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# [12], 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 under-specify 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} -> {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, involutiveness 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 $l = [x_1, x_2, x_3]$, its closure $R_{ob}^*$ would be:

\[
\{(x_1, y) \mid y \in [x_1, x_2, x_3] \} \cup \{(x_2, y) \mid y \in [x_2, x_3] \} \cup \{(x_3, y) \mid y \in [x_3] \}
\]
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 \(\alpha\)-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

\[\{(1,\langle x_1, x_2 \rangle),\langle 1, x_1, x_3 \rangle,\langle 1, x_2, x_3 \rangle\}\]
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:

\[ R_{id} (x) = \{ \{(x)\} \} \]
\[ R_{dup} (x) = \{ \{(x, x)\} \} \]
\[ R_{nestEq} (x) = \{ \{(x)\} - \{\{x\}\} \} \]
\[ R_{eq} (x) = \{ \{\} \} - \{\{\}\} \]

\[ R_{eq} \] is the identity relation, \( R_{dup} \) is a relation that associates a value with a pair that duplicates that value, \( R_{nestEq} \) 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 \( R_{eq} \) 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:

\[ \text{relation } R_{hd} (x :: xs) = \{ \{x\}\} \]

For a concrete list \( l \), \( R_{hd}(l) \) 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 \( xs \), \( (l :: xs, x) \in R_{hd} \), and (2) there does not exist \( x' \) such that \( x' \neq x \) and \( (x :: xs, x') \in R_{hd} \). 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 \( \text{rev} \) reverses the ordering of its input requires specifying a membership relation \( R_{mem} \) that relates a list to every element in \( l \) (regardless of \( l \)'s size). This relation would allow us to define an ordering property such as \( \text{occurs-before} \ or \ \text{occurs-after} \) on precisely those elements that comprise \( \text{rev} \)'s input and output lists. A recursive definition of \( R_{mem} \) looks like:

\[ R_{mem} (x :: xs) = \{ \{x\}\} \cup R_{mem} (xs) \]

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 \( (l, v) \in R \), then \( (l, v) \in R^* \)
- If \( l = x :: xs \) and \( (xs, v) \in R \) then \( (l, v) \in R^* \)

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.

2.2 Parametric Relations

Consider how we might specify a \( \text{zip} \) function over lists, with the following type:

\[ \text{zip} : 'a \ \text{list} \to 'b \ \text{list} \to ('a * 'b) \ \text{list} \]

Any correct implementation of \( \text{zip} \) 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 \( \text{zip} \)'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 \( \text{zip} \), 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:

\[ \text{relation } (R_{hd} R) (x :: xs) = R (x) \]
\[ \{ R_{hd} R \} = \emptyset \]

\[ \text{relation } (R_{mem} R) = (R_{hd} R)^* \]

\[ \text{relation } R_{fst} (x, y) = \{ \{x\}\} \]

\[ \text{relation } R_{snd} (x, y) = \{ \{y\}\} \]

\( \text{zip} \) can now be assigned the following type that faithfully captures the membership relation between its input lists and its output:

\[ \text{zip} : l_1 \to l_2 \to \{ \nu : ((R_{mem} R_{fst}) \nu) = ((R_{mem} R_{hd}) \nu_1) \]
\[ \land ((R_{mem} R_{snd}) \nu) = ((R_{mem} R_{hd}) \nu_2) \} \]

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

\[ \text{relation } (R_{ob} R) (x :: xs) = R (x) \times ((R_{mem} R) xs) \]

Using \( R_{ob} \), the \( \text{dup} \) function described in the previous section can now be specified thus:

\[ \text{dup} : l \to \{ \nu : ((R_{ob} R_{fst})^* \nu) = ((R_{ob} R_{hd})^* \nu_1) \]
\[ \land ((R_{ob} R_{snd})^* \nu) = ((R_{ob} R_{hd})^* \nu_2) \} \]

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

\[ ^1 \text{We drop ML types from dependent type specifications when obvious from context.} \]
relations in a type. To illustrate why such parameterization is useful, consider the following signature for \( \text{foldl} \):

\[
\langle R_{\text{on}} \rangle \text{ foldl} : \{ l : \\text{a list} \} \rightarrow \{ b : \text{b} \} \rightarrow \\
(\{ f : [x : \text{a}] \rightarrow \{ \text{acc : b} \} \rightarrow \\
\{ z : \text{b} | R_{\text{on}}(z) = \{ \{ x \} \} \cup R_{\text{on}}(\text{acc}) \}) \rightarrow \\
(\nu \mid R_{\text{on}}(\nu) = R_{\text{mem}}(1) \cup R_{\text{on}}(b))
\]

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

\[
\text{datatype } \text{a tree} = \text{Leaf} | \text{Tree of } \text{a} \ast (\text{a Tree}) \ast (\text{a Tree})
\]

\[
R_{\text{tdb}} \text{ Leaf } = 0 \\
R_{\text{tdb}} \text{ (Tree (x, t1, t2)) } = (\{ x \})
\]

\[
R_{\text{mem}} = R_{\text{tdb}}
\]

\[
\text{makeTree} : \{ l : \text{a list} \} \rightarrow \\
\{ \nu : \text{a tree } R_{\text{mem}}(\nu) = R_{\text{mem}}(1) \}
\]

\[
\text{val makeTree } = \text{fn} \ l \Rightarrow \\
\text{foldl} (R_{\text{tdb}}) \text{ 1 Leaf treeInsert}
\]

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

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

An alternative type for \( \text{foldl} \) 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 \( (\text{b}) \) has some notion of order captured by a relation such that the result of \( \text{foldl} \)’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 \( (x \text{ and acc}) \). 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 \( \text{foldl} \) that tries to match the abstract order \( (R_{\text{on}}) \) on the result type \( (\text{b}) \) to an occurs-after order on the input list is shown below. For brevity, we avoid reproducing membership invariants from the type of \( \text{foldl} \) from the previous example, using ellipses in their place:

\[
\langle R_{\text{on}}.R_{\text{ho}} \rangle \text{ foldl} : \{ l : \text{a list} \} \rightarrow \{ b : \text{b} \} \rightarrow \\
(\{ f : [x : \text{a}] \rightarrow \{ \text{acc : b} \} \rightarrow \\
\{ z : \text{b} | R_{\text{ho}}(z) = \{ \{ x \} \} \times R_{\text{on}}(\text{acc}) \cup \\
R_{\text{on}}(\text{acc}) \cup R_{\text{ho}}(b)) \cup \\
\}) \rightarrow \\
(\nu \mid R_{\text{ho}}(\nu) = R_{\text{on}}(1) \cup R_{\text{ho}}(b)) \cup \\
R_{\text{mem}}(1) \times R_{\text{on}}(b) . \cup ...\}
\]

An implementation of \( \text{rev} \) that uses \( \text{foldl} \) is given below:

\[
\text{rev} : \{ l : \text{a list} \} \rightarrow \{ \nu : \text{a list } R_{\text{on}}(\nu) = R_{\text{on}}(l) \}
\]

\[
\text{val Cons } = \text{fn} x \Rightarrow \text{fn} xs \Rightarrow \text{x :: xs}
\]

\[
\text{val rev } = \text{fn} l \Rightarrow \text{foldl} (R_{\text{mem}}, R_{\text{on}}) \text{ 1 [] Cons}
\]

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

### 3. Core language

#### 3.1 Syntax

We formalize our ideas using a core calculus \( (\lambda I) \) shown in Fig. 1, an A-normalized extension of the simply-typed lambda calculus. The language supports a primitive type \( (\text{int}) \), a recursive data type \( (\text{intlist}) \), 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, \( (\lambda I) \) is only provided with two constructors, \( \text{Nil} \) and \( \text{Cons} \) 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 \( (\phi) \) in \( \lambda I \) are assertions over relational expressions \( (r) \) which themselves are themselves expressions, which are themselves types, constitute the syntactic class of expressions in our specification language. We refer to the types of relational expressions as \( \text{sorts} \), in order to distinguish them from \( \lambda I \) types. We write \( r :: s \)

\^4\ Proofs for all lemmas and theorems in the paper are also provided in the appendix
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-matching 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-matching or inductive) definition \( \delta \).

A structural relation maps a value to a set of tuples \( \{\theta\} \). We use \( \cdot \rightarrow \) to distinguish such maps from the mapping expressed by dependent function types. For example, the notation:

\[
R_{\text{rel}} : \text{intlist} \rightarrow \{\text{int} \times \text{int}\}
\]

indicates that the sort of relation \( R_{\text{rel}} \) is a map from integer lists to pairs. As reflected by the syntactic class of relation sorts \( (\tau_i) \), the domain of a \( \lambda R \) relation is either intlist or int. For the purposes of the formalization, we assume the existence of a single primitive relation \( R_{\text{rel}} \), whose sort is \( \text{intlist} \rightarrow \{\text{int}\} \) 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 \( \Gamma \), defined as follows:

\[
\Gamma ::= \cdot | \Gamma, x : \tau | \Gamma, \phi
\]

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 \( \Gamma \).

Structural relations are sort checked under an empty type environment. The rules 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

\[
\begin{align*}
\phi^F & \ ::= \ v \mid \phi^E \mid \phi^F \land \phi^F \mid \phi^F \lor \phi^F \mid \phi^F \to \phi^F \\
\phi^L & \ ::= \ w : \mathcal{T} \land \phi^L \\
\tau^F & \ ::= \ bool \mid \mathcal{T}^F \to \tau^F
\end{align*}
\]

\[\phi \land \phi = \{\phi_1 \land \phi_2\}
\]

\[\phi \lor \phi = \{\phi_1 \lor \phi_2\}
\]

Semantics of Relational Expressions \[
\begin{align*}
\mathcal{R}(\mathsf{Cons}
\end{align*}
\]

Auxiliary Definitions

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) = \{\mathsf{Nil} \Rightarrow r_1, \mathsf{Cons}
\]

Semantics of Type Refinements \[
\begin{align*}
\phi^E & \ ::= \ \phi^L \mid \phi^E \land \phi^E \mid \phi^E \lor \phi^E \mid \phi^E \to \phi^E
\]

has the required simple (non-dependent) type. The rule makes use of a simple typing judgment \((\Gamma \vdash t:T)\) under a refinement erased \(\Gamma\) (denoted \([\Gamma]r\)) 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 \((\tau)\) sets its type refinement to \(\mathsf{true}\), effectively erasing the refinement to yield a simple type. For function types, erasure is defined recursively:

\[\|\{v : \mathcal{T} \mid \phi\}\| = T \quad \|\{x : r_1 \to r_2\}\| = \|r_1\| \to \|r_2\|
\]

