Automatically Learning Shape Specifications

He Zhu
Purdue University, USA
zhu103@purdue.edu

Gustavo Petri
Universit Paris Diderot – Paris 7
gpetri@liafa.univ-paris-diderot.fr

Suresh Jagannathan
Purdue Univeristy, USA
suresh@cs.purdue.edu

Abstract

This paper presents a novel automated procedure for discovering expressive shape specifications for sophisticated functional data structures. Our approach extracts potential shape predicates based on the definition of constructors of arbitrary user-defined inductive data types, and combines these predicates within an expressive first-order specification language using a lightweight data-driven learning procedure. Notably, this technique requires *no* programmer annotations, and is equipped with a type-based decision procedure to verify the correctness of discovered specifications. Experimental results indicate that our implementation is both efficient and effective, capable of automatically synthesizing sophisticated shape specifications over a range of complex data types, going well beyond the scope of existing solutions.

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

1. Introduction

Understanding and discovering useful specifications in programs that manipulate sophisticated data structures are central problems in program analysis and verification. A particularly challenging exercise for shape analyses, and the focus of this paper, involves reasoning about ordering specifications that relate the shape of a data structure (*e.g.*, the data structure implements a binary tree) with the values contained therein (*e.g.*, the binary tree traverses its elements *in-order*).

```
type 'a list =
 | Nil
 | Cons 'a *
     'a list

type 'a tree =
 | Leaf
 | Node 'a *
     'a tree *
     'a tree
```

```
// flat:'a tree -> 'a list
//              -> 'a list
let rec flat accu t =
 match t with
  | Leaf -> accu
  | Node (x, l, r) ->
     flat(x::(flat accu r)) l

// elements:'a tree->'a list
let elements t = flat [ ] t
```

Figure 1: Tree flattening function.

To illustrate the issue, consider the `elements` function shown in Fig. 1. The intended behavior of this function is to flatten a binary tree into a list by calling the recursive function `flat` which uses an accumulator list for this purpose. We depict the *input-output* behavior of `elements` with an input tree t and output list ν in Fig. 2b.[1] The meta-variable ν in the figure represents the result of calling `elements` (*i.e.*, in this case ν = `elements` t for the input tree rooted at node t).[2] While there are a number of specifications that we might postulate about this function (*e.g.*, the number of nodes in the output list is the same as the number of nodes in the input tree, or the values contained in the output list are the same as the values contained in the input tree), a more accurate and useful specification, that subsumes the others, is that the *in-order* relation between the elements of the input tree corresponds exactly to the *forward-order* (*i.e.*, *occurs-before*) relation between the elements of the output list.

We are interested in automatically *learning* specifications of this kind that express interesting ordering relations between the elements of a data structure, taking into account properties of the structure's shape, based solely on input-output observations. While having such specifications has obvious benefit for improved program documentation and understanding, they are particularly useful in facilitating modular verification tasks. For example, ordering specifications naturally serve as interface contracts between data structure libraries and client code that can be subsequently leveraged by expressive refinement type checkers [61].

Even for a function as simple as `elements`, however, manually providing a specification that relates the values contained within the input tree and output list is non-

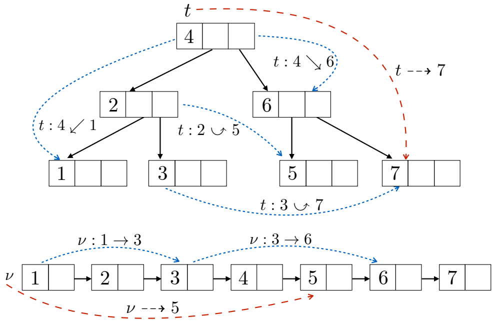
Copyright © ACM [to be supplied]. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

[1] Ignore the non-solid arrows and their labels for the time being.

[2] We preserve this convention throughout the paper.

