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 for definitions of parametric relations for polymorphic data types that enable 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.

Categories and Subject Descriptors D.3.2 [Language Classifications]: Applicative (Functional) Languages; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D.2.4 [Software Engineering]: Software/Program Verification

Keywords Relational Specifications; Inductive Relations; Parametric Relations; Dependent Types; Decidability; Standard ML

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 [24], F* [21, 23], and OCaml [16]. For example, the following Haskell signatures specify how the length of the result list for append and rev relate to their arguments:

\[
\begin{align*}
\text{append} & : \text{List } a \times n \rightarrow \text{List } a \times m \rightarrow \text{List } a \times (\text{Plus } n \times m) \\
\text{rev} & : \text{List } a \times n \rightarrow \text{List } a \times n
\end{align*}
\]

While length-indexed lists capture stronger invariants over append, and rev than possible with just simple types, they still under-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP '14, September 1-6, 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-2873-9/14/09...
http://dx.doi.org/10.1145/10.1145/2628136.2628159

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 abstract 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:

\[
\text{rev} : \{ l : \text{a list} \} \rightarrow \{ \nu : \text{a list} | \nu = \text{rev}'(1) \}
\]

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 like lists, this strategy nonetheless requires that the program be fully described in logic, and reasoned about by the solver in its entirety. Thus, 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, involutivity 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, observe 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 \( l = [1; x_1, x_2, x_3] \), the relation’s closure \( R_{ob} \) would be:

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

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

\[
\text{rev} : \{ l : 'a list \} \rightarrow \{ n : \nu \text{ list } | R_{oa}(1) = R_{ob}(\nu) \}
\]

Since \( R_{oa}(1) \) represents the set of pairs whose elements exhibit the occurs-before property in the input list, and \( R_{ob}(\nu) \) represents the set of pairs whose elements exhibit the occurs-after property in the output list, the above specification effectively asserts that for every pair of elements \( x \) and \( y \) in the input list 1, if \( x \) occurs before \( y \) in 1, then \( x \) has to occur after \( y \) in the result list \( \nu \).

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

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

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

\[
\text{dup} : \{ l : 'a list \} \rightarrow \{ x : \{ 'a* 'a \} list \}
\]

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

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

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 structured as follows. In the next section, we present additional motivation and examples for our ideas. Sec. 3 formalizes the syntax and static semantics of relational refinements in the context of a simply-typed core language. Sec. 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 Sec. 5. Sec. 6 presents a case study. Secs. 7, 8 and 9 present related work, directions for future work, and conclusions, respectively.
2. Structural Relations

Our specification language is primarily the language of relational expressions composed using familiar relational algebraic operators. This language is additionally equipped with pattern matching over constructors of algebraic types to define shape properties in terms of these expressions. A number of built-in polymorphic relations are provided, the most important of which are listed below:

\[
\begin{align*}
R_{id} (x) &= \{\langle x \rangle\} \\
R_{dup} (x) &= \{\langle x, x \rangle\} \\
R_{head} (x) &= \{\langle x \rangle\} - \{\langle x \rangle - \{\langle x \rangle\}\}
\end{align*}
\]

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 \( l \) to every element \( x \) in \( l \) (regardless of \( l \)'s size). This relation would allow us to define an ordering property such as \( \text{occurs-before or occurs-after} \) on precisely those elements that comprise \( \text{rev} \)'s input and output lists. A recursive definition of \( R_{mem} \) looks like:

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

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 \( \text{'a} \) list with elements \( v \) of type \( \text{'a} \). 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.

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

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

The transitive closure of this relation \( R_{OB} \) expresses the \( \text{occurs-before} \) property on every element in the list. The \( \text{occurs-after} \) relation can be defined similarly:

\[
\text{relation } R_{oa} (x :: xs) = R_{mem} (xs) \times \{\langle x \rangle\}
\]

2.2 Parametric Relations

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

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

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 \textit{parametrized} 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:

\[
\begin{align*}
\text{relation } (R_{hd} R) (x :: xs) &= R (x) \\
&\, | (R_{hd} R) [] = \emptyset \\
\text{relation } (R_{mem} R) &= (R_{hd} R)^* \\
\text{relation } R_{fst} (x, y) &= \{\langle x \rangle\} \\
\text{relation } R_{snd} (x, y) &= \{\langle y \rangle\}
\end{align*}
\]

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

\[
\text{zip} : l_1 \to l_2 \to \{\langle v \rangle | ((R_{mem} R_{fst}) v) = ((R_{mem} R_{hd}) l_1) \land ((R_{mem} R_{snd}) v) = ((R_{mem} R_{id}) l_2)\}
\]

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

\[
\begin{align*}
\text{relation } (R_{ob} R) (x :: xs) &= R (x) \times ((R_{mem} R) xs) \\
\text{relation } (R_{oa} R) (x :: xs) &= ((R_{mem} R) xs) \times R (x)
\end{align*}
\]

Using this parametric version of \( R_{oa} \), the \( \text{dup} \) function described in the previous section can now be specified thus:

\[
\text{dup : l} \to \{\langle v \rangle | ((R_{ob} R_{fst})^* v) = ((R_{ob} R_{id})^* l) \land ((R_{ob} R_{snd})^* v) = ((R_{ob} R_{id})^* l)\}
\]