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,

\[\|\Gamma\|, x : \tau, \phi \| = \|\Gamma\|, \phi \|
\]

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

\[
\forall i \in \mathbb{Z}, \ ty(i) = int
\]

The type refinements of \(\phi_0\) and \(\phi\) in the \(\mathcal{T}\)-MATCH rule are conjunctive aggregations of \(\Sigma\) cases of \(\mathcal{T}\)-MATCH (resp.) of all structural relation definitions. To help us precisely define \(\phi_0\) and \(\phi\), we assume the presence of (a) a globally-defined finite map \(\Sigma\) that maps relation names to their pattern-match definitions, and (b) a finite ordered map \(\Gamma\) that maps relation names to their sorts. We implicitly parameterize our typing judgment over \(\Sigma\) (i.e., our \(\Gamma\) is actually \(\Gamma(\Sigma, R, \Gamma)\)). Inductive relations defined using the closure operator are assumed to be unfolded to pattern-match definitions before being bound in \(\Sigma:\)

\[\begin{align*}
R & \triangleq \Sigma_R(R_2) = \{\mathsf{Nil} \Rightarrow r_1, \mathsf{Cons}
\]

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

\[\begin{align*}
\Sigma & = [R \mapsto \{\mathsf{Nil} \Rightarrow \mathsf{Nil}(0) \mid \mathsf{Cons}
\]

The type of \(\mathsf{Nil}\) and \(\mathsf{Cons}\) in such case is as following:

\[\begin{align*}
\mathsf{Nil} & = \{v : \mathsf{intlist} \mid R_1(v) = \mathsf{Nil}(0)\}
\]

The \(\mathcal{T}\)-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 \(\mathsf{Cons}\) branch is also extended with the types of matched pattern variables \((x\) and \(y\). The branch condition for the \(\mathsf{Cons}\) (alternatively, \(\mathsf{Nil}\)) case is obtained by substituting the test value \((v)\) for the bound variable \((v)\) in the type refinement of \(\mathsf{Cons}\) (\(\mathsf{Nil}\)). Intuitively, the branch condition of \(\mathsf{Cons}\) (alternatively, \(\mathsf{Nil}\)) captures the fact that the value \(v\) was obtained by applying the constructor \(\mathsf{Nil}\) (\(\mathsf{Cons}\)); therefore, it should satisfy the invariant of \(\mathsf{Cons}\) (\(\mathsf{Nil}\)). For instance, consider the match expression:

\[\begin{align*}
\text{match } x \text{ with } \mathsf{Cons}
\]

where \(\mathsf{Cons}\) has type^3

\[\begin{align*}
\text{Cons} : x : \mathsf{int} \to y: \mathsf{intlist} \to \{v: \mathsf{intlist} \mid R_{\mathsf{mem}}(v) = R_{\mathsf{aux}}(x) \cup R_{\mathsf{mem}}(y)\}
\]

^3 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:

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

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 $\text{SUBT-BASE}$. The rule refers to the map $\Gamma_R$ that provides sorts for relations occurring free in type refinements. Intuitively, the rule asserts dependent type $\tau_1$ to be a subtype of $\tau_2$, if:

- Their base types match, and,
- Given a logical system $L$, and interpretations of type environment $(\Gamma, \nu : T)$ and the type refinement $\phi_1$ (of $\tau_1$) 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 $\lambda_R$'s type system is defined with respect to a reduction relation ($\rightarrow$) that specifies the language's operational semantics:

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

### 3.3 Semantics of the Specification Language

The semantics of our specification language is defined via a translation from well-formed 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 $\Sigma_R$ map defined previously. For perspicuity, we introduce the following syntactic sugar:

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

Further, we also assume a finite ordered map $\Gamma_R$ 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 $\lambda_R$'s specification language. The semantics is operational in the sense that it describes an algorithm to compute assertions in $\lambda_R$ type refinements to formulas in MSFOL. Our semantics are parameterized over an auxiliary function ($\mathcal{F}$) that maps $\lambda_R$ 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 $\lambda_R$ 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 $\eta_{\text{wrap}}$ 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

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

That is, $\Gamma_R$ maps $R$ to intlist $\rightarrow \{\text{int}\}$. Assume that $[\text{int}] = A_0$ and $[\text{intlist}] = A_1$. Now,

$$
[R] = \eta_{\text{wrap}}(R, \Gamma_R(R))
$$

$$
\eta_{\text{wrap}}(R, [\text{intlist} \rightarrow \{\text{int}\}])
$$

$$
\eta_{\text{wrap}}(R, [\text{intlist}] \rightarrow [\{\text{int}\}])
$$

$$
\eta_{\text{wrap}}(R, A_1 \rightarrow A_0 \rightarrow \text{bool})
$$

$$
\forall (k : A_1). \eta_{\text{wrap}}(R k, A_0 \rightarrow \text{bool})
$$

$$
\forall (k : A_1). \forall (j : A_0). \eta_{\text{wrap}}(R k j, \text{bool})
$$

$$
\forall (k : A_1). \forall (j : A_0). R k j
$$

Auxiliary function $\text{Inst}$ 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:

$$
\forall \exists \phi_1 \cup \forall \exists \phi_2 = \forall \exists (\phi_1 \lor \phi_2)
$$

$$
\forall \exists \phi_1 \times \forall \exists \phi_2 = \forall \exists (\phi_1 \land \phi_2)
$$

Our semantics use syntactic rewrite functions $\gamma_\text{int}$ and $\gamma_\text{intlist}$, 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 $\lambda_R$ assertion involving the occurs-before relation:

$$
(R_{\text{int}} :: \text{intlist} \rightarrow \{\text{int}\}),
$$

and membership relation

$$
(R_{\text{mem}} :: \text{intlist} \rightarrow \{\text{int} \times \text{int}\})
$$

for integer lists:

$$
R_{\text{int}}(\text{intlist}) = R_{id}(\text{intlist}) \times R_{\text{mem}}(\text{intlist})
$$

The steps of series 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 $\lambda_0$, and intlist to sort $\lambda_1$.

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

$$
[\Gamma, x : \{v : T \mid \phi\}] = [\Gamma] \Rightarrow x : [T] \Rightarrow [x/v][\phi]
$$

$$
[\Gamma, \phi] = [\Gamma] \Rightarrow [\phi]
$$

$$
true
$$

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

$$
[\Gamma_R, R :: \tau_R] = [\Gamma_R] \cup \{ R : [\tau_R] \}
$$

$$
\{}\}
$$

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

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

### 3.4 Decidability of $\lambda_R$ Type Checking

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

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

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.
Consequently, decidability of type checking in $\lambda_R$ 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 Bernays-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 $\lambda_R$ a practical exercise.

**Theorem 3.3.** (*Decidability*) Type checking in $\lambda_R$ 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 ($\lambda_R$) 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 $\lambda_{\forall R}$ 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 ($\sigma_R$), akin to type schemes ($\sigma$) of the term language. For example, the non-parametric head relation ($R_{\text{head}}$) from Section 2, when defined over a polymorphic ‘a list will have sort scheme, ‘a a list $\rightarrow$ ‘a. The specification language also contains an expression ($\exists T$) 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 ($R_{\text{mem}} R$), defined in Section 2, can be instantiated with the non-parametric head relation, $R_{\text{head}}$, 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.

The definition ($\Sigma_R$) of a parametric relation is a tuple containing its name ($R$), the name of its relational parameter ($R_{\text{rel}}$), its sort scheme ($\sigma_R$), 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 $R_{\text{head}}$ relation over ‘a list reproduced from Section 2:

$$\text{relation} (R_{\text{head}} R) \ (x:x::xs) = R(x)$$

$R_{\text{head}}$ 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 $R_{\text{head}}$, it is essential that we let the relational parameter have an unrestricted co-domain. Therefore, we let tuple-sort variables ($\theta$) be used in tuple sorts ($\theta$). 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 $R$ 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 ($\delta$). An example of a parametric dependent type is the type of foldl from Section 2. Another example is the type of map shown below:

$$\text{type} \quad \text{map} \quad \forall \text{a}. \text{a list} \rightarrow \text{a list}$$

A note on notation: We use ($R_{\text{mem}}$ $R$) and ($R_{\text{head}}$ $R$) to denote parametric membership and head relations, resp. We continue to use $R_{\text{mem}}$ and $R_{\text{head}}$ to denote their non-parametric versions. We use qualifiers “parametric” and “non-parametric” to disambiguate.
Semantics of Bind Equations \([\psi]\)

\[
[R_2 = \lambda (x : T_1). \, \text{bind} \, (R_1 \, (x), \lambda (k : T_2). \, r)] \quad = \quad \forall (x : [T_1]). \, \gamma \circ ([R_1 \, (x)], \forall (k : [T_2]), \gamma, \, [R_2 \, (x)]) \quad \land \quad \forall (x : [T_1]). \, \gamma \circ ([R_1 \, (x)], \forall (k : [T_2]), \gamma, \, [R_2 \, (x)])
\]

\[
\gamma \circ (\forall (k : T_1). \, \phi^k_1, \, \forall (k : T_1). \, \gamma \circ (\forall (j : T_2). \, \phi^j_2, \, \nu^F)) \quad = \quad \forall (k : T_1). \, \gamma \circ (\forall (j : T_2). \, \phi^j_2 \land \phi^k_1 \Rightarrow \nu^F \, \exists)
\]

Figure 7: Semantics of bind equations for parametric relations in \(\lambda_{\nu R}\)

\((\cdot \, R_1, \, \cdot \, R_2) \, \mapsto :\]

\[
1 \to (f : x \to \{ \nu \mid \, 'R_2 (\nu) = 'R_1 (x) \}) \to \{ \nu \mid ((R_{ob} \, 'R_2)^\nu) = ((R_{ob} \, 'R_1)^\nu \, 1) \}
\]

4.2 Sort and Type Checking

Rules to check sorts of relational expressions and well-formedness of type refinements (\(\Phi\)) in \(\lambda_{\nu R}\) are straightforward extensions of similar rules for \(\lambda_R\) 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 \(\lambda_{\nu R}\).

