamma \vdash e : \tau_1 \quad (H0)$$
$$\tau_1 <: \tau \quad (H1)$$

  Inductive hypothesis:

$$\|\Gamma\| \Vdash e : \|\tau_1\| \quad (IH0)$$

  By inversion on $H1$, it is easy to derive that $\|\tau_1\| = \|\tau\|$. Using this to rewrite $IH0$ produces proof.
- Case T-ABS: $e = \lambda(x : \tau_1).\, e_1$, and $e = \lambda(x : \|\tau_1\|).\, e_1$. Hypotheses:

$$\tau = (x : \tau_1) \to \tau_2 \quad (H4)$$
$$\Gamma, x : \tau_1 \vdash e_1 : \tau_2 \quad (H5)$$
$$\|\tau\| = \|\tau_1\| \to \|\tau_2\| \quad (H6)$$

  Inductive hypotheses (after unfolding $\|\Gamma,\, x : \tau\|$ to $\|\Gamma\|,\, x : \|\tau_1\|$):

$$\|\Gamma\|,\, x : \|\tau_1\| \Vdash e_1 : \tau_2 \quad (IH0)$$

  Applying ST-ABS on $IH0$ gives proof.
- T-LET: Proof closely resembles that for T-ABS case.
- T-MATCH: $e = \texttt{match}\ v\ \texttt{with Cons x y} \Rightarrow e_1\ \texttt{else}\ e_2$, and $e = \texttt{match}\ v\ \texttt{with Cons x y} \Rightarrow e_1\ \texttt{else}\ e_2$. Hypotheses:

$$\Gamma \vdash v : \mathsf{intlist} \quad (H0)$$
$$\Gamma \vdash \mathsf{Nil} : \{\nu : \mathsf{intlist} \mid \phi_n\} \quad (H1)$$
$$\Gamma \vdash \mathsf{Cons} : x : \mathsf{int} \to y : \mathsf{intlist} \to \{\nu : \mathsf{intlist} \mid \phi_c\} \quad (H2)$$
$$\Gamma_c = x : \mathsf{int},\, y : \mathsf{intlist},\, \phi \quad (H4)$$
$$\Gamma_n = \phi_n \quad (H5)$$
$$\Gamma \vdash \tau \quad (H6)$$
$$\Gamma, \Gamma_c \vdash e_1 : \tau \quad (H7)$$
$$\Gamma, \Gamma_n \vdash e_2 : \tau \quad (H8)$$

  Inductive hypotheses:

$$\|\Gamma\| \vdash v : \mathsf{intlist} \quad (IH0)$$
$$\|\Gamma\|,\, \|\Gamma_c\| \vdash e_1 : \|\tau\| \quad (IH1)$$
$$\|\Gamma\|,\, \|\Gamma_n\| \vdash e_2 : \|\tau\| \quad (IH2)$$

  Where,

$$\|\Gamma_c\| = x : \mathsf{int},\, y : \mathsf{intlist}$$
$$\|\Gamma_n\| = \cdot$$

  Applying ST-MATCH on $IH0 - 2$ produces proof.

We assert the substitution lemma for simply typed λ_R, but elide its proof as it closely follows the proof of substitution lemma for simply typed lambda calculus. The lemma is stated thus:

LEMMA A.5. (**Substitution Preserves Simple Typing**) *forall* Γ, x, e, T_1, *and* T_2, *if* $\cdot, x : T_1, \Gamma \vdash e : T_2$ *and* $\cdot \vdash v : T_1$, *then* $\Gamma \vdash [v/x]\, e : T_2$.

LEMMA A.6. (**Substitution Preserves Well-formedness of Relational Expressions**)
forall Γ, *if* $x : \tau_1, \Gamma \vdash r :: \{\theta\}$ *and* $\cdot \vdash v : \tau_1$, *then* $[v/x]\,\Gamma \vdash [v/x]\, r :: \{\theta\}$.

Proof by induction on the derivation of $x : \tau_1, \Gamma \vdash r :: \{\theta\}$. Cases:

- Case S-APP: $r = R(v_1)$, and $[v/x]\ r = R([v/x]\ v_1)$. After expanding $\|x : \tau_1, \Gamma\|$ to $x : \|\tau_1\|,\, \|\Gamma\|$, hypotheses are:

$$x : \|\tau_1\|,\, \|\Gamma\| \Vdash v_1 : T \quad (H0)$$

  Applying Lemma A.4 on $\cdot \vdash v : \tau_1$ gives:

$$\cdot \Vdash v : \|\tau_1\| \quad (H1)$$

  Applying the substitution lemma for simple type judgment of λ_R (Lemma A.5) on $H0$ and $H1$, we derive:

$$\|\Gamma\| \Vdash [v/x]\, v_1 : T \quad (H2)$$

  From the definition of substitution and erasure operations on type environments, we have that $\|[v/x]\,\Gamma\| = \|\Gamma\|$. Using this to rewrite $H2$:

$$\|[v/x]\,\Gamma\| \Vdash [v/x]\, v_1 : T \quad (H3)$$

  Applying S-APP to $H3$ produces proof.
- Cases S-UNION, and S-CROSS: Proof follows trivially from inductive hypotheses.

LEMMA A.7. (**Substitution Preserves Well-formedness**) *forall* Γ, *if* $x : \tau_1, \Gamma \vdash \tau$ *and* $\cdot \vdash v : \tau_1$, *then* $[v/x]\,\Gamma \vdash [v/x]\,\tau$.

Proof Well-formedness judgment of λ_R types directly follows that of type refinements, which is in-turn dependent on well-formedness of relational predicates in type refinements, and ultimately on well-sortedness of relational expressions that constitute such predicates. Therefore, it suffices to show that substitution preserves the sort of relational expressions, which follows from Lemma A.6 ∎

LEMMA A.8. (**Abstract Type of a λ_R Value**) *forall* v, *if* $\cdot \vdash v : \{\nu : T \mid \phi\}$, *then* $\models v : [\![T]\!]$ *and* $\Delta \models [v/\nu]\,[\![\phi]\!]$.

Proof By case analysis on v. Cases:

- Case v is a variable x. Inversion of $\cdot \vdash x : \{\nu : T \mid \phi\}$ leads to absurdity. *ex falso quodlibet*.
- Case v is an abstraction $\lambda x : \tau.\, e$. Again, inversion leads to absurdity, as an abstraction cannot have a dependent base type.
- Case v is an integer c. Induction on $\cdot \vdash c : \{\nu : T \mid \phi\}$ leads to two relevant cases:
  - SubCase T-CONST: $T = \mathsf{int}$ and $\phi = true$. We know that $\models true$ is trivially valid. From the Definition A.1, we also have that $\models c : [\![\mathsf{int}]\!]$.
  - SubCase T-SUB : Hypotheses:

$$\begin{array}{rl} \cdot \vdash c : \tau_1 & (H0) \\ \cdot \vdash \tau_1 <: \{\nu : T \mid \phi\} & (H1) \end{array}$$

    By inversion on $H1$, we have:

$$\begin{array}{rl} \tau_1 = \{\nu : T \mid \phi_1\} & (H2) \\ \cdot \vdash c : \{\nu : T \mid \phi_1\} & (H3) \\ \cdot \vdash \{\nu : T \mid \phi_1\} & (H4) \\ \cdot \vdash \{\nu : T \mid \phi\} & (H4) \\ \Delta \models \nu : [\![T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi]\!] & (H6) \end{array}$$