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

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

This type relates membership properties on foldl’s input list, expressed in terms of a non-parametric \( R_{\text{mem}} \) relation, to an abstract notion of membership over its result type ('b) captured using a relation variable ('R_{\text{foldl}}). This signature constrains foldl to produce a result for which a membership property is a sensible notion. For instance, if foldl were applied to an argument in which \( b \) was of some list type (e.g., ['a]), because it is used as a list transform operator, then 'R_{\text{foldl}} could be trivially instantiated with R_{\text{mem}}. However, allowing types to be parameterized over relation variables enable richer properties to be expressed. For example, consider the function makeTree that uses foldl to generate a binary tree using function treeInsert (not shown):

```
data type 'a tree = Leaf | Tree of 'a * (Tree) * (Tree)
relation R_{\text{tree}} \text{ Leaf } = \emptyset
relation R_{\text{tree}} \text{ (Tree } (x,t_1,t_2)\text{) = } \{ x \}
relation R_{\text{mem}} = R_{\text{tree}}
makeTree : \{ l : 'a \text{ list} \} \to \{ \nu : \text{ Tree of } 'a \to \{ R_{\text{mem}}(\nu) = R_{\text{mem}}(1) \} \}
val \text{ makeTree } = \text{ fn } l \Rightarrow \text{ foldl } (R_{\text{mem}}) \text{ 1 Leaf } \text{ treeInsert}
```

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

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

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

\[
\langle R_{\text{foldl}}, R_{\text{am}} \rangle \text{ foldl} : \{ l : 'a \text{ list} \} \to \{ b : 'b \} \to \{ f : \{ x : 'a \} \to \{ acc : 'b \} \to \{ z : 'b \mid R_{\text{foldl}}(z) = \{ \{ x \} \cup R_{\text{foldl}}(\text{acc}) \} \cup \{ \nu : \{ R_{\text{foldl}}(\nu) = R_{\text{am}}(1) \cup R_{\text{am}}(b) \} \cup \{ \nu : \{ R_{\text{am}}(\nu) = R_{\text{am}}(1) \cup R_{\text{am}}(b) \} \} \}
\]

An implementation of rev that uses foldl is given below:

```
rev : \{ l : 'a \text{ list} \} \to \{ \nu : \text{ Tree of } 'a \to \{ R_{\text{am}}(\nu) = R_{\text{am}}(l) \} \}
val \text{ Cons } = \text{ fn } x \Rightarrow \text{ fn } s x \Rightarrow x :\! s
val \text{ rev } = \text{ fn } l \Rightarrow \text{ foldl } (R_{\text{mem}}, R_{\text{am}}) \text{ l [] Cons}
```

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

3. Core language

3.1 Syntax

We formalize our ideas using a core calculus \( (\lambda_R) \) shown in Fig. 1 an A-normalized extension of the simply-typed lambda calculus. The language supports a primitive type (int), a recursive data type (intlist), along with dependent base and function types. Because the mechanisms and syntax to define and elaborate recursive data types are kept separate from the core, \( \lambda_R \) is only provided with two constructors, Nil and Cons used to build lists. The language has a standard call-by-value operational semantics, details of which can be found in an accompanying technical report [19].

Dependent type refinements (\( \phi \)) in \( \lambda_R \) are assertions over relational expressions (\( r \)); these expressions, which are themselves

\footnote{Proofs for all lemmas and theorems given in this paper are also provided in the report.}
Sort Checking Specification Language

\[ \Gamma \vdash r :: \{ \theta \}, \quad \Gamma \vdash R :: T :\rightarrow \{ \theta \} \]

**S-REL**

\[ R \triangleq (\text{Nil} \Rightarrow r_1, \text{Cons} \, x \, y \Rightarrow r_2) \]

\[ \vdash r_1 :: \{ \theta \}, x :: \text{int}, y :: \text{intlist} \vdash r_2 :: \{ \theta \} \]

\[ \vdash R :: \text{intlist} :\rightarrow \{ \theta \} \]

**S-REL-STAR**

\[ R_1 \triangleq R_2^* \quad \vdash R_1 :: \tau_R \]

**S-APP**

\[ \vdash R :: T :\rightarrow \{ \theta \} \]

**S-REL-ID**

\[ \vdash R_{id} :: \text{int} :\rightarrow \{ \text{int} \} \]

**S-UNION**

\[ \Gamma \vdash r_1 :: \{ \theta \}, \Gamma \vdash r_2 :: \{ \theta \} \]

\[ \vdash r_1 \cup r_2 :: \{ \theta \} \]

**S-CROSS**

\[ \Gamma \vdash r_1 :: \{ \theta_1 \}, \Gamma \vdash r_2 :: \{ \theta_2 \} \]

\[ \vdash r_1 \times r_2 :: \{ \theta_1 \times \theta_2 \} \]

**Well-Formedness**

\[ \Gamma \vdash \phi, \quad \Gamma \vdash \tau \]

**WF-RPred**

\[ \bigcirc \in \{=, \leq, \lt \} \]

\[ \vdash r_1 :: \{ \theta \}, \quad \Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2 \]

\[ \vdash r_1 \circ r_2 :: \{ \theta \} \]

**WF-REF**

\[ \bigcirc \in \{\land, \lor\} \]

\[ \vdash r_1 :: \{ \theta \}, \quad \Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2 \]

\[ \vdash \phi_1 \circ \phi_2 :: \{ \theta \} \]

**WF-BASE**

\[ \nu :: T \vdash \phi \]

\[ \vdash \nu :: T \vdash \phi \]

**WF-FUN**

\[ \Gamma \vdash \phi, \quad \Gamma \vdash \tau \]

\[ \Gamma \vdash \tau_1 \vdash \phi, \quad \Gamma \vdash x :: \tau_1 \vdash \tau_2 \]

