Type-based Data Structure Verification *

Ming Kawaguchi
University of California, San Diego
mwookawa@cs.ucsd.edu

Patrick Rondon
University of California, San Diego
prondon@cs.ucsd.edu

Ranjit Jhala
University of California, San Diego
jhala@cs.ucsd.edu

Abstract

We present a refinement type-based approach for the static verification of complex data structure invariants. Our approach is based on the observation that complex data structures are typically fashioned from two elements: recursion (e.g., lists and trees), and maps (e.g., arrays and hash tables). We introduce two novel type-based mechanisms targeted towards these elements: recursive refinements and polymorphic refinements. These mechanisms automate the challenging work of generalizing and instantiating rich universal invariants by piggybacking simple refinement predicates on top of types, and carefully dividing the labor of analysis between the type system and an SMT solver [6]. Further, the mechanisms permit the use of the abstract interpretation framework of liquid type inference [22] to automatically synthesize complex invariants from simple logical qualifiers, thereby almost completely automating the verification. We have implemented our approach in DSOLVE, which uses liquid types to verify OCAML programs. We present experiments that show that our type-based approach reduces the manual annotation required to verify complex properties like sortedness, balancedness, binary-search-ordering, and acyclicity by more than an order of magnitude.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Dependent Types, Hindley-Milner, Predicate Abstraction, Type Inference

1. Introduction

Recent advances in Satisfiability Modulo Theories (SMT) solving, Model Checking and Abstract Interpretation have made it possible to build tools that automatically verify safety properties of large software systems. However, the inability of these tools to automatically reason about data structures severely limits the kinds of properties they can verify.

The challenge of automating data structure verification stems from the need to reason about relationships between the unbounded number of values comprising the structure. In SMT and theorem proving based approaches, manual effort is needed to help the prover to generalize relationships over specific values into universal facts that hold over the structure, and dually, to instantiate quantified facts to obtain relationships over particular values. In model checking and abstract interpretation based approaches, manual effort is needed to design, for each data structure, an abstract domain that is capable of generalizing and instantiating relevant relationships.

Types provide a robust means for reasoning about coarse-grained properties of an unbounded number of values. For example, if a variable `x` is of the type `int list`, then we are guaranteed that every value in the list is an integer. Similarly, if a variable `g` is of the type $(\texttt{int}, \texttt{int})$ `Map.t`, then we are guaranteed that `g` is a hash-map where every key is an integer that is mapped to another integer. Modern type systems automatically generalize such global properties from local properties of the individual values added to the structure, and dually, automatically instantiate the properties when the structures are accessed.

In this paper, we show how coarse-grained types can be refined with simple, quantifier-free predicates to yield a framework that is expressive enough to enable the specification of a variety of complex data structure properties, yet structured enough to enable automatic verification and inference of the properties. We reconcile expressiveness and automation through two novel type refinement mechanisms targeted towards the two elements from which complex structures are fashioned—recursion (e.g., lists and trees) and maps (e.g., arrays and hash tables).

The first mechanism, *recursive refinements*, allows the recursive type that encodes a structure to be refined with a matrix of predicates that individually refine the elements of the recursive type. Recursive refinements allow us to express uniform properties, for example that a list contains values greater than some program variable `i`, abbreviated as $\{\nu\!:\!\texttt{int} \mid \texttt{i} < \nu\}$ `list`. Further, by recursively propagating the matrix when the type is folded or unfolded, the mechanism allows us to algorithmically analyze rich nested properties like sortedness, distinctness and binary-search-ordering.

The second mechanism, *polymorphic refinements*, allows the polymorphic type schemes of map data types to be refined, enabling the types of stored values to depend on the keys used to access them. For example, the polymorphic refinement type $(\texttt{i}\!:\!\alpha, \beta)$ `Map.t` corresponds to a hash-map where each key `i` has the (refined) type α, and is mapped to a value of type β, which can depend on the key `i`. For example, $(\texttt{i}\!:\!\texttt{int}, \{\nu\!:\!\texttt{int} \mid \texttt{i} < \nu\})$ `Map.t` specifies a hash-map where each integer key is mapped to a value greater than the key. Suppose that the edge-adjacency relation of a graph is represented as a hash-map that maps each node's integer identifier to its list of succes-

* This work was supported by NSF CAREER grant CCF-0644361, NSF PDOS grant CNS-0720802, NSF Collaborative grant CCF-0702603, and a gift from Microsoft Research.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

sors. Then, $(\mathtt{i}\!:\!\mathtt{int}, \{\nu\!:\!\mathtt{int} \mid \mathtt{i} < \nu\}\ \mathtt{list})$ Map.t specifies that the graph is acyclic, as each successor is greater than its source.

Our key insight is that, by piggybacking quantifier-free refinement predicates on top of types, the complementary strengths of types and SMT solvers can be harnessed to automatically verify data structures. The typing rules encode an algorithm for quantifier generalization and instantiation, while subtyping reduces quantified relationships into quantifier-free checks over simple predicates which can then be efficiently discharged by an SMT solver. Further, we can use the abstract interpretation framework of liquid types [22] to automatically infer refinements from a set of simple logical qualifiers, thereby eliminating the prohibitive cost of writing type annotations for all functions and polymorphic instantiations.

We have implemented recursive and polymorphic refinements in DSOLVE, a tool that infers liquid types for OCAML programs. We describe experiments using DSOLVE to verify a variety of complex data structure invariants, including sortedness, balancedness, binary-search-ordering, variable ordering, set-implementation, heap-implementation, and acyclicity, on several benchmarks including textbook data structures, such as sorted lists, union-find, splay heaps, AVL trees, red-black trees, and heavily-used libraries implementing stable sort, heaps, associative maps, extensible vectors, and binary decision diagrams. Previously, the verification of such properties required significant manual annotations in the form of loop invariants, pre- and post- conditions, and proof scripts totaling several times the code size. In contrast, DSOLVE is able to verify complex properties using a handful of simple qualifiers amounting to 3% of the code size.

2. Overview

We begin with an overview of our type-based verification approach. First, we review simple refinement and liquid types. Next, we describe recursive and polymorphic refinements, and illustrate how they can be used to verify data structure invariants.

Refinement Types. Our system is built on the notion of refining ML types with predicates over program values that specify additional constraints which are satisfied by all values of the type [2, 9]. Base values, for example, those of type integer (denoted `int`), can be described as $\{\nu\!:\!\mathtt{int} \mid e\}$ where ν is a special *value variable* not appearing in the program, and e is a boolean-valued expression constraining the value variable called the *refinement predicate*. Intuitively, the base refinement predicate specifies the set of values c of the base type B such that the predicate $e[c/\nu]$ evaluates to true. For example, $\{\nu\!:\!\mathtt{int} \mid \nu \leq \mathtt{n}\}$ specifies the set of integers whose value is less than or equal to the value of the variable `n`. We use the base refinements to build up *dependent function types*, written $x\!:\!T_1 \!\rightarrow\! T_2$. Here, T_1 is the domain type of the function, and the formal parameter x may appear in the base refinements of the range type T_2. For example, $x\!:\!\mathtt{int} \!\rightarrow\! \{\nu\!:\!\mathtt{int} \mid x \leq \nu\}$ is the type of a function that takes an input integer and returns an integer greater than the input. Thus, the type `int` abbreviates $\{\nu\!:\!\mathtt{int} \mid \top\}$, where $\top$ and $\bot$ abbreviate *true* and *false* respectively.

Liquid Types. A *logical qualifier* is a boolean-valued expression (i.e., predicate) over the program variables, the special value variable ν which is distinct from the program variables, and the special placeholder variable $\star$ that can be instantiated with program variables. We say that a qualifier q *matches* the qualifier q' if replacing some subset of the free variables in q with $\star$ yields q'. For example, the qualifier $i \leq \nu$ matches the qualifier $\star \leq \nu$. We write $\mathbb{Q}^\star$ for the set of all qualifiers *not containing* $\star$ that match some qualifier in $\mathbb{Q}$. In the rest of this section, let $\mathbb{Q}$ be the qualifiers

$$\mathbb{Q} = \{0 < \nu,\ \star \leq \nu\}.$$

A *liquid type over* $\mathbb{Q}$ is a dependent type where the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$. We write *liquid type* when $\mathbb{Q}$ is clear from the context. We can automatically *infer* refinement types by requiring that certain expressions like recursive functions have liquid types [22].

Safety Verification. Refinement types can be used to statically prove safety properties by encoding appropriate preconditions into the types of primitive operations. For example, to prove that no divide-by-zero or assertion failures occur at run-time, we can type check the program using the types $\mathtt{int} \!\rightarrow\! \{\nu\!:\!\mathtt{int} \mid \nu \neq 0\} \!\rightarrow\! \mathtt{int}$ and $\{\nu\!:\!\mathtt{bool} \mid \nu\} \!\rightarrow\! \mathtt{unit}$ for division and `assert` respectively.

2.1 Recursive Refinements

The first building block for data structures is recursion, formalized using recursive types, which is used to fashion structures like lists and trees. The key idea behind recursive refinements is that recursive types can be refined using a matrix of predicates, where each predicate applies to a particular element of the recursive type. For ease of exposition, we consider recursive types whose body is a sum-of-products. Each product can be refined using a *product refinement*, which is a vector of predicates where the j^{th} predicate refines the j^{th} element of the product. Each sum-of-products (and hence the entire recursive type), can be refined with a *recursive refinement*, which is a vector of product refinements, where the i^{th} product refinement refines the i^{th} element of the sum.

Uniform Refinements. The ML type for integer lists is:

$$\mu t.\mathtt{Nil}\ +\ \mathtt{Cons}\langle x_1\!:\!\mathtt{int}, x_2\!:\!t\rangle$$

a recursive sum-of-products which we abbreviate to `int list`. We specify a list of integers each of which satisfies a predicate p as $(\langle\langle\rangle;\ \langle p;\ \top\rangle\rangle)$ `int list`. For example, $(\langle\langle\rangle;\ \langle \mathtt{i} \leq \nu;\ \top\rangle\rangle)$ `int list` specifies lists of integers greater than some program variable `i`. This representation reduces specification and inference to determining which logical qualifiers apply at each position of the refinement matrix. In the sequel, for any expression e and relation $\bowtie$ we define the abbreviation $\rho^e_{\bowtie}$ as:

$$\rho^e_{\bowtie} \doteq \langle\langle\rangle;\ \langle e \bowtie \nu;\ \top\rangle\rangle \tag{1}$$

To see how such uniform refinements can be used for verification, consider the program in Figure 1. The function `range` takes two integers, `i` and `j`, and returns the list of integers $\mathtt{i}, \ldots, \mathtt{j}$. The function `harmonic` takes an integer `n` and returns the $\mathtt{n}^{th}$ (scaled) harmonic number. To do so, `harmonic` first calls `range` to get the list of denominators $1, \ldots, \mathtt{n}$, and then calls `List.fold_left` with an accumulator to compute the harmonic number. To prove that the divisions inside `harmonic` are safe, we need to know that all the integers in the list `is` are non-zero. Using $\mathbb{Q}$, our system infers:

$$\mathtt{range} :: \mathtt{i}\!:\!\mathtt{int} \!\rightarrow\! \mathtt{j}\!:\!\mathtt{int} \!\rightarrow\! (\rho^{\mathtt{i}}_{\leq})\ \mathtt{int\ list}$$

By substituting the actuals for the formals in the inferred type for `range`, our system infers that the variable `ds` has the type: $(\rho^1_{\leq})$ `int list`. As the polymorphic ML type for `List.fold_left` is $\forall\alpha,\beta.(\alpha \!\rightarrow\! \beta \!\rightarrow\! \alpha) \!\rightarrow\! \alpha \!\rightarrow\! \beta\ \mathtt{list} \!\rightarrow\! \alpha$, our system infers that, at the application to `List.fold_left` inside `harmonic`, α and β are respectively instantiated with `int` and $\{\nu\!:\!\mathtt{int} \mid 0 < \nu\}$, and hence, that the accumulator has the type:

$$\mathtt{s}\!:\!\mathtt{int} \!\rightarrow\! \mathtt{k}\!:\!\{\nu\!:\!\mathtt{int} \mid 0 < \nu\} \!\rightarrow\! \mathtt{int}$$

As `k` is strictly greater than 0, our system successfully typechecks the application of the division operator $/$, proving the program "division safe". Hence, by refining the base types that appear inside recursive types, we can capture invariants that hold uniformly across all the elements within the recursively defined value.

