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 construct relational abstractions for data structures, which let us automatically verify their structural invariants. Our specification language allows definitions of parametric relations for polymorphic data types. Such parametrization leads to highly composable specifications and naturally generalizes our approach to higher-order polymorphic functions.

We describe an algorithm that translates relational specifications into a decidable first-order logic that can be efficiently discharged by an SMT solver. The implementation of these ideas is manifest in a type checker called CATALYST that is incorporated within the MLton SML compiler. We present examples and a case study to demonstrate the expressive power of our specification language and effectiveness of our type checking and translation mechanisms in supporting such expressivity. Our experimental results indicate that our verification strategy is also practical.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

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 [**?**], F* [**?**], and OCaml [**?**]. 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`, they still underspecify the intended behavior of these operations. For example, a correctly written `append` function must additionally preserve the order of its input lists; a function that incorrectly produces an output list that is a permutation of its inputs would nonetheless satisfy `append`'s type as written above. Similarly, the identity function on lists would clearly satisfy the type given for `rev`; a type that fully captures `rev`'s behavior would also specify that that the order of elements in `rev`'s output list is the inverse of the order of its input. While being precise enough to capture the intended behaviour of `rev`, it should also be general enough to allow for any correct implementation of `rev` to type check. Is it possible to ascribe such an expressive type to `rev`, that can nonetheless be easily described, and efficiently checked?

One approach is to directly state the desired behaviour of functions in type refinements. A straightforward refinement type of `rev` that captures its behaviour could be:

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

where `rev'` represents some reference implementation of `rev`. However, the problem of verifying program equivalence is undecidable in general; so, in order for such types to be machine checkable, the definition of `rev'` should closely resemble that of `rev`. Conversely, an implementation of `rev` is well-typed only if its syntactic structure resembles that of `rev'`. This situation is undesirable as it leads to many correct implementations failing the type check. An alternative approach is to design `rev'` in a way that can be reasoned by an automatic theorem prover. To be feasible, this would require the meaning of (potentially arbitrary) recursive functions like `append` that may be used by `rev'` be understood by the underlying theorem prover used to verify this property. Clearly, unlike the case with length of a list, challenges in building suitable definitions that let us reason about shape properties of a list are substantial.

Yet, the way length of a list is tracked using its length-indexed list type offers a hint of 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 index a list type to help us reason about its 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 list. 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, asserting, for example, that one occurs before the other. By defining an exhaustive enumeration of the set of all such pairs, 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 of the list.

For example, the transitive closure of an *occurs-before* relation, denoted as R^*_{ob} for a concrete list `l=[x1,x2,x3]` is the following set of tuples:

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

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`:

`rev : { l : 'a list} ⟶ {ν: 'a list | R*ob(l) = R*oa(ν)}`

The above specification asserts that for every pair of elements x and y in the input list `l`, if x occurs before y in `l`, then it has to occur after y in the result list ν. This property succintly 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`, thus circumventing the limitations that plague such approach.

We refer to operators like R_{ob} and R_{oa} as *structural relations* because they explicitly describe structural properties of a data structure. As demonstrated with `rev` example, such relations can be used as appropriate abstract domains to reason about the shapes of structures generated by constructor applications in algebraic datatypes. 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 full 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. Consider the `dup` function with following ML type:

```
dup : 'a list -> ('a*'a) list
```

Function `dup` duplicates elements in its input list. An invariant that we can expect from `dup` is that the order of left components of pairs in result list is same as the order of its right components, and both are equal to the order of elements in the input list (`'a list`). Clearly, our definitions of "*occurs-before*" and "*occurs-after*" as binary 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 their 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 taking the cue from parametric polymorphism. We allow structural relations defined over a polymorphic datatype to be parameterized by relations over type variables in the datatype. For instance, R_{ob} relation over `'a list` can be parametrized by a relation R over `'a` leading to parametric *occurs-before* relation. Instead of directly relating the order of two elements x and y in a given `'a list`, a parametric *occurs-before* relation could generically relate the ordering of the elements of two relations - $R(x)$ and $R(y)$; R's specific instantiation would draw from the set of relations defined over the datatype that instantiates the type variable (`'a`).

For example, assume that R_{fst} is a unary singleton relation defined over pairs such that: $R_{fst}((x, y)) = \{\langle x\rangle\}$. Then, the relational parameter (R) of *occurs-before* can be instantiated with R_{fst} to generate a relation that relates `x1` to `x2` if and only if, for some `x3` and `x4`, `(x1,x3)` occurs before `(x2,x4)` in a list of pairs. Dually, Instantiating R with R_{snd}, where $R_{snd}((x, y)) = \{\langle y\rangle\}$, generates a relation that would let us reason about order of right components in a list of pairs. It can be noted that such instantiated relations are sufficient to express the behavior of `dup`. Not surprisingly, 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 powerful specification language based on relational algebra to define and compose structural relations for any algebraic datatype. These definitions are only as complex as the datatype definition itself in the sense that it is possible to construct equivalent relational definitions directly *superimposed* on the datatype. Relations thus defined, including their automatically generated inductive variants, can be used in type refinements to specify shape invariants, and other such relational properties. Our typechecking procedure verifies specifications by interpreting constructor applications as set operations within these abstract relational domain. Typechecking in our system is decidable, a result which follows from the completeness of encoding our specification language in 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 datatypes, 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 parameteric specification can be understood intuitively in second-order logic, we prove 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, providing 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 equipped with support for verifying refinements over structural relations that is incorporated within the MLton SML compiler. We demonstrate the utility of these ideas through a number of benchmarks, and a case study of automatically verifying α-conversion, and capture-avoiding substitution operations of an implementation of untyped lambda calculus.

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 formalization to polymorphic core language, and parametric refinements. Our formalization also presents our translation scheme to translate relational refinements to decidable first-order logic. Details about the implementation, and its performance on benchmarks are given in Section 5. Section 6 presents our case study of automatically verifying untyped lambda calculus implementation. Sections 7 and **??** discuss related work and conclusions, respectively.

2. Structural Relations

Our specification language is primarily the language of relational expressions composed using familiar relational algebraic operators. The language is also equipped with pattern matching over con-

structors of algebraic types to define expressive shape properties in terms of these relational expressions. The language additionally comes equipped with a family of polymorphic relations, the most important of which are listed below:

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

R_{null} does not relate any elements; so it is an empty set. R_{id} is the identity relation, R_{dup} is a relation that associates a value with a pair that duplicates that value, and R_{empty} is an empty relation expressed in such a way that it relates every element to nothing. R_{notEq_k} is a relation indexed by a constant k that relates x to itself, provided x is not equal to k. R_{eq_k} is defined similarly, except it relates x to itself exactly when x equal to k.