**Subtyping**

\[ \Gamma \vdash \tau_1 \vdash \tau_2 \]

**SUBT-BASE**

\[ \Gamma \vdash \nu :: T \vdash \phi \]

\[ \vdash \phi_2 :: \{ \theta \} \]

\[ \Gamma \vdash \tau_1 \vdash \phi \]

**SUBT-ARROW**

\[ \Gamma \vdash \nu :: T \vdash \phi \]

\[ \vdash \phi_2 :: \{ \theta \} \]

\[ \Gamma \vdash \tau_1 \vdash \phi \]

Type Checking Expression Language

\[ \Gamma \vdash e :: \tau \]

**T-VAR**

\[ (x : \tau) \in \Gamma \quad \Gamma \vdash x :: \tau \]

\[ \Gamma \vdash x :: \tau, e :: \tau_1 \vdash e :: \tau_2 \]

\[ \Gamma \vdash \lambda (x : \tau_1) . e :: (x : \tau_1) :: \tau_2 \]

**T-ABS**

\[ \Gamma \vdash \phi :: \tau \]

\[ \Gamma \vdash \lambda (x : \tau_1) . e :: (x : \tau_1) :: \tau_2 \]

**T-CONST**

\[ \Gamma \vdash e :: \tau_1, \Gamma \vdash e :: \tau_2 \]

**T-APP**

\[ \Gamma \vdash e :: (x : \tau_1) :: \tau_2 \]

\[ \Gamma \vdash e :: (x : \tau_1) :: \tau_2 \]

**T-LET**

\[ \Gamma \vdash e :: (x : \tau_1) :: \tau_2 \]

\[ \Gamma \vdash e :: (x : \tau_1) :: \tau_2 \]

\[ \Gamma \vdash \text{let} \; x :: e_1 \; \text{in} \; e_2 :: \tau_2 \]

Figure 3: Static semantics of \( \lambda_R \)

Typed, constitute the syntactic class of expressions in our specification language. We refer to the types of relational expressions as *sorts*, in order to distinguish them from \( \lambda_R \) types. We write \( r :: s \) to denote that a relational expression \( r \) has sort \( s \). A structural relation is a triple, consisting of a unique relation name, its sort, and its definition as (a) a pattern-match sequence that relates constructors of an algebraic data type to a relation expression, or (b) an inductive extension of an existing relation, captured using the closure operator (\( \circ \)). We write \( R \triangleq \delta \) to denote that a relation \( R \) has a (pattern-match or inductive) definition \( \delta \).

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

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

indicates that the sort of relation \( R_{ob} \) is a map from integer lists to pairs. As reflected by the syntactic class of relation sorts (\( \tau_R \)), 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 \( R_{id} \) whose sort is \( \text{int} :\rightarrow \{ \text{int} \} \) 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:

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

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

Structural relations are sort checked under an empty type environment. The rule S-REL type checks a relation definition by ensuring that relational expressions associated with the constructors that comprise the definition all have the same sort. The rule S-REL-
STAR captures the fact that an inductive extension of a relation has the same type as the relation itself. The rule S-APP sort checks relation applications by ensuring that the argument to the relation has the required simple (non-dependent) type. The rule makes use of a simple typing judgment (\(\Gamma\)) under a refinement erased \(\Gamma\) (denoted \(\Gamma^+\)) for this purpose. Rules for simple typing judgments are straightforward, and are elided here; the full set of rules can be found in the accompanying technical report [10].

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

\[
\|\nu : T | \phi\| = T \quad \|\varepsilon : \tau_1 \rightarrow \tau_2\| = \|\tau_1\| \rightarrow \|\tau_2\|
\]

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

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

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

\[
\begin{align*}
\forall i \in \mathbb{Z}, & \quad ty(i) = \text{int} \\
ty(\text{Nil}) & = \{v : \text{int} \mid \phi_v\} \\
\ty(\text{Cons}) & = \{x : \text{int} \rightarrow y : \text{int} \rightarrow \nu : \text{int} \rightarrow \phi_v\}
\end{align*}
\]

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

\[
R \triangleq R^+ \quad \Sigma_R(R_2) = (\text{Nil} \Rightarrow r_1, \text{Cons} x y \Rightarrow r_2) \\
\Sigma_R(\text{Nil}) = (\text{Nil} \Rightarrow r_1, \text{Cons} x y \Rightarrow r_2 \cup R(\text{Nil}))
\]

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 \(\Sigma_R(R) = (\text{Nil} \Rightarrow r_1, \text{Cons} x y \Rightarrow r_2 \cup R(y))\), the notation \(\Sigma_R(R)(\text{Nil})\) denotes \(r_1\), and \(\Sigma_R(R)(\text{Cons} x y)\) denotes \(r_2\). With help of \(\Sigma_R\), we now define \(\phi_v\) and \(\phi_v\) as:

\[
\begin{align*}
\phi_v & = \bigwedge_{R \in \text{dom}(\Sigma_R)} R(\nu) = \Sigma_R(\text{Nil}) \\
\phi_v & = \bigwedge_{R \in \text{dom}(\Sigma_R)} R(\nu) = \Sigma_R(\text{Cons} x y)
\end{align*}
\]

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

\[
\Sigma_R = [R \rightarrow \langle \text{Nil} \Rightarrow R_{id}(0) \rangle \mid \text{Cons} x y \Rightarrow R_{id}(x)]]
\]