Nested Refinements. The function `range` returns an *increasing* sequence of integers. Recursive refinements can capture this invariant

```
let rec range i j =
  if i > j then
    []
  else
    let is = range (i+1) j in
    i::is

let harmonic n =
  let ds = range 1 n in
  List.fold_left
    (fun s k -> s + 10000/k)
    0 ds
```

Figure 1. Divide-by-zero

```
let rec insert x ys =
  match ys with
  | [] -> [x]
  | y::ys' ->
     if x < y then x::y::ys'
     else y::(insert x ys')

let rec insertsort xs =
  match xs with
  | [] -> []
  | x::xs' ->
     insert x (insertsort xs')
```

Figure 2. Insertion Sort

```
let fib i =
  let rec f t0 n =
    if mem t0 n then
      (t0, get t0 n)
    else if n <= 2 then
      (t0, 1)
    else
      let (t1,r1) = f t0 (n-1) in
      let (t2,r2) = f t1 (n-2) in
      let r       = r1 + r2 in
      (set t2 n r, r) in
  snd (f (new 17) i)
```

Figure 3. Memoization

```
let rec build_dag (n, g) =
  let node = random () in
  if (0>node || node>=n) then
    (n, g)
  else
    let succs  = get g node in
    let succs' = (n+1)::succs in
    let g'     = set g node succs' in
    build_dag (n+1, g')

let g0     = set (new 17) 0 []
let (_,g1) = build_dag (1, g0)
```

Figure 4. Acyclic Graph

by applying the recursive refinement to the μ-bound variables *inside* the recursive type. For each type τ define:

$$\tau\ \mathtt{list}_{\leq}\ \doteq\ \mu t.\mathtt{Nil}\ +\ \mathtt{Cons}\langle x_1\!:\!\tau, x_2\!:\!(\rho_{\leq}^{x_1})\ t\rangle \qquad (2)$$

Thus, a list of increasing integers is $\mathtt{int\ list}_{\leq}$. This succinctly captures the fact that the list is increasing, since the result of "unfolding" the type by substituting each occurrence of t with the entire recursively refined type is:

$$\mathtt{Nil}\ +\ \mathtt{Cons}\langle x_1'\!:\!\mathtt{int}, x_2'\!:\!(\rho_{\leq}^{x_1'})\ \mathtt{int\ list}_{\leq}\rangle$$

where x_1', x_2' are fresh names introduced for the top-level "head" and "tail". Intuitively, this unfolded sum type corresponds to a value that is either the empty list or a cons of a head x_1' and a tail which is an increasing list of integers greater than x_1'.

For inference, we need only find which qualifiers flow into the recursive refinement matrices. Using $\mathbb{Q}$, our system infers that `range` returns an increasing list of integers no less than `i`:

$$\mathtt{range} :: \mathtt{i}\!:\!\mathtt{int}\!\rightarrow\!\mathtt{j}\!:\!\mathtt{int}\!\rightarrow\!(\rho_{\leq}^{\mathtt{i}})\ \mathtt{int\ list}_{\leq}$$

As another example, consider the insertion sort function from Figure 2. In a manner similar to the analysis for `range`, using just $\mathbb{Q}$, and no other annotations, our system infers that `insert` has the type $\mathtt{x}\!:\!\alpha\!\rightarrow\!\mathtt{ys}\!:\!\alpha\ \mathtt{list}_{\leq}\!\rightarrow\!\alpha\ \mathtt{list}_{\leq}$, and hence that program correctly sorts lists. Furthermore, the system infers that `insertsort` has the type $\mathtt{xs}\!:\!\alpha\ \mathtt{list}\!\rightarrow\!\alpha\ \mathtt{list}_{\leq}$.

Structure Refinements. Suppose we wish to check that `insertsort`'s output list has the same *set of elements* as the input list. We first specify what we mean by "the set of elements" of the list using a *measure*, an inductively-defined, terminating function that we can soundly use in refinements. The following specifies the set of elements in a list:

```
measure elts = []     -> empty
             | x::xs -> union (single x) (elts xs)
```

where `empty`, `union` and `single` are primitive constants corresponding to the respective set values and operations. Using just the measure specification and the SMT solver's decidable theory of sets our system infers that `insertsort` has the type:

$$\mathtt{xs}\!:\!\alpha\ \mathtt{list}\!\rightarrow\!\{\nu\!:\!\alpha\ \mathtt{list}_{\leq} \mid \mathtt{elts}\ \nu = \mathtt{elts}\ \mathtt{xs}\}$$

i.e., the output list is sorted and has the same elements as the input list. In Section 6 we show how properties like balancedness can be verified by combining measures (to specify heights) and recursive refinements (to specify balancedness at each level).

2.2 Polymorphic Refinements

The second building block for data structures is finite maps, such as arrays, vectors, hash tables, *etc.* The classical way to model maps is using the array axioms [15], and by using algorithmically problematic, universally quantified formulas to capture properties over *all* key-value bindings. Polymorphic refinements allow the implicit representation of universal map invariants within the type system and provide a strategy for generalizing and instantiating quantifiers in order to verify and infer universal map invariants.

To this end, we extend the notion of parametric polymorphism to include refined polytype variables and schemas. Using polymorphic refinements, we can give a polymorphic map the type $(\mathtt{i}\!:\!\alpha, \beta)\ \mathtt{Map.t}$. Intuitively, this type describes a map where every key `i` of type α is mapped to a value of type β and β can refer to `i`. For example, if we instantiate α and β with `int` and $\{\nu\!:\!\mathtt{int} \mid 1 < \nu \wedge \mathtt{i} - 1 < \nu\}$, respectively, the resulting type describes a map from integer keys `i` to integer values strictly greater than 1 and $\mathtt{i} - 1$.

Memoization. Consider the memoized fibonacci function `fib` in Figure 3. The example is shown in the SSA-converted style of Appel [1], with `t0` being the input name of the memo table, and `t1` and `t2` the names after updates. To verify that `fib` always returns a value greater than 1 and (the argument) $\mathtt{i} - 1$, we require the universally quantified invariant that every key `j` in the memo table `t0` is mapped to a value greater than 1 and $\mathtt{j} - 1$. Using the qualifiers $\{1 \leq \nu, \star - 1 \leq \nu\}$, our system infers that the polytype variables α and β can be instantiated as described above, and so the map `t0` has type $(\mathtt{i}\!:\!\mathtt{int}, \{\nu\!:\!\mathtt{int} \mid 1 \leq \nu \wedge \mathtt{i} - 1 \leq \nu\})\ \mathtt{Map.t}$, i.e., every integer key i is mapped to a value greater than 1 and $\mathtt{i} - 1$. Using this, the system infers that `fib` has type $\mathtt{i}\!:\!\mathtt{int}\!\rightarrow\!\{\nu\!:\!\mathtt{int} \mid 1 \leq \nu \wedge i - 1 \leq \nu\}$.

Directed Graphs. Consider the function `build_dag` from Figure 4, which represents a directed graph with the pair $(\mathtt{n}, \mathtt{g})$ where `n` is the number of nodes of the graph, and `g` is a map that encodes the link structure by mapping each node, an integer less than `n`, to the list of its directed successors, each of which is also an integer less than `n`. In each iteration, the function `build_dag` randomly chooses a node in the graph, looks up the map to find its successors, creates a new node $\mathtt{n} + 1$ and adds $\mathtt{n} + 1$ to the successors of the node in the graph. As each node's successors are greater than the node, there can be no cycles in the graph. Using just the qualifiers $\{\nu \leq \star, \star < \nu\}$ our system infers that the function `build_dag` inductively constructs a directed acyclic graph. Formally, the system infers that `g1` has type

$$\mathit{DAG} \doteq (\mathtt{i}\!:\!\mathtt{int}, (\langle\langle\rangle; \langle \mathtt{i} < \nu; \top\rangle\rangle)\ \mathtt{int\ list})\ \mathtt{Map.t} \qquad (3)$$

which specifies that each node `i` has successors that are *strictly greater* than `i`. Thus, by combining recursive and polymorphic refinements over simple quantifier-free predicates, our system infers complex shape invariants about linked structures.

3. Language

We begin by reviewing the core language NanoML from [22], by presenting its syntax and static semantics. NanoML has a strict, call-by-value semantics, formalized using a small-step operational

e	$::=$		*Expressions:*
	$\mid$	x	variable
	$\mid$	$\mathtt{c}$	constant
	$\mid$	$\lambda x.e$	abstraction
	$\mid$	$x\ x$	application
	$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e$	if-then-else
	$\mid$	$\mathtt{let}\ x = e\ \mathtt{in}\ e$	let-binding
	$\mid$	$\mathtt{fix}\ x.e$	fixpoint
	$\mid$	$[\Lambda\alpha]e$	type-abstraction
	$\mid$	$[\tau]e$	type-instantiation
Q	$::=$		*Liquid Refinements*
	$\mid$	$\top$	true
	$\mid$	q	qualifier in $\mathbb{Q}^\star$
	$\mid$	$Q \wedge Q$	conjunction
B	$::=$		*Base:*
	$\mid$	$\mathtt{int}$	integers
	$\mid$	$\mathtt{bool}$	booleans
	$\mid$	α	type variable
$\mathbb{A}(\mathbb{B})$	$::=$		*Unrefined Skeletons:*
	$\mid$	B	base
	$\mid$	$x{:}\mathbb{T}(\mathbb{B}) \rightarrow \mathbb{T}(\mathbb{B})$	function
$\mathbb{T}(\mathbb{B})$	$::=$		*Refined Skeletons:*
	$\mid$	$\{\nu{:}\mathbb{A}(\mathbb{B}) \mid \mathbb{B}\}$	refined type
$\mathbb{S}(\mathbb{B})$	$::=$		*Type Schema Skeletons:*
	$\mid$	$\mathbb{T}(\mathbb{B})$	monotype
	$\mid$	$\forall\alpha.\mathbb{S}(\mathbb{B})$	polytype
τ, σ	$::=$	$\mathbb{T}(\top), \mathbb{S}(\top)$	*Types, Schemas*
T, S	$::=$	$\mathbb{T}(E), \mathbb{S}(E)$	*Depend. Types, Schemas*
$\hat{T}, \hat{S}$	$::=$	$\mathbb{T}(Q), \mathbb{S}(Q)$	*Liquid Types, Schemas*

Figure 5. NanoML**: Syntax**

semantics [21]. In Sections 4 and 5, we extend NanoML with recursive and polymorphic refinements respectively.

Expressions. The expressions of NanoML, summarized in Figure 5, include variables, primitive constants, $\lambda-$abstractions and function application. In addition, NanoML has `let`-bindings and recursive function definitions using the fixpoint operator `fix`. Our type system conservatively extends ML-style parametric polymorphism. Thus, we assume that the ML type inference algorithm automatically places appropriate type generalization and instantiation annotations into the source expression. Finally, we assume that, via α-renaming, each variable is bound at most once in an environment.

Types and Schemas. NanoML has a system of base types, function types, and ML-style parametric polymorphism using type variables α and schemas where the type variables are quantified at the outermost level. We organize types into *unrefined types*, which have no top-level refinement predicates, but which may be composed of types which are themselves refined, and *refined types*, which have a top-level refinement predicate. An *ML type (schema)* is a type (schema) where all the refinement predicates are $\top$. A *liquid type (schema)* is a type (schema) where all the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$. We write τ and σ for ML types and schemas, T and S for refinement types and schemas, and $\hat{T}$ and $\hat{S}$ for liquid types and schemas. We write τ to abbreviate $\{\nu{:}\tau \mid \top\}$, and $\overline{\tau}$ to abbreviate $\{\nu{:}\tau \mid \bot\}$. When τ is clear from context, we write $\{e\}$ to abbreviate $\{\nu{:}\tau \mid e\}$.