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:

$$\texttt{relation}\ R_{hd}\ (x :: xs)\ = \{\langle x\rangle\} \mid R_{hd}\ \texttt{[]}\ = R_{null}$$

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, (1) $(x :: xs, x) \in R_{hd}$, and (2) there does not exist a pattern variable x' such that $x' \neq x$, but $(x :: xs, x') \in R_{hd}$. 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, consider the problem of verifying that an implementation of `rev` preserves the set of elements in its input list. To specify this property, we need a *membership* relation (R_{mem}) that relates a list l to every element in l (regardless of l's size), so that we can assert that if an element is in the membership relation with `rev`'s input list, it must be in a membership relation with the output list the function produces. A recursive definition of R_{mem} could look like:

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

Notably, the above definition is actually an inductive extension of the simple 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:

- $R(l, v) \Rightarrow R^*(l, v)$
- if $l = x :: xs$, then $R(xs, v) \Rightarrow R^*(l, v)$

The above definition is equivalent to the following:

$$R^*(\,x :: xs\,)\ =\ R(\,x :: xs\,) \cup R^*(xs)$$

Using this operator, we can now succintly define a membership relation R_{mem} as:

$$\texttt{relation}\ R_{mem} = R^*_{hd}$$

This relation can now be used to specify the element preservation property for `rev` described informally above. We can think of the induction operator as a controlled abstraction for structural recursion. Based on the recursive structure of an algebraic datatype, sophisticated inductive definitions can be generated from simple structural relations defined for that datatype.

Equipped with R_{mem} we can now precisely define the *occurs-before* relation defined earlier. Recall that relation R_{ob} relates a list to a list of pairs whose first element is the head of the list, and whose second elements are members comprising the tail:

$$\texttt{relation}\ R_{ob}(x :: xs)\ =\ \{\langle x\rangle\} \times R_{mem}(xs) \mid \texttt{[]} = R_{empty}\ \texttt{[]}$$

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

$$\texttt{relation}\ R_{oa}(x :: xs)\ =\ R_{mem}(xs) \times \{\langle x\rangle\} \mid \texttt{[]} = R_{empty}\ \texttt{[]}$$

2.2 Parameterized Relations

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

```
zip : 'a list -> 'b list -> ('a *'b) list
```

We wish to assert that any correct implementation of `zip` must guarantee that the elements of the output list are pairs of elements drawn from both argument lists. The R_{mem} relation defined above provides much of the functionality we require; intuitively, we would like to assert that the first element of every pair in the output list be in a membership relation with the `zip`'s first argument, and the second element of every pair be in a membership relation with `zip`'s second argument. Unfortunately, as currently defined, R_{mem} operates directly on the pair elements comprising the output list, not the pair's individual components. What we require is a mechanism in our relational specification framework that allows R_{mem} to assert the membership property on the pair's components (rather than the pair directly) with respect to either the first or second input list argument.

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

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

`zip` can be assigned the following type:

$$\begin{array}{l} \texttt{zip} : l_1\ \texttt{->}\ l_2\ \texttt{->} \\ \quad \{\ l\ \mid\ (R_{mem}\ (R_{fst}\ R_{id}))\ l = (R_{mem}\ R_{id})\ l_1 \\ \qquad \wedge\ (R_{mem}\ (R_{snd}\ R_{id}))\ l = (R_{mem}\ R_{id})\ l_2\ \} \end{array}$$

The refinement on `zip`'s output list l asserts that the list constructed by projecting the first (resp. second) element from l's pairs contains the same elements as l_1 (resp. l_2), thus accurately capturing the invariant that `zip`'s output pairs are only constructed from its input lists.

As a more substantial example, the following signature for `foldl` relates the membership properties on its input list to its output:

$$('R_1, 'R_2)\ \texttt{foldl}\ :$$

```
{l : 'a list} -> {b : 'b} ->
  ({f: {x :'a} -> {acc : 'b} ->
       {z : 'b | 'R2(z) = 'R1(x) ∪ 'R2(acc)}}) ->
{ν | 'R2(ν) = ((R_mem 'R1) l) ∪ 'R2(b)}
```

To see how we might instantiate relational variables $'R_1$ and $'R_2$ that are intended to capture shape constraints on elements of the input list and output, resp., consider the function `makeTree` that uses `foldl` to generate a binary tree using function `treeInsert` (not shown):

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

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

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

In this example, relation variables $'R_1$ and $'R_2$ in `foldl`'s definition are instanitated to R_{id} and R_{tmem}, resp. The resulting refinement for `foldl`'s result thus requires that the elements in the output tree be the same as elements in the input list. Similarly, the refinement on the result type of the higher-order argument f requires that the output tree returned by f must contain the elements in the tree produced by the accumulator and the elements in the input list whose tree elements have not yet been constructed.

We can construct an alternative type for `foldl` that relates the order of elements in the argument list to some order of the result. The intuition is as following: let us say the result type (`'b`) has some notion of order and a relation to describe the order 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 list, and `f` is applied over elements of the 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 (abstracted by relational variable $'R$ below), given an assertion that describes the order in which `f` is applied over the list. To simplify the presentation, we only parameterize the ordering relation in `foldl`'s signature shown below, and elide the parameteric refinements dealing with membership as described above:

```
('R) foldl :{l : 'a list} -> {b : 'b} ->
              ({f :{x : 'a} -> {acc : 'b} ->
                   {z | 'R3(z) = (⟨x⟩ × 'R_mem(acc)) ∪
                                 'R3(acc)}) ->
            {ν | 'R3(ν) = R*_oa (l) ∪ 'R3(b)) ∪
                          (R_mem(b) × (R_mem (l)))
```

The following program is successfuly typechecked by our implementation given the standard definition of `foldl`, thus verifying that `rev` inverts the order of its input list, while `id` preserves list order.

```
val Cons = fn x => fn xs => x::xs
val Id = fn x => fn xs => xs  [x]
rev : {l : 'a list} -> {v : 'a list | R*_ob(v) = R*_oa(l)}
val rev = fn l => foldl (R*_ob) l [] Cons
val id = fn l => foldl(R*_oa) l [] Id
```

Calculus λ_R

x, y, z, ν	$\in$	*variables*	
C	::=	Cons | Nil	*constructors*
v	::=	$\lambda(x:\tau).\, e \mid v:\tau \mid C\,\overline{v}$	*value*
e	::=	$x \mid e\ x \mid$ `let` $x = e$ `in` e $\mid$ `match` x `with` $C\,\overline{x} \Rightarrow e$ `else` e $\mid e:\tau$	*expression*
T	::=	unit | int | intlist	*datatypes*
τ	::=	$\{\nu : T \mid \phi\} \mid x:\tau \to \tau$	*dependent types*

Specification Language

R	$\in$	*relation names*	
ϕ	::=	$r = r \mid r \subset r \mid \phi \wedge \phi \mid \phi \vee \phi \mid \top$	*type refinement*
r	::=	$R(x) \mid r \cup r \mid r \times r$	*relational expression*
Σ_R	::=	$\langle R, \tau_R, \overline{C\,\overline{x} \Rightarrow r}\rangle$ $\mid \langle R, \tau_R, R^*\rangle$	*relation definition*
θ	::=	$T \mid T * \theta$	*tuple sort*
τ_R	::=	intlist $:\to \{\theta\}$	*relation sort*

Figure 1: Language

3. Core langugage

3.1 Syntax

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

Dependent type refinements (ϕ) in λ_R are assertions over relational expressions(r); these expressions, which are themselves typed, constitute the syntactic class of expressions in our specification language. We refer to types of relational expressions as sorts, in order to distinguish them from types of λ_R expressions. A structural relation definition is a triple, consisting of a relation name, its sort, and either (a) a pattern-match sequence that relates constructors of an algebraic datatype to a relation expression, or (b) an inductive extension of an existing relation, captured using the closure operator (*).