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

\[
ty(\text{Nil}) = \{v : \text{int} \mid R(\nu) = \text{Nil}(0)\} \\
ty(\text{Cons}) = \{x : \text{int} \rightarrow y : \text{int} \rightarrow \nu : \text{int} \rightarrow \nu : \text{int} \rightarrow R_{id}(x)\}
\]

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

\[
\text{match } z \text{ with } \text{Cons } x y \Rightarrow e_1 \text{ else } e_2
\]
In our examples, we assign the same names to formal and actual arguments:

\[
\text{Cons} : x : \text{int} \to y : \text{intlist} \to \\
\{ v : \text{intlist} \mid R_{\text{mem}}(v) = R_d(x) \cup R_{\text{mem}}(y) \}
\]

Expression \( e_1 \) is type-checked under the extended environment:

\[
\Gamma, x : \{ v : \text{intlist} \mid \text{true} \}, y : \{ v : \text{intlist} \mid \text{true} \}, \\
R_{\text{mem}}(z) = R_d(x) \cup R_{\text{mem}}(y)
\]

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 \textsc{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 and only 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 \( L \):
  \[
  \Gamma, \nu : T \models \phi_1 \models \phi_2
  \]

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 \( \cdot \vdash e : \tau \), then either \( e \) is a value, or there exists an \( e' \) such that \( e \rightarrow e' \) and \( \cdot \vdash e' : \tau \).

### 3.3 Semantics of the Specification Language

The semantics of our specification language is defined via a translation from well-typed relational expressions and well-formed type refinements to quantified 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 \( \text{int} \), 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 synaptic sugar:

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

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 \( \Gamma_R(R) = \tau_R \).

Fig. 4 describes the MSFOL semantics of \( \lambda_R \)'s specification language. The semantics is operational in the sense that it describes an algorithm to compile assertions in \( \lambda_R \) type refinements to formulas in MSFOL. Our semantics are parameterized over an auxiliary function \( F \) that maps \( \lambda_R \) datatypes to uninterpreted sorts in MSFOL. The specific uninterpreted sorts type map to are not relevant here. However, \( F \) has to be a total function over \( \lambda_R \) datatypes.

Note that despite treating interpreted types (eg: \( \text{intlist} \) and \( \text{int} \)) 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 prenex quantified logic.

As an example, suppose we have

\[
R :: \text{intlist} :: \to \{ \text{int} \}
\]

That is, \( \Gamma_R \) maps \( R \) to \( \text{intlist} ::= \{ \text{int} \} \). Assume that: \( [\text{int}] = A_0 \) and \( [\text{intlist}] = A_1 \).

\[
[R] = \eta_{\text{wrap}}(R, \Gamma_R)(R) \\
\eta_{\text{wrap}}(R, \text{[intlist]} ::= \{ \text{int} \}) \\
\eta_{\text{wrap}}(R, \text{[intlist]} ::= \{ \text{int} \}) \\
\forall(k : A_1).\eta_{\text{wrap}}(R \ k, A_0 \to \text{bool}) \\
\forall(k : A_1).\forall(j : A_0).\eta_{\text{wrap}}(R \ k j, \text{bool})
\]

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:

\[
\forall \exists \phi_1 \cup \forall \exists \phi_2 = \forall \exists (\phi_1 \lor \phi_2) \\
\forall \exists \phi_1 \times \forall \exists \phi_2 = \forall \exists \forall \exists (\phi_1 \land \phi_2)
\]

Our semantics use syntactic rewrite functions - \( \gamma_{\text{li}} \) and \( \gamma_{\text{tr}} \), 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:

\[
R_{\text{ob}}(x) = R_d(x) \times R_{\text{mem}}(x)\text{int}
\]

involving \textit{membership} and \textit{occurs-before} relations for integer lists:

\[
R_{\text{mem}} ::= \text{intlist} ::= \to \{ \text{int} \} \\
R_{\text{ob}} ::= \text{int} \times \text{int}\text{int}
\]

The series of steps that compile the assertion to an MSFOL formula, which captures the semantics of the assertion, are shown in Fig. 5. The example assumes that \( F \) maps \( \text{int} \) to sort \( A_0 \), and \( \text{intlist} \) to sort \( A_1 \).

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

\[
\Gamma, x : \{ v : T \mid \phi \} = \Gamma' \Rightarrow x : \{ T \} \Rightarrow \{ x / \nu \mid \phi \} \\
\Gamma, \phi = \Gamma' \Rightarrow \{ \phi \} \\
\Gamma, \text{true} = \Gamma'
\]

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)** For all \( \phi, \Gamma \), if \( \Gamma \vdash \phi \), then there exists an MSFOL proposition \( \phi' \) such that \( [\phi] = [\phi'] \).