    Inductive hypotheses:

$$\begin{array}{rl} \models c : [\![T]\!] & (IH0) \\ \Delta \models [c/\nu]\,[\![\phi_1]\!] & (IH1) \end{array}$$

    Since ν occurs free in $H6$, applying the universal quantification introduction rule ($\forall I$):

$$\Delta \models \forall \nu.\, (\nu : [\![T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi]\!])$$

    Now, eliminating the quantifier (rule $\forall E$) by instantiating the bound variable with c:

$$\Delta \models c : [\![T]\!] \Rightarrow [c/\nu]\,[\![\phi_1]\!] \Rightarrow [c/\nu]\,[\![\phi]\!] \quad (H8)$$

    Using weakened form of $IH0$ (with Δ introduced in its context using weakening theorem of first order logic), $IH1$, and $H8$ we have:

$$\models [c/\nu]\,[\![\phi]\!] \quad (H9)$$

    Proof follows from $IH0$ and $H9$.
- Case v is Nil: From Definition A.1, we have that:

$$\begin{array}{rl} T = \mathsf{intlist} & (H0) \\ \models \mathsf{Nil} : [\![\mathsf{intlist}]\!] & (H1) \\ \Delta \models [\mathsf{Nil}/\nu]\,[\![\phi]\!] & (H2) \end{array}$$

  Proof follows straightforwardly from hypotheses.
- Case v is $\mathsf{Cons}\ v_1\ v_2$, such that

$$\cdot \vdash \mathsf{Cons}\ v_1\ v_2 : \{\nu : T \mid \phi\} \quad (H0)$$

  We first show that $T = \mathsf{intlist}$. Applying Lemma A.4 on $H0$, we have:

$$\cdot \Vdash \mathsf{Cons}\ v_1\ v_2 : T \quad (H1)$$

  By inversion on the simple type derivation (Figure 12) in $H1$, we show that $T = \mathsf{intlist}$. Using this to rewrite $H0$:

$$\cdot \vdash \mathsf{Cons}\ v_1\ v_2 : \{\nu : \mathsf{intlist} \mid \phi\} \quad (H2)$$

  From $H1$, and Definition A.1, we prove that

$$\models \mathsf{Cons}\ v_1\ v_2 : [\![\mathsf{intlist}]\!]$$

- Cases when v is Cons, or v is $\mathsf{Cons}\ v_1$, for some value v_1, lead to contradiction as v cannot have a base dependent type in these cases.

LEMMA A.9. (**Substitution Preserves Subtyping**) *forall* Γ*, if* $., x : \tau, \Gamma \vdash \tau_1 <: \tau_2$ *and* $\cdot \vdash v : \tau$*, then* $[v/x]\,\Gamma \vdash [v/x]\,\tau_1 <: [v/x]\,\tau_2$.

Proof By induction on the subtype judgment. Cases:

- Case SUBT-ARROW: Proof is a straightforward application of SUBT-ARROW on inductive hypotheses.
- Case SUBT-BASE: τ_1 is of form $\{\nu : T \mid \phi_1\}$, and τ_2 is of form $\{\nu : T \mid \phi_2\}$. By destructing τ, we have two cases:
  - SubCase $\tau = \{\nu : T_x \mid \phi_x\}$ for some T_x and ϕ_x: Hypotheses:

$$\begin{array}{rl} .\, x : \tau, \Gamma \vdash \tau_1 <: \tau_2 & (H0) \\ \cdot \vdash v : \{\nu : T_x \mid \phi_x\} & (H1) \\ .\, x : \tau, \Gamma \vdash \{\nu : T \mid \phi_1\} & (H2) \\ .\, x : \tau, \Gamma \vdash \{\nu : T \mid \phi_2\} & (H3) \\ \Delta \models [\![x : \{\nu : T_x \mid \phi_x\}, \Gamma, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!] & (H4) \end{array}$$

    Expanding $H4$:

$$\Delta \models x : [\![T_x]\!] \Rightarrow [\![\phi_x]\!] \Rightarrow [\![\Gamma, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!] \quad (H5)$$

    Since x occurs free in the above formula, using the universal quantification introduction rule ($\forall I$) , we have:

$$\Delta \models \forall x.\, x : [\![T_x]\!] \Rightarrow [\![\phi_x]\!] \Rightarrow [\![\Gamma, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!]$$

Now, using the elimination rule of universal quantification ($\forall E$) to instantiate the bound x with v:

$$\Delta \models [v/x]\,(x : [\![T_x]\!] \Rightarrow [\![\phi_x]\!] \Rightarrow [\![\Gamma,\, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \;\Rightarrow\; [\![\phi_2]\!]) \quad (H6)$$

Distributing the substitution operation:

$$\Delta \models (v : [\![T_x]\!] \Rightarrow [\![[v/x]\,\phi_x]\!] \Rightarrow [\![[v/x]\,\Gamma,\, \nu : T]\!] \Rightarrow [\![[v/x]\,\phi_1]\!] \;\Rightarrow\; [\![[v/x]\,\phi_2]\!]) \quad (H7)$$

Now, from Lemma A.8, using the hypothesis $H2$, we derive:

$$\models v : [\![T_x]\!] \quad (H9)$$
$$\Delta \models [\![[v/x]\,\phi_x]\!] \quad (H10)$$

Using $H8$, weakened $H9$, where Δ is introduced in its context, and $H10$, we derive:

$$\Delta \models [\![[v/x]\,\Gamma,\, \nu : T]\!] \Rightarrow [\![[v/x]\,\phi_1]\!] \;\Rightarrow\; [\![[v/x]\,\phi_2]\!]) \quad (H11)$$

Applying Lemma A.7 over $H2$ and $H3$ yields:

$$[v/x]\,\Gamma \vdash \{\nu : T \mid [v/x]\,\phi_1\} \quad (H12)$$
$$[v/x]\,\Gamma \vdash \{\nu : T \mid [v/x]\,\phi_2\} \quad (H13)$$

Applying SUBT-BASE on $H11 - 13$ proves the theorem.

- SubCase $\tau = \tau_{x_1} \to \tau_{x_2}$, for some τ_{x_1} and τ_{x_2}. First, we observe that all free variables in well-formed type refinements (i.e., arguments to relations. Please refer to the syntactic class τ_R in Fig. 1 of the paper.) have type int or intlist, therefore $x : \tau_{x_1} \to \tau_{x_2}$ cannot occur free in ϕ_1, ϕ_2, and type-refinements in Γ. Consequently:

$$[v/x]\,\Gamma = \Gamma \quad (H0)$$
$$[v/x]\,\phi_1 = \phi_1 \quad (H1)$$
$$[v/x]\,\phi_2 = \phi_2 \quad (H2)$$

Rewriting inductive hypotheses (not shown here) using $H0 - 3$ and applying SUBT-BASE results in proof.

LEMMA A.10. (**Weakening**) *if* $\Gamma \vdash e : \tau$ *then forall* Γ', $\quad \Gamma', \Gamma \vdash e : \tau$.