Instantiation. We write $\mathsf{Ins}(S, \alpha, T)$ for the *instantiation* of the type variable α in the scheme S with the refined type T. Intuitively, $\mathsf{Ins}(S, \alpha, \{\nu{:}T' \mid e'\})$ is the refined type obtained by replacing each occurrence of $\{\nu{:}\alpha \mid e\}$ in S with $\{\nu{:}T' \mid e \wedge e'\}$.

Constants. The basic units of computation in NanoML are primitive constants, which are given refinement types that precisely capture their semantics. Primitive constants include basic values, like integers and booleans, as well as primitive functions that define basic operations. The input types for primitive functions describe the values for which each function is defined, and the output types describe the values returned by each function.

Next, we give an overview of our static type system, by describing environments and summarizing the different kinds of judgments.

Environments and Shapes. A *type environment* Γ is a sequence of *type bindings* of the form $x{:}S$ and *guard predicates* e. The guard predicates are used to capture constraints about "path" information corresponding to the branches followed in if-then-else expressions. The *shape* of a refinement type schema S, written as $\mathsf{Shape}(S)$, is the ML type schema obtained by replacing all the refinement predicates with $\top$ (i.e., "erasing" the refinement predicates). The shape of an environment is the ML type environment obtained by applying Shape to each type binding and removing the guards.

Judgments. Our system has four kinds of judgments that relate environments, expressions, recursive refinements and types. Well-formedness judgments ($\Gamma \vdash S$) state that a type schema S is *well-formed* under environment Γ. Intuitively, the judgment holds if the refinement predicates of S are boolean expressions in Γ. Subtyping judgments ($\Gamma \vdash S_1 <: S_2$) state that the type schema S_1 *is a subtype of* the type schema S_2 under environment Γ. Intuitively, the judgments state that, under the value-binding and guard constraints imposed by Γ, the set of values described by S_1 is contained in the set of values described by S_2. Typing judgments ($\Gamma \vdash_{\mathbb{Q}} e{:}S$) state that, using the logical qualifiers $\mathbb{Q}$, the expression e has the type schema S under environment Γ. Intuitively, the judgments state that, under the value-binding and guard constraints imposed by Γ, the expression e will evaluate to a value described by the type S.

Decidable Subtyping and Liquid Type Inference. In order to determine whether one type is a subtype of another, our system uses the subtyping rules (Figure 6) to generate a set of implication checks over refinement predicates. To ensure that these implication checks are decidable, we *embed* the implication checks into a decidable logic of equality, uninterpreted functions, and linear arithmetic (EUFA) that can be decided by an SMT solver [6]. As the types of branches, functions, and polymorphic instantiations are liquid, we can automatically infer liquid types for programs using abstract interpretation [22].

Soundness. We have proven the soundness of the type system by showing that if an expression is well-typed then we are guaranteed that evaluation does not get "stuck", i.e., at run-time, every primitive operation receives valid inputs. We defer the details about the judgments, the proof of soundness, and type inference to [21].

4. Recursive Refinements

We now describe how the core language is extended with recursively refined types which capture invariants of recursive data structures. First, we describe how we extend the syntax of expressions and types, and correspondingly extend the dynamic semantics. Second, we describe measures, a special class of functions that are syntactically guaranteed to terminate, thereby allowing us to use them to refine recursive types with predicates that capture structural properties of recursive values. Finally, we extend the static type system by formalizing the derivation rules that deal with recursive values, and illustrate how the rules are applied to check expressions.

4.1 Syntax and Dynamic Semantics

Figure 7 describes how the expressions and types of NanoML are extended to include recursively-defined values. The language of expressions includes tuples, value constructors, folds, unfolds, and pattern-match expressions. The language of types is extended with

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\mathsf{Shape}(T) = T}{\Gamma \vdash T}\ [\text{WF-Reflex}]$$

$$\frac{\Gamma \vdash T \quad \Gamma; \nu : \mathsf{Shape}(T) \vdash e : \mathtt{bool}}{\Gamma \vdash \{\nu : T \mid e\}}\ [\text{WF-Refine}]$$

$$\frac{\Gamma \vdash T \quad \Gamma; x : \mathsf{Shape}(T) \vdash T'}{\Gamma \vdash x : T \to T'}\ [\text{WF-Fun}]$$

Decidable Subtyping $\boxed{\Gamma \vdash S_1 <: S_2}$

$$\frac{}{\Gamma \vdash T <: T}\ [<:\text{-Reflex}]$$

$$\frac{\Gamma \vdash T <: T' \quad \mathsf{Valid}(\llbracket \Gamma \rrbracket \wedge \llbracket e \rrbracket \Rightarrow \llbracket e' \rrbracket)}{\Gamma \vdash \{\nu : T \mid e\} <: \{\nu : T' \mid e'\}}\ [<:\text{-Refine}]$$

$$\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma; x_2 : T_2 \vdash T_1'[x_2/x_1] <: T_2'}{\Gamma \vdash x_1 : T_1 \to T_1' <: x_2 : T_2 \to T_2'}\ [<:\text{-Fun}]$$

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \quad \Gamma \vdash S <: S' \quad \Gamma \vdash_{\mathbb{Q}} S'}{\Gamma \vdash_{\mathbb{Q}} e : S'}\ [\text{L-Sub}]$$

$$\frac{\Gamma(x) = \{\nu : T \mid e\}}{\Gamma \vdash_{\mathbb{Q}} x : \{\nu : T \mid e \wedge \nu = x\}}\ [\text{L-Var}] \qquad \frac{}{\Gamma \vdash_{\mathbb{Q}} \mathtt{c} : ty(\mathtt{c})}\ [\text{L-Const}]$$

$$\frac{\Gamma \vdash x : \hat{T}_x \to \hat{T} \quad \Gamma; x : \hat{T}_x \vdash_{\mathbb{Q}} e : \hat{T}}{\Gamma \vdash_{\mathbb{Q}} (\lambda x.e) : x : \hat{T}_x \to \hat{T}}\ [\text{L-Fun}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} x_1 : x : T \to T' \quad \Gamma \vdash_{\mathbb{Q}} x_2 : T}{\Gamma \vdash_{\mathbb{Q}} x_1\ x_2 : T'[x_2/x]}\ [\text{L-App}]$$

$$\frac{\Gamma \vdash \hat{T} \quad \Gamma \vdash_{\mathbb{Q}} e_1 : \mathtt{bool} \qquad \Gamma; e_1 \vdash_{\mathbb{Q}} e_2 : \hat{T} \quad \Gamma; \neg e_1 \vdash_{\mathbb{Q}} e_3 : \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \hat{T}}\ [\text{L-If}]$$

$$\frac{\Gamma \vdash \hat{T} \quad \Gamma \vdash_{\mathbb{Q}} e_1 : S_1 \quad \Gamma; x : S_1 \vdash_{\mathbb{Q}} e_2 : \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 : \hat{T}}\ [\text{L-Let}]$$

$$\frac{\Gamma \vdash \hat{S} \quad \Gamma; x : \hat{S} \vdash_{\mathbb{Q}} e : \hat{S}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{fix}\ x.e : \hat{S}}\ [\text{L-Fix}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \qquad \alpha \notin \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha]e : \forall\alpha.S}\ [\text{L-Gen}]$$

$$\frac{\Gamma \vdash T \quad \mathsf{Shape}(T) = \tau \quad \Gamma \vdash_{\mathbb{Q}} e : \forall\alpha.S}{\Gamma \vdash_{\mathbb{Q}} [\tau]e : \mathsf{Ins}(S, \alpha, T)}\ [\text{L-Inst}]$$

Figure 6. Liquid Type Checking Rules

product types, tagged sum types, and a system of *iso-recursive types*. We assume that appropriate `fold` and `unfold` annotations are automatically placed in the source at the standard construction and matching sites, respectively [19]. We assume that different types use disjoint constructors, and that pattern match expressions contain exactly one match binding for each constructor of the appropriate type. The run-time values are extended with tuples, tags and explicit fold/unfold values in the standard manner [19].

Notation. We write $\langle Z \rangle$ for a sequence of values of the kind Z. We write $(Z; \langle Z \rangle)$ for a sequence of values whose first element is Z, and the remaining elements are $\langle Z \rangle$. We write $\langle\rangle$ for the empty sequence. As for functions, we write dependent tuple types using a sequence of name-to-type bindings, and allow refinements for "later" elements in the tuple to refer to previous elements. For

e	$::=$	$\ldots$	*Expressions:*
	$\mid$	$\langle e \rangle$	tuple
	$\mid$	$\mathtt{C}\langle e \rangle$	constructor
	$\mid$	$\mathtt{match}\ e\ \mathtt{with}\ \vert_i\ \mathtt{C}_i\langle x_i \rangle \mapsto e_i$	match-with
	$\mid$	$\mathtt{unfold}\ e$	unfold
	$\mid$	$\mathtt{fold}\ e$	fold
ε	$::=$	$\mathtt{m} \mid \mathtt{c} \mid x \mid \varepsilon\ \varepsilon$	*M-Expressions*
M	$::=$	$(\mathtt{m}, \langle \mathtt{C}_i\langle x_i \rangle \mapsto \varepsilon_i \rangle, \tau, \mu t.\Sigma_i \mathtt{C}_i\langle x_i : \tau_i \rangle)$	*Measures*
$\rho(\mathbb{B})$	$::=$	$\langle\langle \mathbb{B} \rangle^* \rangle^*$	*Recursive Refinement*
$\mathbb{A}(\mathbb{B})$	$::=$	$\ldots$	*Unrefined Skeletons:*
	$\mid$	$\langle x : \mathbb{T}(\mathbb{B}) \rangle$	product
	$\mid$	$\Sigma_i \mathtt{C}_i\langle x : \mathbb{T}(\mathbb{B}) \rangle$	sum
	$\mid$	$(\rho(\mathbb{B}))\ t$	recursive type variable
	$\mid$	$(\rho(\mathbb{B}))\ \mu t.\Sigma_i \mathtt{C}_i\langle x : \mathbb{T}(\mathbb{B}) \rangle$	recursive type

Figure 7. Recursive Refinements: Syntax

example, $\langle x_1 : \mathtt{int};\ x_2 : \{\nu : \mathtt{int} \mid x_1 \leq \nu\}\rangle$ is the type of pairs of integers where the second element is greater than the first.

Measures. Many important attributes of recursive structures, e.g., the length or set of elements of a list, or the height of a tree, are most naturally specified using recursive functions. Unfortunately, the use of arbitrary, potentially non-terminating recursive functions in refinement predicates leads to unsoundness. Thus, we cannot use refinements containing arbitrary functions to encode invariants over recursive attributes. We solve this problem by representing recursively-defined attributes using *measures*, a class of first order functions from the recursive type to any range type. Measures are syntactically guaranteed to terminate, and so we can soundly use them inside refinements. Further, because their domain is the recursive type, we can automatically instantiate them at the appropriate fold and unfold expressions.

Figure 7 shows the syntax of measure functions. A *measure name* $\mathtt{m}$ is a special variable drawn from a set of measure names. A *measure expression* ε is an expression drawn from a restricted language of variables, constants, measure names and application. A *measure for a recursive (ML) type* $\mu t.\Sigma_i \mathtt{C}_i\langle x_i : \tau_i \rangle$ of type τ is a quadruple: $(\mathtt{m}, \langle \mathtt{C}_i\langle x_i \rangle \mapsto \varepsilon_i \rangle, \tau, \mu t.\Sigma_i \mathtt{C}_i\langle x_i : \tau_i \rangle)$. A *measure specification* is an ordered sequence of measures. Figure 8 shows the rules used to check that a measure specification (i.e., $\langle M \rangle$) is well-formed (i.e., $\emptyset \vdash \langle M \rangle$). As measures are defined by structural induction, they are well-founded and hence total.

4.2 Static Semantics

The derivation rules pertaining to recursively refined types, including the rules for product, sum, and recursive types, as well as construction, match-with, fold and unfold expressions are shown in Figure 8. The fold and unfold rules use a judgment called ρ-*application*, written $(\rho)\ T \triangleright T'$. Intuitively, this judgment states that when a recursive refinement ρ is *applied* to a type T, the result is a sum type T' with refinements from ρ. Next, we describe each judgment and the rules relevant to it.