**Transformer Expression.** A transformer expression \((F_R)\) is a \(\lambda_R\) 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 expression is a map (under \(\cdot \to \cdot\)) from tuple-sort \((\theta_R)\) to a set sort \((\{\theta_2\})\). An example of a transformer expression of sort \(\, 'a \to \{ 'a \times 'a \}\) 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:

\[
\forall t_1, t_2, \{ t_1 \} : \to (t_1 : \to \{ t_2 \}) : \to \{ t_2 \}
\]

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 \( 'a \, \text{list}\), the bind expression that binds \(R_{mem}(l)\) with a reflexive transformer is as following:

\[
\text{bind} \, (R_{mem}(l), \, \lambda x. \, R_{id}(x) \times R_{id}(x))
\]

\[
r \quad ::= \quad R(x) \mid r \times r
\]

\[
F_R \quad ::= \quad \lambda (x : T), \, r \quad \text{transformer}
\]

\[
e_b \quad ::= \quad \text{bind} \, (R(x), \, F_R) \quad \text{bind expression}
\]

\[
E_b \quad ::= \quad \lambda (x : T_1). \, \text{bind} \, (R(x), \, F_R) \quad \text{bind abstraction}
\]

\[
\psi \quad ::= \quad R = E_b \quad \text{bind equation}
\]

\[
\Sigma_R \quad ::= \quad \lambda R. \, E_b \quad \text{bind definition}
\]

Figure 8: Bind Syntax

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 \(R_{dup}\):

\[
(R_{mem} \, \text{R}_{dup}) (1) = \text{bind} \, (R_{mem}(1), \, \lambda x. \, R_{id}(x) \times R_{id}(x))
\]

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 \((R_{mem} \, \text{R}_{dup})\) in terms of its ground relation \(R_{mem}\). Indeed, a parametric \(R_{mem}\) relation (call it \(R_{mem}^p\)) can be defined equivalently in terms of its non-parametric variant as:

\[
R_{mem}^p \equiv \lambda R. \, \lambda l. \, \text{bind} \, (R_{mem}(1), \, \lambda x. \, R(x))
\]

We refer to the above definition as the bind definition of \(R_{mem}\). Every well-sorted parametric structural relation definition in \(\lambda_{\nu R}\) 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 \(\lambda_{\nu R}\) type in the tuple sort translates to application of \(R_{id}\). For instance, consider a hypothetical parametric relation \(R_e\) with the following sort:

\[
R_e \quad ::= \quad \forall t. \, (\text{int} : \to \{ \text{int} \}) : \to \{ \text{int} \times \ast \times \ast \}
\]

Its transformer expression is:

\[
\lambda (x, \, y, \, z). \, R_{id}(x) \times R(y) \times R(z)
\]

Renaming the ground relation of \(R_a\) as \(R_{a'}\), we derive the following bind definition of \(R_{a'}\):

\[
\lambda R. \, \lambda l. \, \text{bind} \, (R_{a'}(1), \, \lambda (x, \, y, \, z). \, R_{id}(x) \times R(y) \times R(z))
\]
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 $\lambda_{R_B}$, is a map from $\text{'a list}$ to a set of tuples. Therefore, an instantiated parametric relation can be treated as a new non-parametric relation that is defined using $\text{bind}$. For example, $(R_{\text{mem}} R_{\text{dup}})$ can be treated as a new non-parametric relation $R_I$, defined in terms of $\text{bind}$:

$$R_I = \lambda x. \text{bind}(R_{\text{mem}}(1), \lambda x. R_{\text{id}}(x) \times R_{\text{id}}(x))$$

By rigorously defining the semantics of $\text{bind}$ 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 $(R_{\text{mem}} R_{\text{dup}})$ lets us prove the following implication, which could potentially arise during subtype checking:

$$((R_{\text{mem}} R_{\text{dup}}) I_1) = ((R_{\text{mem}} R_{\text{dup}}) I_2) \Rightarrow R_{\text{mem}}(I_1) = R_{\text{mem}}(I_2)$$

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

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

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

4.4 Decidability of Type Checking

Type refinements ($\Phi$) in $\lambda_{R_B}$ can be elaborated to a conjunction of $\text{bind}$ equations representing semantics of instantiated relations, and a $\lambda_R$ type refinement ($\phi$). Consequently, we have the following result:

**Theorem 4.1. (Decidability)** Type checking in $\lambda_{R_B}$ is decidable.

**Proof** Follows from the decidability proof of EPR logic, to which $\text{bind}$ equations are compiled to, and decidability result (Theorem 3.3) for $\lambda_R$.

5. Implementation

We have implemented our specification language and verification procedure as an extended type-checking pass (called $\text{ATALYST}$) 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 $\text{ATALYST}$ 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 prefix form. The framework only needs to be seeded with the single primitive relation $R_{\text{id}}$. For example, $R_{\text{notEq}}$ can be defined as the following primitive relation:

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

Similarly, $R_{\text{eq}}$ can be defined as:

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

Both $R_{\text{notEq}}$ and $R_{\text{eq}}$ 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 $R_{\text{notEq}} c$ can be used to instantiate the parametric $R_{\text{mem}}$ 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:

$$\text{id} : x \rightarrow \{ v | R_{\text{id}}(v) = R_{\text{id}}(x) \}$$

The type refinement used here is an uninituitive way of expressing the simple fact that $\text{id}$ 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.
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 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 ($R_{\text{torder}}$ in Fig. 9b) is defined in terms of tree membership relation ($R_{\text{mem}}$), 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 ($R_{\text{torder}}$), on a red-black tree, obtained by closing the $R_{\text{torder}}$ 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 f’s inductive total-order relation:

\[
\text{inOrder}: t \rightarrow \{ l \mid \text{inOrder}(t) = \text{inOrder}(l) \}
\]

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:

\[
(R \text{ exists}) : l \rightarrow f : x \rightarrow \{ \nu \mid \nu = \text{true} \Leftrightarrow \forall \nu \in \text{true} \}
\]

The interpretation of the type is as follows: Let there be a relation $R$ such that f returns true if and only if relation $R$ (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:

\[
(R \text{ filter}) : l \rightarrow f : x \rightarrow \{ \nu \mid \nu = \text{false} \Leftrightarrow \lambda \nu \in \text{false} \}
\]

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 $R(x)$ to be equal to $\emptyset$ when $\nu = \text{false}$. 

(a) exists
(b) filter
(c) contains

Figure 10: Examples
**ML Program**

1 | datatype exp = Var of string
2 | | App of exp*exp
3 | | Abs of string*exp
4 | fun freeVars e = case e of
5 | Var id => [id]
6 | | App (e1,e2) =>
7 | | Abs (id,e') => filter (RNeq id)
8 | | concat [freeVars e1, freeVars e2]
9 | (fv fv => not (fv = id)) (freeVars e')
10 | fun alphaConvert e = case e of
11 | Abs (id,e') => if id = id' then e1 else e2
12 | App (e1,e2) => subst e1 id e2
13 | Var id => [id]
14 | fun subst e1 id e2 = case e2 of
15 | Var id' => if id = id' then e1 else e2
16 | App (e21,e22) =>
17 | subst e1 id e21
18 | subst e1 id e22
19 | else Abs(id', subst e1 id e2')
20 | |

**Relational Specification**

relation $R_{\text{free}}$, $(\text{Var } x) = \{x\}$

$R_{\text{free}}$: $(\text{App } (e1,e2)) = R_{\text{free}}(e1) \cup R_{\text{free}}(e2)$

$R_{\text{free}}$: $(\text{Abs } (id,e)) = R_{\text{free}}(e) - \{\{id\}\}$

createNewName : $fvs \rightarrow id \rightarrow \{\nu | not (\nu = id) \wedge not (\{\nu\} \subseteq R_{\text{mem}}(fvs))\}$

freeVars : $e \rightarrow \{l \mid R_{\text{mem}}(l) \subseteq R_{\text{free}}(e)\}$

alphaConvert : $a \rightarrow (\text{ex} \mid R_{\text{free}}(\text{ex}) = R_{\text{free}}(e))$

subst : $e1 \rightarrow id \rightarrow e2 \rightarrow \{\text{ex} \mid \text{if } (((\text{id})) \subseteq R_{\text{free}}(e2)) \text{ then } R_{\text{free}}(\text{ex}) = (R_{\text{free}}(e2) - \{\text{id}\}) \cup R_{\text{free}}(e1) \text{ else } R_{\text{free}}(\text{ex}) = R_{\text{free}}(e2)\}$

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:

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

the relational dependent type of isStr is:

$$\text{isStr : x} \rightarrow \{\nu | R_{\text{eq}_{\text{str}}} (\nu) \neq \emptyset\}$$

This clearly suggests that the relational specification of exists has to be instantiated with $(R_{\text{eq}_{\text{str}}}, \text{str})$. 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 resulting type of hasStr is:

$$\text{hasStr : } \{\nu | \nu=\text{true} \Rightarrow ((R_{\text{mem}} R_{\text{eq}_{\text{str}}}) (\text{str})) \neq \emptyset\}$$

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:

$$\{\nu = \text{true} \Rightarrow ((R_{\text{mem}} R_{\text{eq}_{\text{str}}}) (\text{str})) \neq \emptyset\}$$

the implementation of contains type-checks against the type:

$$\text{l} \rightarrow \text{str} \rightarrow \{\nu | \nu=\text{true} \Rightarrow (R_{\text{eq}_{\text{str}}} (\text{str})) \subseteq R_{\text{mem}}(\text{l})\}$$

6.1 $\alpha$-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 $\alpha$-conversion operations on lambda calculus terms are quite tricky to define correctly [227].

Some of the behaviors exhibited by incorrect implementations include (a) $\alpha$-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 $\alpha$ conversion is given in the bottom-half of Fig. 11.\footnote{We introduce some syntactic sugar in defining type refinements. For example, the branch expression (if $\phi$ then $\phi_1$ else $\phi_2$) in a type refinement translates to ($($\phi \land \phi_1$) \lor ($\neg \phi \land \phi_2$)).} Note that one need not expect notions of capture-avoidance, or other such intricacies, to write down the specification, which is given in terms of a new structural relation $R_{\text{free}}$ 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.
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^* [2-7]. 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 foldr 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


https://www.haskell.org/ghc/.

http://ocaml.org/.

Step-indexed syntactic logical relations for recursive and quantified types.

Relational reasoning in a nominal semantics for storage.
In TLCA, 2005.

Reading, writing and relations: Towards extensional semantics for effect analyses.
In APLAS, 2006.

Relational semantics for effect-based program transformations: Higher-order store.
In PPDP, 2009.

Relational inductive shape analysis.
In POPL, 2008.

Logical step-indexed logical relations.

Generic top-down discrimination for sorting and partitioning in linear time*.
J. Funct. Program.

A relational approach to interprocedural shape analysis.

Type-based data structure verification.
In PLDI, 2009.

Hasochism: The pleasure and pain of dependently typed haskell programming.
In Haskell Symposium, 2013.

Faking it simulating dependent types in haskell.

Shape analysis in the absence of pointers and structure.
In VMCAI, 2010.

Purely Functional Data Structures.

Local type inference.

Deciding effectively propositional logic with equality.

Liquid types.
In PLDI, 2008.