$\nu \dashrightarrow u$	the value u is reachable from the list ν
$t \dashrightarrow u$	the value u is reachable from the tree node t
$\nu : u \rightarrow v$	the value u appears before the value v in list ν
$t : u \swarrow v$	the value v occurs in the *left* subtree of a node containing the value u in tree t
$t : u \searrow v$	the value v occurs in the *right* subtree of a node containing the value u in tree t
$t : u \curvearrowright v$	there is a node in the tree t for which u is contained in its *left* subtree and v is contained in its *right* subtree.

(a) Atomic shape predicates for lists and binary trees.



(b) Pictorial example of atomic shape predicates.

Figure 2: Atomic shape predicates.

trivial [25, 28]. It is even less apparent how we might define a data-driven inference procedure to *automatically discover and verify* such properties. This is because any such procedure must consider the symbiotic interplay of three key components, each of which is complex in its own right: (i) a *specification language* that is both expressive enough to describe properties relating the shape of a data structure and the values that it contains (for example the in-order relation mentioned above), yet which is nonetheless amenable as a target for learning and specification synthesis; (ii) a *learning algorithm* that can perform this synthesis task, yielding input-output specifications from the predicates drawn from the specification language; and, (iii) an *automated verification procedure* that enables formal verification of the implementation with respect to synthesized specifications learnt from observations.

Specification language. Our approach directly leverages the type definition of a data structure to enable generation of a set of *atomic predicates* that state general ordering properties about the values contained in the data structure *with respect to its shape*, given that interesting properties of inductive data structures are typically related to the way in which constructors are composed.

Learning algorithm. Our technique algorithmically uses these atomic predicates to postulate potentially complex shape specifications, learnt exclusively from the input-output behavior of functions that manipulate the data structure.

Notably, existing data-driven learning techniques are ineffective in discovering such specifications. Template-based mining techniques [12, 42, 53] require us to provide the Boolean skeleton of these specifications *a priori*, which we often do not know. Classification-based learning techniques [15, 16, 51, 54, 70] search for specifications that rule out so-called *bad* program states that represent violations of programmer-supplied assertions, usually annotated as post-conditions in source programs. The quality of searched specifications is thus limited by the quality of these annotations. More importantly, because these approaches fail to discover any useful information in the absence of annotated assertions, they would be unable to discover any interesting specification for the assertion-free program given in Fig. 1.

We address these issues by presenting the first (to the best of our knowledge) data-driven technique to automatically discover expressive shape specifications, without assuming any predefined templates, assertions or post-conditions, yet which is nonetheless able to learn the *strongest inductive invariant* (Sec. 4.3) in the solution space from which specifications are drawn.

Verification procedure. Our algorithm automatically verifies the correctness of these specifications in an expressive refinement type system. Cognizant that a presumed specification ψ may only express an unsound approximation of the correct hypothesis, our technique is *progressive*: *i.e.*, provided that the solution space from which specifications are drawn is sufficiently expressive, an unsound ψ serves as a counterexample that can be used to generate additional tests, eventually leading to the strongest inductive invariant in the solution space.

Contributions. Thus, our key contributions are in the development of a principled approach to generate useful atomic predicates for inductive data types drawn from a rich specification language, and a convergent learning algorithm capable of inferring verifiable ordering specifications using these predicates. Specifically, we:

1. Discover predicates for the expression of shape properties and generate their inductive definitions from the type definition of arbitrary user-defined algebraic data types.

2. Devise a data-driven learning technique to perform automatic inference and synthesis of function specifications using these predicates. Importantly, this learning strategy assumes no programmer annotations in source programs.

3. Verify the soundness of discovered specifications leveraging an expressive refinement type system equipped with a decidable notion of subtyping.

4. Evaluate our ideas in a tool, DORDER, which we use to synthesize and verify specifications on a large set of realistic and challenging functional data structure programs.

The remainder of the paper provides an overview of our specification language (Sec. 2); explains the synthesis mechanism through a detailed example (Sec. 3); provides details about type system, verification procedure, as well as soundness and progress results (Sec. 4); and describes generalizations of the core technique, presents implementation results, related work and conclusions (Secs. 5, 6, and 7).