Proof by induction on $\Gamma \vdash e : \tau$ derivation. In most cases, proof follows directly from inductive hypotheses. The only interesting case is T-SUB:

- Case T-SUB: Hypothesis:

$$\Gamma \vdash e : \tau \quad (H0)$$
$$\tau_1 <: \tau \quad (H1)$$

By inductive hypotheses, we have:

$$\Gamma', \Gamma \vdash e : \tau_1 \quad (IH0)$$

To apply SUBT-BASE in order to prove the lemma, it suffices to prove that $\Gamma', \Gamma \vdash \tau_1 <: \tau$:, which we prove by induction on $H1$. Cases:

- SubCase SUBT-BASE: Hypotheses:

$$\tau_1 = \{\nu : T \mid \phi_1\} \quad (H2)$$
$$\tau = \{\nu : T \mid \phi\} \quad (H3)$$
$$\Delta \models [\![\Gamma,\, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi]\!] \quad (H4)$$

To prove that $\Gamma', \Gamma \vdash \tau_1 <: \tau$:, it suffices to prove that

$$[\![\Gamma']\!],\, \Delta \models [\![\Gamma,\, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi]\!]$$

Which follows from $H4$ by monotonicity of entailment (or thinning) in first-order logic.

- SubCase SUBT-ARROW: Proof follows trivially from inductive hypotheses by applying SUBT-ARROW.

LEMMA A.11. (**Well-Typedness Implies Well-Formedness**) *forall* v, *and* τ, *if* $\cdot \vdash v : \tau$ *then* $\cdot \vdash \tau$.

Proof is by case analysis on the structure of v, followed by induction on the typing derivation $\Gamma \vdash v : \tau$, for each case of v. After trivially discharging the cases that result in contradiction, we are left with following cases:

- Case T-SUB: Proof follows from the premises of SUBT-BASE rule, which explicitly assert well-formedness of types involved in subtype judgment.
- Case T-CONST: Type refinement $true$ for integer constants is well-formed under any context. Well-formedness of Nil and Cons type refinements are explicitly asserted in Definition A.1.
- Case T-ABS: Proof by applying WF-FUN on inductive hypotheses.

Since expressions of λ_R are in A-Normal form by construction, we only need substitution lemma for value substitutions.

LEMMA A.12. (**Substitution Preserves Typing**) *forall* Γ, x, e, τ_1, *and* τ_2, *if* $\cdot, x : \tau_1, \Gamma \vdash e : \tau_2$ *and* $\cdot \vdash v : \tau_1$, *then* $[v/x]\,\Gamma \vdash [v/x]\,e : [v/x]\,\tau_2$.

Proof . Hypotheses (after generalizing dependent Γ and τ_2):

$$\forall \Gamma, \forall \tau_2, \cdot, x : \tau_1, \Gamma \vdash e : \tau_2 \quad (H0)$$
$$\cdot \vdash v : \tau_1 \quad (H1)$$

First, using $H1$ and Lemma A.11, we derive the following:

$$\cdot \vdash \tau_1 \quad (H2)$$

Now, we proceed by induction on $H0$. Cases:

- Case T-VAR: e is a variable y such that:

  $$\cdot, x : \tau_1, \Gamma \vdash y : \tau_2 \quad (H3)$$

  We have two subcases:
  - SubCase $y = x$: Since a variable is never bound twice in the environment, by inversion on $\cdot, x : \tau_1, \Gamma \vdash x : \tau_2$, we know that:

    $$\tau_1 = \tau_2 \quad (H4)$$

    Since $[v/x]\, x = v$ it remains to prove that $[v/x]\, \Gamma \vdash v : [v/x]\, \tau_2$. Rewriting using $H4$, the goal is:

    $$[v/x]\, \Gamma \vdash v : [v/x]\, \tau_1$$

    From $H2$, we know that τ_1 is well-formed under empty context; so, its type-refinement is closed. Hence, $[v/x]\, \tau_1 = \tau_1$. Using this to rewrite the goal:

    $$[v/x]\, \Gamma \vdash v : \tau_1$$

    From $H1$, we know that $\cdot \vdash v : \tau_1$. Applying the weakening lemma (Lemma A.10), with bound Γ' instantiated to $[v/x]\, \Gamma$ gives us the required proof.
  - SubCase $y \neq x$: From $H3$, since $y \neq x$, we have:

    $$(y : \tau_2) \in \Gamma \quad (H5)$$

    Applying definition of substitution lifted to type environments yields the following:

    $$(y : [v/x]\, \tau_2) \in [v/x]\, \Gamma \quad (H6)$$

    Since $[v/x]\, y = y$, applying T-VAR rule using $H6$ lets us conclude that

    $$[v/x]\, \Gamma \vdash [v/x]\, y : [v/x]\, \tau_2$$

    which is the required goal.
- Case T-CONST: SubCases:
  - e is an integer constant c: Proof trivial as $[v/x]\, c = c$ and c has type int under any context (Definition A.1 and T-CONST).
  - e is Nil, or e is Cons: Hypotheses:

    $$\Gamma \vdash \mathsf{Nil} : ty(\mathsf{Nil}) \quad (H1)$$
    $$\Gamma \vdash \mathsf{Cons} : ty(\mathsf{Cons}) \quad (H2)$$

    From the definition of ty (Definition A.1), we know that type refinements of Nil and Cons are well-formed under empty context; so, they are closed. Consequently $[v/x]\, ty(\mathsf{Nil}) = ty(\mathsf{Nil})$, and $[v/x]\, ty(\mathsf{Cons}) = ty(\mathsf{Cons})$. Also, $[v/x]\, \mathsf{Nil} = \mathsf{Nil}$ and $[v/x]\, \mathsf{Cons} = \mathsf{Cons}$. Therefore, proof follows from $H1$ and $H2$.
- Case T-APP: e is a function application of form $e_1\, v_1$, where:

  $$\cdot, x : \tau_1, \Gamma \vdash e_1 : (y : \tau_3) \to \tau_2 \quad (H4)$$
  $$\cdot, x : \tau_1, \Gamma \vdash v_1 : \tau_3 \quad (H5)$$

  Inductive hypotheses, after trivially instantiating bound Γ and τ_2:

  $$[v/x]\, \Gamma \vdash [v/x]\, e_1 : [v/x]\, ((y : \tau_3) \to \tau_4) \quad (IH0)$$
  $$[v/x]\, \Gamma \vdash [v/x]\, v_1 : [v/x]\, \tau_3 \quad (IH1)$$

  Pushing the substitution to the level of base types in the arrow type:

  $$[v/x]\, \Gamma \vdash [v/x]\, e_1 : (y : [v/x]\, \tau_3) \to [v/x]\, \tau_4 \quad (H6)$$

  Since $[v/x]\, (e_1\, v_1) = [v/x]\, e_1\, [v/x]\, v_1$, the goal is to prove:

  $$[v/x]\, \Gamma \vdash [v/x]\, e_1\, [v/x]\, v_1 : [v/x]\, \tau_2$$

  Applying T-APP using $H6$ and $IH1$ proves the goal.
- Case T-SUB: Hypotheses:

  $$x : \tau_1, \Gamma \vdash e : \tau_3 \quad (H4)$$
  $$x : \tau_1, \Gamma \vdash \tau_3 <: \tau_2 \quad (H5)$$
  $$[v/x]\, \Gamma \vdash [v/x]\, e : [v/x]\, \tau_3 \quad (IH0)$$

