A Relational Framework for Higher-Order Shape Analysis

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

Abstract

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

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

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (Functional) Languages; F.3.1 [*Logics and*

Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification

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

1. Introduction

Dependent types are well-studied vehicles capable of expressing rich program invariants. A prototypical example is the type of a list that is indexed by a natural number denoting its length. Length-indexed lists can be written in several mainstream languages that support some form of dependent typing, including GHC Haskell [24], F* [21, 23], and OCaml [16]. For example, the following Haskell signatures specify how the length of the result list for `append` and `rev` relate to their arguments:

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

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

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

http://dx.doi.org/10.1145/10.1145/2628136.2628159
specify the intended behavior of these operations. For example, a correctly written `append` function must additionally preserve the order of its input lists; a function that incorrectly produces an output list that is a permutation of its inputs would nonetheless satisfy `append`'s type as written above. Similarly, the identity function would clearly satisfy the type given for `rev`; a type that fully captures `rev`'s behavior would also have to specify that the order of elements in `rev`'s output list is the inverse of the order of its input. Is it possible to 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:

$$\texttt{rev} : \{\texttt{l : 'a list}\} \longrightarrow \{\nu\texttt{: 'a list | } \nu \texttt{ = rev'(l)}\}$$

Here, `rev'` represents some reference implementation of `rev`. Checking `rev`'s implementation against this refinement is tantamount to proving the equivalence of `rev` and `rev'`. Given the undecidability of the general problem, expecting these types to be machine checkable would require the definition of `rev'` to closely resemble `rev`'s. For all but the most trivial of definitions, this approach is unlikely to be fruitful. An alternative approach is to define `rev` within a theorem prover, and directly assert and prove properties on it - for example, that `rev` is involutive. Although modern theorem provers support rich theories over datatypes like lists, this strategy nonetheless requires that the program be fully described in logic, and reasoned about by the solver in its entirety. Thus, defining `rev` in this way also requires an equational definition of `append`, assuming the former is defined in terms of the

latter. For non-trivial programs, this may require equipping provers with arbitrarily complex theories, whose combination may not be decidable. Such a methodology also does not obviously address our original goal of specifying `rev`'s functional correctness, independent of its definition; note that in the case of `rev`, involution does not imply functional correctness. Clearly, the challenges in building suitably typed definitions that let us reason about interesting shape properties of a data structure are substantial.

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

transitive closure of such a relation effectively serving as a faithful representation.

For example, consider a relation R_{ob} that relates a list to a pair if

the first element in the pair *occurs before* the second in the list. For a concrete list `l=[x1,x2,x3]`, the relation's 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]

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

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

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

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

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

makes relational specifications highly amenable for automatic verification.

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

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

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

[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,

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

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 a decidable logic.

The paper makes the following contributions:

1. We present a rich specification language for expressing refinements that are given in terms of relational expressions and familiar relational algebraic operations. The language is equipped

with pattern-matching operations over constructors of algebraic data types, thus allowing the definition of useful shape properties in terms of relational constraints.

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

The remainder of the paper is structured as follows. In the next

section, we present additional motivation and examples for our ideas. Sec. 3 formalizes the syntax and static semantics of relational refinements in the context of a simply-typed core language. Sec. 4 extends the formalization to support parametric refinements within a polymorphic core language. Our formalization also presents a translation scheme from relational refinements to a decidable first-order logic. Details about the implementation are given in Sec. 5. Sec. 6 presents a case study. Secs. 7, 8 and 9 present related work, directions for future work, and conclusions, respectively.

2. Structural Relations

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

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

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

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

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

For a concrete list `l`, $R_{hd}(\mathtt{l})$ produces the set of unary tuples

whose elements are in the *head* relation with `l`. This set is clearly a singleton when the list is non-empty and empty otherwise. The above definition states that for any list pattern constructed using "::" whose head is represented by pattern variable `x` and whose tail is represented by pattern variable `xs`, (1) $\langle \texttt{x} :: \texttt{xs}, \texttt{x} \rangle \in R_{hd}$, 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 equivalently express R_{mem} as an *inductive extension* of the head relation R_{hd} defined above. Suppose R is a structural relation that relates a list l of type `'a list` with elements v of type `'a`. Then, the inductive extension of R (written R^*) is the least relation

that satisfies the following conditions:

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

---

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

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. Because R_{ob} relates a list to a pair whose first element is the head of the list, and whose second element is a member of its tail, it can be expressed in terms of R_{mem} thus:

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

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:

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

2.2 Parametric Relations

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

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

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:

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

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

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

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

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

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

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

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

2.3 Parametric Dependent Types

Our specification language also allows dependent types to be parameterized over relations used in type refinements. In the spirit of type variables, we use relation variables to denote parameterized relations in a type. To illustrate why such parameterization is useful, consider the following signature for `foldl`:

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

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

$$
\begin{array}{l}
\texttt{datatype 'a tree = Leaf} \\
\qquad\qquad\qquad\quad \texttt{| Tree of 'a * ('a Tree) * ('a Tree)} \\
\\
\texttt{relation}\ R_{thd}\ \texttt{Leaf} = \emptyset \\
\qquad\quad \mid R_{thd}\ (\texttt{Tree (x,t}_1\texttt{,t}_2\texttt{)}) = \{\langle \texttt{x} \rangle\} \\
\texttt{relation}\ R_{tmem} = R_{thd}^*
\end{array}
$$

```
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`) to which it is applied. In return, `foldl` guarantees to produce a tree, which contains all the members of its list argument. It should be noted that a correct implementation of `treeInsert` will have the required type of `foldl`'s higher-order argument, after instantiating ${}'R_{bm}$ to R_{tmem}. Thus, the application of `foldl` in the above example typechecks, producing the required invariant of `makeTree`.

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

An alternative type for `foldl` could relate the order of elements in the argument list to some order of the result. The intuition is as follows: suppose the result type (`'b`) has some notion of order captured by a relation such that the result of `foldl`'s higher-order argument (`f`) has a refinement given in terms of this relation; i.e., it

says something about how the order relation of its result (`z`) relates to its arguments (`x` and `acc`). But, `x` comes from the list being folded, and `f` is applied over elements of this list in a pre-defined

Calculus λ_R

$x, y, z, \nu \in$ *variables* $\quad n \in$ *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	$::=$	$v \mid e\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid$ $\texttt{match}\ v\ \texttt{with}\ \mathsf{Cons}\, x\, y\ \Rightarrow\ e\, \texttt{else}\ e$	*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	$::=$	$r = r \mid r \subseteq r \mid \phi \wedge \phi \mid \phi \vee \phi \mid true$	*type refinement*
Δ_R	$::=$	$\langle R,\, \tau_R,\, \mathsf{Cons}\, x\, y \Rightarrow r \mid \mathsf{Nil} \Rightarrow r \rangle$ $\mid \langle R,\, \tau_R,\, R^* \rangle$	*relation def.*
θ	$::=$	$T \mid T * \theta$	*tuple sort*
τ_R	$::=$	$\mathsf{intlist} :\!\to \{\theta\} \mid \mathsf{int} :\!\to \{\theta\}$	*relation sort*

Figure 1: Language

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:

$$
\begin{aligned}
(\text{'}R_{bm},\text{'}R_{bo})\ \texttt{foldl} : &\{\texttt{l : 'a list}\} \rightarrow \{\texttt{b : 'b}\} \rightarrow \\
&(\{\texttt{f} : \{\texttt{x : 'a}\} \rightarrow \{\texttt{acc : 'b}\} \rightarrow \\
&\quad \{\texttt{z} \mid \text{'}R_{bo}(\texttt{z}) = (\{\langle \texttt{x} \rangle\} \times \text{'}R_{bm}(\texttt{acc})) \cup \\
&\qquad \text{'}R_{bo}(\texttt{acc}) \wedge ...\}) \rightarrow \\
&\{\nu \mid \text{'}R_{bo}(\nu) = R^*_{oa}\ (\texttt{l}) \cup \text{'}R_{bo}(\texttt{b})) \cup \\
&\qquad ((R_{mem}\ (\texttt{l})) \times \text{'}R_{bm}(\texttt{b})) \wedge ...\}
\end{aligned}
$$

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 (Rmem,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.

3. Core language

3.1 Syntax

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

standard call-by-value operational semantics, details of which can be found in an accompanying technical report [10].[4]

Dependent type refinements (ϕ) in λ_R are assertions over relational expressions (r); these expressions, which are themselves

[4] Proofs for all lemmas and theorems given in this paper are also provided in the report.

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

S-REL

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

S-REL-STAR

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

S-APP

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

S-REL-ID

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

S-UNION

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

S-CROSS

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

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

WF-RPRED

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

WF-REF

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

WF-BASE

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

WF-FUN

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

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

SUBT-BASE

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

SUBT-ARROW

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

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

T-VAR

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

T-ABS

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

T-MATCH

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

T-CONST

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

T-SUB

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

T-APP

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

T-LET

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

Figure 3: Static semantics of λ_R

typed, constitute the syntactic class of expressions in our specification language. We refer to the types of relational expressions as *sorts*, in order to distinguish them from λ_R types. We write $r :: s$ to denote that a relational expression r has sort s. A structural re-