2. Specification Language

The search space of our data-driven learning procedure includes shape properties defined in terms of atomic predicates stating either the *containment* of a certain value in a data structure, or relations establishing *ordering* between two elements found within the structure. These predicates define the *concept* class from which specifications are generated [2]. We discuss the basic intuition for how these predicates are extracted for the data types defined in our running example in Fig. 1 below.

We first consider possible containment predicates for trees. We are interested in knowing if a certain value u is present in a tree t. By observing the type definition of `'a tree` in Fig. 1, we know that only the constructor `Node` contains a value of type `'a` as its first argument. Therefore we can deduce that if u is present in t then either $t =$ `Node` (u, lt, rt), or $t =$ `Node` (v, lt, rt) and u is contained within lt or rt (with $u \neq v$). A similar argument can be made about lists. Containment predicates like these are denoted with a dashed horizontal arrow ($\nu \dashrightarrow u$ and $t \dashrightarrow u$) as shown in the first two rows of Fig. 2a.

A more interesting predicate class is one that establishes *ordering* relations between two elements of a data structure, u and v. Recall that in the `tree` definition only `Node` constructors contain values. However, since `Node` contains two inductively defined subtrees, there are several cases to consider when establishing an ordering relation among values found within a tree t. If we are interested in cases where the value u appears "before" (according to a specified order) v, we could either have that: (i) the value v occurs in the first (left) subtree from a tree node containing u, described by the notation $t : u \swarrow v$ in Fig. 2, (ii) the value v occurs in the second (right) subtree, described by the notation $t : u \searrow v$, (iii) or both values are in the tree, but u is found in a subtree that is disjoint from the subtree where v occurs. Suppose there exists a node whose first subtree contains u and whose second subtree contains v. This is the last case of Fig. 2a, and it is denoted as $t : u \curvearrowright v$. The symmetric cases are obvious, and we do not describe them. Notice that in this description we have exhausted all possible relations between any two values in a tree. The same argument can be made for `list` , which renders either the forward-order if the value u comes before v in a list l as $l : u \rightarrow v$, or the backwards-order for the symmetric case. Thus, our *ordering* predicates consider all relevant applications of constructors in which u and v are supplied as arguments.

`list`	$l =$ `Nil`	$l =$ `Cons` (u', l')
$l \dashrightarrow u$	**`false`**	$u = u' \vee l' \dashrightarrow u$
$l : u \rightarrow v$	**`false`**	$(u = u' \wedge l' \dashrightarrow v) \vee l' : u \rightarrow v$

`tree`	$t =$ `Leaf`	$t =$ `Node` (u', t_l, t_r)
$t \dashrightarrow u$	**`false`**	$u = u' \vee t_l \dashrightarrow u \vee t_r \dashrightarrow u$
$t : u \swarrow v$	**`false`**	$(u = u' \wedge t_l \dashrightarrow v) \vee$ $t_l : u \swarrow v \vee t_r : u \swarrow v$
$t : u \searrow v$	**`false`**	$(u = u' \wedge t_r \dashrightarrow v) \vee$ $t_l : u \searrow v \vee t_r : u \searrow v$
$t : u \curvearrowright v$	**`false`**	$(t_l \dashrightarrow u \wedge t_r \dashrightarrow v) \vee$ $t_l : u \curvearrowright v \vee t_r : u \curvearrowright v$

Table 1: Ordering and containment for `list` and `tree`.

The inductive definitions of the predicates obtained for lists and trees are presented in Tab. 1. For lists, the containment predicate $l \dashrightarrow u$ recursively inspects each element of a list l and holds only if u can be found in the list. The ordering predicate $l : u \rightarrow v$ relates a pair (u, v) to l if u appears before v in l. Similar definitions are given for trees. For example, the predicate $t : u \curvearrowright v$ is satisfied only if the tree t contains a subtree (including t itself) whose left subtree contains u and right subtree contains v.