  From Lemma A.9, which states that substitution preserves subtyping, we know that:

  $$[v/x]\, \Gamma \vdash [v/x]\, \tau_3 <: [v/x]\, \tau_2 \quad (H6)$$

  Applying T-SUB rule on $IH0$ and $H6$ completes the proof.
- Case T-ABS: e is of form $\lambda y : \tau.\, e_1$. Hypotheses:

  $$\tau_2 = (y : \tau_3) \to \tau_4 \quad (H4)$$
  $$x : \tau_1, \Gamma \vdash \tau_3 \quad (H5)$$
  $$x : \tau_1, \Gamma, y : \tau_3 \vdash e_1 : \tau_4 \quad (H6)$$

  We note that $y \notin freevars(v)$ as $\cdot \vdash v : \tau$ (from $H1$). Therefore, substitution $[v/x]\, e$ is always capture avoiding. Further, we assume that a variable cannot be bound twice in the environment (Γ). This eliminates the case of x and y being equal, leaving us with only case where $x \neq y$:

- SubCase (lambda bound y not same as x): Using Lemma A.7, which asserts that substitution preserves well-formedness of types, we derive the following from $H5$:

$$[v/x]\,\Gamma \vdash [v/x]\,\tau_3 \quad (H7)$$

By instantiating the bound Γ and τ_2 in IH with $(\Gamma,\ y:\tau_3)$ and τ_4, respectively, and using $H6$, we derive the following:

$$[v/x]\,(\Gamma,\ y:\tau_3) \vdash [v/x]\,e_1 \ :\ [v/x]\,\tau_4$$

which, as $y \neq x$, expands to the following:

$$[v/x]\,\Gamma,\ y:[v/x]\,\tau_3 \vdash [v/x]\,e_1 \ :\ [v/x]\,\tau_4 \quad (H8)$$

By the definition of substitution, since $y \neq x$, we have the following:

$$[v/x]\,(\lambda y:\tau.\,e_1) = \lambda(y:[v/x]\,\tau_3.\,[v/x]\,e_1)$$

$$[v/x]\,((y:\tau_3)\to\tau_4) = (y:[v/x]\,\tau_3)\to[v/x]\,\tau_4$$

It remains to show that

$$\Gamma \vdash \lambda(y:[v/z]\,\tau.\,[v/x]\,e_1) \ :\ (y:[v/x]\,\tau_3)\to[v/x]\,\tau_4$$

which follows by applying the rule T-ABS with $H7$ and $H8$.

- Case T-LET: e is of form `let` y `=` e_1 `in` e_2. Hypotheses:

$$\begin{aligned} x:\tau_1,\ \Gamma \vdash e_1 \ :\ \tau_3 \quad & (H4)\\ x:\tau_1,\ \Gamma \vdash \tau_2 \quad & (H5)\\ x:\tau_1,\ \Gamma,\ y:\tau_3 \vdash e_2 \ :\ \tau_2 \quad & (H6)\end{aligned}$$

Since $y \notin freevars(v)$, $[v/x]\,e_2$ avoids variable capture. Since a variable cannot be bound twice in Γ, x and y cannot be equal. This leaves us with one case for $[v/x]\,e$:

- SubCase $y \neq x$: Using Lemma A.7, which asserts well-formedness preservation under substitution, we get the following from $H5$:

$$[v/x]\,\Gamma \vdash [v/x]\,\tau_2 \quad (H7)$$

Instantiating bound Γ and τ_2 in IH appropriately gives us following hypotheses:

$$\begin{aligned}[v/x]\,\Gamma \vdash [v/x]\,e_1 \ :\ [v/x]\,\tau_3 \quad & (H8)\\ [v/x]\,(\Gamma,\ y:\tau_3) \vdash [v/x]\,e_2 \ :\ [v/x]\,\tau_2 \quad & (H9)\end{aligned}$$

Since $x \neq y$, $H9$ is equivalent to:

$$[v/x]\,\Gamma,\ y:[v/x]\,\tau_3 \vdash [v/x]\,e_2 \ :\ [v/x]\,\tau_2 \quad (H10)$$

Applying T-LET rule on $H8$ and $H10$ lets us conclude:

$$[v/x]\,\Gamma \vdash \texttt{let}\ y = [v/x]\,e_1\ \texttt{in}\ [v/x]\,e_2 \ :\ [v/x]\,\tau_2$$

which is what needs to be proven.

- Case T-MATCH: e is of form `match` v' `with Cons` $x'\ y'$ $\Rightarrow$ e_1 `else` e_2, where

$$\begin{aligned} x:\tau_1,\ \Gamma \vdash v' \ :\ \mathsf{intlist} \quad & (H4)\\ x:\tau_1,\ \Gamma \vdash \mathsf{Nil} \ :\ \{\nu:\mathsf{intlist}\,|\,\phi_n\} \quad & (H5)\\ x:\tau_1,\ \Gamma \vdash \mathsf{Cons} \ :\ x':\mathsf{int}\to y':\mathsf{intlist}\to\{\nu:\mathsf{intlist}\,|\,\phi_c\} \quad & (H6)\\ \Gamma_c = x':\mathsf{int}, y':\mathsf{intlist},\ [v'/\nu]\,\phi_c \quad & (H7)\\ \Gamma_n = [v'/\nu]\,\phi_n \quad & (H8)\\ x:\tau_1,\ \Gamma \vdash \tau \quad & (H9)\\ x:\tau_1,\ \Gamma,\ \Gamma_c \vdash e_1 \ :\ \tau \quad & (H10)\\ x:\tau_1,\ \Gamma,\ \Gamma_n \vdash e_2 \ :\ \tau \quad & (H11)\end{aligned}$$

From $H9$, after applying Lemma A.7, we get:

$$[v/x]\,\Gamma \vdash [v/x]\,\tau \quad (H12)$$

We assert that x cannot be equal to x', or y', as it leads to x being bound twice in Γ. Therefore, we are left with one case:

- SubCase $x \neq x'$, and $x \neq y'$: Substitution $[v/x]\,e$ can be expanded to

$$\texttt{match}\ [v/x]\,v'\ \texttt{with Cons}\ x'\ y'\ \Rightarrow\ [v/x]\,e_1\ \texttt{else}\ [v/x]\,e_2$$

Inductive hypotheses (after pushing substitutions into type refinements):

$$\begin{aligned}[v/x]\,\Gamma \vdash [v/x]\,v' \ :\ \mathsf{intlist} \quad & (IH1)\\ [v/x]\,\Gamma \vdash \mathsf{Nil} \ :\ \{\nu:\mathsf{intlist}\,|\,[v/x]\,\phi_n\} \quad & (IH2)\\ [v/x]\,\Gamma \vdash \mathsf{Cons} \ :\ x':\mathsf{int}\to y':\mathsf{intlist}\to\{\nu:\mathsf{intlist}\,|\,[v/x]\,\phi_c\} \quad & (IH3)\\ [v/x]\,\Gamma,\ [v/x]\,\Gamma_c \vdash [v/x]\,e_1 \ :\ [v/x]\,\tau \quad & (IH4)\\ [v/x]\,\Gamma,\ [v/x]\,\Gamma_n \vdash [v/x]\,e_2 \ :\ [v/x]\,\tau \quad & (IH5)\end{aligned}$$

Expansions of $[v/x]\,\Gamma_c$ and $[v/x]\,\Gamma_n$ are given below:

$$\begin{aligned}[v/x]\,\Gamma_c = x':\mathsf{int}, y':\mathsf{intlist},\ [v/x]\,[v'/\nu]\,\phi_c \quad & (H13)\\ [v/x]\,\Gamma_n = [v/x]\,[v'/\nu]\,\phi_n \quad & (H14)\end{aligned}$$