Relational definitions are sort-checked and elaborated into a dependent type refinement for constructors of the domain type. A structural relation maps a value to a set of tuples (θ). We use "$:\to$" to distinguish such maps from the mapping expressed by dependent function types. For example, the notation

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

indicates that the sort of relation R_{ob} is a map from integer lists to pairs. For the purposes of the formalization, we assume the existence of a single primitive relation R_{id} whose sort is `int` $:\to$ {`int`} that defines an identity relation on integers.

Elaboration rules for relational definitions are given in Figure 2. The rules abuse the Σ_R notation, treating it as a map from relation names to their definitions. Rule E-Base populates type environment Γ'; rules E-Cons and E-Nil collect and aggregate type refinements from list constructor applications. Rule E-Rel elaborates the relational expressions found in a definition's pattern-match sequence to type refinements of Nil and Cons using rules E-Cons and E-Nil. Rule E-Rel-Star first unrolls the inductive definition of its argument relation (R_2 in the rule) before elaboration.

$\boxed{\Gamma \looparrowright \Gamma'}$

E-BASE

$$\frac{\begin{array}{c}\Gamma = \cdot,\ \texttt{Nil} : \texttt{intlist} \\ \Gamma' = \Gamma,\ \texttt{Cons} : \texttt{int} \to \texttt{intlist} \\ \to \texttt{intlist}\end{array}}{\cdot \looparrowright \Gamma'}$$

E-CONS

$$\frac{\begin{array}{c}\Gamma' = \Gamma,\ \texttt{x} : \texttt{int}\,,\ \texttt{xs} : \texttt{int} \\ \Gamma' \vdash \texttt{Cons x xs} \ :\ \{\nu : \texttt{intlist} \mid \phi\} \quad \Gamma' \vdash r \\ \Gamma'' = \cdot,\ \texttt{Cons} : \texttt{x} : \texttt{int} \to \texttt{xs} : \texttt{intlist} \\ \to \{\nu : \texttt{intlist} \mid \phi \wedge (\texttt{R}(\nu) = \texttt{r})\}\end{array}}{\Gamma \oplus \langle\, \texttt{Cons}\,, R, r\rangle \looparrowright \Gamma, \Gamma''}$$

E-NIL

$$\frac{\begin{array}{c}\Gamma \vdash r \\ \Gamma \vdash \ \texttt{Nil} : \{\nu : \texttt{intlist} \mid \phi\} \ : \\ \Gamma' = \cdot,\ \texttt{Nil} : \{\nu : \texttt{intlist} \mid \phi \wedge (\texttt{R}(\nu) = \texttt{r})\}\end{array}}{\Gamma \oplus \langle\, \texttt{Nil}\,, R, r\rangle \looparrowright \Gamma, \Gamma'}$$

E-REL

$$\frac{\begin{array}{c}\Sigma_R(R) = \langle\, \texttt{Nil} \Rightarrow r_1,\ \texttt{Cons x xs} \Rightarrow r_2\rangle \\ \Gamma \vdash r1 :: \theta \quad \Gamma \vdash r2 :: \theta \quad \Gamma' = \Gamma,\ R :: \text{intlist} :\to \{\theta\} \\ \Gamma \oplus \langle\, \texttt{Nil}\,, R, r_1\rangle \oplus \langle\, \texttt{Cons}\,, R, r_2\rangle \looparrowright \Gamma'\end{array}}{\Gamma \looparrowright \Gamma'}$$

E-REL-STAR

$$\frac{\begin{array}{c}\Sigma_R(R_1) = R_2^* \quad \Sigma_R(R_2) = \langle\, \texttt{Nil} \Rightarrow r_1,\ \texttt{Cons x xs} \Rightarrow r_2\rangle \\ \Gamma \vdash R_2 :: \tau_R \quad \Gamma' = \Gamma,\ R_2 :: \tau_R \\ \Gamma \oplus \langle\, \texttt{Nil}\,, R_1, r_1\rangle \oplus \langle\, \texttt{Cons}\,, R_1, r_2 \cup R_1(xs)\rangle \looparrowright \Gamma'\end{array}}{\Gamma \looparrowright \Gamma'}$$

Figure 2: Elaboration rules for structural relations

Sort Checking Specification Language $\boxed{\Gamma \vdash r :: \{\theta\}}$

S-REL

$$\frac{(R :: \tau_R) \in \Gamma}{\Gamma \vdash R :: \tau_R}$$

S-APP

$$\frac{\Gamma \vdash R :: T :\to \{\theta\} \quad \Gamma \vdash x : T}{\Gamma \vdash R(x) :: \{\theta\}}$$

S-UNION

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

S-CROSS

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

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

WF-RPRED

$$\frac{\begin{array}{c}\odot \in \{=, \subset\} \\ \Gamma \vdash r_1 :: \theta \qquad \Gamma \vdash r_2 :: \theta\end{array}}{\Gamma \vdash r_1 \odot r_2}$$

WF-REF

$$\frac{\begin{array}{c}\odot \in \{\wedge, \vee, \Rightarrow\} \\ \Gamma \vdash \phi_1 \qquad \Gamma \vdash \phi_2\end{array}}{\Gamma \vdash \phi_1 \odot \phi_2}$$

WF-BASE

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

WF-FUN

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

Sub-typing $\boxed{\Gamma \vdash \tau_1 <: \tau_2}$

SUBT-BASE

$$\frac{\begin{array}{c}\Gamma \vdash \{\nu : T \mid \phi_1\} \qquad \Gamma \vdash \{\nu : T \mid \phi_2\} \\ [[\Gamma, \nu : T]]_L \wedge [[\phi_1]]_L \models_L [[\phi_2]]_L\end{array}}{\Gamma \vdash \{\nu : T \mid \phi_1\} <: \{\nu : T \mid \phi_2\}}$$

SUBT-ARROW

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

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

T-VAR

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

T-APP

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

T-ABS

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

T-SUB

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

T-LET

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

T-MATCH

$$\frac{\begin{array}{c}\Gamma \vdash C : (\overline{x : \tau_x} \to \{\nu : T_x \mid \phi_c\}) \quad \Gamma \vdash y : \{\nu : T_x \mid \phi\} \\ fresh(z) \quad \Gamma' = \overline{x : \tau_x},\ z : \{\nu : \mathsf{unit} \mid [y/\nu]\phi_c\} \quad \Gamma, \Gamma' \vdash e1 : \tau \quad \Gamma \vdash e2 : \tau\end{array}}{\Gamma \vdash \texttt{match}\ y\ \texttt{with}\ C\,\overline{x} \Rightarrow e_1\ \texttt{else}\ e_2 : \tau}$$

Figure 3: Static semantics of λ_R

3.2 Sorts, Types and Well-formedness

Figure 3 defines rules to check sorts of relational expressions and establish well-formedness of type refinements. Both judgements make use of environment Γ, defined as follows:

$$\Gamma \ ::= \ \cdot \ \mid \ \Gamma,\, R :: \tau_R \ \mid \ \Gamma,\, x : \tau$$

We use Γ to maintain sort bindings for relations, and type bindings for λ_R variables. As evident in the definition, Γ is an ordered environment. Furthermore, we assume that any syntactic object is bound only once in Γ. Any later binding for an object in Γ overrides the previous binding.