To enable verification using off-the-shelf SMT solvers, our specification language disallows quantifier alternations (specifications are in prenex normal form, with universal quantification only permitted at the top-level), but nonetheless retains expressivity by allowing arbitrary Boolean combinations of the predicates. For example, we can specify `elements` (Fig. 1) with the following two specifications:

$$\begin{array}{l}(\forall u,\ \nu \dashrightarrow u \iff t \dashrightarrow u)\\ (\forall u\ v,\ \nu : u \rightarrow v \iff \begin{pmatrix} t : v \swarrow u \vee \\ t : u \curvearrowright v \vee \\ t : u \searrow v \end{pmatrix})\end{array} \qquad (1)$$

where the free variables u, v of Fig. 2a are universally quantified. In words, the specifications state that: (i) the values contained in the input tree t and the output list ν are exactly the same and (ii) for any two values u and v that appear in the *forward-order* in the output list ν, they are in the *in-order* of the input tree and vice versa. These specifications accurately capture the intended behavior of the function.

The full power of our specification language is realized in a practical extension (Sec. 5.2) that combines shape predicates with *relational data ordering constraints*, which are *binary predicates*, resulting in what we refer to as *shape-data* properties. For example, the following specification describes the characteristics of a binary search tree (`BST`), such as the instantiation (tree t) given in Fig. 2b:

$$(\forall u\ v,\ (t : u \swarrow v \Rightarrow u > v) \wedge (t : u \searrow v \Rightarrow u < v))$$

We can refine the specification of `elements` when applied to a `BST` to yield an accurate shape-data property that states the output list must be sorted: $(\forall u\ v, \nu : u \rightarrow v \Rightarrow u < v)$.

Hypothesis Domain. Equipped with these inductive definitions, we can define the hypothesis domain of containment and ordering properties which we denote as Ω. Given a function f, our hypothesis domain consists of a set of atomic predicates which relate the inputs and outputs of f. Assume that $\theta(f)$ is the set of function parameters and return values for f. Moreover, assume that $\theta_D(f)$ is the subset of $\theta(f)$ that includes all variables with data structure type (*e.g.*, `list` or `tree`) and $\theta_B(f)$ is the subset of $\theta(f)$ that includes all variables with base type (*e.g.*, `bool` or type variables).

The set of containment and ordering atomic predicates corresponding to a data structure variable $d \in \theta_D(f)$ included in the hypothesis domain of f contains the following predicates:

$$\Omega(d) = \{d \dashrightarrow u, d \dashrightarrow v\} \cup$$
$$\begin{cases} \{d : u \rightarrow v, d : v \rightarrow u\} & \texttt{\textbf{typeof}}\,(d) = \texttt{list} \\ \left\{\begin{array}{l} d : u \swarrow v, d : u \searrow v, d : u \curvearrowright v, \\ d : v \swarrow u, d : v \searrow u, d : v \curvearrowright u \end{array}\right\} & \texttt{\textbf{typeof}}\,(d) = \texttt{tree} \end{cases}$$

The logical variables u and v are free here, and will be universally quantified in the resulting specifications. For a variable $x \in \theta_B(f)$ of a base type we define:

$$\Omega(x) = \begin{cases} x & \texttt{\textbf{typeof}}\,(x) = \texttt{bool} \\ \{u = x, v = x\} \cup & \texttt{\textbf{otherwise}} \\ \{d \dashrightarrow x \mid d \in \theta_D(f)\} & \end{cases}$$

Finally, the hypothesis domain of a function f consists of the atomic predicates described by the definition of $\Omega(f)$ below.

$$\Omega(f) = \bigcup_{x \in \theta(f)} \Omega(x)$$

Specification Space. Assume that we denote with $BF(\Omega)$ the smallest set of Boolean formulas containing all the atomic predicates of Ω and closed by standard propositional logic connectives. The *specification space* of a function f, denoted by $Spec(\Omega, f)$, is the set of *input-output specifications* derivable from $BF(\Omega(f))$:

$$Spec(\Omega, f) = \{(\forall u\ v, \xi) \mid \xi \in BF(\Omega(f))\}$$

The free variables u and v occurring in the predicates found in ξ are universally quantified. Our construction guarantees that the specifications in $Spec(\Omega, f)$ can be encoded within the BSR (Bernays-Schönfinkel-Ramsey) first-order logic.