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

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

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

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

3.2 Sorts, Types and Well-formedness

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

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

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

Structural relations are sort checked under an empty type environment. The rule S-REL type checks a relation definition by ensuring that relational expressions associated with the constructors that comprise the definition all have the same sort. The rule S-REL-

MSFOL

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

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

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

Auxiliary Definitions

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

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

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

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

$$
\begin{array}{rcl}
[\![\phi_1 \wedge \phi_2]\!] & = & [\![\phi_1]\!] \wedge [\![\phi_2]\!] \\
[\![\phi_1 \vee \phi_2]\!] & = & [\![\phi_1]\!] \vee [\![\phi_2]\!]
\end{array}
$$

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

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

Figure 4: Semantics of Specification Language

STAR captures the fact that an inductive extension of a relation has the same type as the relation itself. The rule S-APP sort checks relation applications by ensuring that the argument to the relation has the required simple (non-dependent) type. The rule makes use of a simple typing judgment ($\Vdash$) under a *refinement erased* Γ (denoted $\|\Gamma\|$) for this purpose. Rules for simple typing judgments are straightforward, and are elided here; the full set of rules can be found in the accompanying technical report [10].

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

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

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:

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

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

to be unfolded to pattern-match definitions before being bound in Σ_R:

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

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

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

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

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

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

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

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

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

where Cons has type[5]

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

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

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

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

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

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

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

3.3 Semantics of the Specification Language

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

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

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

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

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

Fig. 4 describes the MSFOL semantics of λ_R's specification language. The semantics is operational in the sense that it describes an algorithm to compile assertions in λ_R type refinements to formulas in MSFOL. Our semantics are parameterized over an auxiliary function ($\mathcal{F}$) that maps λ_R datatypes to uninterpreted sorts in MSFOL. The specific uninterpreted sorts types map to are not relevant here. However, $\mathcal{F}$ has to be a total function over λ_R datatypes. Note that despite treating interpreted types (eg: intlist and int) as

uninterpreted sorts in the underlying logic, the exercise of ascribing a semantics to the type refinement language is complete. This is because the interpretation of any type is the collection of operations allowed on that type, and our type refinement language does not contain operations that are specific to values of any specific type.

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

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

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

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

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

Auxiliary function *Inst* instantiates a prenex-quantified formula.

We employ the standard interpretation of set union and cross product operations, when sets are represented using prenex-quantified propositions:

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

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:

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

involving *membership* and *occurs-before* relations for integer lists:

$$\begin{aligned} R_{mem} \ &:: \mathsf{intlist} :\rightarrow \{\mathsf{int}\} \\ R_{ob} \ &:: \mathsf{intlist} :\rightarrow \{\mathsf{int}\texttt{*}\mathsf{int}\} \end{aligned}$$

The series of steps that compile the assertion to an MSFOL formula, which captures the semantics of the assertion, are shown in Fig. 5.[6] The example assumes that $\mathcal{F}$ maps `int` to sort $\mathtt{A}_0$, and `intlist` to sort $\mathtt{A}_1$.

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

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

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

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

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

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

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

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

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

$$Inst[\![\mathtt{R}_{ob}]\!]\ l \tag{2.1}$$

$$Inst(\ \forall(\mathtt{i} : [\![\mathtt{intlist}]\!]).\forall(\mathtt{j} : [\![\mathtt{int}]\!])\,. \tag{2.2}$$

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

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

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

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

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

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