Decision procedures for algebraic data types with abstractions.
In POPL, 2010.

Abstract refinement types.
In ESOP, 2013.

Binders unbound.
In ICFP, 2011.

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 $\lambda_R$ We define simply typed $\lambda_R$ 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 $\Gamma$ to denote simple type environment against dependent type environment in the dependent type rules of $\lambda_R$ that maps variables to their (unrefined) types. Recall that the domain of a $\lambda_R$ 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 $\Gamma$ relies on well-sortedness of relation applications under same $\Gamma$, 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 $pty$ 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:

\[
\forall i \in \mathbb{Z}, \ pty(i) = \text{int}
\]

\[
pty(\text{Nil}) = \text{intlist}
\]

\[
pty(\text{Cons}) = \text{int} \rightarrow \text{intlist} \rightarrow \text{intlist}
\]

Definition (VC Prelude) Sec. 3.3 of the paper introduces $\Gamma_R$ as an ordered map from structural relations to their colon-arrow sorts. MSFOL translation of $\Gamma_R$ (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 $\lambda_R$, we call the set as VC prelude.

Formally, $\Delta$ is the smallest set of MSFOL formulas such that for all $R, \Gamma_R$ such that $\forall R, \Gamma' \vdash R :: T :: \{\theta\}$,

\[
R : [T ::= \{\theta\}] \in \Delta
\]

Remark (Entailment) In our proofs, we use $\models$ to denote semantic entailment in first-order logic. We write $\phi_1 \vdash \phi_2$ to denote that $\phi_1$ semantically entails $\phi_2$. 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 $\Gamma$) of hypotheses. We adopt few standard theorems of first-order deductive systems to our setting:

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

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

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

\[
[v/z] \lambda(x : \tau). e = \lambda(x : [v/z] \tau). e \quad \text{if } z = x
\]

\[
[v/z] \lambda(x : \tau) . e = [v/z] \text{alphaConvert}(\lambda(x : \tau) . e) \quad \text{if } x \in \text{freevars}(v)
\]

\[
[v/z] \lambda(x : \tau). e = \lambda(x : [v/z] \tau). [v/z] e \quad \text{otherwise}
\]

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:

\[
[v/z] \{ \nu : T | \phi \} = \{ \nu : [v/z] T | \phi \}
\]

\[
[v/z] (x : \tau_1) \rightarrow \tau_2 = (x : [v/z] \tau_1) \rightarrow \tau_2 \quad \text{if } z = x
\]

\[
[v/z] (x : \tau_1) \rightarrow \tau_2 = [v/z] \text{alphaConvert}(x : \tau_1) \rightarrow \tau_2 \quad \text{if } x \in \text{freevars}(v)
\]

\[
[v/z] (x : \tau_1) \rightarrow \tau_2 = (x : [v/z] \tau_1) \rightarrow [v/z] \tau_2 \quad \text{otherwise}
\]

We assume a function $\text{alphaConvert}$ 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:

\[
[v/z] R(v') = R([v/z] v')
\]

Definition (Basic Axioms) Basic axioms assert sorts, and validity of type refinements of constants in MSFOL. This accounts for the $\text{T-CONST}$ rule of $\lambda_R$ type system, which seeds the typing judgment with assumptions on types of constants. The axioms are stated below:

- Type of Integers: For all integer constants $c$, $\models c : [\text{int}]$
- Type of Nil: $\models \text{Nil} : [\text{intlist}]$
- Type of Cons: $x : [\text{int}], y : [\text{intlist}] \models \text{Cons } x y : [\text{intlist}]$
- Validity of $\phi_a : \Delta \models \text{Nil} / v \phi_a$
- Validity of $\phi_c : \Delta, x : [\text{int}], y : [\text{intlist}] \models \text{Cons } x y / v \phi_c$
### Evaluation Rules

<table>
<thead>
<tr>
<th>ST-VAR</th>
<th>ST-APP</th>
<th>ST-ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (x : T) \in \Gamma )</td>
<td>( \Gamma \vdash e : T_1 \rightarrow T_2 )</td>
<td>( \Gamma, x : T_1 \vdash e : T_2 )</td>
</tr>
<tr>
<td>( \Gamma \vdash x : T )</td>
<td>( \Gamma \vdash v : T_2 )</td>
<td>( \Gamma \vdash \lambda(x : T_1). e : \Gamma_1 \rightarrow T_2 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ST-CONST</th>
<th>ST-LET</th>
<th>ST-MATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash e : \mu g(c) )</td>
<td>( \Gamma \vdash e_1 : T_1 ) ( \Gamma, x : T_1 \vdash e_2 : T_2 )</td>
<td>( \Gamma \vdash v : \text{intlist} ) ( \Gamma \vdash e_1 : T ) ( \Gamma \vdash e_2 : T )</td>
</tr>
<tr>
<td>( \Gamma \vdash x = e_1 ) ( \in e_2 : T_2 )</td>
<td>( \Gamma \vdash \text{let} \ x = e_1 \in e_2 : T_2 )</td>
<td>( \Gamma \vdash \text{match} \ v ) ( \text{with} \ \text{Cons} \ x \ y \Rightarrow e_1 ) ( \text{else} \ e_2 : T )</td>
</tr>
</tbody>
</table>

#### Figure 12: Type Rules for simply typed \( \lambda_R \)

### Evaluation Context

\[
E ::= \bullet | \bullet \nu | \text{let} \ x = \bullet \in e
\]

#### Figure 13: Operational Semantics of Core Calculus (\( \lambda_R \))

### A.2 Type Safety

We now prove\(^\text{10}\) the type safety of \( \lambda_R \)'s type system by proving its progress and preservation properties. Call-by-value operational semantics of \( \lambda_R \) 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 \( e \rightarrow e' \).

**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 \( \cdot \vdash e_1 : (x : \tau_1) \rightarrow \tau_2 \), and \( \cdot \vdash v : \tau_1 \). By IH, either \( e_1 \) can take a step or \( e_1 \) is a value.
  - If \( e_1 \rightarrow e_2 \). As per our definition of evaluation contexts (Fig. 13), \( e_1 \rightarrow e_2 \); So, \( e' = e_2 v \).
  - If \( e_1 \) is a value \( v_1 \). By inversion on \( v_1 \), followed by eliminating cases using the assumption \( \cdot \vdash v_1 : (x : \tau_1) \rightarrow \tau_2 \), we are left with following cases:
    - If \( v_1 \) is of form \( \lambda x : \tau. e_3 \), for some \( e_3 \): Consequently, \( e = \lambda x : \tau. e_3 \), which reduces by E-\( \beta \) to \( \lambda x : \tau. e_3 \).
    - If \( v_1 \) is Cons: \( Cons \ v_1 \) is a value.
  - If \( v_1 \) is Cons \( v_2 \), for some \( v_2 \): Cons \( v_2 \) \( v \) is a value.
- **Case T-ABS:** \( e = \lambda x : \tau. e_1 \), which is a value.
- **Case T-LET:** \( e = \text{let} \ x = e_1 \in e_2 \), where \( \cdot \vdash e_1 : \tau_1 \), and \( \cdot \vdash x : \tau_1 \vdash e_2 : \tau_2 \). Similar to T-APP, we have two cases:
  - If \( e_1 \rightarrow e' \), in which case \( e \rightarrow e' \) (as per our definition of evaluation contexts).
  - If \( e_1 \) is a value \( v_1 \), in which case \( e \) reduces by rule E-\( \text{LET} \) to \( [v_1/x][v_2/y]e_2 \).
- **Case T-MATCH:** \( e = \text{match} \ v \) \( \text{with} \ \text{Cons} \ x \ y \Rightarrow e_1 \) \( \text{else} \ e_2 \), where \( \cdot \vdash v : \{ \nu : \text{intlist} | \phi \} \). By inversion on \( v \) and eliminating abstract cases, we are left with two cases:
  - If \( v = \text{Cons} v_1 v_2 \), in which case \( e \) reduces by E-MCON to \( [v_1/x][v_2/y]e_1 \).
  - If \( v = \text{Nil} \), in which case \( e \) reduces by M-ENIL to \( e_2 \).

**Lemma A.2. (Context Invariance for Well-Formedness)** If \( \Gamma \vdash \tau \), then for all \( \Gamma' \) 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:

\[
\text{for all } r, \text{ if } \Gamma \vdash r :: \{ \theta \}, \text{ then } \Gamma' \vdash r :: \{ \theta \}.
\]

---

\(^{10}\) 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 \( H[0 \rightarrow 9]+ \) and \( IH[0 \rightarrow 9]+ \) 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 $\Gamma \vdash \tau :: \{ \theta \}$. Cases S-UNION and S-CROSS follow directly from inductive hypotheses. The only interesting case is S-APP, where $\tau = R(v)$, for some $R$ and $v$. The proof is by inversion on $\Gamma \vdash R(v) :: \{ \theta \}$. Hypotheses:

\[
\begin{align*}
\Gamma \vdash R :: T \rightarrow \{ \theta \} & \quad (H1) \\
\Gamma \vdash v :: T & \quad (H2)
\end{align*}
\]

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)** For all $x, e, \tau, \phi, \text{and } \Gamma$, if $\Delta \vdash [\phi]$, and $\phi, \Gamma \vdash e : \tau$, then $\Gamma \vdash e : \tau$

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

- Case T-VAR: $e$ is a variable $y$. Inversion of $\phi, \Gamma \vdash y : \tau$ produces $y : \tau \in \phi, \Gamma$. 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:

\[
\begin{align*}
\tau &= \tau_2 & (H0) \\
\Delta &\vdash [\phi] & (H1) \\
\phi, \Gamma \vdash \tau_1 : \tau_2 & \quad (H3)
\end{align*}
\]

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:

\[
\begin{align*}
\tau_1 &= \{ \nu : T | \phi_1 \} & \quad (H4) \\
\tau_2 &= \{ \nu : T | \phi_2 \} & \quad (H5) \\
\phi, \Gamma \vdash \nu : T & \quad (H6) \\
\phi, \Gamma \vdash \nu : T & \quad (H7) \\
\Delta &\vdash [\phi] \Rightarrow [\Gamma] \Rightarrow [\phi_1] \Rightarrow [\phi_2] & \quad (H8)
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash \{ \nu : T | \phi_1 \} & \quad (H8) \\
\Gamma \vdash \{ \nu : T | \phi_2 \} & \quad (H9)
\end{align*}
\]

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

\[
\begin{align*}
\Delta, [\phi], [\Gamma], [\phi_1] &\vdash [\phi_2] & \quad (H10)
\end{align*}
\]

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

\[
\begin{align*}
\Delta, [\Gamma], [\phi_1] &\vdash [\phi_2] & \quad (H11)
\end{align*}
\]