3. Specification Inference

Fig. 3 illustrates the design and implementation of our specification inference system. The input to our system is a data structure program. To bootstrap the inference process, we can use any advanced testing techniques for data structures.

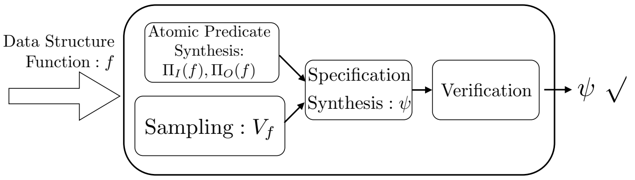


Figure 3: Specification synthesis architecture.

For simplicity, we use a *random testing* approach based on QUICKCHECK [8], which runs the program with a random sequence of calls to the API (interface functions) of the data structure. During this phase, we collect a set of inputs and outputs for each data structure function f into a sample set (which we generally denote with V_f). The bookkeeping of inputs and outputs simply records the mappings of variables to values, which in the case of inductive data structures uses a trivial serialization.[3]

Our system analyzes the data type definitions in the program and *automatically generates a set of atomic predicates* (c.f. Sec. 2), defining the hypothesis domain for the learning phase. For each function f, we partition its hypothesis domain $\Omega(f)$ into $\Pi_I(\Omega(f))$: the predicates over input variables of f (*e.g.*, t and $accu$ for the `flat` function in Fig. 1), and $\Pi_O(\Omega(f))$: the predicates over the functions output (the implicit variable ν). When the context is clear, we use $\Pi_I(f)$ or Π_I to abbreviate $\Pi_I(\Omega(f))$. This convention also applies to $\Pi_O(f)$ and Π_O. The extraction of predicates is abstractly depicted in the top left component of Fig. 3.

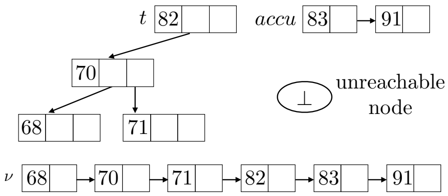
We then apply our learning algorithm to the samples in V_f, learning input-output relations over the atomic predicates of $\Pi_I(f)$ and $\Pi_O(f)$ that hold for all the samples. We obtain a candidate specification ψ for f, which is then fed into our verification system. In case verification fails, we show in Sec. 4 that our technique can make progress towards a valid specification for f by adding more tests systematically, provided that one such specification exists in the specification space of f. We illustrate the entire process by considering the verification of the `flat` function in Fig. 1.

3.1 Sampling

We first instrument the entry and exit points of functions to collect their inputs and outputs during testing. We use $V_{\texttt{flat}}$ to denote the set of samples collected during sampling for `flat`. Intuitively, $V_{\texttt{flat}}$ represents a coarse underapproximation of `flat`'s input and output behavior. Abstractly, we regard a sample σ as a function that maps program variables to concrete values in the case of base types, or a serialized data structure in the case of inductive data types.

Tab. 2a presents a pictorial view of a sample resulting from a call to `flat`. The sample manipulated by `flat` contains the input variables t and $accu$, as well as the result

[3] We assume the existence of generic serialization and deserialization functions, with the obvious recursive structure on the definition of the types.