ρ-Application and Unfolding. Formally, a recursive type $(\rho)\ \mu t.T$ is unfolded in two steps. First, we *apply* the recursive refinement ρ to the *body* of the recursive type T to get the result T' (written $(\rho)\ T \triangleright T'$). Second, in the result of the application T', we *replace* the μ-bound recursive type variable t with the entire original recursive type ($(\rho)\ \mu t.T$), and *normalize* by replacing adjacent refinements $(\rho)(\rho')$ with a new refinement ρ'' which contains the conjunctions of corresponding predicates from ρ and ρ'.

Example. Consider the type which describes increasing lists of integers greater than some x_1':

$$(\rho_{\leq}^{x_1'})\ \mu t.\mathtt{Nil}\ +\ \mathtt{Cons}\langle x_1 : \mathtt{int}, x_2 : (\rho_{\leq}^{x_1})\ t\rangle$$

Measure Well-formedness $\boxed{\Gamma \vdash \langle M \rangle}$

$$\frac{\begin{array}{c}\Gamma;\, \mathtt{m}\!:\!\mu t.\Sigma_i \mathtt{C}_i \langle x_i : \tau_i\rangle \rightarrow \tau \vdash \langle M\rangle \\ \forall i:\ \Gamma;\, \mathtt{m}\!:\!\mu t.\Sigma_i \mathtt{C}_i \langle x_i : \tau_i\rangle \rightarrow \tau;\ \langle x_i : \tau_i\rangle \vdash \varepsilon_i : \tau\end{array}}{\Gamma \vdash (\mathtt{m}, \langle \mathtt{C}_i \langle x_i\rangle \mapsto \varepsilon_i\rangle, \tau, \mu t.\Sigma_i \mathtt{C}_i\langle x_i : \tau_i\rangle);\ \langle M\rangle}\ [\text{WF-M}]$$

ρ-Application $\boxed{(\rho)\ T \triangleright T'}$

$$\frac{\text{fresh } x'\ (\langle e\rangle[x'/x])\ \langle x : T[x'/x]\rangle \triangleright \langle x' : T'\rangle}{(e;\ \langle e\rangle)\ (x : \{\nu : T \mid e_x\};\ \langle x : T\rangle) \triangleright (x' : \{\nu : T \mid e \wedge e_x\};\ \langle x' : T'\rangle)}\ [\triangleright\text{-PROD}]$$

$$\frac{\forall i:\ (\rho_i)\ \langle x_i : T_i\rangle \triangleright \langle x'_i : T'_i\rangle}{(\rho)\ \Sigma_i \mathtt{C}_i\langle x_i : T_i\rangle \triangleright \Sigma_i \mathtt{C}_i\langle x'_i : T'_i\rangle}\ [\triangleright\text{-SUM}]$$

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\Gamma \vdash T \quad \Gamma;\, x : \mathsf{Shape}(T) \vdash \langle x : T\rangle}{\Gamma \vdash x : T;\ \langle x : T\rangle}\ [\text{WF-PROD}]$$

$$\frac{\forall i:\ \Gamma \vdash \langle x_i : T_i\rangle}{\Gamma \vdash \Sigma_i \mathtt{C}_i\langle x_i : T_i\rangle}\ [\text{WF-SUM}]$$

$$\frac{(\rho)\ T \triangleright T' \quad \Gamma \vdash T'[\mathsf{Shape}(\mu t.T)/t]}{\Gamma \vdash (\rho)\ \mu t.T}\ [\text{WF-REC}]$$

Decidable Subtyping $\boxed{\Gamma \vdash S_1 <: S_2}$

$$\frac{\Gamma \vdash T <: T' \quad \Gamma;\, x : T \vdash \langle x : T\rangle <: \langle x' : T'[x/x']\rangle}{\Gamma \vdash x : T;\ \langle x : T\rangle <: x' : T';\ \langle x' : T'\rangle}\ [<:\text{-PROD}]$$

$$\frac{\forall i:\ \Gamma \vdash \langle x_i : T_i\rangle <: \langle x'_i : T'_i\rangle}{\Gamma \vdash \Sigma_i \mathtt{C}_i\langle x_i : T_i\rangle <: \Sigma_i \mathtt{C}_i\langle x'_i : T'_i\rangle}\ [<:\text{-SUM}]$$

$$\frac{\begin{array}{c}(\rho_1)\ T_1 \triangleright T'_1 \qquad (\rho_2)\ T_2 \triangleright T'_2 \\ \tau = \mathsf{Shape}(\mu t.T_1) = \mathsf{Shape}(\mu t.T_2) \\ \Gamma \vdash T'_1[\tau/t] <: T'_2[\tau/t]\end{array}}{\Gamma \vdash (\rho_1)\ \mu t.T_1 <: (\rho_2)\ \mu t.T_2}\ [<:\text{-REC}]$$

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$

$$\frac{\begin{array}{c}\Gamma \vdash (\hat\rho)\ \mu t.\hat T \quad (\rho)\ \hat T \triangleright \hat T' \\ \Gamma \vdash_{\mathbb{Q}} e : \{\nu : \hat T'[(\hat\rho)\ \mu t.\hat T/t] \mid e'\}\end{array}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{fold}\, e : \{\nu : (\hat\rho)\ \mu t.\hat T \mid e'\}}\ [\text{L-FOLD-M}]$$

$$\frac{(\rho)\ T \triangleright T' \quad \Gamma \vdash e : \{\nu : (\rho)\ \mu t.T \mid e'\}}{\Gamma \vdash \mathtt{unfold}\, e : \{\nu : T'[(\rho)\ \mu t.T/t] \mid e'\}}\ [\text{L-UNFOLD-M}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} \langle e\rangle : \langle x : T\rangle}{\Gamma \vdash_{\mathbb{Q}} \mathtt{C}_j\langle e\rangle : \{\nu : \mathtt{C}_j\langle x : T\rangle + \Sigma_{i\neq j}\mathtt{C}_i\langle x_i : \overline{\tau_i}\rangle \mid \wedge_{\mathtt{m}}\mathtt{m}(\nu) = \varepsilon_j(\langle e\rangle)\}}\ [\text{L-SUM-M}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \Sigma_i \mathtt{C}_i\langle x_i : T_i\rangle \quad \Gamma \vdash T \\ \forall i\ \Gamma;\ \langle x_i : T_i\rangle;\ \wedge_{\mathtt{m}}\mathtt{m}(e) = \varepsilon_i(\langle x_i\rangle) \vdash e_i : \hat T\end{array}}{\Gamma \vdash \mathtt{match}\, e\, \mathtt{with}\, |_i\ \mathtt{C}_i\langle x_i\rangle \mapsto e_i : \hat T}\ [\text{L-MATCH-M}]$$

Figure 8. Recursive Refinements: Static Semantics

To unfold this type, we first apply the recursive refinement $\rho_{\le}^{x'_1}$ to the recursive type's body, $\mathtt{Nil} + \mathtt{Cons}\langle x_1 : \mathtt{int}, x_2 : (\rho_{\le}^{x_1})t\rangle$. To apply the recursive refinement to the above sum type, we use rule [$\triangleright$-SUM] to apply the product refinements $\langle\rangle$ and $\langle x'_1 \le \nu;\ \top\rangle$ of $\rho_{\le}^{x'_1}$ to the products corresponding to the Nil and Cons constructors, respectively. To apply the refinements to each product, we use the rule [$\triangleright$-PROD] to obtain the result:

$$\mathtt{Nil} + \mathtt{Cons}\langle x''_1 : \{\nu : \mathtt{int} \mid x'_1 \le \nu\}, x''_2 : (\rho_{\le}^{x''_1})t\rangle \qquad (4)$$

Notice that the result is a sum type with fresh names for the "head" and "tail" of the unfolded list. Observe that this renaming allows us to soundly use the head's value to refine the tail, via a recursive refinement stipulating all elements in the tail are greater than the head, x''_1. To complete the unfolding, we replace t with the entire recursive type and normalize to get:

$$\mathtt{Nil} + \mathtt{Cons}\langle x''_1 : \{\nu : \mathtt{int} \mid x'_1 \le \nu\}, x''_2 : (\rho)\ \mathtt{int\ list}_{\le}\rangle$$

where $\rho \doteq \langle\langle\rangle;\ \langle x'_1 \le \nu \wedge x''_1 \le \nu;\ \top\rangle\rangle$. Intuitively, the result of the unfolding is a type that specifies an empty list, or a non-empty list with a head greater than x'_1 and an increasing tail whose elements are greater than x'_1 and x''_1.

Well-formedness. Rule [WF-REC] checks if a recursive type $(\rho)\ \mu t.T$ is well-formed in an environment Γ. First, the rule applies the refinement ρ to T, the body of the recursive type, to obtain T'. Next, the rule replaces the μ-bound variable t in T' with the *shape* of the recursive type and checks the well-formedness of the result of the substitution.

Example. When $\rho_\top \doteq \langle\langle\rangle;\ \langle\top;\ \top\rangle\rangle$, the check

$$\emptyset \vdash (\rho_\top)\ \mu t.\mathtt{Nil} + \mathtt{Cons}\langle x_1 : \mathtt{int}, x_2 : (\rho_{\le}^{x_1})\ t\rangle$$

is reduced to checking, using [WF-REFINE] that in the environment where x'_1 (the fresh name given to the unfolded list's "head") has type int, the refinement $\{\nu : \mathtt{int} \mid x'_1 \le \nu\}$ (applied to the elements of the unfolded list's "tail") is well-formed.

Subtyping. Rule [<:-PROD] (resp. [<:-SUM]) determines whether two products (resp. sums) satisfy the subtyping relation by checking subtyping between corresponding elements.

Example. When $\Gamma \doteq x'_1 : \mathtt{int}$, the check

$$\Gamma \vdash \langle y_1 : \{x'_1 < \nu\}, y_2 : \mathtt{int\ list}\rangle <: \langle z_1 : \{x'_1 \neq \nu\}, z_2 : \mathtt{int\ list}\rangle \qquad (5)$$

is reduced by [<:-PROD] to:

$$\Gamma \vdash \{\nu : \mathtt{int} \mid x'_1 < \nu\} <: \{\nu : \mathtt{int} \mid x'_1 \neq \nu\}$$
$$\Gamma;\, y_1 : \mathtt{int} \vdash \mathtt{int\ list} <: \mathtt{int\ list}$$

[<:-REFLEX] ensures the latter. [<:-REFINE] reduces the former to checking the validity of $x'_1 < \nu \Rightarrow x'_1 \neq \nu$ in EUFA.

Rule [<:-REC] determines whether the subtyping relation holds between two recursively refined types. The rule first applies the outer refinements to the bodies of the recursive types, then substitutes the μ-bound variable with the *shape* (as for well-formedness), and then checks that the resulting sums are subtypes.

Example. Consider the subtyping check

$$x'_1 : \mathtt{int} \vdash (\rho_{<}^{x'_1})\ \mathtt{int\ list} <: (\rho_{\neq}^{x'_1})\ \mathtt{int\ list} \qquad (6)$$

that is, the list of integers greater than x'_1 is a subtype of the list of integers distinct from x'_1. Rule [<:-REC] applies the refinements to the bodies of the recursive types and then substitutes the shapes, reducing the above (after using [<:-SUM], [<:-PROD] and [<:-REFLEX]) to (5). Finally, applying the rule [<:-REC] yields the judgment

$$\emptyset \vdash \mathtt{int\ list}_{<} <: \mathtt{int\ list}_{\neq} \qquad (7)$$
$$\mathtt{int\ list}_{<} \doteq (\rho_\top)\ \mu t.\mathtt{Nil} + \mathtt{Cons}\langle x_1 : \mathtt{int}, x_2 : (\rho_{<}^{x_1})\ t\rangle$$
$$\mathtt{int\ list}_{\neq} \doteq (\rho_\top)\ \mu t.\mathtt{Nil} + \mathtt{Cons}\langle x_1 : \mathtt{int}, x_2 : (\rho_{\neq}^{x_1})\ t\rangle$$

that is, that the list of strictly increasing integers is a subtype of the list of distinct integers. To see why, observe that after applying the trivial top-level recursive refinement $\rho_\top$ to the body, substituting the shapes, and applying the [<:-SUM], [<:-PROD] and [<:-REFLEX] rules, the check reduces to (6).