Since Nil and Cons are constants, from the definition of ty (Definition A.1) and T-CONST:

$$\begin{aligned}\cdot \vdash \mathsf{Nil} \ :\ \{\nu:\mathsf{intlist}\,|\,\phi_n\}\\ \cdot \vdash \mathsf{Cons} \ :\ x':\mathsf{int}\to y':\mathsf{intlist}\to\{\nu:\mathsf{intlist}\,|\,\phi_c\}\end{aligned}$$

From Lemma A.11, we know that types of Nil and Cons are well-formed under empty context. By inverting well-formedness derivation of Nil type (via WF-BASE rule in Fig. 3 of the paper), and well-formedness derivation of Cons type (twice through WF-FUN, and once through WF-BASE), we derive:

$$\begin{array}{rl} \cdot \vdash \phi_n & (H15) \\ x' : \mathsf{int},\, y' : \mathsf{intlist} \vdash \phi_c & (H16) \end{array}$$

From $H15$, we conclude that:

$$x \notin freevars(\phi_n) \quad (H17)$$

Similarly, from $H17$, since $x \neq x'$ and $x \neq y'$, we conclude:

$$x \notin freevars(\phi_c) \quad (H18)$$

Using $H17 - 18$, and the definition of substitution operation, we rewrite $H13 - 14$ as:

$$\begin{array}{rl} [v/x]\,\Gamma_c = x' : \mathsf{int}, y' : \mathsf{intlist},\, [[v/x]\,v'/\nu]\,\phi_c & (H19) \\ [v/x]\,\Gamma_n = [[v/x]\,v'/\nu]\,\phi_n & (H20) \end{array}$$

Finally, by applying T-MATCH on $IH1 - 5$ and $H19 - 20$ leads us to conclude that:

$$[v/x]\,\Gamma \vdash \texttt{match}\ [v/x]\,v'\ \texttt{with Cons}\ x'\ y'\ \Rightarrow\ [v/x]\,e_1\ \texttt{else}\ [v/x]\,e_2 : [v/x]\,\tau$$

Which is what needs to be proven.

THEOREM A.13. (**Preservation**) *if* $\cdot \vdash e : \tau$, *and* $e \longrightarrow e'$, *then* $\cdot \vdash e' : \tau$.

Proof by induction on type derivation $\cdot \vdash e : \tau$. Cases:

- Case T-VAR: e is a variable x. By inversion on $\cdot \vdash x : \tau$ we get $x : \tau \in \cdot$, which is absurd. Proof follows from *ex falso quodlibet*.
- Cases T-CONST and T-ABS: Constants and abstractions are values, therefore cannot take a step.
- Case T-APP: e is of form $e_1\ v$, where $\cdot \vdash e_1 : (x : \tau_1) \to \tau_2$, and $\cdot \vdash v : \tau_1$. Therefore, $\cdot \vdash e : [v/x]\,\tau_2$. Inversion on $e_1\ v \longrightarrow e'$. Cases:
  - SubCase E-CO: $e_1 \longrightarrow e_1'$. Therefore, $e_1\ v \longrightarrow e_1'\ v$. By IH, $\cdot \vdash e_1' : (x : \tau_1) \to \tau_2$. Hence, $\cdot \vdash e_1'\ v : [v/x]\,\tau_2$.
  - SubCase E-APP: Inversion on $\cdot \vdash e_1 : (x : \tau_1) \to \tau_2$. Cases:
    - SubSubCase $e_1 = \lambda x : \tau_3.\, e_2$. Therefore, $e_1\ v \longrightarrow [v/x]\,e_2$. By inversion on $\cdot \vdash \lambda x : \tau_3.\, e_2 : \tau_1 \to \tau_2$, we know that $\tau_3 = \tau_1$, and

      $$\cdot, x : \tau_1 \vdash e_2 : \tau_2 \quad (H0)$$

      Now, using $H0$ and the substitution lemma (Lemma A.12), with bound Γ instantiated with $\cdot$, we get $\cdot \vdash [v/x]\,e_2 : [v/x]\,\tau_2$. Hence, $e' = [v/x]\,e_2$ has same type as e.
    - SubSubCase $e_1 = \mathsf{Cons}$, or $e_1 = \mathsf{Cons}\ v_1$, for some v_1: Both $\mathsf{Cons}\ v$, and $\mathsf{Cons}\ v_1\ v$ are values, therefore cannot take a step.
- Case T-SUB: Hypotheses:

  $$\begin{array}{rl} \cdot \vdash e : \tau_1 & (H0) \\ \tau_1 <: \tau & (H1) \end{array}$$

  Inductive hypothesis:

  $$If \quad e \longrightarrow e', \quad then \quad \cdot \vdash e' : \tau_1 \quad (IH)$$

  Proof follows from IH and $H1$.
- Case T-LET: e is of form $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$, where:

  $$\begin{array}{rl} \cdot \vdash e_1 : \tau_1 & (H0) \\ \cdot, x : \tau_1 \vdash e_2 : \tau & (H1) \\ \cdot \vdash \tau & (H2) \end{array}$$

  Inductive Hypothesis:

  $$If \quad e_1 \longrightarrow e_1', \quad then \quad \cdot \vdash e_1' : \tau_1 \quad (IH)$$

  By inversion on $e \longrightarrow e'$, we have two cases:
  - SubCase E-CO: $e_1 \longrightarrow e_1'$ and $e \longrightarrow \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2$. Applying IH, we know that:

    $$\cdot \vdash e_1' : \tau_1 \quad (H3)$$

    Applying T-LET using hypotheses $H3$, $H1$ and $H2$, we conclude that $\cdot \vdash \texttt{let}\ x = e_1'\ \texttt{in}\ e_2 : \tau$.
  - SubCase E-LET: $e_1 \longrightarrow v$ and $e \longrightarrow [v/x]\,e_2$. Using $H0$ and $H1$, and applying the substitution lemma (Lemma A.12), we derive the following:

    $$\cdot \vdash [v/x]\,e_2 : [v/x]\,\tau \quad (H4)$$

    From hypothesis $H2$, we know that τ is well-formed under empty context. Rewriting $H4$ with $[v/x]\,\tau = \tau$ lets us conclude:

    $$\cdot \vdash [v/x]\,e_2 : \tau \quad (H4)$$

$\boxed{\cdot \vdash \phi^F : \tau^F, \quad \cdot \vdash \phi^L : \tau^F}$

QF-PROP-SORT

$$\frac{}{\cdot \vdash \phi^F : \mathit{bool}}$$

Q-PROP-SORT

$$\frac{\cdot \vdash \phi^L : \tau^F}{\cdot \vdash \forall (k : T^F).\, \phi^L : T^F \to \mathit{bool}}$$

Figure 14: Nominal Type System for MSFOL propositions

- Case T-MATCH: e is of form $\texttt{match}\ v\ \texttt{with}\ \mathsf{Cons}\ x\ y \Rightarrow e_1\ \texttt{else}\ e_2$, where:

$$\begin{array}{rl}
\cdot \vdash v : \mathsf{intlist} & (H0) \\
\cdot \vdash \mathsf{Nil} : \{\nu : \mathsf{intlist} \mid \phi_n\} & (H1) \\
\cdot \vdash \mathsf{Cons} : x : \mathsf{int} \to y : \mathsf{intlist} \to \{\nu : \mathsf{intlist} \mid \phi_c\} & (H2) \\
\Gamma_c = x : \mathsf{int}, y : \mathsf{intlist}, [v/\nu]\,\phi_c & (H4) \\
\Gamma_n = [v/\nu]\,\phi_n & (H5) \\
\cdot \vdash \tau & (H6) \\
\Gamma_c \vdash e_1 : \tau & (H7) \\
\Gamma_n \vdash e_2 : \tau & (H8)
\end{array}$$