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

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

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

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

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

$$\gamma_{\sqcup}(\ (\forall(\mathtt{j} : \mathtt{A}_0).\forall(\mathtt{k} : \mathtt{A}_0).(\mathtt{Rob\ x\ j\ k}))\ ,\ \Leftrightarrow, \tag{1.3}$$

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

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

Figure 5: Compiling a λ_R assertion to MSFOL

3.4 Decidability of λ_R Type Checking

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

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

Consequently, decidability of type checking in λ_R reduces to decidability of semantic entailment in MSFOL. Although semantic entailment is undecidable for full first-order logic, our subset of MSFOL is a carefully chosen decidable fragment. This fragment, known as Effectively Propositional (EPR) first-order logic, or Bernay-Schönfinkel-Ramsey (BSR) logic, consists of prenex quantified propositions with uninterpreted relations and equality. Off-the-shelf SMT solvers (e.g., Z3) are equipped with efficient decision procedures for EPR logic [19], making type checking in λ_R a practical exercise.

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

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

4. Parametricity

4.1 Syntax

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

The only algebraic data type in $\lambda_{\forall R}$ is a polymorphic list, which is the domain for structural relations. Consequently, structural relations have sort schemes (σ_R), akin to type schemes (σ) of the term

language. For example, the non-parametric head relation (R_{hd}) from Section 2, when defined over a polymorphic `'a list` will have sort scheme, $\forall$ `'a. 'a list` $:\!\!\rightarrow$ `'a`. The specification language also contains an expression ($\mathcal{R}\,T$) to instantiate a generalized type variable in parametric relation sorts.

A parametric relation generalizes a structural relation, just as a polymorphic list generalizes a monomorphic one. Our syntax

Calculus $\lambda_{\forall R}$

$$
\begin{array}{llll}
t & \in & \textit{tuple} - \textit{sort variables} & x, y, k \in \textit{variables} \\
\texttt{'a}, \texttt{'b} & \in & \textit{type variables} &
\end{array}
$$

$$
\begin{array}{lll}
T & ::= & \texttt{'a} \mid \texttt{'a list} \mid \mathsf{int} \qquad \textit{datatypes} \\
\tau & ::= & \{\nu : T \mid \Phi\} \mid (x:\tau) \to \tau \qquad \textit{dependent type} \\
\delta & ::= & \forall t. \forall (R :: \texttt{'a} :\!\!\rightarrow t).\ \delta \mid \tau \qquad \textit{parametric dep. type} \\
\sigma & ::= & \forall \texttt{'a}.\ \sigma \mid \delta \qquad \textit{type scheme}
\end{array}
$$

Specification Language

$$
\begin{array}{lll}
\Phi & ::= & \rho = \rho \mid \rho \subseteq \rho \mid \Phi \wedge \Phi \mid \mathit{true} \qquad \textit{type refinement} \\
\rho & ::= & \mathcal{R}(x) \mid \rho \cup \rho \mid \rho \times \rho \qquad \textit{rel. expression} \\
\mathcal{R} & ::= & \mathcal{R}\,T \mid \mathcal{R}\,\theta\,\mathcal{R} \mid R \qquad \textit{instantiation} \\
\theta & ::= & t \mid t * \theta \mid T * \theta \mid T \qquad \textit{tuple sort} \\
\tau_R & ::= & \forall t.\ (\texttt{'a} :\!\!\rightarrow t) :\!\!\rightarrow (\texttt{'a list} :\!\!\rightarrow \theta) \qquad \textit{relation sort} \\
 & \mid & \texttt{'a list} :\!\!\rightarrow \theta \\
\sigma_R & ::= & \forall \texttt{'a}.\ \tau_R \mid \tau_R \qquad \textit{sort scheme} \\
\Delta_R & ::= & \langle R,\ R_p,\ \sigma_R,\ \mathsf{Cons}\,x\,y \Rightarrow r \mid \mathsf{Nil} \Rightarrow r \rangle \qquad \textit{rel. definition} \\
 & \mid & \langle R,\ R_p,\ \sigma_R,\ \mathcal{R}^* \rangle
\end{array}
$$

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

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 the type variable 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 Sec. 2, can be instantiated with the non-parametric head relation, R_{hd}[7], after instantiating `'a` in its sort scheme with a `'b list`. The resulting relation can now be applied to a list of lists (i.e., a `'b list list`) to denote the set of head elements in the constituent lists.

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

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

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

In order to use a parametric relation in a type refinement, its relational parameter has to be instantiated. Polymorphism in $\lambda_{\forall R}$

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

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

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

Figure 7: Bind Syntax

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:

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

4.2 Sort and Type Checking

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

typed calculi, such as System F, as are rules to type-check generalization and instantiation expressions.

4.3 Semantics of Parametric Relations

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

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

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

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

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

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

We name the operator `bind`, after set monadic bind. The syntax of bind expressions is given in Fig. 7. For brevity, we exclude sort annotations on bind expressions (F_R) and bind abstractions (E_b) in our examples.

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

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

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

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

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

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

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

syntactic sugar over its underlying bind definition.

4.3.1 Elaboration to Bind Definition

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

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

We let R denote the relational parameter of R_x. The ground relation of R_x (call it $R_{x'}$) is the instantiated parametric relation $(R_x \ R_{id})$, which has the sort $\mathsf{int\ list} :\rightarrow \{\mathsf{int} * \mathsf{int} * \mathsf{int}\}$. From the type of `bind`, we know that the sort of the required transformer expression (F_R) is $(\mathsf{int}*\mathsf{int}*\mathsf{int}) :\rightarrow \{\mathsf{int}*t*t\}$. Recalling that F_R is a lambda bound relational expression, which is a cross product combination of relation applications (Fig. 7), we observe that the only possible solution for F_R is:

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

Consequently, we derive the following bind definition of R_x:

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

4.3.2 Bind Equations

By substituting parametric relations with their bind definitions, every instantiation of a parametric relation can be reduced to a bind abstraction (E_b in Figure 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`:

Semantics of Bind Equations $[\![\psi]\!]$

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

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

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

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:

$$\begin{array}{c}((R_{mem}\ R_{dup})\ l_1) = ((R_{mem}\ R_{dup})\ l_2) \\ \Rightarrow R_{mem}(l_1) = R_{mem}(l_2)\end{array}$$

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 $\langle y\rangle \in ((R_{mem}\ R_{dup})\ l)$.
- If $\langle y\rangle \in ((R_{mem}\ R_{dup})\ l)$, 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, and the 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 [15], a whole-program optimizing compiler for Standard ML (SML).[8] The input to our system is CoreML, an A-normalized intermediate representation with pattern-matching, but with all SML module constructs elaborated and removed. SML programs are annotated with relational specifications, defined in terms of relational dependent types that decorate function signatures, along with definitions of parameterized structural relations over the program's datatypes. The type system is a conservative

[8] The source code for the implementation as well as a Web interface to the system is available online from: *https://github.com/tycon/catalyst*.

extension of SML's, so all programs that are well-typed under CATALYST are well-typed SML programs. Our type-checking and verification process closely follows the description given in the previous sections. Verification conditions, representing the consequent of the SUBT-BASE type-checking rule (Fig. 3) are compiled to a first-order formula, as described in Sections 3 and 4, and checked for validity (satisfiability of its negation) using the Z3 SMT solver.

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

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

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

Similarly, R_{eq_k} can be defined as:

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

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

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

(a) balance

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

(b) Relational specification of `balance`

Figure 9: Red-Black Tree Example

For non-recursive function applications, although it is possible to infer instantiation annotations for parametric relations with the help of an expensive fixpoint computation that generates an exhaustive list of all possible instantiations, CATALYST relies on manual annotations for parameter instantiations to avoid this cost. An example of such annotation is shown in Fig. 10c (the `contains` function).

5.1 Experiments

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

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

For several of these benchmarks (especially those in (1) and (2)), CATALYST was able to successfully verify specifications to the extent of full functional correctness. Excluding the time take by the MLton compiler to elaborate and type check these Standard ML programs, none of our benchmarks take more than 0.2s to verify;

this time includes A-Normalization, specification elaboration, VC generation, and SMT solving through Z3.

Red-Black Tree. The specification of the red-black tree `balance` function, shown in Fig. 9b, illustrates the kind of specifications that were automatically verified by CATALYST in our experiments. The specification asserts that the `balance` function on red-black trees (Fig. 9a) preserves a *total-order* among members of the tree. The non-inductive *total-order* relation (R_{to} in Fig. 9b) is defined in terms of the tree *membership* relation (R_{tmem}) described in Sec. 2.3, and relates (a) elements in the left sub-tree to the root element, (b) root to the elements in the right sub-tree, and (c) elements in the left sub-tree to those in right. The inductive *total-order* relation (R^*_{to}) on a red-black tree, obtained by closing the R_{to} relation over the tree, relates every pair of elements in the tree that are *in-order*. Consequently, the specification of the `balance` function effectively asserts that in-order traversal over an unbalanced red-black tree, and in-order traversal on its balanced version, return the same sequence of elements.

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

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

SSA. An important intermediate representation used in MLton is

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

A program in SSA form is represented as a tree of basic blocks, where each block consists of a set of straight-line instructions (e.g., definitions, assignments, primitive applications). The specification of an SSA program makes use of several inductive relations: R_{du}, the *def-use* relation, R_{ud}, the *use-def* relation, and $R_{use\text{-}refl}$, the reflective variant of R_{use}, the *use* relation, that collects all variables used on the right-hand side of an assignment. The *def-use* relation relates a *def*, i.e., a variable that is defined using an assignment statement, to all *use*s that are dominated by the definition. Conversely, R_{ud} relates a *use* to all *def*s that it dominates. With these definitions, we can express the type of an SSA tree thus:

$$\texttt{type ssa\_tree} = \{\nu : \texttt{block tree} \mid R_{use\text{-}refl}(\nu) \subseteq R_{du}(\nu) \wedge R_{use\text{-}refl}(\nu) \cap R_{ud}(\nu) = \emptyset\}$$

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

$$\texttt{ssa\_tree} \rightarrow \texttt{ssa\_tree}$$

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

6. Case Study

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

exists. Consider the higher-order `exists` function over lists shown in Fig. 10a; dependent type signatures are elided for brevity.


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

(a) `exists`

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

(b) `filter`

Figure 10: Examples

ML Program