It should be noted that domain of a relation is always a primitive type. A relation application is sensible if its argument has the

required type. The rule S-APP captures this semantics. The rules for union and cross-product are straightforward.

Dependent type checking rules for λ_R expressions are shown in Figure 3. The rule to type check `match` expression first ensures that actual arguments to the constructor (C) in pattern match with its formal arguments. Next, the (un-refined) co-domain type of the constructor (C) is checked against that of test variable (y). It then type checks expression in `else` branch under the same type environment (Γ), but `with` branch under an extended environment (Γ'). The extended environment contains bindings for matched pattern variables, and records guard condition as type refinement of a fresh dummy variable (z). For instance, consider the following match expression:

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

where y is an `intlist`, and Cons has type[1]:

$$\texttt{Cons : x:int} \rightarrow \texttt{xs:intlist} \rightarrow \{\nu\texttt{:intlist | Rmem(l) = RId(x)} \cup \texttt{Rmem(xs)}\}$$

When type checking the expression under the environment Γ, e_2 is type checked under the original environment (Γ), but e_1 is checked under the extended environment:

$$\Gamma,\ x : \{\nu : \texttt{int} \mid \top\},\ xs : \{\nu : \texttt{intlist} \mid \top\},\ z : \{\nu : 1 \mid \texttt{Rmem}(y) = \texttt{RId}(x) \cup \texttt{Rmem}(xs)\}$$

Rule T-SUBT lets the type system take advantage of sub-typing by allowing safe type cast from sub-type to super-type. The cornerstone of subtyping is the sub-typing judgement between base dependent types defined by the rule SUBT-BASE. Intuitively, the rule adjudges a dependent type τ_1 to be a sub-type of τ_2, if:

- Their base types match, and,
- Given a logical system L, the conjunction of the interpretation of type environment (Γ) in L and the interpretation of type refinement ϕ_1 (of τ_1) in L *semantically entails* the interpretation of type refinement ϕ_2 (of τ_2) in L.

We chose Many-Sorted First Order Logic to give interpretations to λ_R type refinements and type environment (Γ). Sub-section 3.4 describes our semantics in detail.

3.3 Properties of Type Checking

With sub-typing factored out as a separate judgement, type system of λ_R is similar to SMT-base dependent type systems found in the literature [**? ? ?**], and enjoys the usual progress and preservation properties:

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

LEMMA 3.2. (**Preservation**) *if* $\cdot \vdash e : \tau$ *and* $e \longrightarrow e'$*, then* $\cdot \vdash e' : \tau$

THEOREM 3.3. (**Type Safety**) *if* $\cdot \vdash e : \tau$*, then there exists a value* ν *such that* $e \longrightarrow * \nu$*.*

3.4 Semantics of Specification Language

We define semantics of our specification language by translating well-typed relational expressions and well-formed type refinements to propositions of many-sorted first-order logic (MSFOL).

Many-sorted first-order logic extends first-order logic (FOL) with sorts (types) for variables. For our purpose, we only consider the extension with booleans and uninterpreted sorts, i.e., sorts that, unlike `int`, do not have an attached interpretation. Ground terms, or quantifier-free formulas, of MSFOL are drawn from the propositional logic with equality and n-ary uninterpreted functions. Ascribing MSFOL semantics to λ_R specification language involves interpreting a λ_R variable as an MSFOL variable, a λ_R primitive type as an uninterpreted sort, and a relation as an uninterpreted function with boolean co-domain. Subsequently, set operations, such as union and cross-product, naturally translate to quantified assertions involving conjunctions and disjunctions of uninterpreted function applications. As an example, consider the following assertion involving relation `Rmem` from `intlist` to $\{\texttt{int}\}$:

$$\texttt{Rmem(l) = RId(x)} \cup \texttt{Rmem(xs)}$$

Conjunctions and disjunctions of such assertions make up type refinements λ_R. To ascribe semantics to the above assertion in MSFOL, we start by interpreting `intlist` type as an uninterpreted sort A_0, and `int` as uninterpreted sort A_1. Variables `l` and `xs`, and `x` are interpreted as similarly named variables of sort A_0 and A_1, respectively. Relation `Rmem :: intlist` $:\rightarrow \{\texttt{int}\}$ is treated as a binary uninterpreted function $Rmem$ from $A_0 \times A_1$ to booleans. Likewise, `RId :: int` $:\rightarrow \{\texttt{int}\}$ is interpreted as a binary uninterpreted function RId from $A_1 \times A_1$ to booleans. Under these interpretations, the assertion itself tranlates to the following quantified MSFOL formula:

$$\forall(k : A_0).\ Rmem(l, k) \Leftrightarrow RId(x, k) \vee Rmem(xs, k)$$

Special relation `RId` defines identity relation; so, $RId(x, k)$ if and only if $x = k$. We use this to rewrite the MSFOL formula to arrive at the meaning of the assertion we wrote:

$$\forall(k : A_0).\ Rmem(l, k) \Leftrightarrow x = k \vee Rmem(xs, k)$$

By similar reasoning, one would interpret the meaning of the assertion:

$$\texttt{Rob(l) = RId(x)} \times \texttt{Rmem(xs)}$$

as the following MSFOL formula:

$$\forall(j : A_0).(k : A_0).\ Rob(l, j, k) \Leftrightarrow x = j \wedge Rmem(xs, k)$$

Where, Rob is an uninterpreted function from $A_1 \times A_0 \times A_0$ to booleans.

It has to be noted that despite treating interpreted types (eg: `list` and `int`) as uninterpreted sorts, the exericise of ascribing semantics to type refinement language in MSFOL is complete. This can be established by the argument that 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 a type.

Formal semantics of specification language of λ_R are presented in Figure 4. The semantics are operational in the sense that they describe an algorithm to compile assertions in type refinements of λ_R to formulas in MSFOL. In order to capture the instantiation semantics of quantified formulas, we find it quite useful to represent them using lambda terms. We, therefore, use a typed calculus with propositional atoms, as an intermediate language of compilation to logic. We refer to the calculus as λ_ϕ . Primitive types of λ_ϕ consist of booleans, and a class of uninterpreted types (A). There exists an uninterpreted type (A) in calculus for every primitive type in λ_R. We assume the presence of a function $\mathcal{F}$ that peforms this mapping, and we parametrize our semantics over $\mathcal{F}$.

Variables and relation names of λ_R and type refinement language are also valid variables of λ_ϕ . They occur as free-variables in λ_ϕ expressions. We let x range over λ_R variables any type, R over relation names, which are variables of function type in λ_ϕ , and k over locally bound variables. Propositions (ϕ^F) and lambda expressions representing prenex-quantified propositions, constitute

[1] We assign same names to formal and actual arguments for convenience. Formal arguments can always be consistently renamed.

