A Relational Framework for Higher-Order Shape Analysis

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

Abstract

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

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

1. Introduction

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

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

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

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

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

Here, `rev'` represents some reference implementation of `rev`. Checking `rev`'s implementation against this refinement is tantamount to proving the equivalence of `rev` and `rev'`. Given the undecidability of the general problem, expecting these types to be machine checkable would require the definition of `rev'` to closely resemble `rev`'s. For all but the most trivial of definitions, this approach is unlikely to be fruitful. An alternative approach is to design `rev'` in a way that makes it amenable for reasoning by an automated theorem prover. To be feasible, this would require the meaning of (potentially arbitrary) recursive functions like `append` that may be used by `rev'` to be understood by the prover, which is again likely to be infeasible in practice. Clearly, the challenges in building suitably typed definitions that let us reason about interesting shape properties of a data structure are substantial.

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

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

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

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

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

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)}`

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

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

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

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

```
dup : 'a list → ('a*'a) list
```

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

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

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 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 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 that is incorporated within the MLton Standard ML compiler, and demonstrate the utility of these ideas through a detailed case study that automatically verifies the correctness of α-conversion and capture-avoiding substitution operations of the untyped lambda calculus, whose types are expressed using relational expressions.

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

2. Structural Relations

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

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

R_{id} is the identity relation, R_{dup} is a relation that associates a value with a pair that duplicates that value, R_{notEq_k} is a relation indexed by a constant `k` (of some base type) that relates `x` to itself,

provided `x` is not equal to `k`, and R_{eq_k} is defined similarly, except it relates `x` to itself exactly when `x` is equal to `k`. Apart from the relations defined above, the language also includes the relation $\emptyset$ that denotes the empty set.

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

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

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) $\langle \texttt{x} :: \texttt{xs}, \texttt{x}\rangle \in R_{hd}$, and (2) there does not exist an `x'` such that $\texttt{x}' \neq \texttt{x}$ and $\langle \texttt{x} :: \texttt{xs}, \texttt{x}'\rangle \in R_{hd}$. The declarative syntax of the kind shown above is the primary means of defining structural relations in our system.

2.1 Relational Composition

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

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

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

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

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

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

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 pair whose first element is the head of the list, and whose second element is a member of its tail:

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

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

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

[2] We often elide the case for empty list (`[]`). It defaults to the empty set ($\emptyset$)

2.2 Parametric Relations

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

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

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

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

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

`zip` can now be assigned the following type that precisely captures the desired specification[3]:

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

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

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

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

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

2.3 Parametric Dependent Types

Our specification language also allows dependent types to be parameterized over relations used in type refinements. In the spirit of type variables, we use relation variables ($'R$) to denote parameterized relations in a type. Consider the following signature for `foldl`:

$$\begin{array}{l}(' R_{bm})\ \texttt{foldl :}\\ \quad \{\texttt{l : 'a list}\} \rightarrow \{\texttt{b : 'b}\} \rightarrow\\ \quad\ \ (\{\texttt{f: \{x :'a\}} \rightarrow \texttt{\{acc : 'b\}} \rightarrow\\ \qquad \{\texttt{z : 'b}\ \mid\ '\!R_{bm}\texttt{(z)} = \{\langle \texttt{x}\rangle\} \cup\ '\!R_{bm}\texttt{(acc)}\ \}\}) \rightarrow\\ \quad \{\ \nu\ \mid\ '\!R_{bm}\,(\nu) = R_{mem}\ \texttt{(l)} \cup '\!R_{bm}\texttt{(b)}\}\end{array}$$

This type relates membership properties on `foldl`'s input list (non-parametric R_{mem}) to an abstract notion of membership over its result type (`'b`) captured using a relation variable ($'R_{bm}$). To see how we might instantiate this relation variable, consider the

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

function `makeTree` that uses `foldl` to generate a binary tree using function `treeInsert` (not shown):

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

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

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

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

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

An alternative type for `foldl` could relate the order of elements in the argument list to some order of the result. The intuition is as following: let us say 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 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, given that we know the order in which `f` is applied over the list. The type of `foldl` that tries to match the abstract order (R_{bo}) on the result type (`'b`) to an occurs-after order on the input list is shown below. For brevity, we avoid reproducing membership invariants from the type of `foldl` from the previous example, using ellipses in their place:

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

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

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

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

Calculus λ_R

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

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

Specification Language

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

Figure 1: Language

3. Core language

3.1 Syntax

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

Dependent type refinements (ϕ) in λ_R are assertions over relational expressions (r); these expressions, which are themselves typed, constitute the syntactic class of expressions in our specification language. We refer to the types of relational expressions as *sorts*, in order to distinguish them from the λ_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 (*). We write $R \triangleq \delta$ to denote that a relation R has (pattern-match or inductive) definition δ.