\footnote{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.}
\[ \lambda \text{tended calculus as phism, and the specification language with parametric relations - } \lambda \text{ have sort scheme, from Section 2, when defined over a polymorphic language. For example, the non-parametric head relation (} \forall \text{ of MSFOL is a carefully chosen decidable fragment. This frag-} \text{tic entailment is undecidable for full first-order logic, our subset } \lambda \text{ practical exercise.} \]

\[ \text{Follows from Lemma 3.2 and decidability proof of EPR } \]

\[ \text{THEOREM 3.3. (Decidability) Type checking in } \lambda_R \text{ is decidable.} \]

**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 \( \lambda_{IR} \). Figure 3 shows the type and specification language of \( \lambda_{IR} \). We have elided \( \lambda_{IR} \)'s expression language in the interest of space. Unsyntactic modified forms of \( \lambda_{IR} \) are also elided. The only algebraic data type in \( \lambda_{IR} \) 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 (\( R_{\text{hd}} \)) from Section 2 when defined over a polymorphic \( \text{'a' list} \) will have sort scheme, \( \forall \text{ 'a, 'a list} \rightarrow \text{'a} \). The specification language also contains an expression (\( R \)) 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 \( \lambda_{IR} \) are parameterized over one relational parameter. The domain of a relational parameter to a structural relation over a \( \text{'a list} \) should be \( \text{'a} \). When the type variable in \( \text{'a list} \) is instantiated with, e.g., \( \text{'b list} \), the parameter of a parametric relation over \( \text{'a list} \) can be instantiated with a structural relation over \( \text{'b list} \). For instance, the relational parameter \( R \) in the parametric membership relation (\( R_{\text{mem}} R \)), defined in Sec. 2, can be instantiated with the non-parametric head relation, \( R_{\text{hd}} \), after instantiating \( \text{'a list} \) in its sort scheme with a \( \text{'b list} \). The resulting relation can now be applied to a list of lists (i.e., a \( \text{'b list list} \)) to denote the set of head elements in the constituent lists.

The definition (\( \Delta_R \)) of a parametric relation is a tuple containing its name (\( R \)), the name of its relational parameter (\( R_{\text{pr}} \)), 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 \( R_{\text{hd}} \) relation over \( \text{'a list} \) reproduced from Section 2:

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

\[ \text{ | } (R_{\text{hd}} R) \text{ [] } = \emptyset \]

\( R_{\text{hd}} \) requires that the domain of its parameter be \( \text{'a} \), but it places no restriction on the co-domain of \( R \). In order to have a truly parametric definition of \( R_{\text{hd}} \), it is essential that we let the relational parameter have an unrestricted co-domain. Therefore, we let tuple-sort variables (\( t \)) be used in tuple sorts (\( \theta \)). 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 has to be instantiated. Polymorphism in \( \lambda_{IR} \) is predicative so parameterization over relations in \( \lambda_{IR} \) 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.

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

*Figure 6: \( \lambda_{IR} \) - Language with parametric relations*
relations from the level of type refinements to the level of types. We
map shown below:
\[
\text{foldl} \quad \lambda(x : T), r \quad \text{transformer}
\]
e₆ : bind (R(x), F₆) \quad \text{bind expression}
E₆ : \lambda(x : T). bind (R(x), F₆) \quad \text{bind abstraction}
ψ₆ : R = E₆ \quad \text{bind equation}
Σ₆ : R = \lambda R. E₆ \quad \text{bind definition}

Figure 7: Bind Syntax

To extend the generality of parametric relations to dependent
types of the term language, we lift the parameterization over re-
lations from the level of type refinements to the level of types. We
depend on types parameterized over relations as parametric
dependent types (\(\delta\)). An example of a parametric dependent type
is the type of \textit{foldl} from Section 2. Another example is the type of
map shown below:
\[
\langle \tau R, \tau R \rangle \text{ map} : 1 \rightarrow (f : x \rightarrow \{\nu | \tau R (\nu) = \tau R (x)\}) \rightarrow
\{\nu | ((\text{fob } \tau R) \tau R (\nu) = ((\text{fob } \tau R) \tau R (x)) \ast 1)\}
\]

4.2 Sort and Type Checking

Rules to check sorts of relational expressions and well-formedness
of type refinements (\(\Phi\)) in \(\lambda_{\nu \delta R}\) are straightforward extensions of
similar rules for \(\lambda_{\nu R}\) and are omitted here. Sort-checking a parametric
relation definition reduces to sort-checking a non-parametric rela-
tion 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 general-
ization 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 \(R_{\text{id}}\) 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 \(R_{\text{id}}\) is
always sort-safe. Therefore, there exists a ground relation for every
parametric relation in \(\lambda_{\nu R}\).

Transformer Expression. A transformer expression (\(F_\delta\)) 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
\(\ast \rightarrow \ast\)) from tuple-sort (\(\theta_t\)) to a set sort \(\{\theta_2\}\). An example of a transformer expression of sort
\(\forall a : \{a \ast a\}\) is the reflexive transformer:
\[
\lambda x. R_{\text{id}}(x) \times R_{\text{id}}(x)
\]

Bind Expressions. Consider an operator that accepts a relation
application and a transformer expression (\(F_\delta\)) applies \(F_\delta\) over
every tuple in the set representing a relation application, and sub-
sequently folds the resulting set of sets using set union. Such an
operator has following sort:
\[
\forall t_1, t_2, \{t_1\} : \{t_1 : \{t_2\}\} : \{t_2\}
\]
We name the operator \textit{bind}, after set monadic bind. The syntax
of bind expressions is given in Fig. 7. For brevity, we exclude sort
annotations on bind expressions (\(F_\delta\)) and bind abstractions (\(E_\delta\)) in
our examples.