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

\[
\begin{align*}
\Delta &\vdash [\Gamma] \Rightarrow [\phi_1] \Rightarrow [\phi_2] & \quad (H12)
\end{align*}
\]

Finally, applying SUBT-BASE rule to $H8, H9$, and $H12$ leads us to conclude that $\Gamma \vdash \tau_1 : \tau_2$

- Case SUBT-ARR: One can derive proof for this case by simply applying SUBT-ARR 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. ($\lambda_T$ Type System is Conservative) for all $\Gamma$, 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 $\|\text{int}\| = \text{int}$, and $ty(c) = pty(c)$, gives us proof.
  - $c$ is Nil: Observing that $ty(\text{Nil}) = \{ \nu : \text{intlist} | \phi_0 \}$, and $\|ty(\text{Nil})\| = \text{intlist} = pty(\text{Nil})$, gives us proof.
  - $c$ is Cons: Proof similar to the Nil case.
- Case T-APP: $e = e_1 v$, where:

\[
\begin{align*}
\Gamma \vdash e_1 :: (x : \tau_1) & \rightarrow \tau_2 & \quad (H0) \\
\Gamma \vdash v_1 :: \tau_1 & \quad (H1)
\end{align*}
\]

Inductive hypotheses (after simplifying $\|\Gamma\| \vdash e_1 :: \|\tau_1\| \rightarrow \|\tau_2\|$):

\[
\begin{align*}
\|\Gamma\| \vdash e_1 :: \|\tau_1\| & \rightarrow \|\tau_2\| & \quad (IH0) \\
\|\Gamma\| \vdash v_1 :: \|\tau_1\| & \quad (IH1)
\end{align*}
\]

Applying S-APP on $IH0 - 1$ gives proof.
• Case T-SUB: Hypotheses:

\[ \Gamma \vdash e : \tau_1 \quad (H0) \]
\[ \tau_1 \vdash \tau \quad (H1) \]

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:

\[ \tau = (x : \tau_1) \rightarrow \tau_2 \quad (H4) \]
\[ \Gamma, x : \tau_1 \vdash e_1 : \tau_2 \quad (H5) \]
\[ \|\tau\| = \|\tau_1\| \rightarrow \|\tau_2\| \quad (H6) \]

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

\[ \|\Gamma\|, x : \|\tau_1\| \vdash e_1 : \tau_2 \quad (IH0) \]

Applying ST-ABS on IH0 gives proof.

• T-LET: Proof closely resembles that for T-ABS case.

• T-MATCH: \(e = \text{match } v \text{ with } \text{Cons } x y \Rightarrow e_1 \text{ else } e_2\), and \(e = \text{match } v \text{ with } \text{Cons } x y \Rightarrow e_1 \text{ else } e_2\). Hypotheses:

\[ \Gamma \vdash v : \text{intlist} \quad (H0) \]
\[ \Gamma \vdash \text{Nil} : \{v : \text{intlist} | \phi_n\} \quad (H1) \]
\[ \Gamma \vdash \text{Cons} : x : \text{int} \rightarrow y : \text{intlist} \rightarrow \{v : \text{intlist} | \phi_c\} \quad (H2) \]
\[ \Gamma_v = x : \text{int}, y : \text{intlist}, \phi \quad (H4) \]
\[ \Gamma_n = \phi_n \quad (H5) \]
\[ \Gamma \vdash \tau \quad (H6) \]
\[ \Gamma, \Gamma_v \vdash e_1 : \tau \quad (H7) \]
\[ \Gamma, \Gamma_n \vdash e_2 : \tau \quad (H8) \]

Inductive hypotheses:

\[ \|\Gamma\| \vdash v : \text{intlist} \quad (IH0) \]
\[ \|\Gamma\|, \|\Gamma_v\| \vdash e_1 : \|\tau\| \quad (IH1) \]
\[ \|\Gamma\|, \|\Gamma_n\| \vdash e_2 : \|\tau\| \quad (IH2) \]

Where,

\[ \|\Gamma_v\| = x : \text{int}, y : \text{intlist} \]
\[ \|\Gamma_n\| = . \]

Applying ST-MATCH on IH0 – 2 produces proof.

We assert the substitution lemma for simply typed \(\lambda_R\), 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 \(\Gamma, x, e, T_1,\) 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 \(\Gamma\), if \(x : \tau_1, \Gamma \vdash r :: \{\emptyset\}\) and \(\cdot \vdash v : \tau_1\), then \([v/x] \Gamma \vdash [v/x] r :: \{\emptyset\}\).

**Proof** by induction on the derivation of \(x : \tau_1, \Gamma \vdash r :: \{\emptyset\}\). 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:

\[ \vdash v : \|\tau_1\| \quad (H1) \]

Applying the substitution lemma for simple type judgment of \(\lambda_R\) (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 \(\|\Gamma\| \vdash v : \|\Gamma\|\). 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 \(\Gamma, if \ x : \tau_1, \Gamma \vdash \tau \) and \(\cdot \vdash v : \tau_1\), then \([v/x] \Gamma \vdash [v/x] \tau\).

**Proof** Well-formedness judgment of \(\lambda_R\) 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 \(\lambda_R\) Value)** forall \(v, if \cdot \vdash v : \{v : T | \phi \}, then \|v : [T] \) and \(\Delta \vdash [v/v] \|\phi\|.\)
**Proof** By case analysis on \( v \). Cases:

- Case \( v \) is a variable \( x \). Inversion of \( \vdash x : \{ \nu : T \ | \phi \} \) leads to absurdity, \textit{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 \( \vdash c : \{ \nu : T \ | \phi \} \) leads to two relevant cases:
  - SubCase T-CONST: \( T = \text{int} \) and \( \phi = \text{true} \). We know that \( \models \text{true} \) is trivially valid. From the Definition A.1, we also have that \( \models c : \text{[int]} \).
  - SubCase T-SUB: Hypotheses:

    \[
    \begin{align*}
    \vdash c : \tau_1 & \quad \text{(H0)} \\
    \vdash \tau_1 <: \{ \nu : T \ | \phi \} & \quad \text{(H1)}
    \end{align*}
    \]

    By inversion on \( H1 \), we have:

    \[
    \begin{align*}
    \tau_1 &= \{ \nu : T \ | \phi_1 \} & \quad \text{(H2)} \\
    \vdash c : \{ \nu : T \ | \phi_1 \} & \quad \text{(H3)} \\
    \vdash \{ \nu : T \ | \phi_1 \} & \quad \text{(H4)} \\
    \vdash \nu : [\tau] & \Rightarrow [\phi_1] & \Rightarrow [\phi] & \quad \text{(H6)} \\
    \end{align*}
    \]

    Inductive hypotheses:

    \[
    \begin{align*}
    \models c : [\tau] & \quad \text{(IH0)} \\
    \Delta & \models [c/\nu] [\phi_1] & \quad \text{(IH1)}
    \end{align*}
    \]

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

    \[
    \Delta \models \forall \nu. (\nu : [\tau] \Rightarrow [\phi_1] \Rightarrow [\phi])
    \]

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

    \[
    \Delta \models c : [\tau] \Rightarrow [c/\nu] [\phi_1] \Rightarrow [c/\nu] [\phi] & \quad \text{(H8)}
    \]

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

    \[
    \models [c/\nu] [\phi] & \quad \text{(H9)}
    \]

    Proof follows from \( IH0 \) and \( H9 \).

- Case \( v \) is Nil: From Definition A.1, we have that:

    \[
    T = \text{intlist} & \quad \text{(H0)} \\
    \models \text{Nil} : \text{[intlist]} & \quad \text{(H1)} \\
    \Delta & \models [\text{Nil}/\nu/\nu] [\phi] & \quad \text{(H2)}
    \]

    Proof follows straightforwardly from hypotheses.

- Case \( v \) is Cons \( v_1 \ v_2 \), such that

    \[
    \vdash \text{Cons} v_1 v_2 : \{ \nu : T \ | \phi \} & \quad \text{(H0)}
    \]

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

    \[
    \vdash \text{Cons} v_1 v_2 : T & \quad \text{(H1)}
    \]

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

    \[
    \vdash \text{Cons} v_1 v_2 : \{ \nu : \text{intlist} \ | \phi \} & \quad \text{(H2)}
    \]

    From \( H1 \), and Definition A.1, we prove that

    \[
    \models \text{Cons} v_1 v_2 : \text{[intlist]}
    \]

- Cases when \( v \) is Cons, or \( v \) is 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)** For all \( \Gamma \), if \( \vdash x : \tau, \Gamma \vdash \tau_1 <: \tau_2 \) and \( \vdash \nu : \tau, then [\nu/x] \, \Gamma \vdash [\nu/x] \, \tau_1 <: [\nu/x] \, \tau_2 \).

**Proof** By induction on the subtype judgment. Cases:

- Case \( \text{SUBT-ARROW} \): Proof is a straightforward application of \( \text{SUBT-ARROW} \) on inductive hypotheses.
- Case \( \text{SUBT-BASE} \): \( \tau_1 \) is of form \( \{ \nu : T \ | \phi_1 \} \), and \( \tau_2 \) is of form \( \{ \nu : T \ | \phi_2 \} \). By destructing \( \tau \), we have two cases:
  - SubCase \( \tau = \{ \nu : T \ | \phi_2 \} \) for some \( T_2 \) and \( \phi_2 \) Hypotheses:

    \[
    \begin{align*}
    \vdash x : \tau, \Gamma \vdash \tau_1 <: \tau_2 & \quad \text{(H0)} \\
    \vdash \nu : \{ \nu : T \ | \phi_2 \} & \quad \text{(H1)} \\
    \vdash \nu : \{ \nu : T \ | \phi_2 \} & \quad \text{(H2)} \\
    \vdash \nu : \{ \nu : T \ | \phi_2 \} & \quad \text{(H3)} \\
    \Delta & \models [x : \{ \nu : T_2 \ | \phi_2 \}, \Gamma, \nu : T] \Rightarrow [\phi_1] \Rightarrow [\phi_2] & \quad \text{(H4)}
    \end{align*}
    \]

    Expanding \( H4 \):

    \[
    \Delta \models [x : [T_2] \Rightarrow [\phi_2] \Rightarrow [\Gamma, \nu : T] \Rightarrow [\phi_1] \Rightarrow [\phi_2] & \quad \text{(H5)}
    \]

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

    \[
    \Delta \models \forall x. [x : [T_2] \Rightarrow [\phi_2] \Rightarrow [\Gamma, \nu : T] \Rightarrow [\phi_1] \Rightarrow [\phi_2]
    \]

    Proof follows from \( IH0 \) and \( H9 \).