MSFOL

$$
\begin{array}{llll}
x & \in & \lambda_R \; variable & k, j \in quantifying\; variable \\
R & \in & uninterpreted\; relation &
\end{array}
$$

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

Auxiliary Definitions

$$
\begin{array}{lcl}
\mathcal{F} & : & T \to A \\
Inst & : & \phi^L \times v \to \phi^L \\
Inst(\forall(k : T^F).\phi^L,\, y) & = & [y/k]\phi^L \\
\eta_{wrap} & : & \phi^F \times \tau^F \to \phi^L \\
\eta_{wrap}(\phi^F, \tau_1^F \to \tau_2^F) & = & \forall(k : \tau_1^F).\eta_{wrap}(\phi^F\; k, \tau_2^F) \\
\eta_{wrap}(\phi^F, bool) & = & \phi^F
\end{array}
$$

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

$$
\begin{array}{lcl}
[[T]]_L & \stackrel{def}{=} & \mathcal{F}(T) \\
[[T * \theta]]_L & \stackrel{def}{=} & [[T]]_L \to [[\theta]]_L \\
[[\texttt{islist} :\to \{\theta\}]]_L & \stackrel{def}{=} & [[\texttt{islist}]]_L \to [[\theta]]_L \to bool \\
[[R_{id}]]_L & \stackrel{def}{=} & \forall(j : [[\texttt{int}]]_L).\; \forall(k : [[\texttt{int}]]_L).j = k \\
[[R :: \tau_R]]_L & \stackrel{def}{=} & \eta_{wrap}(R, [[\tau_R]]_L)
\end{array}
\qquad
\begin{array}{lcl}
[[R(x)]]_L & \stackrel{def}{=} & Inst([[R]]_L\,, x) \\
[[r_1 \cup r_2]]_L & \stackrel{def}{=} & \gamma_\sqcup([[r_1]]_L\,, \vee, [[r_2]]_L) \\
[[r_1 \times r_2]]_L & \stackrel{def}{=} & \gamma_\bowtie([[r_1]]_L\,, \wedge, [[r_2]]_L) \\
\gamma_\sqcup(\forall(k : T^F)e_1, \odot, \forall(k : T^F)e_2) & \hookrightarrow & \forall(k : T^F)\gamma_\sqcup(e_1, \odot, e_2) \\
\gamma_\sqcup(\phi_1^F, \odot, \phi_2^F) & \hookrightarrow & \phi_1^F \odot \phi_2^F \\
\gamma_\bowtie(\forall j : T_j^F \phi_1^F, \odot, \forall k : T_k^F \phi_1^F) & \hookrightarrow & \forall(j : T_j^F)\forall(k : T_k^F)\phi_1^F \odot \phi_2^F
\end{array}
$$

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

$$
\begin{array}{lcl}
[[\phi_1 \wedge \phi_2]]_L & \stackrel{def}{=} & \gamma_\bowtie([[\phi_1]]_L\,, \wedge, [[\phi_2]]_L) \\
[[\phi_1 \vee \phi_2]]_L & \stackrel{def}{=} & \gamma_\sqcup([[\phi_1]]_L\,, \vee, [[\phi_2]]_L)
\end{array}
\qquad
\begin{array}{lcl}
[[r_1 = r_2]]_L & \stackrel{def}{=} & \gamma_\sqcup([[r_1]]_L\,, \Leftrightarrow, [[r_2]]_L) \\
[[r_1 \subset r_2]]_L & \stackrel{def}{=} & \gamma_\sqcup([[r_1]]_L\,, \Rightarrow, [[r_2]]_L)
\end{array}
$$

Figure 4: Semantics of Specification Language

syntactic values of λ_ϕ . We collate bound variables under a single binder and represent the later form equivalently as $\lambda(\overline{k : T^F}).\, \phi^F$. A relation ($R$) application expression is a proposition, hence a value. This lets us continue treating R's as uninterpreted. Operational semantics of λ_ϕ are call-by-value. The only reduction rule is standard beta reduction.

Rules in Figure 4 provide interpretation of expressions and types of specification language as expressions and types of λ_ϕ . The interpretation of a λ_R type is its image with respect to $\mathcal{F}$. A colon-arrow type of relation is interpreted as a function type with boolean co-domain. While its domain is expected to be a product type, we chose to curry it to simplify presentation. A relation (R) of type τ_R is interpreted as a variable R of (function) type $[[\tau_R]]_L$. To compose relation applications using set operators (eg: union), it is convenient to have R under a binder. We, therefore, consider its eta-equivalent form as standard interpretation of R. We use an auxiliary function (η_{wrap}) that accepts a λ_ϕ proposition expression and its type scheme, and wraps it under an eta-equivalent binder. Beta reduction in λ_ϕ always produce a value; so, evaluation relation is a total function ($Eval$) over well-typed terms. We use $Eval$ to provide semantics to relation application expressions.

The standard interpretation of set-union and cross-product operations, when sets are represented using prenex-quantified propositions is as following:

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

Our semantics for union and cross-product operations are straightforward application of this interpretation to lambda-bound propositions. We use syntactic rewrite functions - $\gamma_\sqcup$ and $\gamma_\bowtie$, to for this purpose. Finally, we use one more rewrite function - $\gamma_\forall$, to rewrite λ_ϕ propositions bound under lambda to quantified propositions of MSFOL.

To demonstrate the compilation process, we consider the following λ_R assertion, which is legal, albeit tautological, type refinement:

```
RId(x) ∪ RId(y) = RId(y) ∪ RId(x)
```

The series of steps that compile the assertion to an MSFOL formula are shown in Figure 5[2]. It can be observed how the interpretation of `RId` and `Rmem` as λ_ϕ binders aids in compiling the assertion to MSFOL formula. **Change of example needed to include Rmem.**.

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

$$
\begin{array}{lcl}
[[\Gamma, x : \{\nu : T \mid \phi\}]]_L & \stackrel{def}{=} & [[\Gamma]]_L \wedge \nu : [[T]]_L \wedge [[\phi]]_L \\
[[\cdot]]_L & \stackrel{def}{=} & true
\end{array}
$$

LEMMA 3.4. (Completeness of semantics) *For every type refinement ϕ, if $\Gamma \vdash \phi$, then* `compile` (Γ, ϕ) *terminates and produces an MSFOL formula.*

3.5 Decidability of λ_R Type Checking

Sub-typing judgement in our core language (λ_R) relies on the semantic entailment judgement of MSFOL. The premise of SUBTBASE contains the following:

$$[[\Gamma, \nu : T]]_L \wedge [[\phi_1]]_L \models_L [[\phi_2]]_L$$

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

$$[[\,\mathtt{R_{id}(x) \cup R_{id}(y) = R_{id}(y) \cup R_{id}(x)}\,]]_L \quad (1.1)$$

$$\gamma_{\sqcup}([\underline{[[\,\mathtt{R_{id}(x) \cup R_{id}(y)}\,]]_L}, \Leftrightarrow, [[\,\mathtt{R_{id}(y) \cup R_{id}(x)}\,]]_L) \quad (1.2)$$

$$\gamma_{\sqcup}([\underline{[[\,\mathtt{R_{id}(x)}\,]]_L}, \cup, [[\,\mathtt{R_{id}(y)}\,]]_L) \quad (2.1)$$