  By inversion on $e \longrightarrow e'$, we have two cases:
  - SubCase E-MCONS: Hypotheses:

$$\begin{array}{rl}
v = \mathsf{Cons}\ v_1\ v_2 & (H9) \\
e' = [v_2/y]\,[v_1/x]\,e_1 & (H10)
\end{array}$$

    By inversion on $H0$, we derive:

$$\begin{array}{rl}
\cdot \vdash v_1 : \mathsf{int} & (H11) \\
\cdot \vdash v_2 : \mathsf{intlist} & (H12)
\end{array}$$

    Using $H4$ and $H7$, and twice applying the substitution lemma (Lemma A.12), we derive the following:

$$[v_2/y]\,[v_1/x]\,[v/\nu]\,\phi_c \vdash [v_2/y]\,[v_1/x]\,e_1 : [v_2/y]\,[v_1/x]\,\tau \quad (H13)$$

    From Definition A.1, which asserts the validity of type refinements of Cons and Nil, we get:

$$\Delta,\ x : [\![\mathsf{int}]\!],\ y : [\![\mathsf{intlist}]\!] \models [\![[\mathsf{Cons}\ x\ y/\nu]\,\phi_c]\!] \quad (H14)$$

    Using $H11 - 12$, and instantiating x, and y with v_1 and v_2, respectively:

$$\Delta \models [\![[v_2/y]\,[v_1/x]\,[v/\nu]\,\phi_c]\!] \quad (H15)$$

    Now, applying the cut elimination lemma (Lemma A.3) on $H13$ and $H15$, we derive:

$$\cdot \vdash [v_2/y]\,[v_1/x]\,e_1 : [v_2/y]\,[v_1/x]\,\tau \quad (H16)$$

    From $H6$, we know that type refinement of τ is a closed term; therefore, $[v_2/y]\,[v_1/x]\,\tau = \tau$. Using this fact to rewrite $H16$, we conclude that:

$$\cdot \vdash [v_2/y]\,[v_1/x]\,e_1 : \tau$$

  - SubCase E-MNIL: $v = \mathsf{Nil}$. $e' = e_2$. From $H5$ and $H8$:

$$[v/\nu]\,\phi_n \vdash e_2 : \tau \quad (H9)$$

    As in the case of E-MCONS, using Definition A.1 and cut elimination lemma (Lemma A.3), we conclude that:

$$\cdot \vdash e_2 : \tau$$

THEOREM A.14. (**Type Safety**) *if* $\cdot \vdash e : \tau$*, then either* e *is a value, or* $e \longrightarrow e'$ *and* $\cdot \vdash e' : \tau$.

Proof follows directly from progress (Theorem A.1), and preservation (Theorem A.13) properties. ∎

A.3 MSFOL Semantics of Type Refinements

We now prove that the exercise of ascribing MSFOL semantics to type refinements is complete. The metatheory relies on certain definitions given below:

Definition *Nominal Type System for MSFOL lanugage* We define a nominal type system that assigns MSFOL sorts (τ^F) to MSFOL propositions. The type system is nominal in the sense that the sorts assigned by the type system need not necessarily relate to the actual sort of a proposition under MSFOL. Sorts are always assigned under an empty sort environment. The type system is defined in Fig 14.

Definition *Join of Nominal Types* We define join operation on nominal types recursively as:

$$\begin{array}{lcl}
\mathit{bool} \bowtie \mathit{bool} & = & \mathit{bool} \\
\mathit{bool} \bowtie T^F \to \tau^F & = & T^F \to \tau^F \\
T^F \to \tau^F \bowtie \mathit{bool} & = & T^F \to \tau^F \\
T^F \to \tau_1^F \bowtie \tau_2^F & = & T^F \to (\tau_1^F \bowtie \tau_2^F)
\end{array}$$

Definition *Substitution Operation on Propositions.* Capture avoiding substitution, where variable capture is with respect to the variable bound by quantifiers, is assumed on MSFOL propositions.

Definition In our meta-theory, we use $\odot$ to denote any boolean connective such that, for any two MSFOL formulas, ϕ_1^L and ϕ_2^L, $\phi_1^L \odot \phi_2^L$ is an MSFOL formula.

LEMMA A.15. (η_{wrap} **is type safe**) *forall* ϕ^F*, there exists and MSFOL proposition* ϕ^L *such that* $\eta_{wrap}(\phi^F, \tau^F) = \phi^L$*, and* $\cdot \vdash \phi^L : \tau^F$

Proof By induction on τ^F. Cases:

- Case $\tau^F = bool$: $\eta_{wrap}(\phi^F, bool) = \phi^F$, which is an MSFOL proposition. From QF-PROP-SORT, we also know that $\cdot \vdash \phi^F : bool$.
- Case $\tau^F = T^F \to \tau_2^F$: Eta-wrap expansion is:$\eta_{wrap}(\phi^F, T^F \to \tau^F) = \forall(k : T^F).\, \eta_{wrap}(\phi^F\, k, \tau_2^F)$. Inductive hypothesis tells us that there exists an MSFOL proposition ϕ_k^L, such that:
$$\begin{array}{ll} \phi_k^L = \eta_{wrap}(\phi^F\, k, \tau_2^F) & (IH0) \\ \cdot \vdash \phi_k^L : \tau_2^F & (IH1) \end{array}$$
Now, $\phi^L = \forall(k : T^F).\, \phi_k^L$ is an MSFOL proposition. Further, applying Q-PROP-SORT on $IH1$ lets us conclude:
$$\cdot \vdash \phi^L : T^F \to \tau_2^F$$

LEMMA A.16. (**MSFOL sort of a relation**) *forall R, if* $\cdot \vdash R :: \tau_R$*, then there exists an MSFOL proposition* ϕ_R^L *such that* $[\![R]\!] = \phi_R^L$*, and* $\cdot \vdash \phi_R^L : [\![\tau_R]\!]$.

Proof We proceed by case analysis on R.

- Case $R = R_{id}$: From S-REL-ID, we know that:
$$\cdot \vdash R_{id} :: \mathsf{int} \to \{\mathsf{int}\} \quad (H0)$$
Also, from Fig. 4:
$$\begin{array}{ll} [\![\mathsf{int} \to \{\mathsf{int}\}]\!] = [\![\mathsf{int}]\!] \to [\![\mathsf{int}]\!] \to bool & (H1) \\ [\![R_{id}]\!] = \forall(j : [\![\mathsf{int}]\!]).\, \forall(k : [\![\mathsf{int}]\!]).j = k & (H2) \end{array}$$
From the definition of MSFOL encoding of types, we know that $[\![\mathsf{int}]\!] = \mathcal{F}(\mathsf{int})$. Using this to rewrite $H2$, we deduce that:
$$\phi_R^L = \forall(j : \mathcal{F}(\mathsf{int})).\, \forall(k : \mathcal{F}(\mathsf{int})).j = k \quad (H3)$$
Applying Q-PROP-SORT twice, we also know that:
$$\cdot \vdash \forall(j : [\![\mathsf{int}]\!]).\, \forall(k : [\![\mathsf{int}]\!]).j = k \;:\; [\![\mathsf{int}]\!] \to [\![\mathsf{int}]\!] \to bool \quad (H4)$$
Therefore:
$$\cdot \vdash \forall(j : \mathcal{F}(\mathsf{int})).\, \forall(k : \mathcal{F}(\mathsf{int})).j = k :\; [\![\mathsf{int}]\!] \to [\![\mathsf{int}]\!] \to bool \quad (H5)$$
From $H3$, and rewriting $H5$ with $H1$ gives us proof.
- R is any relation that is not R_{id}: From Fig. 4:
$$[\![R]\!] = \eta_{wrap}(R, [\![\Gamma_R(R)]\!]) \quad (H0)$$
Since $\cdot \vdash R :: \tau_R$, from the definition of ordered map Γ_R, we know that $\Gamma_R(R) = \tau_R$. Rewriting $H0$:
$$[\![R]\!] = \eta_{wrap}(R, \tau_R) \quad (H1)$$
Now, applying Lemma A.15 gives us proof.