Local Subtyping. Our recursive refinements represent universally quantified properties over the elements of a structure. Hence, we reduce subtyping between two recursively-refined structures to *local* subtype checks between corresponding elements of two *arbitrarily* chosen values of the recursive types. The judgment [<:-REC] carries out this reduction via ρ-application and unfolding and enforces the local subtyping requirement with the final antecedent.

Typing. We now turn to the rules for type checking expressions. Rules [L-UNFOLD-M] and [L-MATCH-M] describe how values *extracted* from recursively constructed values are type checked. First, [L-UNFOLD-M] is applied, which performs one unfolding of the recursive type, as described above. Second, the rule [L-MATCH-M] is used on the resulting sum type. This rule stipulates that the entire expression has some type $\hat{T}$ if the type is well-formed in the current environment, and that, for each case of the match (i.e., for each element of the sum), the body expression has type $\hat{T}$ in the environment extended with: (a) the corresponding match bindings and (b) the guard predicate that captures the relationship between the measure of the matched expression and the variables bound by the matched pattern.

Example. Consider the following function:

```
let rec sortcheck xs =
  match xs with
  | x::(x'::_ as xs') ->
     assert (x <= x'); sortcheck xs'
  | _ -> ()
```

Let Γ be $\mathtt{xs}\!:\!\alpha\ \mathtt{list}_{\leq}$. From rule [L-UNFOLD], we have

$$\Gamma \vdash_{\mathbb{Q}} \mathtt{unfold\ xs} : \mathtt{Nil} + \mathtt{Cons}\langle \mathtt{x}\!:\!\alpha;\ \mathtt{xs}'\!:\!(\rho^{\mathtt{x}}_{\leq})\ \alpha\ \mathtt{list}_{\leq}\rangle$$

For clarity, we assume that the fresh names are those used in the match-bindings. For the Cons pattern in the outer match, we use the rule [L-MATCH-M] to get the environment Γ', which is Γ extended with $\mathtt{x}\!:\!\alpha$, $\mathtt{xs}'\!:\!(\rho^{\mathtt{x}}_{\leq})\ \alpha\ \mathtt{list}_{\leq}$. Thus, [L-UNFOLD] yields

$$\Gamma' \vdash_{\mathbb{Q}} \mathtt{unfold\ xs}' : \mathtt{Nil} + \mathtt{Cons}\langle \mathtt{x}'\!:\!\{\nu\!:\!\alpha \mid \mathtt{x} \leq \nu\};\ \mathtt{xs}''\!:\!\ldots\rangle$$

Hence, in the environment:

$$\mathtt{xs}\!:\!\alpha\ \mathtt{list}_{\leq};\ \mathtt{x}\!:\!\alpha;\ \mathtt{xs}'\!:\!(\rho^{\mathtt{x}}_{\leq})\ \alpha\ \mathtt{list}_{\leq};\ \mathtt{x}'\!:\!\{\nu\!:\!\alpha \mid \mathtt{x} \leq \nu\}$$

which corresponds to the extension of Γ' with the (inner) pattern-match bindings, the argument type $\{\nu = \mathtt{x} \leq \mathtt{x}'\}$ is a subtype of the input type $\{\nu\}$ and system verifies that the assert cannot fail.

Rules [L-SUM-M] and [L-FOLD-M] describe how recursively constructed values are type checked. First, [L-SUM-M] is applied, which uses the constructor $\mathtt{C}_j$ to determine which sum type the tuple should be injected into. Notice that, in the resulting sum: (a) the refinement predicate for all other sum elements is $\bot$, capturing the fact that $\mathtt{C}_j$ is the only inhabited constructor within the sum value, and (b) for each measure m defined for the corresponding recursive type, a predicate specifying the value of the measure m is conjoined to the refinement for the constructed sum. Second, the rule [L-FOLD-M] folds the (sum) type into a recursive type.

Example. Consider the expression $\mathtt{Cons}(\mathtt{i}, \mathtt{is})$ which is returned by the function range from Figure 1. Let $\Gamma \doteq \mathtt{i}\!:\!\mathtt{int};\ \mathtt{is}\!:\!(\rho^{\mathtt{i}+1}_{\leq})\ \mathtt{int\ list}_{\leq}$. Rule [<:-REFINE] yields:

$$\Gamma;\ x'_1\!:\!\{\mathtt{i} = \nu\} \vdash \{\mathtt{i} + 1 \leq \nu\} <: \{x'_1 \leq \nu \wedge i \leq \nu\}$$

Consequently, using the rules for subtyping, we derive:

$$\Gamma;\ x'_1\!:\!\{\mathtt{i} = \nu\} \vdash (\rho^{\mathtt{i}+1}_{\leq})\ \mathtt{int\ list}_{\leq} <: (\rho')\ \mathtt{int\ list}_{\leq} \quad (8)$$

where ρ' is $\langle\langle\rangle;\ \langle x'_1 \leq \nu \wedge i \leq \nu;\ \top\rangle\rangle$. Using [L-PROD]:

$$\Gamma \vdash_{\mathbb{Q}} (\mathtt{i}, \mathtt{is}) : \langle x'_1\!:\!\{\mathtt{i} = \nu\};\ x'_2\!:\!(\rho')\ \mathtt{int\ list}_{\leq}\rangle$$

and so, using the subsumption rule [L-SUB] and (8) we have:

$$\Gamma \vdash_{\mathbb{Q}} (\mathtt{i}, \mathtt{is}) : \langle x'_1\!:\!\{\mathtt{i} \leq \nu\};\ x'_2\!:\!(\rho')\ \mathtt{int\ list}_{\leq}\rangle$$

i.e., the first element of the pair is greater than i and the second element is an increasing list of values greater than than the first element and i. Thus, applying [L-SUM], we get:

$$\Gamma \vdash_{\mathbb{Q}} \mathtt{Cons}(\mathtt{i}, \mathtt{is}) : \mathtt{Nil} + \mathtt{Cons}\ \langle x'_1\!:\!\{\mathtt{i} \leq \nu\};\ x'_2\!:\!(\rho')\ \mathtt{int\ list}_{\leq}\rangle$$

As $(\rho^{\mathtt{i}}_{\leq})\ \mathtt{int\ list}_{\leq}$ unfolds to the above type, [L-FOLD] yields

$$\Gamma \vdash_{\mathbb{Q}} \mathtt{fold}(\mathtt{Cons}(\mathtt{i}, \mathtt{is})) : (\rho^{\mathtt{i}}_{\leq})\ \mathtt{int\ list}_{\leq}$$

i.e., range returns an increasing list of integers greater than i.

Example: Measures. Next, let us type check the below expression using the elts measure specification from Section 2.

```
let a = [] in let b = 1::a in
match b with x::xs -> () | [] -> assert false
```

Using [L-SUM-M] and [L-FOLD-M] (and [L-LET]), we derive:

$$\emptyset \vdash_{\mathbb{Q}} \mathtt{Nil} : \{\mathtt{elts}\ \nu = \mathtt{empty}\}$$

$$\mathtt{a}\!:\!\{\mathtt{elts}\ \nu = \mathtt{empty}\} \vdash_{\mathbb{Q}} \mathtt{Cons}(1, \mathtt{a}) : T_{1::\mathtt{a}}$$

where $T_{1::\mathtt{a}}$ abbreviates $\{\mathtt{elts}\ \nu = \mathtt{union}\ (\mathtt{single}\ 1)\ (\mathtt{elts\ a})\}$. Due to [L-UNFOLD-M] and [L-MATCH-M], the assert in the Nil pattern-match body is checked in an environment Γ containing $\mathtt{a}\!:\!\{\mathtt{elts}\ \nu = \mathtt{empty}\}$, $\mathtt{b}\!:\!T_{1::\mathtt{a}}$ and the guard $(\mathtt{elts\ b} = \mathtt{empty})$ from the definition of elts for Nil. Under this *inconsistent* Γ, the argument type $\{\mathtt{not}\ \nu\}$ is a subtype of the input type $\{\nu\}$ and so the call to assert type checks, proving the Nil case is not reachable.

4.3 Type Inference

Next, we summarize the main issues that had to be addressed to extend the liquid type inference algorithm of [22] to the setting of recursive refinements. For details, see [21].

Polymorphic Recursion. Recall the function insert from Figure 2. Suppose that ys is an increasing list, i.e., it has the type $\alpha\ \mathtt{list}_{\leq}$. In the case when ys is not empty, and x is not less than y, the system must infer that the list $\mathtt{Cons}(\mathtt{y}, \mathtt{insert\ x\ ys}')$ is an increasing list. To do so, it must reason that the recursive call to insert returns a list of values that are (a) increasing and, (b) greater than y. Fact (a) can be derived from the (inductively inferred) output type of insert. However, fact (b) is specific to this particular call site – y is not even in scope at other call sites, or at the point at which insert is defined. Notice, though, that the branch condition at the recursive call tells us that the first parameter passed to insert, namely, x, is greater than y. As y and $\mathtt{ys}'$ are the head and tail of the increasing list ys, we know also that every element of $\mathtt{ys}'$ is greater than y. The ML type of insert is $\forall\alpha.\alpha{\to}\alpha\ \mathtt{list}{\to}\alpha\ \mathtt{list}$. By instantiating α at this call site with $\{\nu\!:\!\alpha \mid \mathtt{y} \leq \nu\}$, we can deduce fact (b).

This kind of reasoning is critical for establishing recursive properties. In our system it is formalized by the combination of [L-INST], a rule for dependent polymorphic instantiation, and [L-FIX], a dependent version of Mycroft's rule [16] that allows the instantiation of the polymorphic type of a recursive function within the body of the function. Although Mycroft's rule is known to render ML type inference undecidable [11], this is not so for our system, as it conservatively extends the ML type system. In other words, since only well-typed ML programs are typable in our system, we can use Milner's rule to construct an ML type derivation tree in a first pass, and subsequently use Mycroft's rule and the generalized types inferred in the first pass to instantiate dependent types at (polymorphic) recursive call sites.

Conjunctive Templates. Recall from Section 3 that polymorphic instantiation using Ins (rule [L-INST]) results in types whose refinements are a conjunction of the refinements applied to the original type variables (i.e., α) within the scheme (i.e., S) and the top-level refinements applied to the type used for instantiation (i.e., T).

B	$::=$	$\ldots$	*Base:*
	$\mid$	$\alpha[y/x]$	refined polytype variable
$\mathbb{S}(\mathbb{B})$	$::=$	$\ldots$	*Type Schema Skeletons:*
	$\mid$	$\forall\alpha\langle x:\tau\rangle.\mathbb{S}(\mathbb{B})$	refined polytype schema

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\alpha\langle x:\tau\rangle \in \Gamma \quad \Gamma \vdash y:\tau}{\Gamma \vdash \alpha[y/x]}\ [\text{WF-REFVAR}]$$

$$\frac{\Gamma; \alpha\langle x:\tau\rangle \vdash S}{\Gamma \vdash \forall\alpha\langle x:\tau\rangle.S}\ [\text{WF-REFPOLY}]$$

Decidable Subtyping $\boxed{\Gamma \vdash S_1 <: S_2}$

$$\frac{\alpha\langle x:\tau\rangle \in \Gamma \quad \Gamma \vdash \{\nu:\tau \mid \nu = y_1\} <: \{\nu:\tau \mid \nu = y_2\}}{\Gamma \vdash \alpha[y_1/x] <: \alpha[y_2/x]}\ [\text{<:-REFVAR}]$$

$$\frac{\Gamma; \alpha\langle x:\tau\rangle \vdash S_1 <: S_2}{\Gamma \vdash \forall\alpha\langle x:\tau\rangle.S_1 <: \forall\alpha\langle x:\tau\rangle.S_2}\ [\text{<:-REFPOLY}]$$

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$

$$\frac{\Gamma; \alpha\langle x:\tau\rangle \vdash_{\mathbb{Q}} e : S \quad \alpha\langle x:\tau\rangle \notin \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha\langle x:\tau\rangle]e : \forall\alpha\langle x:\tau\rangle.S}\ [\text{L-REFGEN}]$$

$$\frac{\Gamma; x:\tau_x \vdash T \quad \mathsf{Shape}(T) = \tau \quad \Gamma \vdash_{\mathbb{Q}} e : \forall\alpha\langle x:\tau_x\rangle.S}{\Gamma \vdash_{\mathbb{Q}} [\tau]e : \mathsf{Ins}(S, \alpha, T)}\ [\text{L-REFINST}]$$

Figure 9. Polymorphic Refinements: Syntax, Static Semantics

Similarly, the normalizing of recursive refinements (i.e., collapsing adjacent refinements $(\rho)(\rho')$) results in new refinements that conjoin refinements from ρ and ρ'. Due to these mechanisms, the type inference engine must solve implication constraints over *conjunctive refinement templates* described by the grammar

$$\begin{array}{lll} \theta & ::= & \epsilon \mid [e/x];\theta \qquad \text{(Pending Substitutions)} \\ L & ::= & e \mid \theta\cdot\kappa \wedge L \qquad \text{(Refinement Template)} \end{array}$$

where κ are *liquid type variables* [22]. We use the fact that

$$P \Rightarrow (Q \wedge R) \text{ is valid iff } P \Rightarrow Q \text{ and } P \Rightarrow R \text{ are valid}$$

to reduce each implication constraint over a conjunctive template into a set of constraints over simple templates (with a single conjunct). The iterative weakening algorithm from [22] suffices to solve the reduced constraints, and hence infer liquid types.