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

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

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

3.2 Sorts, Types and Well-formedness

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

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

S-REL

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

S-REL-STAR

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

S-APP

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

S-REL-ID

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

S-UNION

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

S-CROSS

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

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

WF-RPRED

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

WF-REF

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

WF-BASE

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

WF-FUN

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

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

SUBT-BASE

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

SUBT-ARROW

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

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

T-VAR

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

T-ABS

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

T-MATCH

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

T-CONST

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

T-SUB

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

T-APP

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

T-LET

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

Figure 3: Static semantics of λ_R

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

Environment Γ is an ordered set of assertions that make up the typing context. The assertions are either (a) type bindings for variables, or (b) branch conditions collected from `match` expressions. We assume that any variable is bound only once in Γ.

Structural relations are sort checked under an empty type environment. The rule S-REL type checks a relation definition by ensuring that relational expressions associated with the constructors 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 judgement ($\Vdash$) under a *refinement erased* Γ (denoted $\|\Gamma\|$) for this purpose. Rules for simple typing judgement are straightforward, and are elided due to space constraints. They can be found in the appendix.

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

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

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

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

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

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

The type refinements of Nil (ϕ_n) and Cons (ϕ_c) are conjunctive aggregations of Nil and Cons cases (resp.) of all structural relation definitions. To help us precisely define ϕ_n and ϕ_c, we assume the

MSFOL

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

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

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

Auxiliary Definitions

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

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

$$\begin{array}{lcl} [\![R(\mathsf{Cons}\, v_1\, v_2)]\!]_L & = & [\![\Sigma_R(R)(\mathsf{Cons}\, v_1\, v_2)]\!]_L \\ [\![R(\mathsf{Nil})]\!]_L & = & [\![\Sigma_R(R)(\mathsf{Nil})]\!]_L \\ [\![T]\!]_L & = & \mathcal{F}(T) \\ [\![T * \theta]\!]_L & = & [\![T]\!]_L \to [\![\theta]\!]_L \\ [\![\mathsf{intlist} :\to \{\theta\}]\!]_L & = & [\![\mathsf{intlist}]\!]_L \to [\![\theta]\!]_L \to \textit{bool} \\ [\![R_{id}]\!]_L & = & \forall (j : [\![\,\mathtt{int}\,]\!]_L).\ \forall (k : [\![\,\mathtt{int}\,]\!]_L).j = k \end{array}$$

$$\begin{array}{lcl} [\![R]\!]_L & = & \eta_{wrap}(R, [\![\Gamma_R(R)]\!]_L) \\ [\![R(x)]\!]_L & = & Inst([\![R]\!]_L,\, x) \\ [\![r_1 \cup r_2]\!]_L & = & \gamma_{\sqcup}([\![r_1]\!]_L, \vee, [\![r_2]\!]_L) \\ [\![r_1 \times r_2]\!]_L & = & \gamma_{\bowtie}([\![r_1]\!]_L, \wedge, [\![r_2]\!]_L) \\ \gamma_{\sqcup}(\forall (k : T^F)e_1, \odot, \forall (k : T^F)e_2) & = & \forall (k : T^F).\gamma_{\sqcup}(e_1, \odot, e_2) \\ \gamma_{\sqcup}(\phi_1^F, \odot, \phi_2^F) & = & \phi_1^F \odot \phi_2^F \\ \gamma_{\bowtie}(\forall j : T_j^F \phi_1^F, \odot, \forall \overline{k : T_k^F} \phi_1^F) & = & \forall (j : T_j^F).\forall (\overline{k : T_k^F}).\, \phi_1^F \odot \phi_2^F \end{array}$$

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

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

Figure 4: Semantics of Specification Language

presence of (a) a finite map (Σ_R) that maps relation names to their pattern-match definitions, and (b) a finite ordered map Γ_R that maps relation names to their sorts. We implicitly parameterize our typing judgement over Σ_R (i.e., our $\vdash$, is actually $\vdash_{\langle \Sigma_R, \Gamma_R \rangle}$). Inductive relations defined using the closure operator are assumed to be unfolded to pattern-match definitions before being bound in Σ_R:

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

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

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

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

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

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

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

The T-MATCH rule type checks each branch of the `match` expression under an environment that records the corresponding branch condition. Additionally, the type environment for Cons branch is also extended with types of matched pattern variables (x and y). The branch condition for Cons (alternatively, Nil) case is obtained by substituting the test value (v) for the bound variable (ν) in the type refinement of Cons (Nil). Intuitively, 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:

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

where Cons has type[4]

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

Expression e_1 is type-checked under the extended environment:

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

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

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

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

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

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

Soundness property of λ_R's type system refers to the reduction relation ($\longrightarrow$) defined by its operational semantics , and is stated thus :[5]

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

3.3 Semantics of Specification Language

We define the 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 propositional logic with equality and n-ary uninterpreted functions.

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

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

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

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

Relations translate to uninterpreted functions with a Boolean co-domain in MSFOL. We choose to curry sorts of uninterpreted functions representing relations (R) to simplify the semantics. Auxiliary function η_{wrap} wraps an uninterpreted function under a quantified formula; this can be construed as an eta-equivalent abstraction of an uninterpreted function in quantified logic. Auxiliary function $Inst$ instantiates a prenex-quantified formula. The standard interpretation of set-union and cross-product operations, when sets are represented using prenex-quantified propositions is as following:

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

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

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

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

and membership relation

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

for integer lists:

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

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

$$\begin{aligned}
&[\![\mathtt{R_{ob}(l)} = \mathtt{R_{id}(x)} \times \mathtt{R_{mem}(xs)}]\!]_L && (1.1)\\
&\gamma_\sqcup(\underline{[\![\mathtt{R_{ob}(l)}]\!]_L}, \Leftrightarrow, [\![\mathtt{R_{id}(x)} \times \mathtt{R_{mem}(xs)}]\!]_L) && (1.2)\\
&Inst[\![\mathtt{R_{ob}}]\!]_L\ l && (2.1)\\
&Inst(\,\forall(\mathtt{i} : [\![\mathtt{intlist}]\!]_L).\forall(\mathtt{j} : [\![\mathtt{int}]\!]_L)\,. && (2.2)\\
&\qquad \forall(\mathtt{k} : [\![\mathtt{int}]\!]_L).(\mathtt{Rob\ i\ j\ k})\,)\ \mathtt{x} &&\\
&Inst\,(\forall(\mathtt{i} : \mathtt{A_1}).\forall(\mathtt{j} : \mathtt{A_0}).\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{Rob\ i\ j\ k}))\ \mathtt{x} && (2.3)\\
&(\forall(\mathtt{j} : \mathtt{A_0}).\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{Rob\ x\ j\ k})) && (2.4)\\
&\gamma_\sqcup([\![\mathtt{R_{ob}(l)}]\!]_L, \Leftrightarrow, \underline{[\![\mathtt{R_{id}(x)} \times \mathtt{R_{mem}(xs)}]\!]_L}) && (1.2)\\
&\gamma_\bowtie(\underline{[\![\mathtt{R_{id}(x)}]\!]_L}, \wedge, [\![\mathtt{R_{mem}(xs)}]\!]_L) && (3.1)\\
&Inst\,(\forall(\mathtt{i} : [\![\mathtt{int}]\!]_L.\forall(\mathtt{j} : [\![\mathtt{int}]\!]_L).(\mathtt{i} = \mathtt{j}))\ \mathtt{x} && (4.1)\\
&\forall(\mathtt{j} : \mathtt{A_0}).(\mathtt{x} = \mathtt{j}) && (4.2)\\
&\gamma_\bowtie([\![\mathtt{R_{id}(x)}]\!]_L, \wedge, \underline{[\![\mathtt{R_{mem}(xs)}]\!]_L}) && (3.1)\\
&(\forall(\mathtt{k} : \mathtt{A_0})(\mathtt{Rmem\ xs\ k})) && (5.1)\\
&\gamma_\bowtie(\,\forall(\mathtt{j} : \mathtt{A_0}).(\mathtt{x} = \mathtt{j})\,, \wedge,\ (\forall(\mathtt{k} : \mathtt{A_0})(\mathtt{Rmem\ xs\ k}))\,) && (3.2)\\
&\forall(\mathtt{j} : \mathtt{A_0}).\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{x} = \mathtt{j}) \wedge (\mathtt{Rmem\ xs\ k}) && (3.3)\\
&\gamma_\sqcup(\,(\forall(\mathtt{j} : \mathtt{A_0}).\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{Rob\ x\ j\ k}))\,, \Leftrightarrow, && (1.3)\\
&\qquad \forall(\mathtt{j} : \mathtt{A_0}).\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{x} = \mathtt{j}) \wedge (\mathtt{Rmem\ xs\ k})\,) &&\\
&\forall(\mathtt{j} : \mathtt{A_0}).\forall(\mathtt{k} : \mathtt{A_0}).(\mathtt{Rob\ l\ j\ k}) \Leftrightarrow (\mathtt{x} = \mathtt{j}) \wedge (\mathtt{Rmem\ xs\ k}) && (1.4)
\end{aligned}$$