```
1  datatype exp =     Var of string
2                   | App of exp*exp                    21 and subst e1
3                         | Abs of string*exp
4
```

```
5  fun freeVars e = case e of
6      Var id => [id]
7    | App (e1,e2) =>
8       concat [freeVars e1, freeVars e2]
9    | Abs (id,e') => filter (RNeq id)
10       (fn fv => not (fv = id)) (freeVars e')
11
12 fun alphaConvert e = case e of
13     Abs (id,e') =>
14     let
15       val fv_e' = freeVars e'
16       val id' = createNewName fv_e' id
17     in
18       Abs(id',subst(Var id',id,e'))
19     end
20   | _ => raise Error
```

Relational Specification

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

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

```
      fun contains l str =
        let
          val isStr = fn x => x=str
          (* Instantiate the implicit
```

```
        * relational parameter in type
        * of "exists" with (REq str) *)
      val hasStr = exists (REq str)
          isStr l
  in
    hasStr
  end
```

(c) contains

```
22  id e2  = case e2 of
23        Var id' => if id = id'
24          then e1 else e2
25      | App(e21,e22) =>
26        let
27          val e21' = subst e1 id e21,
28          val e22' = subst e1 id e22
29        in
30          App (e21',e22')
31        end
32      | Abs(id',e2') => if id' = id then e2 else
33        let
34          val fv_e1 = freeVars e1
35        in
36          if contains fv_e1 id'
37          then subst e1 id (alphaConvert e2)
38          else Abs(id',subst e1 id e2')
```