5. Polymorphic Refinements

We now describe how the language is extended to capture invariants of finite maps by describing the syntax and static semantics of polymorphic refinements. The syntax of expressions and dynamic semantics are unchanged.

5.1 Syntax

Figure 9 shows how the syntax of types is extended to include polymorphically refined types (in addition to standard polymorphic types). First, type schemas can be universally quantified over *refined polytype variables* $\alpha\langle x:\tau\rangle$. Second, the body of the type schema can contain *refined polytype variable instances* $\alpha[y/x]$.

Intuitively, the quantification over $\alpha\langle x:\tau\rangle$ indicates that α can be instantiated with a refined type containing a free variable x of type τ. It is straightforward to extend the system to allow multiple free variables; we omit this for clarity. A refined polytype instance $\alpha[y/x]$ is a standard polymorphic type variable α with a *pending substitution* $[y/x]$ that gets applied *after* α is instantiated.

The polytype variable instance is syntactic sugar for $\exists x.\{\nu:\alpha \mid y = x\}$. That is, y serves as a *witness* for the existentially bound x. To keep the quantification implicit, the instantiation function Ins eagerly applies the pending substitution whenever each polytype variable is instantiated. When the polytype variable is instantiated with a type containing other polytype instances, it suffices to telescope the pending substitutions by replacing $\alpha[x_1/x][y/x_2]$ with $\alpha[y/x]$ if $x_1 \equiv x_2$ and with $\alpha[x_1/x]$ otherwise.

Example. The following refined polytype schema signature specifies the behavior of the key operations of a *finite dependent map*.

$\texttt{new} :: \forall\alpha,\beta\langle x:\alpha\rangle.\texttt{int}\rightarrow(\texttt{i}:\alpha,\beta[\texttt{i}/x])\ \texttt{t}$

$\texttt{set} :: \forall\alpha,\beta\langle x:\alpha\rangle.(\texttt{i}:\alpha,\beta[\texttt{i}/x])\ \texttt{t}\rightarrow\texttt{k}:\alpha\rightarrow\beta[\texttt{k}/x]\rightarrow(\texttt{j}:\alpha,\beta[\texttt{j}/x])\ \texttt{t}$

$\texttt{get} :: \forall\alpha,\beta\langle x:\alpha\rangle.(\texttt{i}:\alpha,\beta[\texttt{i}/x])\ \texttt{t}\rightarrow\texttt{k}:\alpha\rightarrow\beta[\texttt{k}/x]$

$\texttt{mem} :: \forall\alpha,\beta\langle x:\alpha\rangle.(\texttt{i}:\alpha,\beta[\texttt{i}/x])\ \texttt{t}\rightarrow\texttt{k}:\alpha\rightarrow\texttt{bool}$

In the signature, `t` is a polymorphic type constructor that takes two arguments corresponding to the types of the keys and values stored in the map. If the signature is implemented using association lists, the type $(\texttt{i}:\alpha,\beta[\texttt{i}/x])\ \texttt{t}$ is an abbreviation for $\langle\texttt{i}:\alpha,\beta[\texttt{i}/x]\rangle\ \texttt{list}$. In general, this signature can be implemented by any appropriate data structure, such as balanced search trees, hash tables, *etc.* The signature specifies that certain relationships must hold between the keys and values when new elements are added to the map (using `set`), and it ensures that subsequent reads (using `get`) return values that satisfy the relationship. For example, when α and β are instantiated with `int` and $\{\nu:\texttt{int} \mid x \leq \nu\}$ respectively, we get a finite map $(\texttt{i}:\texttt{int},\{\nu:\texttt{int} \mid \texttt{i} \leq \nu\})\ \texttt{t}$ where, *for each* key-value binding, the value is greater than the key. For this map, the type for `set` ensures that when a new binding is added to the map, the value to be added has the type $\{x \leq \nu\}[\texttt{k}/x]$, which is $\{\texttt{k} \leq \nu\}$, i.e., is greater than the key `k`. Dually, when `get` is used to query the map with a key `k`, the type specifies that the returned value is guaranteed to be greater than the key `k`.

5.2 Static Semantics

Figure 9 summarizes the rules for polymorphic refinements.

Well-formedness. We extend our well-formedness rules with a rule [WF-REFVAR] which states that a polytype instance $\alpha[y/x]$ is well-formed if: (a) the instance occurs in a schema quantified over $\alpha\langle x:\tau\rangle$, i.e., where α can have a free occurrence of x of type τ, and (b) the free variable y that replaces x is bound in the environment to a type τ.

Subtyping. We extend our subtyping rules with a rule [<:-REFVAR] for subtyping refined polytype variable instances. The intuition behind the rule follows from the existential interpretation of the refined polytype instances and the fact that $\forall\nu.(\nu = y_1) \Rightarrow (\nu = y_2)$ implies $y_1 = y_2$, which implies $(\exists x.P \wedge x = y_1) \Rightarrow (\exists x.P \wedge x = y_2)$ for every logical formula P in which x occurs free. Thus, to check that the subtyping holds *for any* possible instantiation of α containing a free x, it suffices to check that the replacement variables y_1 and y_2 are equal.

Example. It is straightforward to check that each of the schemas for `set`, `get`, *etc.* are well-formed. Next, let us see how the following implementation of the `get` function

```
let rec get xs k =
  match xs with
  | [] -> diverge ()
  | (k',d')::xs' -> if k=k' then d' else get xs' k
```

implements the refined polytype schema shown above. From the input assumption that `xs` has the type $\langle\texttt{i}:\alpha,\beta[\texttt{i}/x]\rangle\ \texttt{list}$, and the rules for unfolding and pattern matching ([L-UNFOLD-M] and [L-MATCH-M]) we have that, at the point where $\texttt{d}'$ is returned, the environment Γ contains the type binding $\texttt{d}':\beta[\texttt{i}/x][\texttt{k}'/\texttt{i}]$ which, after telescoping the substitutions, is equivalent to the binding $\mathtt{d}':\beta[\mathtt{k}'/x]$. Due to [L-IF], the branch condition $\mathtt{k}=\mathtt{k}'$ is in Γ, and so $\Gamma \vdash \{\nu:\alpha \mid \nu=\mathtt{k}'\} <: \{\nu:\alpha \mid \nu=\mathtt{k}\}$. Thus, from rule [<:-REFVAR], and subsumption, we derive that the `then` branch has the type $\beta[\mathtt{k}/x]$ from which the schema follows.

Typing. We extend the typing rules with rules that handle refined polytype generalization ([L-REFGEN]) and instantiation ([L-REFINST]). A refined polytype variable $\alpha\langle x:\tau\rangle$ can be instantiated with a dependent type T that contains a free occurrence of x of type τ. This is ensured by the well-formedness antecedent for [L-REFINST] which checks that T is well-formed in the environment extended with the appropriate binding for x. However, once T is substituted into the body of the schema S, the different pending substitutions at each of the refined polytype *instances* of α are applied and hence x *does not appear free* in the instantiated type, which is consistent with the existential interpretation of polymorphic refinements.

Polymorphic Refinements vs. Array Axioms. Our technique of specifying the behavior of finite maps using polymorphic refinements is orthogonal, and complementary, to the classical approach that uses McCarthy's array axioms [15]. In this approach, one models reads and writes to arrays, or, more generally, finite maps, using two operators. The first, $Sel(m,i)$, takes a *map* m and an *address* i and returns the value stored in the map at that address. The second, $Upd(m,i,v)$, takes a map m, an address i and a *value* v and returns the new map which corresponds to m updated at the address i with the new value v. The two efficiently decidable axioms

$$\forall m,i,v.\ Sel(Upd(m,i,v),i)=v$$
$$\forall m,i,j,v.\ i=j \vee Sel(Upd(m,i,v),j)=Sel(m,j)$$

specify the behavior of the operators. Thus, an analysis can use the operators to algorithmically reason about the exact contents of explicitly named addresses within a map. For example, the predicate $Sel(\mathtt{m},\mathtt{i})=0$ specifies that `m` maps the key `i` to the value 0. However, to capture invariants that hold for all key-value bindings in the map, one must use universally quantified formulas, which make algorithmic reasoning brittle and unpredictable. In contrast, polymorphic refinements can smoothly capture and reason about the relationship between *all* the addresses and values but do not, as described so far, let us refer to particular named addresses.

We can have the best of both worlds in our system by combining these techniques. Using polymorphic refinements, we can reason about universal relationships between keys and values and by refining the output types of `set` and `get` with the predicates $(\nu = Upd(\mathtt{m},\mathtt{k},\mathtt{v}))$ and $(\nu = Sel(\mathtt{m},\mathtt{k}))$, respectively, we can simultaneously reason about the specific keys and values in a map.

Example. Polymorphic refinements can be used to verify properties of linked structures, as each link field corresponds to a map from the set of source structures to the set of link targets. For example, a field `f` corresponds to a map `f`, a field read `x.f` corresponds to `get f x`, and a field write `x.f` $\leftarrow e$ corresponds to `set f x e`. Consider the following SSA-converted [1] implementation of the textbook `find` function for the *union-find* data structure.

```
let rec find rank parent0 x =
  let px = get parent0 x in
  if px = x then (parent0, x) else
    let (parent1, px') = find rank parent0 px in
    let parent2        = set parent1 x px' in
    (parent2, px')
```

The function `find` takes two maps as input: `rank` and `parent0`, corresponding to the rank and parent fields in an imperative implementation, and an element `x`, and finds the "root" of `x` by transitively following the `parent` link, until it reaches an element that is its own parent. The function implements *path-compression*, i.e., it destructively updates the parent map so that subsequent queries jump straight to the root. The data structure maintains the acyclicity invariant that each non-root element's rank is strictly smaller than the rank of the element's parent. The acyclicity invariant of the `parent` map is captured by the type

$$(\mathtt{i}:\mathtt{int}, \{\nu:\mathtt{int} \mid (\mathtt{i}=\nu) \vee Sel(\mathtt{rank},\mathtt{i}) < Sel(\mathtt{rank},\nu)\})\ \mathtt{t}$$

which states that for each key `i`, the parent ν is such that, either the key is its own parent or the key's rank is less than the parent's rank. Our system verifies that when `find` is called with a parent map that satisfies the invariant, the output map also satisfies the invariant. To do so, it automatically instantiates the refined polytype variables in the signatures for `get` and `set` with the appropriate refinement types, after which the rules from Section 3 (Figure 6) suffice to establish the invariant. Similarly, our system verifies the `union` function where the rank of a root is incremented when two roots of equal ranks are linked. Thus, polymorphic refinements enable the verification of complex acyclicity invariants of mutable data structures.