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 \(\ast \ast \text{ list}\), the bind expression that binds
\(R_{\text{mem}}(l)\) with a reflexive transformer is as following:
\[
\text{bind}(R_{\text{mem}}(l), \lambda x. R_{\text{id}}(x) \times R_{\text{id}}(x))
\]
The result of evaluating this expression is the set of reflexive pairs
of elements in the list, which is equivalent to instantiating \(R_{\text{mem}}\)
with \(R_{\text{id}}:\n
\[
(R_{\text{mem}} R_{\text{id}})(l) = \text{bind}(R_{\text{mem}}(l), \lambda x. R_{\text{id}}(x) \times R_{\text{id}}(x))
\]
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 \(R_{\text{mem}}(R_{\text{id}})\) in terms of its
ground relation \(R_{\text{mem}}\). Indeed, a parametric \(R_{\text{mem}}\) relation (call it
\(R_{\text{mem}}\)) can be defined equivalently in terms of its non-parametric
variant as:
\[
R_{\text{mem}}^n = \lambda R. \lambda l. \text{bind}(R_{\text{mem}}(l), \lambda x. R(x))
\]
We refer to the above definition as the bind definition of parametric
\(R_{\text{mem}}\) relation. Every well-sorted parametric structural relation
definition in \(\lambda_{\nu \delta R}\) 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 re-
quires that we construct its ground relation, and a transformer ex-
pression (\(F_\delta\)). A ground relation definition is derived by instantiat-
ating its parametric definition with \(R_{\text{id}}\), as stated previously. Con-
structing 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 \textit{bind}). A sort variable in the tuple
sort is interpreted as application of its parameter relation, an aster-
isk in the sort translates to a cross-product, and a \(\lambda_{\nu \delta R}\) type in
the tuple sort translates to application of \(R_{\text{id}}\). For instance, consider a
hypothetical parametric relation \(R_e\) with the following sort:
\[
R_e : \forall t. (\text{int} : \{t\}) \rightarrow (\text{int list} : \{\text{int} \ast \text{int} \ast \text{int}\})
\]
We let \(R\) denote the relational parameter of \(R_e\). The ground relation
of \(R\) (call it \(R_{\text{id}}\)) is the instantiated parametric relation \((R_e R_{\text{id}})\),
which has the sort \(\text{int list} : \{\text{int} \ast \text{int} \ast \text{int}\}\). From the type of
\textit{bind}, we know that the sort of the required transformer expression
\((F_\delta)\) is \(\text{int \ast int \ast int} \rightarrow \text{int \ast int \ast int}\). Recalling that \(F_\delta\) is a lambda
bound relational expression, which is a cross product combination
of relation applications (Fig. 7), we observe that the only possible
solution for \(F_\delta\) is:
\[
\lambda x, y, z. R_{\text{id}}(x) \times R(y) \times R(z)
\]
Consequently, we derive the following bind definition of \(R_e\):
\[
\lambda R. \lambda l. \text{bind}(R_e(l), \lambda x, y, z. R_{\text{id}}(x) \times R(y) \times R(z))
\]

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 \((E_\delta)\) in Figure 7, which, like any non-parametric
structural relation in \(\lambda_{\nu R}\), is a map from a \(\ast \ast \text{ list}\) to a set of
tuples. Therefore, an instantiated parametric relation can be treated
as a new non-parametric relation that is defined using \textit{bind}. For
example, \((R_{\text{mem}} R_{\text{id}})\) can be treated as a new non-parametric
relation \(R_t\), defined in terms of \textit{bind}:
\[
\begin{align*}
\text{bind}(R_{\text{mem}}(l), \lambda x. R_{\text{id}}(x) \times R_{\text{id}}(x)) \\
\end{align*}
\]
By rigorously defining the semantics of *bind* equations as above, we can effectively capture the semantics of any instantiation of a parametric relation in terms of its ground relation. This is the insight that allows us to use parametric relations seamlessly in type refinements. For instance, the bind semantics for \((R_{\text{mem}}, R_{\text{dup}})\) lets us prove the following implication, which could potentially arise during subtype checking:

\[
(R_{\text{mem}} \wedge R_{\text{dup}})(l_1) = ((R_{\text{mem}} \wedge R_{\text{dup}})(l_2) \implies R_{\text{mem}}(l_1) = R_{\text{mem}}(l_2))
\]

The formal semantics of bind equations, which also define an algorithm to compile bind equations into MSFOL formulas, is described in Fig. 8. Under our 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 \((x) \in R_{\text{mem}}(l_1)\) and \((y) \in R_{\text{id}}(x) \times R_{\text{id}}(x)\), then \((y) \in ((R_{\text{mem}} \wedge R_{\text{dup}})(l))\).
- If \((y) \in ((R_{\text{mem}} \wedge R_{\text{dup}})(l))\), then there must exist \(x\) such that \((x) \in R_{\text{mem}}(l_1)\) 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 \(((R_{\text{mem}} \wedge R_{\text{dup}})(l))\) in terms of \((R_{\text{mem}}(l))\).

### 4.4 Decidability of Type Checking

Type refinements (\(\Phi\)) in \(\lambda R\) can be elaborated to a conjunction of 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 \(\lambda R\) is decidable.