LEMMA A.17. (**Substitution Preserves Nominal Typing**) *forall* ϕ^L*, x, and y, if* $\cdot \vdash \phi^L : \tau^F$*, then* $\cdot \vdash [y/x]\, \phi^L : \tau^F$

Proof trivial, as substitution operation substitutes one variable for another in an MSFOL formula, and all variables have type *bool* under nominal type system. ∎

LEMMA A.18. ($\gamma_\sqcup$ **correctness**) *forall* ϕ_1^L*,* ϕ_2^L*, and* τ^F*, if* $\cdot \vdash \phi_1^L : \tau^F$*, and* $\cdot \vdash \phi_2^L : \tau^F$*, then there exists an MSFOL proposition* ϕ^L *such that* $\gamma_\sqcup(\phi_1^L, \odot, \phi_2^L) = \phi^L$*, and* $\cdot \vdash \phi^L : \tau^F$.

Proof by simultaneous induction (i.e., double induction followed by elimination of absurd cases) on nominal typing derivations $\cdot \vdash \phi_1^L : \tau^F$, and $\cdot \vdash \phi_2^L : \tau^F$. Cases:

- Case Q-PROP-SORT : τ^F is of form $T^F \to \tau_2^F$. Propositions ϕ_1^L, and ϕ_2^L are of form $\forall(k : T^F).\, \phi_{11}^L$ and $\forall(k : T^F).\, \phi_{21}^L$, respectively, such that:
$$\begin{array}{ll} \cdot \vdash \phi_{11}^L : \tau_2^F & (H0) \\ \cdot \vdash \phi_{12}^L : \tau_2^F & (H1) \end{array}$$
From inductive hypotheses, we know that there exists an MSFOL prop ϕ_3^L such that:
$$\begin{array}{ll} \gamma_\sqcup(\phi_{11}^L, \odot, \phi_{12}^L) = \phi_3^L & (IH0) \\ \cdot \vdash \phi_3^L : \tau_2 & (IH1) \end{array}$$
From the definition of $\gamma_\sqcup$, we have that:
$$\gamma_\sqcup(\forall(k : T^F).\, \phi_{11}^L, \odot, \forall(k : T^F).\, \phi_{12}^L) = \forall(k : T^F).\, \gamma_\sqcup(\phi_{11}^L, \odot, \phi_{12}^L) \quad (H2)$$
Rewriting $H2$ using $IH0$, we have:
$$\gamma_\sqcup(\forall(k : T^F).\, \phi_{11}^L, \odot, \forall(k : T^F).\, \phi_{12}^L) = \forall(k : T^F).\, \phi_3^L \quad (H2)$$
Hence, ϕ^L is $\forall(k : T^F).\, \phi_3^L$. It remains to prove that $\cdot \vdash \forall(k : T^F).\, \phi_3^L : \tau^F$, which can be done by applying Q-PROP-SORT on $IH1$.

- Case QF-PROP-SORT: $\tau^F = \mathit{bool}$. Inversion on $\cdot \vdash \phi_1^L : \mathit{bool}$, and $\cdot \vdash \phi_2^L : \mathit{bool}$ lets us infer that ϕ_1^L and ϕ_2^L are quantifier-free propositions ϕ_1^F and ϕ_2^F, respectively. From the definition of $\gamma_\sqcup$, we know that:
$$\gamma_\sqcup(\phi_1^F, \odot, \phi_2^F) = \phi_1^F \odot \phi_2^F$$
Proof is obtained by observing that $\phi_1^F \odot \phi_2^F$ is a quantifier-free MSFOL formula, which, by QF-PROP-SORT rule has type bool.

LEMMA A.19. ($\gamma_\bowtie$ **correctness**) *forall ϕ_1^L, ϕ_2^L, τ_1^F, and τ_2^F, if $\cdot \vdash \phi_1^L : \tau_1^F$, and $\cdot \vdash \phi_2^L : \tau_2^F$, then there exists an MSFOL proposition ϕ^L such that $\gamma_\bowtie(\phi_1^L, \odot, \phi_2^L) = \phi^L$, and $\cdot \vdash \phi^L : \tau_1^F \bowtie \tau_2^F$.*

Proof by simultaneous inductions on nominal typing derivations $\cdot \vdash \phi_1^L : \tau_1^F$, and $\cdot \vdash \phi_2^L : \tau_2^F$. We will have four cases, one for each case of join. Proof proceeds similar to the proof of Lemma A.18. ∎

LEMMA A.20. (**Translation for relational expressions**) *Forall Γ, r, and θ, if $\Gamma \vdash r :: \{\theta\}$, then there exists a ϕ^L such that $[\![r]\!] = \phi^L$, and $\cdot \vdash \phi^L : [\![\{\theta\}]\!]$.*

Proof By induction on the sort derivation $\Gamma \vdash r :: \{\theta\}$. Cases:

- Case S-UNION : $r = r_1 \cup r_2$, for some r_1, r_2. Hypotheses:
$$\begin{array}{ll} \Gamma \vdash r_1 :: \{\theta\} & (H0) \\ \Gamma \vdash r_2 :: \{\theta\} & (H1) \end{array}$$
From inductive hypotheses, we know that there exist two propositions, ϕ_1^L and ϕ_2^L, such that:
$$\begin{array}{ll} [\![r_1]\!] = \phi_1^L & (IH0) \\ [\![r_2]\!] = \phi_2^L & (IH0) \\ \cdot \vdash \phi_1^L : [\![\theta]\!] & (IH2) \\ \cdot \vdash \phi_2^L : [\![\theta]\!] & (IH3) \end{array}$$
We know that $[\![r_1 \cup r_2]\!] = \gamma_\sqcup([\![r_1]\!], \vee, [\![r_2]\!])$. Therefore, the goal is to prove that there exists a ϕ^L, such that:
$$\begin{array}{l} \gamma_\sqcup([\![r_1]\!], \vee, [\![r_2]\!]) = \phi^L \\ \cdot \vdash \phi^L : [\![\theta]\!] \end{array}$$
Applying Lemma A.18 using $IH2 - 3$ proves the goal.
- Case S-CROSS: Similar to S-UNION case. We make use of Lemma A.19 to prove the goal.
- Case S-APP : r is of form $R\,v$, for some relation R, and λ_R value v. Hypotheses:
$$\begin{array}{ll} \cdot \vdash R :: T :\to \{\theta\} & (H0) \\ \|\Gamma\| \Vdash v : T & (H1) \end{array}$$
From definition of MSFOL encoding for colon-arrow types:
$$[\![T :\to \{\theta\}]\!] = [\![T]\!] \to [\![\theta]\!] \quad (H2)$$
Since $T \in \{\mathsf{int}, \mathsf{intlist}\}$, we have the following cases for v:
  - SubCase v is a variable x: $r = R(x)$. From the definition of MSFOL encoding:
$$[\![R(x)]\!] = Inst([\![R]\!], x) \quad (H3)$$
From Lemma A.16, we know that $[\![R]\!]$ is an MSFOL formula ϕ_R^L such that $\cdot \vdash \phi_R^L : [\![T :\to \{\theta\}]\!]$. Rewriting using $H2$:
$$\cdot \vdash \phi_R^L : [\![T]\!] \to [\![\theta]\!] \quad (H4)$$
By inversion on $H4$, we know that ϕ_R^L is of form $\forall(k : [\![T]\!]).\, \phi_k^L$, where
$$\cdot \vdash \phi_k^L : [\![\theta]\!] \quad (H5)$$
Rewriting $H3$:
$$[\![R(x)]\!] = Inst(\forall(k : [\![T]\!]).\, \phi_k^L, x) \quad (H6)$$
From the definition of $Inst$, $H6$ reduces to:
$$[\![R(x)]\!] = [x/k]\, \phi_k^L \quad (H7)$$
Therefore, $[\![R(x)]\!]$ is an MSFOL formula. Further, Applying substitution lemma of nominal typing (Lemma **??**) on $H5$, we also have that
$$\cdot \vdash [x/k]\, \phi_k^L : [\![\theta]\!]$$
This concludes the proof for current SubCase.
  - SubCase v is Nil: r is $R(\mathsf{Nil})$. Hypotheses:
$$\cdot \vdash R(\mathsf{Nil}) :: \{\theta\} \quad (H0)$$
By inversion on $H0$:
$$\cdot \vdash R :: \mathsf{intlist} :\to \{\theta\} \quad (H1)$$
By inversion on $H1$, for some relational expressions r_1 and r_2:
$$\begin{array}{ll} R \triangleq \langle \mathsf{Nil} \Rightarrow r_1 \mid \mathsf{Cons}\, x\, y \Rightarrow r_2 \rangle & (H2) \\ \cdot \vdash r_1 :: \{\theta\} & (H3) \end{array}$$

Using $H3$, the inductive hypothesis tells us that there exists a ϕ^L, such that

$$\begin{array}{ll} [\![r_1]\!] = \phi^L & (H4) \\ \cdot \vdash \phi^L :: [\![\theta]\!] & (H5) \end{array}$$

From the definition of MSFOL encoding:

$$[\![R(\mathsf{Nil})]\!] = [\![\Sigma_R(R)(\mathsf{Nil})]\!]$$

Recall that Σ_R maps relation names to their definitions, and we treat the relation definition as a map from constructor patterns to relational expressions. Therefore, $\Sigma_R(R)(\mathsf{Nil}) = r_1$, and $[\![\Sigma_R(R)(\mathsf{Nil})]\!] = [\![r_1]\!]$. Consequently, proof follows directly from $H4 - 5$.

- Case v is $\mathsf{Cons}\, v_1\, v_2$, for some λ_R values v_1 and v_2. From the type of Cons under simple type system, we know that:

$$\begin{array}{l} \|\Gamma\| \Vdash v_1 : \mathsf{int} \\ \|\Gamma\| \Vdash v_2 : \mathsf{intlist} \end{array}$$

As a corollary of Lemma A.4, from previous two hypotheses, we have:

$$\begin{array}{ll} \|\Gamma\| \vdash v_1 : \mathsf{int} & (H0) \\ \|\Gamma\| \vdash v_2 : \mathsf{intlist} & (H1) \end{array}$$

Further, we have the following hypothesis:

$$\Gamma \vdash R(\mathsf{Cons}\, v_1\, v_2) :: \{\theta\} \quad (H2)$$

By inversion on $H0$:

$$\Gamma \vdash R :: \mathsf{intlist} :\to \{\theta\} \quad (H3)$$

By inversion on $H1$, for some relational expressions r_1 and r_2:

$$\begin{array}{ll} R \triangleq \langle \mathsf{Nil} \Rightarrow r_1 \mid \mathsf{Cons}\, x\, y \Rightarrow r_2 \rangle & (H4) \\ \cdot, x : \mathsf{int}, y : \mathsf{intlist} \vdash r_2 :: \{\theta\} & (H5) \end{array}$$

From $H5$, and $H0 - 1$:

$$\cdot \vdash [v_2/y]\,[v_1/x]\, r_2 :: \{\theta\} \quad (H5)$$

From the definition of MSFOL encoding:

$$[\![R(\mathsf{Cons}\, v_1\, v_2)]\!] = [\![\Sigma_R(R)(\mathsf{Cons}\, v_1\, v_2)]\!] \quad (H6)$$

Using the definition of Σ_R, followed by desugaring:

$$[\![R(\mathsf{Cons}\, v_1\, v_2)]\!] = [v_2/y]\,[v_1/x]\, r_2 \quad (H7)$$

From $H5$, which asserts that $[v_2/y]\,[v_1/x]\, r_2$ is well-sorted under empty environment not containing type bindings for Cons and Nil, we know that arguments to relations in r_2 are smaller than $\mathsf{Cons}\, v_1\, v_2$. The goal can now be proved by induction on the size of relation arguments.

THEOREM A.21. (**Completeness of MSFOL semantics**) *Forall ϕ, Γ, if $\Gamma \vdash \phi$, then there exists an MSFOL proposition ϕ^L such that $[\![\phi]\!] = \phi^L$.*

Proof by induction on $\Gamma \vdash \phi$. Cases:

- Case WF-REF: $\phi = \phi_1 \wedge \phi_2$, or $\phi = \phi_1 \vee \phi_2$. From inductive hypothesis, we have that $[\![\phi_1]\!]$ and $[\![\phi_2]\!]$ are both MSFOL formulas. Since conjunctions and disjunctions of MSFOL formulas are MSFOL formulas themselves, proof follows.
- Case WF-PRED: ϕ is of form $r_1 = r_2$, or $r_1 \subseteq r_2$, for some relational expressions r_1 and r_2. Hypotheses:

$$\begin{array}{ll} \Gamma \vdash r_1 :: \{\theta\} & (H0) \\ \Gamma \vdash r_2 :: \{\theta\} & (H1) \end{array}$$

where, θ is a tuple sort. From lemma A.20, we know that there exist two MSFOL propositions ϕ_1^L and ϕ_2^L, such that:

$$\begin{array}{ll} \cdot \vdash \phi_1^L : [\![\{\theta\}]\!] & (H2) \\ \cdot \vdash \phi_2^L : [\![\{\theta\}]\!] & (H3) \end{array}$$

Now, since:

$$\begin{array}{ll} [\![r_1 = r_2]\!] = \gamma_\sqcup([\![r_1]\!], \Leftrightarrow, [\![r_2]\!]) & (H4) \\ [\![r_1 \subseteq r_2]\!] = \gamma_\sqcup([\![r_1]\!], \Rightarrow, [\![r_2]\!]) & (H4) \end{array}$$

applying Lemma A.18, using $H2 - 3$ gives us the proof.