6. Evaluation

We have implemented our type-based data structure verification techniques in DSOLVE, which takes as input an OCAML program (a `.ml` file), a property specification (a `.mlq` file), and a set of logical qualifiers (a `.quals` file). The program corresponds to a module, and the specification comprises measure definitions and types against which the interface functions of the module should be checked. DSOLVE combines the manually supplied qualifiers (`.quals`) with qualifiers scraped from the properties to be proved (`.mlq`) to obtain the set $\mathbb{Q}$ used to infer types for verification. DSOLVE produces as output the list of possible refinement type errors, and a `.annot` file containing the inferred liquid types for all the program expressions.

Benchmarks. We applied DSOLVE to the following set of benchmarks, designed to demonstrate: *Expressiveness*—that our approach can be used to verify a variety of complex properties across a diverse set of data structures, including textbook structures, and structures designed for particular problem domains; *Efficiency*—that our approach scales to large, realistic data structure implementations; and *Automation*—that, due to liquid type inference, our approach requires a small set of manual qualifier annotations.

- `List-sort`: a collection of textbook list-based sorting routines, including insertion-sort, merge-sort and quick-sort,
- `Map`: an ordered AVL-tree based implementation of finite maps, (from the OCAML standard library)
- `Ralist`: a random-access lists library, (due to Xi [23])
- `Redblack`: a red-black tree insertion implementation (without deletion), (due to Dunfield [7])
- `Stablesort`: a tail recursive mergesort, (from the OCAML standard library)
- `Vec`: a tree-based vector library (due to de Alfaro [5])
- `Heap`: a binary heap library, (due to Filliâtre [8])
- `Splayheap`: a splay tree based heap, (due to Okasaki [18])
- `Malloc`: a resource management library,
- `Bdd`: a binary decision diagram library (due to Filliâtre [8])
- `Unionfind`: the textbook union-find data structure,
- `Subvsolve`: a DAG-based type inference algorithm [12]

On the programs, we check the following properties: Sorted, the output list is sorted, Elts, the output list has the same elements as the input, Balance, the output trees are balanced, BST, the output trees are binary search ordered, Set, the structure implements a set interface, e.g., the outputs of the `add`, `remove`, `merge` functions correspond to the addition of, removal of, union of, (resp.) the

Program	LOC	Ann.	T(s)	Property
List-sort	110	7	11	Sorted, Elts
Map	95	3	23	Balance, BST, Set
Ralist	91	3	3	Len
Redblack	105	3	32	Balance, Color, BST
Stablesort	161	1	6	Sorted
Vec	343	9	103	Balance, Len1, Len2
Heap	120	2	41	Heap, Min, Set
Splayheap	128	3	7	BST, Min, Set
Malloc	71	2	2	Alloc
Bdd	205	3	38	VariableOrder
Unionfind	61	2	5	Acyclic
Subvsolve	264	2	26	Acyclic
Total	1754	40	297	

Figure 10. Results: LOC is the number of lines of code without comments, **Property** is the properties verified, **Ann.** is the number of manual qualifier annotations, and **T(s)** is the time in seconds DSOLVE requires to verify each property.

elements or sets corresponding to the inputs, Len, the various operations appropriately change the length of the list, Color, the output trees satisfy the red-black color invariant, Heap, the output trees are heap-ordered, Min, the `extractmin` function returns the smallest element, Alloc, the used and free resource lists only contain used and free resources,VariableOrder, the output BDDs have the variable ordering property, Acyclic, the output graphs are acyclic. The complete benchmark suite is available in [21].

Results. The results of running DSOLVE on the benchmarks are summarized in Figure 6. Even for the larger benchmarks, very few qualifiers are required for verification. These qualifiers capture simple relationships between variables that are not difficult to specify after understanding the program. Due to liquid type inference, the total amount of manual annotation remains extremely small — just 3% of code size, which is acceptable given the complexity of the implementation and the properties being verified. Next, we describe the subtleties of some of the benchmarks and describe how DSOLVE verifies the key invariants.

Sorting. We used DSOLVE to verify that implementations of various list sorting algorithms returned sorted lists whose elements were the same as the input lists', i.e., that the sorting functions had type $\mathtt{xs}{:}\alpha\ \mathtt{list}{\rightarrow}\{\nu{:}\alpha\ \mathtt{list}_{\leq} \mid \mathtt{elts}\ \nu = \mathtt{elts}\ \mathtt{xs}\}$. We checked the above properties on `insertsort` (shown in Figure 2), `mergesort` [23] which recursively halves the lists, sorts, and merges the results, `mergesort2` [23] which chops the list into a list of (sorted) lists of size two, and then repeatedly passes over the list of lists, merging adjacent lists, until it is reduced to a singleton which is the fully sorted list, `quicksort`, which partitions the list around a pivot, then sorts and appends the two partitions, and `stablesort`, from the OCAML standard library's `List` module, which is a tail-recursive mergesort that uses two mutually recursive functions, one which returns an increasing list, another a decreasing list. For each benchmark, DSOLVE infers that the sort function has type Sorted using only the qualifier $\{\nu \leq \star\}$. To prove Elts, we need a few simple qualifiers relating the elements of the output list to those of the input. DSOLVE cannot check Elts for `stablesort` due to its (currently) limited handling of OCAML's pattern syntax.

Non-aliasing. Nested refinements can be useful not just to verify properties like sortedness, but also to ensure non-aliasing across an unbounded and unordered collection of values. As an example, consider the two functions `alloc` and `free` of Figure 11. The functions manipulate a "world" which is a triple comprised of: `m`, a bitmap indicating whether addresses are free (0) or used (1), `us`, a list of addresses marked used, and `fs`, a list of addresses marked free. Suppose that the map functions have types:

```
let alloc (m, us, fs) =
  match fs with
  | [] ->
     assert false
  | p::fs' ->
     let m'  = set m p 1 in
     let us' = p::us in
     ((m', us', fs'), p)
```

```
let free (m, us, fs) p =
  if get m p = 0 then
   (m, us, fs)
  else
   let m'  = set m p 0 in
   let us' = delete p us in
   let fs' = p::fs in
   (m', us', fs')
```

Figure 11. Excerpt from `Malloc`

$$\mathtt{set} :: \mathtt{m}{:}(\alpha,\beta)\ \mathtt{t}{\rightarrow}\mathtt{k}{:}\alpha{\rightarrow}\mathtt{d}{:}\beta{\rightarrow}\{\nu{:}(\alpha,\beta)\ \mathtt{t} \mid \nu = \mathit{Upd}(\mathtt{m},\mathtt{k},\mathtt{d})\}$$
$$\mathtt{get} :: \mathtt{m}{:}(\alpha,\beta)\ \mathtt{t}{\rightarrow}\mathtt{k}{:}\alpha{\rightarrow}\{\nu{:}\beta \mid \nu = \mathit{Sel}(\mathtt{m},\mathtt{k})\}$$

We can formalize the invariants on the lists using the product type:

$$\rho_c \doteq \langle\langle\rangle;\ \langle \mathit{Sel}(m,\nu) = c;\ \top\rangle\rangle$$
$$\sigma^c_{\bowtie} \doteq (\rho_c)\ \mathtt{int\ list}_{\bowtie}$$
$$\mathit{RES}_{\bowtie} \doteq \langle \mathtt{m}{:}(\mathtt{int},\mathtt{int})\ \mathtt{t}, \mathtt{us}{:}\sigma^1_{\bowtie}, \mathtt{fs}{:}\sigma^0_{\bowtie}\rangle$$

The function `alloc` picks an address from the free list, sets the used bit of the address in the bitmap, adds it to the used list and returns this address together with the updated world. Dually, the function `free` checks if the given address is in use and, if so, removes it from the used list, unsets the used bit of the address, and adds it to the free list. We would like to verify that that the functions preserve the invariant on the world above, i.e.,

$$\mathtt{alloc} :: \mathit{RES}{\rightarrow}\langle \mathit{RES}, \mathtt{int}\rangle,\ \mathtt{free} :: \mathit{RES}{\rightarrow}\mathtt{int}{\rightarrow}\mathit{RES}.$$

However, the functions *do not* have these types. Suppose that the free list, `fs`, passed to `alloc`, contains *duplicates*. In particular, suppose that the head element `p` also appears inside the tail $\mathtt{fs}'$. In that case, setting `p`'s used bit will cause there to be an element of the output free list $\mathtt{fs}'$, namely `p`, whose used bit is set, violating the output invariant. Hence, we need to capture the invariant that there are no duplicates in the used or free lists, i.e., that no two elements of the used or free lists are *aliases* for the same address. In our system, this is expressed by the type $\mathtt{int\ list}_{\neq}$, as defined by (1). Hence, If the input world has type $\mathit{RES}_{\neq}$, then when `p`'s bit is set, the SMT solver uses the array axioms to determine that for each $\nu \neq \mathtt{p}$, $\mathit{Sel}(\mathtt{m}',\nu) = \mathit{Sel}(\mathtt{m},\nu)$ and hence, the used bit of each address in $\mathtt{fs}'$ remains unset in the new map $\mathtt{m}'$. Dually, our system infers that the no-duplicates invariant holds on `p :: us` as `p` (whose bit is unset in `m`) is different from all the elements of `us` (whose bits are set in `m`). Thus, our system automatically verifies:

$$\mathtt{alloc} :: \mathit{RES}_{\neq}{\rightarrow}\langle \mathit{RES}_{\neq}, \mathtt{int}\rangle,\ \mathtt{free} :: \mathit{RES}_{\neq}{\rightarrow}\mathtt{int}{\rightarrow}\mathit{RES}_{\neq}$$

Maps. We applied DSOLVE to verify OCAML's tree-based functional `Map` library. The trees have the ML type:

```
type ('a,'b) t =
  E | N of 'a * 'b * ('a,'b) t * ('a,'b) t * int
```

The `N` constructor takes as input a key of type `'a`, a datum of type `'b`, two subtrees, and an integer representing the *height* of the resulting tree. The library implements a variant of AVL trees where, internally, the heights of siblings can differ by at most 2. A *binary search tree* (BST) is one where, for each node, the keys in the left (resp. right) subtree are smaller (resp. greater) than the node's key. A tree is *balanced* if, at each node, the heights of its subtrees differ by *at most* two. Formally, after defining a height measure `ht`:

```
measure ht =  E -> 0 | N (_,_,l,r,_) ->
  if ht l < ht r then 1 + ht r else 1 + ht l
```

the balance and BST invariants are respectively specified by:

$$(\rho_{\mathsf{bal}})\ \mu t.\,\mathtt{E} + \mathtt{N}\langle k\!:\!\alpha, d\!:\!\beta, l\!:\!t, r\!:\!t, h\!:\!\mathtt{int}\rangle \qquad (\mathsf{Balance})$$

$$\mu t.\,\mathtt{E} + \mathtt{N}\langle k\!:\!\alpha, d\!:\!\beta, l\!:\!(\rho_<)\,t, r\!:\!(\rho_>)\,t, h\!:\!\mathtt{int}\rangle \quad (\mathsf{BST})$$

$$\rho_{\bowtie} \doteq \langle\langle\rangle;\ \langle\nu \bowtie k;\ \top;\ \top;\ \top;\ \top\rangle\rangle \text{ for } \bowtie \in \{<,>\}$$

$$\rho_{\mathsf{bal}} \doteq \langle\langle\rangle;\ \langle\top;\ \top;\ \top;\ e_b;\ e_h\rangle\rangle$$

$$e_h \doteq (\mathtt{ht}\ l < \mathtt{ht}\ r)?(\nu = 1 + \mathtt{ht}\ r) : (\nu = 1 + \mathtt{ht}\ l)$$

$$e_b \doteq (\mathtt{ht}\ l - \mathtt{ht}\ \nu \leq 2) \wedge (\mathtt{ht}\ \nu - \mathtt{ht}\ l \leq 2)$$

DSOLVE verifies that all trees returned by API functions are balanced binary search trees, that no programmer-specified assertion fails at run-time, and that the library implements a set interface.