**Proof.** Follows from the decidability proof of EPR logic, to which bind equations are compiled, and the 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 CATALYST) in MLton [13], a whole-program optimizing compiler for Standard ML (SML). The input to our system is CoreML, an A-normalized intermediate representation with pattern-matching, but with all SML module constructs elaborated and removed. SML programs are annotated with relational specifications, defined in terms of relational dependent types that decorate function signatures, along with definitions of parameterized structural relations over the program’s datatypes. The type system is a conservative extension of SML’s, so all programs that are well-typed under CATALYST are well-typed SML programs. Our type-checking and verification process closely follows the description given in the previous sections. Verification conditions, representing the consequent of the SUBT-BASE type-checking rule (Fig. 2) 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 \(R_{\text{eq}}\). For example, \(R_{\text{notEq}}\) can be defined as the following primitive relation:

\[
R_{\text{notEq}} = \lambda k. \text{ax} \cdot R_{\text{id}}(x) - R_{\text{id}}(x)
\]

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

\[
R_{\text{eq}} = \lambda k. \text{ax} \cdot R_{\text{id}}(x) - (R_{\text{id}}(x) - R_{\text{id}}(x))
\]

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:

\[
id : x \rightarrow \{ v \mid R_{\text{id}}(v) = R_{\text{id}}(x) \}
\]

The type refinement used here is an intuitive way of expressing the simple fact that \(\text{id}\) returns its argument. To avoid such needless verbosity, we admit non-relational assertions (called *base 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 [18], 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.

---

8 The source code for the implementation as well as a Web interface to the system is available online from: https://github.com/tycon/catalyst.
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. An example of such annotation is shown in Fig. 10c (the contains function).

5.1 Experiments

We have investigated the automatic verification of expressive shape invariants using CATALYST on a number of programs, including:

1. List library functions, such as concat, rev, revAppend, foldl, foldr, zip, unzip etc. (some of these specifications have been discussed in Sec. 3 and 4), and
2. Okasaki’s red-black tree [17] library functions, such as balance, multiple order traversal functions, and mirrorImage.
3. Compiler transformations over MLton’s SSA (Static Single Assignment) intermediate representation.

For several of these benchmarks (especially those in (1) and (2)), CATALYST was able to successfully verify specifications to the extent of full functional correctness. Excluding the time take by the MLton compiler to elaborate and type check these Standard ML programs, none of our benchmarks take more than 0.2s to verify; this time includes A-Normalization, specification elaboration, VC generation, and SMT solving through Z3.

Red-Black Tree. The specification of the red-black tree balance function, shown in Fig. 9a 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 a total-order among members of the tree. The non-inductive total-order relation \( R_{to} \) in Fig. 9b is defined in terms of the tree membership relation \( R_{mem} \) described in Sec. 2 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. The inductive total-order relation \( R_{to}^* \) on a red-black tree, obtained by closing the \( R_{to} \) relation over the tree, relates every pair of elements in the tree that are in-order. Consequently, the specification of the balance function effectively asserts that in-order traversal over an unbalanced red-black tree, and in-order traversal on its balanced version, return the same sequence of elements.

CATALYST can verify full functional correctness of standard tree traversal functions that return a list of elements. The relational specifications for such functions essentially relate different order relations on the input tree to an 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 the same as that of \( t \)’s inductive total-order relation:

\[
\text{inOrder} : t \rightarrow \{ l \mid R_{to}^*(l) = R_{to}^*(t) \}
\]

SSA. An important intermediate representation used in MLton is a variant of SSA that is operated upon by a number of optimization passes. After each such pass, MLton checks the well-formedness of the output by checking, for example, that variable definitions dominate variable uses in the SSA dominator tree. Because MLton performs this check after every optimization pass, compile times can suffer, especially as program size scales. A potential application of CATALYST is to statically typecheck the integrity of SSA optimization passes, thereby eliminating this overhead.

A program in SSA form is represented as a tree of basic blocks, where each block consists of a set of straight-line instructions (e.g., definitions, assignments, primitive applications). The specification of an SSA program makes use of several inductive relations: \( R_{du} \), the def-use relation, \( R_{ud} \), the use-def relation, and \( R_{use-ref} \), the refl ective variant of SSA. An important intermediate representation used in MLton is the re-}


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

(a) balance

Figure 9: Red-Black Tree Example

\[ (*) \text{ Tree head (root) relation } (*) \]
relation \( R_{hd} \) \( (T(c,l,n,r)) = \{ (n) \} \)

\[ (*) \text{ Tree membership relation } (*) \]
relation \( R_{mem} = R_{hd}^* \):

\[ (* \text{ Total-order relation among tree members } *) \]
relation \( R_{to} \) \( (T(c,l,n,r)) = R_{mem} (1) \times \{ (n) \} \)
关系 \( \cup (\{ (n) \} \times R_{mem} (r)) \)

\[ (* \text{ "balance" preserves the total-order among members of the tree } *) \]
balance : t \rightarrow \{ t' \mid R_{to}^*(t') = R_{to}^*(t) \}

(b) Relational specification of balance

\[ (*) \text{ Tree head (root) relation } (*) \]
relation \( R_{hd} \) \( (T(c,l,n,r)) = \{ (n) \} \)

\[ (*) \text{ Tree membership relation } (*) \]
relation \( R_{mem} = R_{hd}^* \):

\[ (* \text{ Total-order relation among tree members } *) \]
relation \( R_{to} \) \( (T(c,l,n,r)) = R_{mem} (1) \times \{ (n) \} \)
关系 \( \cup (\{ (n) \} \times R_{mem} (r)) \)

\[ (*) \text{ "balance" preserves the total-order among members of the tree } *) \]
balance : t \rightarrow \{ t' \mid R_{to}^*(t') = R_{to}^*(t) \}

(c) elements in the left sub-tree to those in right. The inductive total-order relation \( R_{to}^* \) on a red-black tree, obtained by closing the \( R_{to} \) relation over the tree, relates every pair of elements in the tree that are in-order. Consequently, the specification of the balance function effectively asserts that in-order traversal over an unbalanced red-black tree, and in-order traversal on its balanced version, return the same sequence of elements.

\[ \text{inOrder} : t \rightarrow \{ l \mid R_{to}^*(l) = R_{to}^*(t) \} \]

This type captures the two essential structural properties of SSA: (1) every use of a variable must be dominated by its definition; and (2) no definition of a variable is ever dominated by its use. Verifying that a transformation pass over the SSA IR has the type:

\[ \text{ssa_tree} \rightarrow \text{ssa_tree} \]

is tantamount to proving the transformation preserves the salient SSA invariant that definitions always dominate uses.