$$\underline{Inst[[\,\mathtt{R_{id}}\,]]_L}\ x \quad (3.1)$$

$$Inst\,(\forall(\mathtt{j} : [[\,\mathtt{int}\,]]_{\mathrm{L}}.\forall(\mathtt{k} : [[\,\mathtt{int}\,]]_{\mathrm{L}}).(\mathtt{j} = \mathtt{k}))\mathtt{x} \quad (3.2)$$

$$Inst\,(\forall(\mathtt{j} : \mathtt{A_0}).\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{j} = \mathtt{k}))\mathtt{x} \quad (3.3)$$

$$\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{x} = \mathtt{k}) \quad (3.4)$$

$$\gamma_{\sqcup}(\,\forall\mathtt{k}.(\mathtt{x} = \mathtt{k})\,,\ \cup,\ \forall\mathtt{k}.(\mathtt{y} = \mathtt{k})\,) \quad (2.2)$$

$$\forall(\mathtt{k} : \mathtt{A_0}).(\gamma_{\sqcup}((\mathtt{x} = \mathtt{k}),\ \cup,\ (\mathtt{y} = \mathtt{k}))) \quad (2.3)$$

$$\forall(\mathtt{k} : \mathtt{A_0}).((\mathtt{x} = \mathtt{k} \vee \mathtt{y} = \mathtt{k})) \quad (2.4)$$

$$\gamma_{\sqcup}(\,\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{x} = \mathtt{k} \vee \mathtt{y} = \mathtt{k}),\ \Leftrightarrow,\ \forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{y} = \mathtt{k} \vee \mathtt{x} = \mathtt{k})\,) \quad (1.3)$$

$$\forall(\mathtt{k} : \mathtt{A_0}).(\gamma_{\sqcup}(\mathtt{x} = \mathtt{k} \vee \mathtt{y} = \mathtt{k},\ \Leftrightarrow,\ \mathtt{y} = \mathtt{k} \vee \mathtt{x} = \mathtt{k})) \quad (1.4)$$

$$\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{x} = \mathtt{k} \vee \mathtt{y} = \mathtt{k} \ \Leftrightarrow\ \mathtt{y} = \mathtt{k} \vee \mathtt{x} = \mathtt{k}) \quad (1.5)$$

Figure 5: Compiling a λ_R assertion to MSFOL

Consequently, decidability of type checking in λ_R reduces to decidability of semantic entailment in MSFOL. Although semantic entailment is undecidable for full first-order logic, our MSFOL is a carefully chosen fragment of first-order logic for which the problem is known to be decidable. The fragment, known as Effectively Propositional (EPR) first-order logic, or Bernay-Schonfinkel-Ramsay (BSR) class of logic, consists of prenex quantified propositions with uninterpreted relations and equality. Off-the-shelf SMT solvers (eg: Z3) are also effecient decision procedures for EPR logic [**?**]. Therefore, type checking in λ_R is also practical. Our experimental results, provided in Section 5 corroborate this statement.

THEOREM 3.5. (**Decidability**) *Type checking in λ_R is decidable.*

Proof Follows from Lemma 3.4 and decidability proof of EPR logic. ∎

4. Parametricity

4.1 Syntax

We now extend our core language (λ_R) with parametric polymorphism, and its specification language with parametric relations - relations parametrized over other relations . We refer to the extended calculus as $\lambda_{\forall R}$. Figure 6 shows type and specification language of $\lambda_{\forall R}$. We have elided expression language of $\lambda_{\forall R}$, as it is not relevant to the discussion. Unmodified syntactic forms are also elided using ellipsis.

The only algebraic data type in $\lambda_{\forall R}$ is a polymorphic list, which is the domain for structural relations. Consequently, structural relations have sort schemes (σ_R), akin to type schemes (σ) of the term language. For example, the non-parametric head relation (`Rhd`) from Section 2, when defined over a `'a list` will have a sort scheme of $\forall$ `'a. 'a list` $:\!\!\rightarrow$ `'a`. Specification language contains an expression ($\mathcal{R}\,T$) to instantiate 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 parametrized over one relational parameter. The domain of the relational parameter to a structural relation over a `'a list` should be `'a`. When `'a` in `'a list` is instantiated with `'b list`, the relational parameter of a parametric relation over `'a list` can be instantiated with a structural relation over `'b list`. For instance, the relational parameter R in the parametric membership relation (R_{mem} R), defined in Section 2, can be instantiated with the non-parametric head relation [3] (R_{hd}) after instantiating `'a` in its sort scheme with a `'b list`. The resulting relation can now be applied to a list of lists (i.e., a `'b list list`) to denote the set of head elements in the constituent lists.

Calculus $\lambda_{\forall R}$

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

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

Specification Language

Φ	$::=$	$\rho = \rho \mid \rho \subset \rho \mid \Phi \wedge \Phi$	*type refinement*
	$\mid$	$\Phi \vee \Phi \mid \top$	
ρ	$::=$	$\mathcal{R}(x) \mid \rho \cup \rho \mid \rho \times \rho$	*rel. expression*
$\mathcal{R}$	$::=$	$\mathcal{R}\,T \mid \mathcal{R}\,\theta\,\mathcal{R} \mid R$	*instantiation*
θ	$::=$	$t \mid t * \theta \mid$ ...	*tuple sort*
τ_R	$::=$	$\forall t.\,(\mathtt{'a} :\!\!\rightarrow t) :\!\!\rightarrow (\mathtt{'a\ list} :\!\!\rightarrow \theta)$	*relation sort*
	$\mid$	...	
σ_R	$::=$	$\forall a.\ \tau_R \mid \tau_R$	*sort scheme*
Σ_R	$::=$	$\langle R_s,\ R_p,\ \sigma_R,\ \overline{C\,\overline{x} \Rightarrow r}\rangle$	*rel. definition*
	$\mid$	$\langle R_s,\ R_p,\ \sigma_R,\ \mathcal{R}^*\rangle \mid$ ...	

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

The definition of a parametric relation very often does not place constraints over co-domain of its relational parameters. For instance, consider the parametric R_{hd} relation over `'a list` reproduced from Section 2:

`relation` (R_{hd} R) `(x::xs) =` R`(x) | ([]) =` R_{null}

Relation `Rhd` requires that domain of its parameter be `a`, but it places no restriction on co-domain of `R`. In order to have a truly parametric definition of `Rhd`, it is imperative that we let the relational parameter to have an unrestricted co-domain. Therefore, we let tuple-sort variables (t) to be used in tuple sorts (θ). A tuple-sort variable (or simply a sort variable) can be instantiated with a tuple sort, such as `int*int`. To simplify presentation, we require the relational parameter to always have a sort variable as their co-domain, so that sort of a parametric relation always has a quantified sort variable (t) at the beginning. For instance, parametric relation `Rhd` over an `int list` has the following sort:

$$\forall\mathtt{t.\ (int} :\!\!\rightarrow \mathtt{\{t\})} :\!\!\rightarrow \mathtt{(int\ list} :\!\!\rightarrow \mathtt{\{t\})}$$

The above sort is obtained by instantiating type variable `a` with `int` in its sort scheme, which is shown below:

$$\forall\mathtt{a.}\ \forall\mathtt{t.\ (a} :\!\!\rightarrow \mathtt{\{t\})} :\!\!\rightarrow \mathtt{(a\ list} :\!\!\rightarrow \mathtt{\{t\})}$$

In order to use a parametric relation in type refinement, its relational parameter has to be instantiated. Polymorphism in $\lambda_{\forall R}$ is predicative; so, parametrization 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 term language,

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

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

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

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

and can also be used to instantiate other parametric relations . For instance, `(Rhd RId)` is an instantiated relation. It can be applied to a list `l`, or can be used to instantiate another parameteric relation (eg: `(Rmem R)`). The class of relational expressions (ρ) of $\lambda_{\forall R}$ generalizes relation applicaton expression ($R(x)$) of λ_R to instantiated relation application ($\mathcal{R}(x)$).