Vectors. We applied DSOLVE to verify various invariants in a library that uses binary trees to represent C++-style extensible vectors [5]. To ensure that various operations are efficient, the heights of the subtrees at each level are allowed to differ by at most two. DSOLVE verifies that that all the trees returned by API functions, which include appending to the end of a vector, updating values, deleting sub-vectors, concatenation, *etc.*, are balanced, vector operations performed with valid index operands, i.e., indices between 0 and the number of elements in the vector don't fail (Len1), and that all functions passed as arguments to the iteration, fold and map procedures are called with integer arguments in the appropriate range (Len2). For example, DSOLVE proves:

$$\mathtt{iteri} :: \mathtt{v}\!:\!\alpha\ \mathtt{t} \rightarrow (\{0 \leq \nu < \mathtt{len\ v}\} \rightarrow \alpha \rightarrow \mathtt{unit}) \rightarrow \mathtt{unit}$$

That is, the second argument passed to the higher-order iterator is only called with inputs between 0 and the length of the vector, i.e., the number of elements in each vector. DSOLVE found a subtle bug in the rebalancing procedure; by using the inferred types, we were able to find a minimal and complete fix which the author adopted.

Binary Decision Diagrams. A *Binary Decision Diagram* (BDD) is a reduced decision tree used to represent boolean formulas. Each node is labeled by a propositional variable drawn from some ordered set $x_1 < \ldots < x_n$. The nodes satisfy a *variable ordering* invariant that if a node labeled x_i has a child labeled x_j then $x_i < x_j$. A combination of hash-consing/memoization and variable ordering ensures that each formula has a canonical BDD representation. Using just three similar, elementary qualifiers, DSOLVE verifies the variable ordering invariant in Filliâtre's OCAML BDD library [8]. The verification requires recursive refinements to handle the ordering invariant and polymorphic refinements to handle the memoization that is crucial for efficient implementations. The type used to encode BDDs, simplified for exposition, is:

```
type var = int
type bdd = Z of int | O of int
         | N of var * bdd * bdd * int
```

where `var` is the variable at a node, and the `int` elements are hash-cons tags for the corresponding sub-BDDs. To capture the order invariant, we write a measure that represents the index of the root variable of a BDD:

```
measure var = Z _ | O _  -> maxvar + 1
            | N (x,_,_,_) -> x
```

where `maxvar` is the total number of propositional variables being used to construct BDDs. After defining

$$\mathtt{bdd} \doteq \mu t.\,\mathtt{Z\ int} + \mathtt{O\ int} + \mathtt{N}\langle x\!:\!\mathtt{int}, t, t, \mathtt{int}\rangle$$

$$\rho_{\mathtt{V}} \doteq \langle\langle\top\rangle;\ \langle\top\rangle;\ \langle\top;\ (x < \mathtt{var}\ \nu);\ (x < \mathtt{var}\ \nu);\ \top\rangle\rangle$$

we can specify BDDs satisfying the variable ordering VariableOrder invariant as $(\rho_{\mathtt{V}})$ `bdd`. The following code shows the function that computes the BDD corresponding to the negation of the input `x`, by using the table `cache` for memoization.

```
let mk_not x =
  let cache = Hash.create cache_default_size in
  let rec mk_not_rec x =
    if Hash.mem cache x then Hash.find cache x else
      let res = match x with
        | Z _ -> one | O _ -> zero
        | N (v, l, h, _) ->
            mk v (mk_not_rec l) (mk_not_rec h) in
      Hash.add cache x res; res in
  mk_not_rec x
```

Using the polymorphically refined signatures for the hash table operations (`set`, `get`, *etc.* from Section 5), with the proviso, already enforced by OCAML, that the key's type be treated as invariant, DSOLVE is able to verify that variable ordering (VariableOrder) is preserved on the entire library. To do so, DSOLVE uses the qualifier $\mathtt{var}\ \star \leq \mathtt{var}\ \nu$ to automatically instantiate the refined polytype variables α and $\beta\langle x\!:\!\alpha\rangle$ in the signatures for `Hash.find` and `Hash.add` with `bdd` and $\{\nu\!:\!\mathtt{bdd} \mid \mathtt{var}\ x \leq \mathtt{var}\ \nu\}$, which, with the other rules, suffices to infer that:

$$\mathtt{mk\_not} :: \mathtt{x}\!:\!\mathtt{bdd} \rightarrow \{\nu\!:\!\mathtt{bdd} \mid \mathtt{var}\ x \leq \mathtt{var}\ \nu\}$$

Bit-level Type Inference. We applied DSOLVE to verify an implementation of a graph-based algorithm for inferring bit-level types from the bit-level operations of a C program [12]. The bit-level types are represented as a sequence of *blocks*, each of which is represented as a node in a graph. Mask or shift operations on the block cause the block to be *split* into sub-blocks, which are represented by the list of successors of the block node. Finally, the fact that value-flow can cause different bit-level types to have unified subsequences of blocks is captured by having different nodes *share* successor blocks. The key invariant maintained by the algorithm is that the graph contains no cycles. DSOLVE combines recursive and polymorphic refinements to verify that the graph satisfies an acyclicity invariant like DAG from (3) in Section 2.2.

6.1 Limitations and Future Work

Our case studies reveal several expressiveness limitations of our system. Currently, simple modifications allow each program to typecheck. We intend to address these limitations in future work.

First-order Refinements. To preserve decidability, our EUFA embedding leaves all function applications in the refinement predicates uninterpreted. This prevents checking `quicksort` with the standard higher-order partition function:

$$\mathtt{part} :: \alpha\ \mathtt{list} \rightarrow \mathtt{p}\!:\!(\alpha \rightarrow \mathtt{bool}) \rightarrow \{\mathtt{p}(\nu)\}\ \mathtt{list} * \{\neg\mathtt{p}(\nu)\}\ \mathtt{list}$$

where `p` is the higher-order predicate used for partitioning. The function applications $\mathtt{p}(\nu)$ in `part`'s output type are left uninterpreted, so we cannot use them in verification. Instead, we change the program so that the type of `p` is $\alpha \rightarrow (\beta, \gamma)$ `either` where (β, γ) `either` has constructors `T` of β and `F` of γ. The new `part` function then collects `T` and `F` values into a pair of separate lists. Thus, if `w` is the pivot, we can verify `quicksort` by passing `part` a higher-order predicate of type:

$$\alpha \rightarrow (\{\nu\!:\!\alpha \mid \nu \geq \mathtt{w}\}, \{\nu\!:\!\alpha \mid \nu < \mathtt{w}\})\ \mathtt{either}.$$

Existential Witnesses. Recall that expressing the key acyclicity invariant in *union-find*'s `find` function (Section 5.2) required referencing the `rank` map. Although `rank` is not used in the body of the `find` function, omitting it from the parameter list causes this acyclicity invariant to be ill-formed within `find`. Instead of complicating our system with existentially quantified types, we add `rank` as a *witness* parameter to `find`. Similarly, notice that the list obtained by appending two sorted lists `xs` and `ys` is sorted iff there exists some `w` such that the elements of `xs` (resp. `ys`) are less than (resp. greater than) `w`. In the case of `quicksort`, this `w` is exactly the pivot. Hence, we add `w` as a witness parameter to `append`, and pass in the pivot at the callsite, after which the system infers:

$$\texttt{append} :: \texttt{w}{:}\alpha \rightarrow \{\nu \leq \texttt{w}\}\ \texttt{list}_{\leq} \rightarrow \{\texttt{w} \leq \nu\}\ \texttt{list}_{\leq} \rightarrow \alpha\ \texttt{list}_{\leq}$$

and hence that `quicksort` has type Sorted. Similar witness parameters are needed for the tail-recursive merges used in `stablesort`.

Context-Sensitivity. Finally, there were cases where functions have different behavior in different calling contexts. For example, `stablesort` uses a function that reverses a list. Depending upon the context, the function is either (1) passed an increasing list as an argument and returns a decreasing list as output, or, (2) passed a decreasing list as an argument and returns an increasing list. Our system lacks intersection types (e.g., [7]), and we cannot capture the above, which forces us to duplicate code at each callsite. In our experience so far, this has been rare (out of all our benchmarks, only one duplicate function in `stablesort` was required), but nevertheless, in the future, we would like to investigate how our system can be extended to intersection types.

7. Related Work

Indexed Type based approaches use types augmented with *indices* which capture invariants of recursive types [24, 4, 7]. Recursive refinements offer several significant advantages over indexed types. First, they compose more easily. While indexed types can be used to specify invariants like sortedness and binary-search-ordering, expressing additional invariants requires the programmer to add more type indices, altering both the type and functions defined on that type. Thus, it is cumbersome to use indices to encode auxiliary invariants: e.g., one must manually define different types (and constructors) to describe decreasing lists, or sorted lists with elements in some range, as well as different functions to manipulate data of these types. In contrast, by *separating* the refinements from the underlying types, we allow the programmer to compose different refinements with the same type skeleton, which is essential to the usability of the system. Second, no inference algorithm for indexed types has been demonstrated, which greatly limits their usability due to the drastically increased annotation burden; the programmer must annotate *all* functions and polymorphic instantiations. For example, at each call to `Hash.add`, the programmer would have to specify the appropriate indexed type instantiations for the type variables α, β. In contrast, by *separating* the refinements from the underlying types, and *uniformly* representing refinements as conjunctions of qualifiers, our system permits inference, which is essential for usability.

Hoare Logic based approaches require that the programmer write pre- and post-conditions and loop invariants for functions in a rich, higher-order specification logic. From these and the code, *verification conditions* (VCs) are generated whose validity implies that the code meets the specification. The VCs are then proved using automatic [13] or interactive theorem proving [17, 26, 20]. These approaches allow for the specification of far more expressive properties than is possible in our system. However, they require significantly more manual effort in interacting with the prover. Of the above proposals, the latter three have equal or more expressiveness than our approach, but require, at a conservative estimate, annotations amounting to more than three times the code size to prove similar properties. A direct comparison is difficult as [17, 26] consider lower-level languages and [20] considers some different properties. Nevertheless, the order-of-magnitude annotation reduction proves the utility of our approach.

Abstract Interpretation based approaches focus on inferring lower-level *shape* properties (e.g., that a structure is a singly-linked list or tree) in the presence of destructive heap updates. These techniques work by carefully controlling generalization (i.e., "blurring") and instantiation (i.e., "focusing") using a combination of user-defined recursive predicates [14, 25] and abstract domains tailored to the structure being analyzed [10, 3]. Our insight is that in high-level languages, shape invariants can be guaranteed using a rich type system. Furthermore, by piggybacking refinements on top of the types, one can use abstract interpretation (in the form of liquid type inference) to verify properties which have hitherto been beyond the scope of automation. In future work we would like to apply our techniques to lower-level languages, by first using shape analysis to reconstruct the recursive type information, and then using recursive and polymorphic refinements over the reconstructed shapes to verify high-level properties.

Acknowledgments. We thank Robbie Findler and Amal Ahmed for valuable discussions about Polymorphic Refinements.

References

[1] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4), 1998.

[2] L. Augustsson. Cayenne - a language with dependent types. In *ICFP*, 1998.

[3] B. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[4] S. Cui, K. Donnelly, and H. Xi. Ats: A language that combines programming with theorem proving. In *FroCos*, 2005.

[5] Luca de Alfaro. Vec: Extensible, functional arrays for ocaml. http://www.dealfaro.com/vec.html.

[6] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[7] Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[8] J.C. Filliâtre. Ocaml software. http://www.lri.fr/ filliatr/software.en.html.

[9] C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.

[10] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.

[11] F. Henglein. Type inference with polymorphic recursion. *ACM TOPLAS*, 15(2):253–289, 1993.

[12] R. Jhala and R. Majumdar. Bit-level types for high-level reasoning. In *FSE*. ACM, 2006.

[13] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL*, 2008.

[14] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, LNCS 1824, pages 280–301. Springer, 2000.

[15] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[16] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Symposium on Programming*, pages 217–228, 1984.

[17] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP*, 2008.

[18] C. Okasaki. *Purely Functional Data Structures*. CUP, 1999.

[19] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[20] Y. Regis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In *MPC*, 2008. To appear.

[21] P. Rondon, M. Kawaguchi, and R. Jhala. Type based data structure verification. http://pho.ucsd.edu/liquid.

[22] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 158–169, 2008.

[23] H. Xi. DML code examples. http://www.cs.bu.edu/fac/hwxi/DML/.

[24] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

[25] H. Yang, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.

[26] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.