(a) $V_{\texttt{flat}}$: input (t and $accu$), and output (ν) sampled data structures from `flat`.

$$\Pi_I(\texttt{flat}) \begin{cases} \Pi_0 \equiv t : u \swarrow v \quad \Pi_1 \equiv t : u \searrow v \quad \Pi_2 \equiv t : u \curvearrowright v \\ \Pi_3 \equiv t : v \swarrow u \quad \Pi_4 \equiv t : v \searrow u \quad \Pi_5 \equiv t : v \curvearrowright u \\ \Pi_6 \equiv t \dashrightarrow u \qquad \Pi_7 \equiv t \dashrightarrow v \\ \Pi_8 \equiv accu : u \rightarrow v \qquad \Pi_9 \equiv accu : v \rightarrow u \\ \Pi_{10} \equiv accu \dashrightarrow u \qquad \Pi_{11} \equiv accu \dashrightarrow v \end{cases}$$

$$\Pi_O(\texttt{flat}) \qquad \Pi_{12} \equiv \nu \dashrightarrow u \qquad \Pi_{13} \equiv \nu : u \rightarrow v$$

(b) Hypothesis domain ($\Omega(\texttt{flat})$): $\Pi_I(\Omega(\texttt{flat})) = \{\Pi_0, \cdots, \Pi_{11}\}$, and $\Pi_O(\Omega(\texttt{flat})) = \{\Pi_{12}, \Pi_{13}\}$.

	(u, v)	Π_0	Π_1	Π_2	Π_3	Π_4	Π_5	Π_6	Π_7	Π_8	Π_9	Π_{10}	Π_{11}	Π_{12}	Π_{13}
S	(68, 70)	0	0	0	1	0	0	1	1	0	0	0	0	1	1
	(83, 91)	0	0	0	0	0	0	0	0	1	0	1	1	1	1
	(82, 83)	0	0	0	0	0	0	1	0	0	0	0	1	1	1
	(68, 71)	0	0	1	0	0	0	1	1	0	0	0	0	1	1
	(70, 71)	0	1	0	0	0	0	1	1	0	0	0	0	1	1
U	(91, 83)	0	0	0	0	0	0	0	0	0	1	1	1	1	0
	(91, 70)	0	0	0	0	0	0	0	1	0	0	1	0	1	0
	(71, 68)	0	0	0	0	0	1	1	1	0	0	0	0	1	0
	(82, 70)	1	0	0	0	0	0	1	1	0	0	0	0	1	0
	(71, 70)	0	0	0	0	1	0	1	1	0	0	0	0	1	0
	(82, ⊥)	0	0	0	0	0	0	1	0	0	0	0	0	1	0
	(⊥, 82)	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	(83, ⊥)	0	0	0	0	0	0	0	0	0	0	1	0	1	0
	(⊥, 83)	0	0	0	0	0	0	0	0	0	0	0	1	0	0

(c) $V^b_{\texttt{flat}}$ is the evaluation of $V_{\texttt{flat}}$ expressed in terms of the predicates of Tab. 2b.

	Π_1	Π_2	Π_3	Π_6	Π_8	Π_{11}	Π_{13}
S	0	0	1	1	0	0	1
	0	0	0	0	1	1	1
	0	0	0	1	0	1	1
	0	1	0	1	0	0	1
	1	0	0	1	0	0	1
U	0	0	0	0	0	1	0
	0	0	0	1	0	0	0
	0	0	0	0	0	0	0

Boolean Formula from Tab. 2c

$$\Pi_{13} \iff \Pi_3 \vee \Pi_8 \vee (\Pi_6 \wedge \Pi_{11}) \vee \Pi_2 \vee \Pi_1$$

(d) Predicates selected for separation w.r.t. Π_{13}.

Table 2: Learning shape specifications for the `flat` function in Fig. 1.

ν (*i.e.* ν = `flat` t $accu$). In the figure, t is a root node with value 82, a link to a left subtree rooted at a node with value 70, and no right subtree; $accu$ is a two node list. In the sample, the result of the evaluation of `flat` is a list in which the in-order traversal of t is appended to $accu$.

Unreachables. While recording input/output pairs for runs of the function allows us to learn how its arguments and result are manipulated, it is also important to establish that data structures that are *not used* by the function cannot affect its behavior. To express such facts, we establish a *frame property* that delimits the behavior of the function f. The property manifests through a synthetic value $\bot$, which symbolically represents an arbitrary value known to be unrelated to the data structures manipulated by f. Our learning algorithm considers the behavior of predicates in the hypothesis domain with respect to this value. By stating atomic containment and ordering predicates in terms of $\bot$, we ensure that specifications inferred for f focus on values found in the data structures directly manipulated by f, preventing those specifications from unsoundly approximating values unrelated to the data structures manipulated by f.