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

$$\begin{array}{c}
\texttt{('}R_1\texttt{,'}R_2\texttt{) map : l} \rightarrow \texttt{(f : x} \rightarrow \{\nu \mid \texttt{'}R_2\texttt{(}\nu\texttt{) = '}R_1\texttt{(x)}\}\texttt{)} \\
\rightarrow \{\nu \mid \texttt{(}R_{obs}\ \texttt{'}R_2\texttt{)(}\nu\texttt{) = (}R_{obs}\ \texttt{'}R_1\texttt{)(l)}\}
\end{array}$$

The above type of map uses R_{obs} as transitive occurs-before relation (R^*_{ob}). More examples of parametric dependent types can be found in Section 6.

4.2 Sort and Type Checking

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

4.3 Semantics of Parametric Relations

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

Ground Relation A ground relation of a parametric relation (R) is a non-parametric relation obtained by instantiating the relational parameter with identity (`RId`) relation in its definition. For instance, ground relation of parametric membership (`Rmem`) relation is `Rmem RId`, whose definition is obtained by substituting `RId` for relational parameter in the definition of `(Rmem R)` . The substituted definition is effectively the definition of non-parametric membership relation (`Rmem`). Since we require the co-domain of the relational parameter in $\lambda_{\forall R}$ to be a tuple-sort variable (t), instantiating with `RId` is always sort-safe. Therefore, there exists a ground relation for every parametric relation in $\lambda_{\forall R}$.

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

$$\lambda\,(\texttt{x})\,.\ \texttt{RId(x)}\ \times\ \texttt{RId(x)}$$

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

Figure 8: Bind Syntax

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

$$\forall\ \texttt{t1,t2.}\ \{\texttt{t1}\} :\rightarrow (\texttt{t1} :\rightarrow \{\texttt{t2}\}) :\rightarrow \{\texttt{t2}\}$$

We name the operator `bind`, after set monadic bind. Formal syntax of a bind expression are given in Figure 8.

By *binding* a relation application with a transformer expression, a bind expression effectively creates a new relation. For instance, for an `a list l`, bind expression that binds `Rmem(l)` with reflexive transformer is as following:

$$\texttt{bind(Rmem(l),}\ \lambda\texttt{x.RId(x)}\times\texttt{RId(x))}$$

As per our informal semantics of `bind`, reflexive transformer is applied on every tuple in the set `Rmem(l)` , and the the resulting set of sets is folded using union. The result is the set of reflexive pairs of elements in the list. It should be observed that this is the same set produced by parametric `(Rmem R)` , if it is instantiated with primitive $\texttt{R}_{\texttt{dup}}$ relation (Section 2). Therefore, we note that:

$$\texttt{(Rmem R}_{\texttt{dup}}\texttt{)(l) = bind(Rmem(l),}\lambda\texttt{x.RId(x)}\times\texttt{RId(x))}$$

Where, equality should be interpreted as equality of sets on both sides. Since semantics of a relation application is the set of tuples, the above equation defines the semantics of $\texttt{(Rmem R}_{\texttt{dup}}\texttt{)}$ *in terms* of its ground relation `Rmem`. It should be noted that similar observation can be made about every instantiation of `Rmem`. Therefore, `Rmem` can be defined equivalently as:

$$\texttt{Rmem}\ = \lambda\,\texttt{R}\,.\lambda\,\texttt{l}\,.\ \texttt{bind(Rmem(l),}\lambda\texttt{x.R(x))}$$

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

definition. We briefly describe the elaboration to bind definition in following sub-section. However, we do not establish a formal proof of equivalence between a pattern-match based definition and its elaborated bind definition. For formal treatment, we assume that a parametric relation is defined using `bind` syntax from Figure 8.

4.3.1 Elaboration to Bind Definition

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

$$\texttt{Rx :: } \forall \texttt{t. (int} :\rightarrow \{\texttt{t}\}) :\rightarrow \texttt{(int list} :\rightarrow \{\texttt{int*t*t}\})$$

We construct the following transformer expression by looking at its co-domain sort:

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

Naming the ground relation of `Rx` as `Rx'`, the equivalent bind definition of `Rx` is:

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

4.3.2 Bind Equations

By substituting parametric relation with its bind definition, every instantiation of a parametric relation can be beta-reduced to a bind abstraction (E_b in Figure 8), which, like any non-parametric structural relation in $\lambda_{\forall R}$, is a map from `a list` to set of tuples. Therefore, instantiated parametric relation can be treated as a new non-parametric relation that is defined using `bind`. For example, $\texttt{(Rmem R}_{\texttt{dup}}\texttt{)}$ can be treated as a new non-parametric relation `R1`, defined in terms of `bind`:

$$\texttt{R1 = }\lambda\texttt{l.bind(Rmem(l),}\lambda\texttt{x.RId(x)}\times\texttt{RId(x))}$$

By rigourously defining semantics of *bind equations* such as above, we can effectively capture 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, bind semantics for $\texttt{(Rmem R}_{\texttt{dup}}\texttt{)}$ let us prove the following implication, which could potentially arise during subtype checking:

$$\texttt{(Rmem R}_{\texttt{dup}}\texttt{)(l1) = (Rmem R}_{\texttt{dup}}\texttt{)(l2)} \Rightarrow \texttt{Rmem(l1) = Rmem(l2)}$$

Formal semantics of bind equations, which also define an algorithm to compile bind equations to MSFOL formulas, are described in Figure 7. Our semantics extend MSFOL semantics of relational expressions from Figure 4 to bind equations. Under our semantics, the bind equation for $\texttt{Rmem R}_{\texttt{dup}}$ is interpreted as a conjunction of following first-order formulas (elaborated for clarity):

- If `x` $\in$ `Rmem(l)`, and `y` $\in$ `RId(x)` $\times$ `RId(x)`, then `y` $\in$ `R1(l)`.
- If `y` $\in$ `R1(l)`, then there must exist an `x` such that `x` $\in$ `Rmem(l)` and `y` $\in$ `RId(x)` $\times$ `RId(x)`

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

4.4 Decidability of Type Checking

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

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

Proof Follows from the decidability proof of EPR logic, to which bind equations are compiled to, and decidability result (Theorem 3.5) for λ_R. ∎

5. Implementation

We have implemented our specification language and verification procedure as an extended type-checking pass (called CATALYST) in MLton, a whole-program optimizing compiler for Standard ML[4] 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 paramterized structural relations over the program's datatypes. The type system is a conservation 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 (Figure 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:

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

   $$R_{\neq}\ \texttt{=}\ \lambda\ \texttt{k.}\ \lambda\ \texttt{x.}\ R_{id}\texttt{(x)}\ \texttt{-}\ R_{id}\texttt{(k)}$$

   Similarly, $R_{=_k}$ can be defined as:

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

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

2. *Base Predicates*: Consider the following polymorphic identity function, augmented with its obvious relational refinement:

   $$\texttt{id : x} \rightarrow \{\texttt{v} \mid R_{id}\texttt{(v) = }R_{id}\texttt{(x)}\ \}$$
   $$\texttt{val id = fn x => x}$$

   The post-condition is an unintuitive way of expressing the simple fact that `id` returns its argument. To avoid such needless verbosity, we admit non-relational assertions (called *base predicates*), drawn from propositional logic with equality, to our

[4] The source code for the implementation is available online from: *github URL.*

specification language; these predicates may be freely composed in type refinements using logical connectives.