Figure 5: Compiling a λ_R assertion to MSFOL

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

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

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

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

A completeness lemma for translation, which states that translation to MSFOL is complete for a well-formed type refinement, can now be stated thus [7]

LEMMA 3.2. (Completeness of semantics) *For every type refinement* ϕ*, if* $\Gamma \vdash \phi$*, then* $[\![\phi]\!]_L$ *terminates and produces an MSFOL formula.*

3.4 Decidability of λ_R Type Checking

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

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

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

[5] Type safety is restated and proved as Lemma 1.14. in the appendix

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

[7] Proof can be found at : *http://tycon.github.io/catalyst/meta.pdf*

Calculus $\lambda_{\forall R}$

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

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

Specification Language

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

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

sion procedures for EPR logic [18], making type checking in λ_R a practical exercise.

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

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

4. Parametricity

4.1 Syntax

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

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

A parametric relation generalizes a structural relation, just as a polymorphic list generalizes a monomorphic one. Our syntax and semantics for parametric relations are based on this correspondence. Since the `list` type constructor takes only one type argument, structural relations in $\lambda_{\forall R}$ are parameterized over one relational parameter. The domain of a relational parameter to a structural relation over a `'a list` should be `'a`. When `'a` in `'a list` is instantiated with, e.g., `'b list`, the parameter of a parametric relation over `'a list` can be instantiated with a structural relation over `'b list`. For instance, the relational parameter R in the parametric membership relation (R_{mem} R), defined in Section 2, can be instantiated with the non-parametric head relation, R_{hd}[8], after instantiating `'a` in its sort scheme with a `'b list`. The resulting relation can now be applied to a list of lists (i.e., a `'b list list`) to denote the set of head elements in the constituent lists.

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

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

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

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

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

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

4.2 Sort and Type Checking

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

4.3 Semantics of Parametric Relations

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

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

Transformer Expression. A transformer expression (F_R) is a λ_R relational expression under a binder that binds a tuple of variables. A transformer expression is expected to transform the tuple to a set of tuples through a cross-product combination of relation applications. The sort of a transformer application is a map (under

[8] 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.

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

Figure 7: Bind Syntax

' :->') from tuple-sort (θ_1) to a set sort ($\{\theta_2\}$). An example of a transformer expression of sort 'a :→ { 'a*'a } is the *reflexive transformer*:

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

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

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

We name the operator `bind`, after set monadic bind. The syntax of bind expressions is given in Fig. 7.

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

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

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

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

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

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

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

4.3.1 Elaboration to Bind Definition

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

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

Its transformer expression is:

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

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

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

4.3.2 Bind Equations

By substituting parametric relations with their bind definitions, every instantiation of a parametric relation can be reduced to a bind abstraction (E_b in Figure 7), which, like any non-parametric structural relation in $\lambda_{\forall R}$, is a map from a 'a list to a set of tuples. Therefore, an instantiated parametric relation can be treated as a new non-parametric relation that is defined using `bind`. For example, $(R_{mem}R_{dup})$ can be treated as a new non-parametric relation R_1, defined in terms of `bind`:

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

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

$$\begin{array}{c}((R_{mem}\ R_{dup})\ \texttt{l}_1\texttt{)} = ((R_{mem}\ R_{dup})\ \texttt{l}_2\texttt{)} \\ \Rightarrow R_{mem}(\texttt{l}_1) = R_{mem}(\texttt{l}_2)\end{array}$$