```
38      end
```

```
not ({(v)} ⊆ Rmem(fvs))};
```

```
= (R_fv(e2) - {(id)}) ∪ R_fv(e1) else R_fv(ex) = R_fv(e2)};
```

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

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:

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

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 element in list.

filter. A parametric dependent type for `filter`, shown in Fig.

10b is given below:

$$\texttt{('}R\ \texttt{filter) :}$$
$$\texttt{l} \rightarrow \texttt{f : x} \rightarrow \{\ \nu\ \texttt{|}\ \nu\texttt{=false} \Rightarrow \texttt{'}R(\texttt{x}) = \emptyset$$
$$\wedge\ \nu\texttt{=true} \Rightarrow \texttt{'}R(\texttt{x}) = R_{id}(\texttt{x})\ \} \rightarrow$$
$$\{\nu\ \texttt{|}\ R_{mem}(\nu)\ \texttt{=}\ ((R_{mem}\ \texttt{'}R)\ \texttt{l})\ \}$$

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

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

```
val isStr = fn x => x=str
```

the relational dependent type of `isStr` is:

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

This clearly suggests that the relational parameter of `exists` has to be instantiated with $R_{eq_{str}}$. 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:

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

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 [6, 26]. Some of the behaviors exhibited by incorrect implementations include (a) α-conversion renames a free variable, or fails to rename a bound variable; (b) substitution fails to substitute free occurrences of the variable (`id`), or substitutes a bound occurrence of the variable; or (c) substitution is not capture-avoiding, i.e., substituting `e1` for `id` in `e2` captures variables of `e1` , which are otherwise free.

The relational specification of substitution and α conversion is given in the bottom-half of Fig. 11.[9] Note that one need not expose notions of capture-avoidance, or other such intricacies, to write

down the specification, which is given in terms of a new structural relation R_{fv} that relates an expression of the calculus to its free variables. Function `freeVars` returns a list, whose members are free variables of its input expression. Its type represents this fact.

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

7. Related Work

Type systems of mainstream functional languages, such as GHC Haskell and OCaml, support a basic form of dependent typing [12, 13] using GADTs [27]. At a high level, a structural relation of a data type is similar to a GADT insofar as it corresponds to an index that tracks an inductively definable relation over the data type. However, unlike the indexed type systems of Haskell and OCaml, where types are kept separate from terms, ours is a dependent type system. In this sense, our type system is similar to the refinement based dependent type system of F* [23]. Type refinements in F* are drawn from unrestricted (higher-order) logic extended with theories, whereas our specification language for ML programs is an abstraction over first-order logic that was tailor-made for equational and relational reasoning. The expressivity gained by allowing unrestricted type refinements in F* comes at the cost of decidability

of type checking.

Structural relations, in their operational manifestation, can be compared to the structurally recursive *measures* of liquid types

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

[11, 25] where the co-domain is always a set. Parametric structural relations may be viewed as generalizing such measures to higher-order measures. Relationally parametric dependent types can be compared to liquid types with abstract refinements [25], which let liquid types parameterize over type refinements (Boolean predicates). Once applied to a value, an abstract refinement becomes a concrete refinement, which can only be used to refine a type. On the other hand, a relational parameter can be treated just as any other relation in our type refinements, including being passed as an argument to other parametric relations. We require this generality to reason about shape invariants of higher-order catamorphisms such as `map` and `foldr`. For example, using only abstract refinements, it is not possible to verify that projecting a list of pairs using `map` and `fst` preserves ordering, or that an implementation of list `append` that uses `foldr` is correct.

Measures are an example of structurally recursive abstraction functions that map an algebraic data type to an abstract domain, such as natural numbers or sets. Suter *et al.* [22] describe decision procedures for the theory of algebraic data types extended with abstraction functions to decidable abstract domains. Our encoding does not require such extensions since a structural relation directly translates to an uninterpreted relation in first-order logic. Our encoding also supports parametric relations, which would otherwise require higher-order abstraction functions.

Imperative shape analyses have previously used relations to cap-