- Case \( \text{SUBT-CONST} \): Proof cannot be given in this form.
Now, using the elimination rule of universal quantification ($\forall E$) to instantiate the bound $x$ with $v$:

\[ \Delta \models [u/x] (x : T_\alpha \Rightarrow [\phi_\alpha] \Rightarrow [\Gamma, \nu : T] \Rightarrow [\phi_1] \Rightarrow [\phi_2]) \quad (H6) \]

Distributing the substitution operation:

\[ \Delta \models (v : [T_\alpha] \Rightarrow \llbracket [v/x] \phi_\alpha \rrbracket \Rightarrow \llbracket [v/x] \Gamma, \nu : T \rrbracket \Rightarrow \llbracket [v/x] \phi_1 \rrbracket \Rightarrow \llbracket [v/x] \phi_2 \rrbracket) \quad (H7) \]

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

\[ \models v : [T_\alpha] \quad (H9) \]

\[ \Delta \models \llbracket [v/x] \phi_\alpha \rrbracket \quad (H10) \]

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

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

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

\[ [v/x] \Gamma \vdash \{v : T | [v/x] \phi_1\} \quad (H12) \]

\[ [v/x] \Gamma \vdash \{v : T | [v/x] \phi_2\} \quad (H13) \]

Applying Subt-Base on $H11 - 13$ proves the theorem.

- SubCase $T = \tau_{a1} \rightarrow \tau_{a2}$, for some $\tau_{a1}$ and $\tau_{a2}$. First, we observe that all free variables in well-formed type refinements (i.e., arguments to relations. Please refer to the syntactic class $\tau_R$ in Fig. 1 of the paper.) have type int or intlist, therefore $x : \tau_{a1} \rightarrow \tau_{a2}$ cannot occur free in $\phi_1, \phi_2$, and type-refinements in $\Gamma$. 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 for all $\Gamma'$, $\Gamma', \Gamma \vdash e : \tau$.

**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:

\[ \tau_1 = \{v : T | \phi_1\} \quad (H2) \]

\[ \tau = \{v : T | \phi\} \quad (H3) \]

\[ \Delta \models [\Gamma, \nu : T] \Rightarrow [\phi_1] \Rightarrow [\phi] \quad (H4) \]

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)** For all $v$ and $\tau$, if $\vdash v : \tau$ then $\vdash v : \tau$.

**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 $\lambda_R$ are in $A$-Normal form by construction, we only need substitution lemma for value substitutions.

**Lemma A.12. (Substitution Preserves Typing)** For all $\Gamma$, $x$, $e$, $\tau_1$, and $\tau_2$, if $\vdash e : \tau_1$, $\Gamma \vdash e : \tau_2$ and $\vdash v : \tau_1$, then $[v/x] \Gamma \vdash [v/x] e : [v/x] \tau_2$.

**Proof**. Hypotheses (after generalizing dependent $\Gamma$ and $\tau_2$):

\[ \forall \Gamma, \forall \tau_2, \cdot : \tau_1, \Gamma \vdash e : \tau_2 \quad (H0) \]

\[ \cdot \vdash v : \tau_1 \quad (H1) \]
Now, we proceed by induction on $H$. 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 $\tau_1$ 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:

\[ [v/x]\Gamma \vdash [v/x]y : [v/x]\tau_1 \]

From $H1$, we know that $\cdot \vdash v : \tau_1$. Applying the weakening lemma (Lemma A.10), with bound $\Gamma'$ 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 $\text{Nil}$, or $e$ is $\text{Cons}$:
    Hypotheses:

    \[
    \Gamma \vdash \text{Nil} : ty(\text{Nil}) \\
    \Gamma \vdash \text{Cons} : ty(\text{Cons})
    \]

    From the definition of $ty$ (Definition A.1), we know that type refinements of $\text{Nil}$ and $\text{Cons}$ are well-formed under empty context; so, they are closed. Consequently $[v/x]ty(\text{Nil}) = ty(\text{Nil})$, and $[v/x]ty(\text{Cons}) = ty(\text{Cons})$. Also, $[v/x]\text{Nil} = \text{Nil}$ and $[v/x]\text{Cons} = \text{Cons}$. Therefore, proof follows from $H1$ and $H2$.

- **Case $T$-App**: $e$ is a function application of form $e_1 v_1$, where:

\[ \cdot, x : \tau_1, \Gamma \vdash e_1 : (y : \tau_3) \rightarrow \tau_2 \quad (H4) \]

\[ \cdot, x : \tau_1, \Gamma \vdash v_1 : \tau_3 \quad (H5) \]

Inductive hypotheses, after trivially instantiating bound $\Gamma$ and $\tau_2$:

\[ [v/x]\Gamma \vdash [v/x]e_1 : [v/x](y : \tau_3) \rightarrow \tau_4 \quad (IH0) \]

\[ [v/x]\Gamma \vdash [v/x]v_1 : [v/x]\tau_3 \quad (IH1) \]

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

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

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:

\[ \tau_2 = (y : \tau_3) \rightarrow \tau_4 \quad (H4) \]

\[ x : \tau_1, \Gamma \vdash \tau_3 \quad (H5) \]

\[ x : \tau_1, \Gamma, y : \tau_3 \vdash e_1 : \tau_4 \quad (H6) \]

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 ($\Gamma$). 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 \( \Gamma \) and \( \tau_2 \) in IH with \( (\Gamma, y : \tau_3) \) and \( \tau_4 \), 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:

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

\[ [v/x][((y : \tau_3) \rightarrow \tau_4) = (y : [v/x] \tau_3) \rightarrow [v/x] \tau_4 \]

It remains to show that

\[ \Gamma \vdash \lambda(y : [v/x] \tau.[v/x] e_1) : (y : [v/x] \tau_3) \rightarrow [v/x] \tau_4 \]

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

• Case T-LET: e is of form let \( y = e_1 \) in \( e_2 \). Hypotheses:

\[ x : \tau_1, \Gamma \vdash e_1 : \tau_3 \quad (H4) \]
\[ x : \tau_1, \Gamma \vdash \tau_2 \quad (H5) \]
\[ x : \tau_1, \Gamma, y : \tau_3 \vdash e_2 : \tau_2 \quad (H6) \]

Since \( y \notin freevars(v) \), \([v/x] e_2\) avoids variable capture. Since a variable cannot be bound twice in \( \Gamma \), \( x \) 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 \( \Gamma \) and \( \tau_2 \) in IH appropriately gives us following hypotheses:

\[ [v/x] \Gamma \vdash [v/x] e_1 : [v/x] \tau_3 \quad (H8) \]
\[ [v/x](\Gamma, y : \tau_3) \vdash [v/x] e_2 : [v/x] \tau_2 \quad (H9) \]

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:

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

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

\[ x : \tau_1, \Gamma \vdash v' : \text{intlist} \quad (H4) \]
\[ x : \tau_1, \Gamma \vdash \text{Nil} : \{\nu : \text{intlist} \mid \phi_n\} \quad (H5) \]
\[ x : \tau_1, \Gamma \vdash \text{Cons} : x' : \text{int} \rightarrow y' : \text{intlist} \rightarrow \{\nu : \text{intlist} \mid \phi_c\} \quad (H6) \]
\[ \Gamma_c = \{\nu : \text{intlist} \mid \phi_c\} \quad (H7) \]
\[ \Gamma_n = \{\nu' : \nu \mid \phi_n\} \quad (H8) \]
\[ x : \tau_1, \Gamma \vdash \tau \quad (H9) \]
\[ x : \tau_1, \Gamma, \Gamma_c \vdash e_1 : \tau \quad (H10) \]
\[ x : \tau_1, \Gamma, \Gamma_n \vdash e_2 : \tau \quad (H11) \]

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 \( \Gamma \). Therefore, we are left with one case:

• SubCase \( x \neq x' \), and \( x \neq y' \): Substitution \([v/x] e\) can be expanded to

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

Inductive hypotheses (after pushing substitutions into type refinements):

\[ [v/x] \Gamma \vdash [v/x] v' : \text{intlist} \quad (IH1) \]
\[ [v/x] \Gamma \vdash \text{Nil} : \{\nu : \text{intlist} \mid [v/x] \phi_n\} \quad (IH2) \]
\[ [v/x] \Gamma \vdash \text{Cons} : x' : \text{int} \rightarrow y' : \text{intlist} \rightarrow \{\nu : \text{intlist} \mid [v/x] \phi_c\} \quad (IH3) \]
\[ [v/x] \Gamma, [v/x] \Gamma_c \vdash [v/x] e_1 : [v/x] \tau \quad (IH4) \]
\[ [v/x] \Gamma, [v/x] \Gamma_n \vdash [v/x] e_2 : [v/x] \tau \quad (IH5) \]

Expansions of \([v/x] \Gamma_c \) and \([v/x] \Gamma_n \) are given below:

\[ [v/x] \Gamma_c = x' : \text{int}, y' : \text{intlist}, [v/x] [v'/\nu] \phi_c \quad (H13) \]
\[ [v/x] \Gamma_n = [v/x] [v'/\nu] \phi_n \quad (H14) \]

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

\[ \vdash \text{Nil} : \{\nu : \text{intlist} \mid \phi_n\} \]
\[ \vdash \text{Cons} : x' : \text{int} \rightarrow y' : \text{intlist} \rightarrow \{\nu : \text{intlist} \mid \phi_c\} \]
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:

\[ \vdash \phi_n \]  \\
\[ x' : \text{int}, y' : \text{intlist} \vdash \phi_c \]  \\

From H15, we conclude that:

\[ x \notin \text{freevars}(\phi_n) \]  \\

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

\[ x \notin \text{freevars}(\phi_c) \]  \\

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

\[ [v/x] \Gamma \vdash x' : \text{int}, y' : \text{intlist}, [[v/x] \nu'/\nu] \phi_c \]  \\
\[ [v/x] \Gamma_n = [[v/x] \nu'/\nu] \phi_n \]  \\

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

\[ [v/x] \Gamma \vdash \text{match } [v/x] \nu' \text{ with } \text{Cons } x' y' \rightarrow [v/x] e_1 \text{ else } [v/x] e_2 : [v/x] \tau \]

Which is what needs to be proven.