The formal semantics of bind equations, which also define an algorithm to compile bind equations to MSFOL formulas, is described in Fig. 8. Under our semantics, the bind equation for $(R_{mem}\ R_{dup})$ is interpreted as a conjunction of following first-order formulas (elaborated for clarity):

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

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

4.4 Decidability of Type Checking

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

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

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

5. Implementation

We have implemented our specification language and verification procedure as an extended type-checking pass (called CATALYST) in MLton, a whole-program optimizing compiler for Standard ML (SML)[9] 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

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

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

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

Figure 8: Semantics of bind equations for parametric relations in $\lambda_{\forall R}$

dependent types that decorate function signatures, along with definitions of parameterized structural relations over the program's datatypes. The type system is a conservative extension of SML's, so all programs that are well-typed under CATALYST are well-typed SML programs. Our type-checking and verification process closely follows the description given in the previous sections. Verification conditions, representing the consequent of the SUBT-BASE type-checking rule (Fig. 3) are compiled to a first-order formula, as described in Sections 3 and 4, and checked for validity (satisfiability of its negation) using the Z3 SMT solver.

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

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

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

Similarly, R_{eq_k} can be defined as:

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

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

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

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

The type refinement used here is an unintuitive way of expressing the simple fact that `id` returns its argument. To avoid such needless verbosity, we admit non-relational assertions (called *base predicates*), drawn from propositional logic with equality, to our specification language; these predicates may be freely composed in type refinements using logical connectives.

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

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

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

5.1 Experiments

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

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

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

6. Case Study

An SML implementation of the untyped lambda calculus is shown in Fig. 10. The implementation makes use of auxiliary functions, such as `filter` and `contains`, directly, and `exists` through `contains`. By the virtue of being compositional, our verification process relies on expressive relational types of these auxiliary functions, which we present below:

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:

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

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

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

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

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

(a) `exists`

```
fun filter l f = case l of
  [] => []
| x::xs =>
  let
    val xs' = filter xs f
  in
    if f x then x::xs'
           else 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)}
       | Rfv (App (e1,e2)) = Rfv(e1) ∪ Rfv(e2)
       | Rfv (Abs (id,e)) = Rfv(e) - {(id)};

createNewName : fvs -> id -> {v | not (v = id) ∧ ¬ ({(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)}) ∪ Rfv(e1) else Rfv(ex) = Rfv(e2)};
```

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

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

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:

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

the relational dependent type of `isStr` is:

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

This clearly suggests that 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 it is a non-recursive function.

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

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

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

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

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

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

6.1 α-conversion

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

It is widely agreed that substitution and α-conversion operations on lambda calculus terms are quite tricky to define correctly [19, 22]. Some of the behaviors exhibited by incorrect implementations include (a) α-conversion renames a free variable, or fails to rename a bound variable; (b) substitution fails to substitute free occurrences of the variable (`id`), or substitutes a bound occurrence of the variable; or (c) substitution is not capture-avoiding, i.e., substituting `e1` for `id` in `e2` captures variables of `e1`, which are otherwise free.

The relational specification of substitution and α conversion is given in the bottom-half of Fig. 10 [10] 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. Function `freeVars` returns a list, whose members are free variables of its input expression. Its type represents this fact.

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

7. Related Work

Type systems of mainstream functional languages, such as GHC Haskell and OCaml, support a basic form of dependent typing [13, 14] using GADTs [23]. At a high level, a structural relation of a data type is similar to a GADT insofar as it corresponds to an index that tracks an inductively definable relation over the data type. However, unlike the indexed type systems of Haskell and OCaml, where types are kept separate from terms, ours is a dependent type system. In this sense, our type system is similar to the refinement based dependent type system of F* [1]. Type refinements in F* are drawn from full first-order logic extended with theories that an SMT solver can reason with, whereas our specification language for ML programs is an abstraction over first-order logic that was tailor-made for equational and relational reasoning. The expressivity of using full-first order logic in F* comes at the cost of decidability of type checking. Further, even with access to full first-order logic in type refinements, a relationally parametric type cannot be directly expressed in F*; second-order quantification is required.

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

[10] 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 $((\phi \wedge \phi_1) \vee (\neg\phi \wedge \phi_2))$.

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

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

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

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

8. 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.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[19] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. A locally nameless representation for a natural semantics for lazy evaluation. In *ICTAC*, 2012.

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

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

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

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