3. *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 [**?**], 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` expression 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 parameteric relations with help of a an expensive fix-point computation that generates an exhaustive list of all possible instantiations, CATALYST relies on manual annotations for parameter instantiations in these cases.

5.1 Experiments

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

- List library functions, such as as `concat`, `rev`, `revAppend`, `foldl`, `foldr`, `map`, `exists`, `filter`, `zip` etc., and
- Okasaki's red-black tree [**?**] 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, including A-Nomalization, specification elaboration, VC generation, and SAT solving through Z3.

6. Examples and Case Study

6.1 Examples

exists. Consider the higher-order `exists` function over lists shown in Fig. 9a (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:

$$\texttt{('}R\ \texttt{exists) :}\ \texttt{l} \rightarrow \texttt{f : x} \rightarrow \{\ \nu\ \texttt{|}\ \nu\texttt{=true} \Leftrightarrow \texttt{'}R(\texttt{x}) \neq R_{null})\} \rightarrow \{\nu\ \texttt{|}\ \nu\texttt{=true} \Leftrightarrow\ (R_{mem}\ \texttt{'}R)(\nu)\ \neq\ R_{null}\ \}$$

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

A parametric dependent type for `filter`, shown in Fig. 9b is given below:

$$\texttt{('}R\ \texttt{filter) :}\ \texttt{l} \rightarrow \texttt{f : x} \rightarrow \{\ \nu\ \texttt{|}\ \nu\texttt{=true} \Leftrightarrow \texttt{'}R(\texttt{x}) = R_{null})\} \rightarrow \{\nu\ \texttt{|}\ (R_{mem}\ \texttt{'}R)(\nu)\ \texttt{=}\ R_{mem}(\texttt{l})\ \}$$

The intuition behind this type is the same as that of `exists`. However, since filter *removes* those elements for which its higher-order argument returns true, `f`'s dependent relation type is $'R(\texttt{x}) = R_{null}$.

contains Consider the definition of the `contains` function shown in Fig. 9c 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:

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

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

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

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

6.2 α-conversion

An SML implementation of the untyped lambda calculus is shown in Figure 10. 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 free variables in an expression.

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

The relational specification of substitution and α conversion is given in the bottom-half of Fig. 10 [5] 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 `Rfv` that relates an expression of the calculus to its free variables[6] 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 specification for `subst` and `alphaConvert` does not constrain how these functions are realized in ML. For instance, an implementation of `subst` that pro-actively 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 [**? ?**

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

[6] We write `Rmem` instead of `(Rmem RId)` in the specification when there is no ambiguity.

```
fun exists l f = case l of
  [] => false
| x::xs =>
  let
    val v1 = exists xs f
    val v2 = f x
    val v3 = v1 orelse v2
  in
    v3
  end
```

(a) `exists`

```
fun filter l f = case l of
  [] => []
| x::xs =>
  let
    val xs' = filter xs f
    val v2 = f x
  in
    if v2 then xs'
          else x::xs'
  end
```

(b) `filter`

```
fun contains l str =
  let
    val isStr =
        fn x => x=str
    val hasStr = exists (REq str) l
         isStr
  in
    hasStr
  end
```

(c) `contains`

Figure 9: Examples

ML Program

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

Relational Specification

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

concat : ll -> {v | (Rmem RId)(v) = (Rmem (Rmem RId))(ll)};
createNewName : fvs -> id -> {v | not (v = id) /\ not ({(v)} ⊆ Rmem(fvs))};
freeVars : e -> {l | Rmem(l) = Rfv(e)};
alphaConvert : e -> {ex | Rfv(ex) = Rfv(e)};
subst : e1 -> id -> e2
        -> {ex | if ({(id)} ⊆ Rfv(e2)) then Rfv(ex) = (Rfv(e2) - {(id)}) U Rfv(e1) else Rfv(ex) = Rfv(e2)};
```

Figure 10: SML implementation and specification of the untyped lambda calculus.

] using GADTs [**?**]. 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. Variables and constants of term language can freely enter the type universe. In this sense, our type system is similar to the refinement based dependent type system of F* [**?**]. Type refinements in F* are drawn from full first-order logic extended with theories that an SMT solver can reason with, whereas our specification language for ML programs is an abstraction over first-order logic that was tailor-made for equational and relational reasoning. The expressivity of using full-first order logic in F* comes at the cost of decidability of type checking. Further, even with access to full first-order logic in type refinements, a relationally parametric type cannot be directly expressed in F*; second-order quantification is required.

Structural relations, in their operational manifestation, can be compared to the structurally recursive *measures* of Liquid Types [**?** **?**] where the co-domain is always a set. Parametric structural relations may be viewed as generalizing such measures to higher-order measures. However, the focus in liquid types is on infererring type refinements with help of user-provided templates, which requires templates to be simple so that inference remains trackable. Relationally parametric dependent types can be compared to liquid types with abstract refinements [**?**], which let liquid types parametrize 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 a higher-order argument to parametric relations. We require this generality to reason about shape invariants of higher-order catamorphisms such as `foldl` .

Liquid type 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.[**?**] describe decision procedures for the theory of algebraic data types extended with abstraction functions to decidable abstract domains. However, our encoding does not require such theory extensions to logic. A structural relation straightforwardly translates to an uninterpreted

relation in the first-order logic. Using bind semantics, our encoding was also able to support parametric relations, which would otherwise require higher-order abstraction functions in logic.

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

Logical relations have been used extensively to reason about contextual equivalence [**?**]. Whereas a logical relation relates two terms of a (possibly recursive) type, a structural relation relates a term of an iso-recursive type to its constituent values. Parametric logical relations have been used to reason about contextual equivalence for effectful programs [**? ? ?**]. In these works, 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 reason about contextual equivalence. This is a possible avenue for future research.

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

8. Conclusions

In this paper, we have presented 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 described how parametric relations can be used to enable compositional verification in presence of parametric polymorphism and higher-order functions. We demonstrated that even with such expressive specification language, type checking can remain decidable and practical. We have presented several examples and a case study to show wide applicability and practicality of our approach.

A. Appendix

LEMMA A.1. (**Completeness of semantics**) *For every type refinement ϕ, if $\Gamma \vdash \phi$, then* `compile` (Γ, ϕ) *terminates and produces an MSFOL formula.*

Proof Proof by induction on well-formedness judgement.