**Theorem A.13. (Preservation)** If \( \vdash e : \tau, \) and \( e \rightarrow e', \) then \( \vdash e' : \tau. \)

**Proof** by induction on type derivation \( \vdash e : \tau. \) Cases:

- Case T-VAR: \( e \) is a variable \( x. \) By inversion on \( \vdash x : \tau \) 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 \( e_1 \ nu, \) where \( \vdash e_1 : (x : \tau_1) \rightarrow \tau_2 \) and \( \vdash v : \tau_1. \) Therefore, \( \vdash e : [v/x] \tau_2. \) Inversion on \( e_1 v \rightarrow e' \).
  - SubCase E-Co: \( e_1 \rightarrow e_1'. \) Therefore, \( e_1 v \rightarrow e_1' v. \) By IH, \( \vdash e_1' : (x : \tau_1) \rightarrow \tau_2. \) Hence, \( \vdash e_1' v : [v/x] \tau_2. \)
  - SubCase E-App: Inversion on \( \vdash e_1 : (x : \tau_1) \rightarrow \tau_2. \) Cases:
    - SubSubCase \( e_1 = \lambda x : \tau_3. e_2. \) Therefore, \( e_1 v \rightarrow [v/x] e_2. \) By inversion on \( \vdash \lambda x : \tau_3. e_2 : \tau_1 \rightarrow \tau_2, \) we know that \( \tau_3 = \tau_1, \) and
    - \( x : \tau_1 \vdash e_2 : \tau_2 \) (H0)
      - Now, using H0 and the substitution lemma (Lemma A.12), with bound \( \Gamma \) instantiated with \( x, \) we get \( \vdash [v/x] e_2 : [v/x] \tau_2. \)
      - Hence, \( e' = [v/x] e_2 \) has same type as \( e. \)
  - SubSubCase \( e_1 = \text{Cons}, \) or \( e_1 = \text{Cons } v_1, \) for some \( v_1; \) Both \( \text{Cons } v, \) and \( \text{Cons } v_1 \) are values, therefore cannot take a step.
- Case T-SUB: Hypotheses:
  
  \[ \vdash e : \tau_1 \]  \\
  \( \tau_1 < \tau \)  \\

  Inductive hypothesis:

  \[ \text{If } e \rightarrow e', \text{ then } \vdash e' : \tau_1 \]  \\
  (IH)

  Proof follows from IH and H1.
- Case T-LET: \( e \) is of form \( \text{let } x = e_1 \ \text{in } e_2, \) where:
  
  \[ \vdash e_1 : \tau_1 \]  \\
  \( \tau_1 < \tau \)  \\

  Inductive Hypothesis:

  \[ \text{If } e_1 \rightarrow e_1', \text{ then } \vdash e_1' : \tau_1 \]  \\
  (IH)

  By inversion on \( e \rightarrow e', \) we have two cases:
  - SubCase E-Co: \( e_1 \rightarrow e_1' \) and \( e \rightarrow \text{let } x = e_1' \ \text{in } e_2. \) Applying IH, we know that:
    
    \[ \vdash e_1' : \tau_1 \]  \\
  - SubCase E-LET: \( e \rightarrow v \) and \( e \rightarrow [v/x] e_2. \) Using H0 and H1, and applying the substitution lemma (Lemma A.12), we derive the following:
    
    \[ \vdash [v/x] e_2 : [v/x] \tau \]  \\
  From hypothesis H2, we know that \( \tau \) is well-formed under empty context. Rewriting H4 with \([v/x] \tau = \tau \) lets us conclude:
    
    \[ \vdash [v/x] e_2 : \tau \]  \\

\[
\phi^Q : \tau^F, \quad \phi^L : \tau^F
\]

**Q-PROP-SORT**

\[
\cdot \vdash \phi^Q : \text{bool}
\]

\[
\cdot \vdash \phi^L : \tau^F
\]

\[
\cdot \vdash \forall (k : T^F), \phi^L : T^F \rightarrow \text{bool}
\]

Figure 14: Nominal Type System for MSFOL propositions

- Case T-MATCH: \(e\) is of form \(v\) with \(\text{Cons } x\ y \Rightarrow e_1\ else\ e_2\), where:

  \[
  \vdash v : \text{intlist} \quad (H0)
  \]

  \[
  \cdot \vdash \text{Nil} : \{v : \text{intlist} \mid \phi_n\} \quad (H1)
  \]

  \[
  \cdot \vdash \text{Cons} : x \rightarrow y : \text{intlist} \rightarrow \{v : \text{intlist} \mid \phi_n\} \quad (H2)
  \]

  \[
  \Gamma_e = x : \text{int}, y : \text{intlist}, \{v/y\} \phi_c \quad (H4)
  \]

  \[
  \Gamma_n = \{v/y\} \phi_n \quad (H5)
  \]

  \[
  \cdot \vdash \tau \quad (H6)
  \]

  \[
  \Gamma_e \vdash e_1 : \tau \quad (H7)
  \]

  \[
  \Gamma_n \vdash e_2 : \tau \quad (H8)
  \]

By inversion on \(e \rightsquigarrow e'\), we have two cases:

- **SubCase E-MCONS**: Hypotheses:

  \[
  v = \text{Cons } v_1\ v_2 \quad (H9)
  \]

  \[
  e' = [v_2/y][v_1/x]e_1 \quad (H10)
  \]

  By inversion on \(H0\), we derive:

  \[
  \cdot \vdash v_1 : \text{int} \quad (H11)
  \]

  \[
  \cdot \vdash v_2 : \text{intlist} \quad (H12)
  \]

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

  \[
  [v_2/y][v_1/x]v/\phi \vdash [v_2/y][v_1/x]e_1 : [v_2/y][v_1/x]\tau \quad (H13)
  \]

  From Definition A.1, which asserts the validity of type refinements of \(\text{Cons}\) and \(\text{Nil}\), we get:

  \[
  \Delta, x : [\text{int}], y : [\text{intlist}] \vdash [[\text{Cons } x\ y]/v] \phi \quad (H14)
  \]

  Using \(H11 - 12\), and instantiating \(x\) and \(y\) with \(v_1\) and \(v_2\), respectively:

  \[
  \Delta \vdash [[v_2/y][v_1/x]v/\phi] \quad (H15)
  \]

  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 \(\tau\) is a closed term; therefore, \([v_2/y][v_1/x]\tau = \tau\). Using this fact to rewrite \(H16\), we conclude that:

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

- **SubCase E-MNIL**: \(v = \text{Nil}.\ e' = e_2\). From \(H5\) and \(H8\):

  \[
  [v/y]\phi_n \vdash e_2 : \tau \quad (H9)
  \]

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

  \[
  \cdot \vdash e_2 : \tau \quad (H9)
  \]

**Theorem A.14.** (Type Safety) If \(\cdot \vdash e : \tau\), then either \(e\) is a value, or \(e \rightsquigarrow e'\) 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 language We define a nominal type system that assigns MSFOL sorts (\(\tau^F\)) 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:

\[
\begin{align*}
\text{bool} \triangleright \text{bool} & = \text{bool} \\
\text{bool} \triangleright T^F \rightarrow \tau^F & = T^F \rightarrow \tau^F \\
T^F \rightarrow \tau^F \triangleright \text{bool} & = T^F \rightarrow \tau^F \\
T^F \rightarrow \tau_1^F \triangleright \tau_2^F & = T^F \rightarrow \tau_2^F \triangleright \tau_1^F
\end{align*}
\]

**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, $\phi_1^L$ and $\phi_2^L$, $\phi_1^L \odot \phi_2^L$ is an MSFOL formula.

Lemma A.15. $(\eta_{\text{wrap}}$ is type safe) for all $\phi^F$, there exists and MSFOL proposition $\phi^L$ such that $\eta_{\text{wrap}}(\phi^F, \tau^F) = \phi^L$, and $\vdash \phi^L : \tau^F$

Proof By induction on $\tau^F$. Cases:
- Case $\tau^F = \text{bool}$: $\eta_{\text{wrap}}(\phi^F, \text{bool}) = \phi^F$, which is an MSFOL proposition. From QF-PROP-SORT, we also know that $\vdash \phi^F : \text{bool}$.
- Case $\tau^F = T^F \rightarrow \tau_2^F$: Eta-wrap expansion is: $\eta_{\text{wrap}}(\phi^F, T^F \rightarrow \tau_2^F) = \forall (k : T^F). \eta_{\text{wrap}}(\phi^F, k, \tau_2^F)$. Inductive hypothesis tells us that there exists an MSFOL proposition $\phi_2^L$, such that:

$$\phi_2^L = \eta_{\text{wrap}}(\phi^F, k, \tau_2^F) \quad (IH0)$$

Now, $\phi^L = \forall (k : T^F). \phi_2^L$ is an MSFOL proposition. Further, applying Q-PROP-SORT on $IH1$ lets us conclude:

$$\vdash \phi^L : T^F \rightarrow \tau_2^F \quad (IH2)$$

Lemma A.16. (MSFOL sort of a relation) for all $R$, if $\vdash R :: \tau_R$, then there exists an MSFOL proposition $\phi_R^L$ such that $[R] = \phi_R^L$, and $\vdash \phi_R^L : [\tau_R]$.

Proof We proceed by case analysis on $R$.
- Case $R = R_{id}$: From S-REL-1D, we know that:

$$\vdash R_{id} :: \text{int} \rightarrow \{ \text{int} \} \quad (H0)$$

Also, from Fig. 4:

$$[\text{int} \rightarrow \{ \text{int} \}] = [\text{int}] \rightarrow [\text{int}] \rightarrow \text{bool} \quad (H1)$$

$$[R_{id}] = \forall (j : [\text{int}]). \forall (k : [\text{int}]), j = k \quad (H2)$$

From the definition of MSFOL encoding of types, we know that $[\text{int}] = F(\text{int})$. Using this to rewrite $H2$, we deduce that:

$$\phi_{R_{id}}^L = \forall (j : F(\text{int})). \forall (k : F(\text{int})), j = k \quad (H3)$$

Applying Q-PROP-SORT twice, we also know that:

$$\vdash \forall (j : [\text{int}]). \forall (k : [\text{int}]), j = k : [\text{int}] \rightarrow [\text{int}] \rightarrow \text{bool} \quad (H4)$$

Therefore:

$$\vdash \forall (j : F(\text{int})). \forall (k : F(\text{int})), j = k : [\text{int}] \rightarrow [\text{int}] \rightarrow \text{bool} \quad (H5)$$

From $H3$, and rewriting $H5$ with $H1$ gives us proof.
- $R$ is any relation that is not $R_{id}$: From Fig. 4:

$$[R] = \eta_{\text{wrap}}(R, [\Gamma_R(R)]) \quad (H0)$$

Since $\vdash R :: \tau_R$, from the definition of ordered map $\Gamma_R$, we know that $\Gamma_R(R) = \tau_R$. Rewriting $H0$:

$$[R] = \eta_{\text{wrap}}(R, \tau_R) \quad (H1)$$

Now, applying Lemma A.15 gives us proof.