Atomic Predicates. Given the atomic predicates in the hypothesis domain $\Omega($`flat`$)$ which are divided into Π_I and Π_O as shown in Tab. 2b,[4] we next relate observed samples with these predicates. Tab. 2c (ignore the first column labeled with S and U for the moment) shows the result of evaluating the atomic predicates of $\Omega($`flat`$)$ – which are essentially recursive functions over the data structure – with different instantiations for u and v derived from the sampled input/output pairs.[5] The variables u and v, which are always universally quantified in the final specifications, range over values observed in the sampled data structures as well as the synthetic value $\bot$. Importantly, since rows containing identical valuations for the predicates do not aid in learning, we keep *at most* one row with a unique valuation, discarding any repetitions. We denote the samples represented by this table as $V^b_{\texttt{flat}}$, a Boolean abstraction (or abstract samples) of $V_{\texttt{flat}}$ according to $\Omega($`flat`$)$.

For instance, the first row considers the pair where the variable u has the value 68, and the variable v has the value 70. The last four rows of Tab. 2c, containing pairs with the synthetic value $\bot$, and marked in blue, generalize observed data structures, relating them to hypothetical elements $\bot$ not accessible by the data structures of `flat`. Thus, the pair $(82, \bot)$ evaluates to true in Π_6 because 82 is reachable in t; all ordering predicates related to t where $u = \bot$ or $v = \bot$

[4] Π_O is simplified by removing the symmetric cases for ease of exposition.

[5] Entries are labeled 0 for false, and 1 for true.

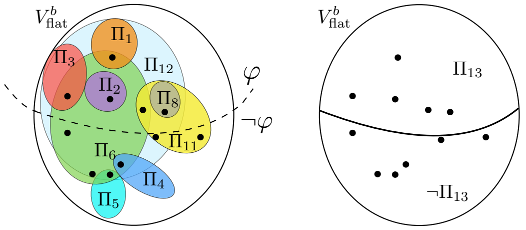


Figure 4: Learning a classifier φ w.r.t. Π_{13}. The two large sets $V^b_{\texttt{flat}}$ represent identical copies of the whole space of abstract samples. Each dot represents a valuation of in Tab. 2c. Each set marked with a predicate Π represents the samples that satisfy Π. We omit some predicates not used in the final specification.

(*i.e.*, $\Pi_0 - \Pi_5$) are false since there is no ordering relation between 82 and a value unreachable from t (see Tab. 2a).

3.2 Learning Specifications

Fig. 4 depicts our specification learning algorithm, in which the full set of observed abstracted samples $V^b_{\texttt{flat}}$ are depicted twice. On the left hand side of the picture, we show the subsets of samples that satisfy each predicate from Π_0 to Π_{12}.[6] On the right hand side, we depict the separation of $V^b_{\texttt{flat}}$ according to Π_{13}. The objective of our learning is to obtain a classifier φ in terms of the input predicates of Π_I (from Π_0 to Π_{11}) and Π_{12} which captures the same set of samples that are included in the output predicate Π_{13}. Once we find one such classifier φ, we know that in all samples the following predicate holds: $\Pi_{13} \iff \varphi$. This predicate can be considered a specification abstractly relating the function inputs to its outputs, according to the predicate Π_{13}.[7]

To synthesize this candidate specification by means of the output predicate Π_{13}, we split the samples in $V^b_{\texttt{flat}}$ according to whether the predicate Π_{13} holds in the sample or not. In Tab. 2c we mark with S the samples **S**atisfying Π_{13}, and with U the samples for which Π_{13} is **U**nsatisfied. Then, the goal of our learning algorithm is to produce a *classifier* predicate over Π_I (from Π_0 to Π_{11}) and Π_{12} which can separate the samples in S from the samples in U.