ture some inductive properties [5], and to describe memory configurations [9]. However, their applicability has been limited owing to destructive updates and pointer manipulations in imperative programs. In [14], 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 [1, 7]. 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 [2–4]. In these efforts, a binary logical relation that relates effectful expressions is parameterized by a relation that relates their states. In contrast, a parametric structural relation is a structural relation over a polymorphic data type, that is parameterized by relations over type variables in the data type. While the primary purpose of structural relations is to enable specification and static verification, there is a possibility of sufficiently equipping our framework to reason about invariance of arbitrary relations, which is the key to reasoning about contextual equivalence. This is a possible avenue for future research.

Henglein [8] describes a domain-specific language to define ordering relations for composite data types such as lists and trees. However, the notion of order explored is the domain order used to compare two elements of same domain, such as a lexicographic order. In contrast, the order relation in our system describes relative

ordering of elements in a composite data type.

8. Future Work

Due to the undecidability of program equivalence in general, it is impossible for any specification language that is based on a decidable logic to completely specify functional correctness of all possible ML programs. The expressivity of our specification language is inherently bound by the limits imposed by our choice of the un-

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

One noticeable limitation of our current system is the lack of a general type inference mechanism. Given that relational specifications which make use of parametric relations to express rich invariants are non-trivial, and can be quite verbose, writing such specifications sometimes requires considerable manual effort. While providing higher level abstractions in the specification language can mitigate the problem by enabling the programmer to reason directly at the level of properties, rather than at the level of relations, the approach can be substantiated with a lightweight type inference mechanism based on refinement templates [20] to reduce the burden of manual annotation. The integration of such mechanisms within CATALYST is another avenue we anticipate pursuing.

9. Conclusions

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

Acknowledgments

We thank Matt Might, Ranjit Jhala, and the anonymous reviewers for their detailed comments and suggestions. This work is supported by the National Science Foundation under grants CCF-1216613 and CCF-1318227.

References

[1] A. Ahmed. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *ESOP'06*, pages 69–83, 2006.

[2] N. Benton and B. Leperchey. Relational Reasoning in a Nominal Semantics for Storage. In *TLCA*, pages 86–101, 2005.

[3] N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, Writing and Relations: Towards Extensional Semantics for Effect Analyses. In *APLAS*, pages 114–130, 2006.

[4] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational

Semantics for Effect-based Program Transformations: Higher-order Store. In *PPDP*, pages 301–312, 2009.

[5] B.-Y. E. Chang and X. Rival. Relational Inductive Shape Analysis. In *POPL*, pages 247–260, 2008.

[6] A. Charguraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012. ISSN 0168-7433.

[7] D. Dreyer, A. Ahmed, and L. Birkedal. Logical Step-Indexed Logical Relations. In *LICS'09*, pages 71–80, 2009.

[8] F. Henglein. Generic Top-down Discrimination for Sorting and Partitioning in Linear Time*. *J. Funct. Program.*, pages 300–374, 2012.

[9] 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.

[10] G. Kaki and S. Jagannathan. A Relational Framework for Higher-Order Shape Analysis. Technical Report TR-14-002, Purdue University, 2014. URL `http://docs.lib.purdue.edu/cstech/1772/`.

[11] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based Data Structure Verification. In *PLDI*, pages 304–315, 2009.

[12] S. Lindley and C. McBride. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In *Haskell Symposium*, pages 81–92, 2013.

[13] C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

[14] M. Might. Shape Analysis in the Absence of Pointers and Structure. In *VMCAI*, pages 263–278, 2010.

[15] MLton. `http://mlton.org/`.

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

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

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

[19] R. Piskac, L. de Moura, and N. Bjørner. Deciding Effectively Propositional Logic with Equality. Technical Report MSR-TR-2008-181.

[20] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid Types. In *PLDI*, pages 159–169, 2008.

[21] P.-Y. Strub, N. Swamy, C. Fournet, and J. Chen. Self-Certification: Bootstrapping Certified Typecheckers in F* with Coq. In *POPL*, pages 571–584, 2012.

[22] P. Suter, M. Dotta, and V. Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *POPL*, pages 199–210, 2010.

[23] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278, 2011.

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

[25] N. Vazou, P. M. Rondon, and R. Jhala. Abstract Refinement Types. In *ESOP*, pages 209–228, 2013.

[26] S. Weirich, B. A. Yorgey, and T. Sheard. Binders Unbound. In *ICFP*, pages 333–345, 2011.

[27] H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors. In *POPL*, pages 224–235, 2003.