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 \( \text{filter} \) and \( \text{contains} \), directly, and \( \text{exists} \) through \( \text{contains} \). 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:

\[ \text{exists}. \] Consider the higher-order \( \text{exists} \) function over lists shown in Fig. 10a. Dependent type signatures are elided for brevity.
fun exists f l = case l of
  [] => false
  | x::xs =>
    let
      val v1 = exists f xs
    in
      if f x then x::xs' else xs'
    end
end

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

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

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

Figure 10: Examples

Relational Specification

relation R_{f_0} (Var x) = \{(x)\}
| R_{f_0} (App (e1,e2)) = R_{f_0}(e1) \cup R_{f_0}(e2)
| R_{f_0} (Abs (id,e)) = R_{f_0}(e) - \{(id)\};

createNewName : fvs \rightarrow \{v \mid not (v = id) \land not (\{(v)\} \subseteq R_{mem}(fvs))\};
freeVars : e \rightarrow \{(v) \mid R_{mem}(\{v\}) = R_{f_0}(e)\};
alphaConvert : e \rightarrow \{(ex) \mid R_{f_0}(ex) = R_{f_0}(e)\};
subst : e1 \rightarrow id \rightarrow e2 \rightarrow
\{ex | if (((\{(id)\}) \subseteq R_{f_0}(e2)) then R_{f_0}(ex) = (R_{f_0}(e2) - \{(id)\}) \cup R_{f_0}(e1) else R_{f_0}(ex) = R_{f_0}(e2)\};

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:

\langle \forall R. \exists x. \forall f. (f: x \rightarrow \{v \mid v = true \iff R(x) \neq \emptyset\}) \rightarrow \{v \mid v = true \iff (R_{mem} \ast R(v)) \neq \emptyset\}\rangle

The intuition behind this type is same as that of exists. Filter retains only those elements for which its higher-order argument returns true.

contains. Consider the definition of the contains function shown in Fig. 10b 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:

val isStr = fn x => x=str

the relational dependent type of isStr is:

\langle \forall \nu. (\nu \mid \nu_{eqap}(\nu) \neq \emptyset)\rangle

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

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

hasStr : \{\nu \mid \nu_{eqap} \iff (R_{eqap} \ast \nu) \neq \emptyset\}
The type refinement for hasStr indicates that hasStr is true if and only if the set of all elements of list 1 that are equal to str is not empty. Due to the equivalence of its first-order encoding to that of the following assertion:

\[ \{ \nu | \nu = \text{true} \Leftrightarrow R_{\text{id}}(\text{str}) \subseteq R_{\text{mem}}(1) \} \]

the implementation of contains type-checks against the type:

\[ 1 \rightarrow \text{str} \rightarrow \{ \nu | \nu = \text{true} \Leftrightarrow R_{\text{id}}(\text{str}) \subseteq R_{\text{mem}}(1) \} \]

6.1 \( \alpha \)-conversion

The substitution operation \( \text{subst} \) substitutes a free variable \( \text{id} \) in an expression \( e2 \) with another expression \( e1 \). Function alphaConvert consistently renames occurrences of the bound variable in an abstraction expression. Observe that \( \text{subst} \) 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 \([6,26]\). 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 \( \text{id} \), or substitutes a bound occurrence of the variable; or (c) substitution is not capture-avoiding, i.e., substituting \( e1 \) for \( \text{id} \) 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. Note that one need not expose notions of capture-avoidance, or other such intricacies, to write down the specification, which is given in terms of a new structural relation \( R_{\text{id}} \) 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 \( \text{subst} \) and alphaConvert do not constrain how these functions are realized in ML. For instance, an implementation of \( \text{subst} \) 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 \([12,13]\) using GADTs \([22]\). 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* \([23]\). Type refinements in F* are drawn from unrestricted (higher-order) logic extended with theories, 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 gained by allowing unrestricted type refinements in F* comes at the cost of decidability of type checking.

Structural relations, in their operational manifestation, can be compared to the structurally recursive measures of liquid types \([11,25]\) 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 \([23]\), 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 flat 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. \([22]\) 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 \([3]\), and to describe memory configurations \([9]\). However, their applicability has been limited owing to destructive updates and pointer manipulations in imperative programs. In \([13]\), 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 \([1,7]\). Whereas a logical relation relates two terms of (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 \([2,4]\). In these efforts, a binary logical relation that relates effectful expressions is parameterized by a relation that relates their states. In contrast, a parametric structural relation is a structural relation over a polymorphic data type, that is parameterized 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 \([8]\) 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. 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 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 un-
delying 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 \((\text{op} +)\) 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 such a way that the combination remains decidable. We intend to explore both these extensions as part of future work.

One noticeable limitation of our current system is the lack of a general type inference mechanism. Given that relational specifications which make use of parametric relations to express rich invariants are non-trivial, and can be quite verbose, writing such specifications sometimes 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 \[20\] 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, and the anonymous reviewers for their detailed comments and suggestions. This work is supported by the National Science Foundation under grants CCF-1216613 and CCF-1318227.

References