Lemma A.17. (Substitution Preserves Nominal Typing) for all $\phi^L$, $x$, and $y$, if $\vdash \phi^L : \tau^F$, then $\vdash [y/x] \phi^L : \tau^F$

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. $(\gamma_{\otimes}$ correctness) for all $\phi_1^L$, $\phi_2^L$, and $\tau^F$, if $\vdash \phi_1^L : \tau^F$, and $\vdash \phi_2^L : \tau^F$, then there exists an MSFOL proposition $\phi_{\otimes}^L$ such that $\gamma_{\otimes}(\phi_1^L, \phi_2^L) = \phi_{\otimes}^L$, and $\vdash \phi_{\otimes}^L : \tau^F$.

Proof by simultaneous induction (i.e., double induction followed by elimination of absurd cases) on nominal typing derivations $\vdash \phi^L : \tau^F$, and $\vdash \phi_2^L : \tau^F$. Cases:
- Case Q-PROP-SORT : $\tau^F$ is of form $T^F \rightarrow \tau_2^F$. Propositions $\phi_1^L$ and $\phi_2^L$ are of form $\forall (k : T^F), \phi_{11}^L$ and $\forall (k : T^F), \phi_{21}^L$, respectively, such that:

$$\vdash \phi_1^L : \tau_2^L \quad (H0)$$

From inductive hypotheses, we know that there exists an MSFOL prop $\phi_{11}^L$ such that:

$$\gamma_{\otimes}(\phi_{11}^L, \phi_2^L) = \phi_{\otimes}^L \quad (IH0)$$

From the definition of $\gamma_{\otimes}$, we have that:

$$\gamma_{\otimes}(\forall (k : T^F), \phi_{11}^L, \forall (k : T^F), \phi_{21}^L) = \forall (k : T^F), \gamma_{\otimes}(\phi_{11}^L, \phi_{21}^L) \quad (H2)$$

Rewriting $H2$ using $IH0$, we have:

$$\gamma_{\otimes}(\forall (k : T^F), \phi_{11}^L, \forall (k : T^F), \phi_{21}^L) = \forall (k : T^F), \phi_{\otimes}^L \quad (H2)$$

Hence, $\phi^L$ is $\forall (k : T^F), \phi_{\otimes}^L$. It remains to prove that $\vdash \forall (k : T^F), \phi_{\otimes}^L : \tau^F$, which can be done by applying Q-PROP-SORT on $IH1$.\]
Case QF-PROP-SORT: $\tau^R = \text{bool}$. Inversion on $\vdash \phi^R_x : \text{bool}$, and $\vdash \phi^R_y : \text{bool}$ lets us infer that $\phi^R_x$ and $\phi^R_y$ are quantifier-free propositions $\phi^R_1$ and $\phi^R_2$, respectively. From the definition of $\gamma_{\text{un}}$, we know that:

$$\gamma_{\text{un}}(\phi^R_1, \circ, \phi^R_2) = \phi^R_3$$

Proof is obtained by observing that $\phi^R_1 \circ \phi^R_2$ is a quantifier-free MSFOL formula, which, by QF-PROP-SORT rule has type \text{bool}.

**Lemma A.19. (Correctness)** For all $\phi^R_1, \phi^R_2, \tau^R_1, \tau^R_2$, if $\vdash \phi^R_1 : \tau^R_1$, and $\vdash \phi^R_2 : \tau^R_2$, then there exists an MSFOL proposition $\phi^L$ such that $\gamma_{\text{un}}(\phi^R_1, \circ, \phi^R_2) = \phi^L$, and $\vdash \phi^L : \tau^R_1 \otimes \tau^R_2$.

**Proof** by simultaneous inductions on nominal typing derivations $\vdash \phi^R_1 : \tau^R_1$, and $\vdash \phi^R_2 : \tau^R_2$. 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)** For all $\Gamma, r$, and $\theta$, if $\Gamma \vdash r : \{\theta\}$, then there exists a $\phi^L$ such that $[r] = \phi^L$, and $\vdash \phi^L : \{\theta\}$.

**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:
  $$\Gamma \vdash r_1 : \{\theta\} \quad \text{(H0)}$$
  $$\Gamma \vdash r_2 : \{\theta\} \quad \text{(H1)}$$

  From inductive hypotheses, we know that there exist two propositions, $\phi^R_1$ and $\phi^R_2$, such that:

  $\begin{align*}
  [r_1] &= \phi^R_1 \quad \text{(IH0)}
  
  [r_2] &= \phi^R_2 \quad \text{(IH0)}
  
  \vdash \phi^R_1 : \{\theta\} \quad \text{(IH2)}
  
  \vdash \phi^R_2 : \{\theta\} \quad \text{(IH3)}
  \end{align*}$

  We know that $[r_1 \cup r_2] = \gamma_{\text{un}}([r_1], \lor, [r_2])$. Therefore, the goal is to prove that there exists a $\phi^L$, such that:

  $$\gamma_{\text{un}}([r_1], \lor, [r_2]) = \phi^L, \quad \vdash \phi^L : \{\theta\}$$

  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 $\lambda_R$ value $v$. Hypotheses:

  $$\vdash R : T \rightarrow \{\theta\} \quad \text{(H0)}$$
  $$\vdash v : T \quad \text{(H1)}$$

  From definition of MSFOL encoding for colon-arrow types:

  $$[T \rightarrow \{\theta\}] = [T] \rightarrow [\theta] \quad \text{(H2)}$$

  Since $T \in \{\text{int}, \text{intlist}\}$, we have the following cases for $v$:

  - **SubCase** $v$ is a variable $x : r = R(x)$. From the definition of MSFOL encoding:
    $$[R(x)] = \text{Inst}([R], x) \quad \text{(H3)}$$

    From Lemma A.16, we know that $[R]$ is an MSFOL formula $\phi^R_R$ such that $\vdash \phi^R_R : [T \rightarrow \{\theta\}]$. Rewriting using H2:

    $$\vdash \phi^R_R : [T] \rightarrow [\theta] \quad \text{(H4)}$$

    By inversion on H4, we know that $\phi^R_R$ is of form $\forall k : [T], \phi^L_k$, where

    $$\vdash \phi^L_k : \{\theta\} \quad \text{(H5)}$$

    Rewriting H3:

    $$[R(x)] = \text{Inst}(\forall k : [T], \phi^L_k, x) \quad \text{(H6)}$$

    From the definition of $\text{Inst}$, H6 reduces to:

    $$[R(x)] = [x/k] \phi^L_k \quad \text{(H7)}$$

    Therefore, $[R(x)]$ is an MSFOL formula. Further, Applying substitution lemma of nominal typing (Lemma ??) on H5, we also have that

    $$\vdash [x/k] \phi^L_k : \{\theta\}$$

    This concludes the proof for current SubCase.

  - **SubCase** $v$ is Nil: $r = R(\text{Nil})$. Hypotheses:

    $$\vdash R(\text{Nil}) : \{\theta\} \quad \text{(H0)}$$

    By inversion on H0:

    $$\vdash R(\text{Nil}) \vdash \text{Nil} \rightarrow \{\theta\} \quad \text{(H1)}$$

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

    $$R \triangleq \text{Nil} ightarrow r_1 \mid \text{Cons} x y \rightarrow r_2 \quad \text{(H2)}$$

    $$\vdash r_1 : \{\theta\} \quad \text{(H3)}$$
Using $H3$, the inductive hypothesis tells us that there exists a $\phi^L$, such that
\[
[r_1] = \phi^L \quad (H4)
\]
\[
\vdash \phi^L : [\theta] \quad (H5)
\]

From the definition of MSFOL encoding:
\[
[R(\text{Nil})] = [\Sigma_R(R)(\text{Nil})]
\]

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

- Case $v$ is $\text{Cons} \ v_1 \ v_2$, for some $\lambda_R$ values $v_1$ and $v_2$. From the type of $\text{Cons}$ under simple type system, we know that:
\[
\| \Gamma \| \vdash v_1 : \text{int}
\]
\[
\| \Gamma \| \vdash v_2 : \text{intlist}
\]

As a corollary of Lemma A.4, from previous two hypotheses, we have:
\[
\| \Gamma \| \vdash v_1 : \text{int} \quad (H0)
\]
\[
\| \Gamma \| \vdash v_2 : \text{intlist} \quad (H1)
\]

Further, we have the following hypothesis:
\[
\Gamma \vdash R(\text{Cons} \ v_1 \ v_2) :: \{ \theta \} \quad (H2)
\]

By inversion on $H0$:
\[
\Gamma \vdash R :: \text{intlist} \rightarrow \{ \theta \} \quad (H3)
\]

By inversion on $H1$, for some relational expressions $r_1$ and $r_2$:
\[
R \triangleq (\text{Nil} \Rightarrow r_1 | \text{Cons} \ x \ y \Rightarrow r_2) \quad (H4)
\]
\[
\vdash [v_2/y] [v_1/x] r_2 :: \{ \theta \} \quad (H5)
\]

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

From the definition of MSFOL encoding:
\[
[R(\text{Cons} \ v_1 \ v_2)] = [\Sigma_R(R)(\text{Cons} \ v_1 \ v_2)] \quad (H6)
\]

Using the definition of $\Sigma_R$, followed by desugaring:
\[
[R(\text{Cons} \ v_1 \ v_2)] = [v_2/y] [v_1/x] r_2 \quad (H7)
\]

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

**Theorem A.21. (Completeness of MSFOL semantics)** For all $\phi$, $\Gamma$, if $\Gamma \vdash \phi$, then there exists an MSFOL proposition $\phi^L$ such that $[\phi] = \phi^L$.

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

- Case $\text{WF-REF}$: $\phi = \phi_1 \land \phi_2$, or $\phi = \phi_1 \lor \phi_2$. 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 $\text{WF-PRED}$: $\phi$ is of form $r_1 = r_2$, or $r_1 \subseteq r_2$, for some relational expressions $r_1$ and $r_2$. Hypotheses:
\[
\Gamma \vdash r_1 :: \{ \theta \} \quad (H0)
\]
\[
\Gamma \vdash r_2 :: \{ \theta \} \quad (H1)
\]

where, $\theta$ is a tuple sort. From lemma A.20, we know that there exist two MSFOL propositions $\phi^L_1$ and $\phi^L_2$, such that:
\[
\vdash \phi^L_1 : \{ \theta \} \quad (H2)
\]
\[
\vdash \phi^L_2 : \{ \theta \} \quad (H3)
\]

Now, since:
\[
[r_1 = r_2] = \gamma_\subseteq([r_1] \leftrightarrow [r_2]) \quad (H4)
\]
\[
[r_1 \subseteq r_2] = \gamma_\subseteq([r_1] \rightarrow [r_2]) \quad (H4)
\]

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