However, the potential search space for a candidate specification is often large, possibly exponential in the number atomic predicates in the hypothesis domain. To circumvent this problem, our technique is inspired by the observation that a simple specification is more likely to generalize in the program than a complex one [1, 22].

To synthesize a simple specification, a learning algorithm should select a minimum subset of the predicates that can achieve the separation. The details of the learning algorithm are presented in Sec. 3.3, but we show the final selection informally in Tab. 2d: Π_1, Π_2, Π_3, Π_6, Π_8 and Π_{11} constitute a sufficient classifier. To compute a final candidate classifier, we generate its truth table from Tab. 2d. The truth table should accept all the samples in S from Tab. 2d and conservatively reject every other sample. This step is conservative because we only generalize the samples in U (the truth table rejects more valuations than the ones sampled in Tab. 2d). We omit this step in our example in Tab. 2.

```
let rec insert x t =
 match t with
  | Leaf -> T (x, Leaf, Leaf)
  | Node (y, l, r) ->
    if x < y then Node (y, insert x l, r)
    else if y < x then Node (y, l, insert x r)
    else t
```

Figure 5: Binary search tree insertion function.

Once this truth table is obtained for the selected predicates, we apply standard logic minimization [39] techniques to infer the Boolean structure of the classifier. The obtained solution is shown in Tab. 2d, which in turn represents the following candidate specification by unfolding the definitions of the predicates Π_I and Π_O:

$$\big(\forall u\ v,\ \nu : u \rightarrow v \iff \begin{pmatrix} t : v \swarrow u \ \vee\ accu : u \rightarrow v \\ \vee\ (t \dashrightarrow u \ \wedge\ accu \dashrightarrow v) \\ \vee\ t : u \circlearrowright v \ \vee\ t : u \searrow v \end{pmatrix}\big) \tag{2}$$

Notice we add quantifiers to bind u and v, which essentially generalizes the specification to all other unseen samples.

To construct all salient input-output relations between Π_I and Π_O in Tab. 2b, we enumerate the predicates in Π_O. In a similar way, we use the other output predicate $\Pi_{12} \equiv \nu \dashrightarrow u$ to partition $V^b_{\texttt{flat}}$, learning the following specification:

$$\big(\forall u,\ \nu \dashrightarrow u \iff (t \dashrightarrow u \vee accu \dashrightarrow u)\big) \tag{3}$$

Verification. The conjunction of these two specifications are subsequently encoded into our verification system as the candidate specification for `flat` . We have implemented an automatic verification algorithm (described in Sec. 4), which can validate specifications of this kind.

Precision. The structure of Tab. 2c allowed us to find a classifier separating S from U, and thus provided us with a " $\iff$ " specification precisely relating Π_I with Π_{12} or Π_{13}. Unfortunately, but unsurprisingly, this is not always the case.

To see why, consider how we might infer a shape specification for the `insert` function of a binary search tree (see Fig. 5), whose hypothesis domain is shown in Tab. 3. As before, we proceed by executing the function, generating an abstract view of the function's concrete samples of $V^b_{\texttt{insert}}$ shown in Tab. 4.

As we have seen earlier, we would use Π_{10} to partition $V^b_{\texttt{insert}}$ to establish a relation between the predicates in Π_I and the predicates in Π_O of Tab. 3. If we consider the first S sample in Tab. 4, we see that it is exactly the same as the

[6] For perspicuity, the picture does not present an exact representation of the sets shown in Tab. 2c; in particular some predicates not used in the final specification are omitted.

[7] A similar construction of the input-output relation according to the output predicate Π_{12}, which is also in Π_O, will be considered later.

Π_I	$\Pi_0 \equiv t : u \swarrow v$	$\Pi_1 \equiv t : u \searrow v$	$\Pi_2 \equiv t : u \curvearrowright v$
	$\Pi_3 \equiv t : v \swarrow u$	$\Pi_4 \equiv t : v \searrow u$	$\Pi_5 \equiv t : v \curvearrowright u$
	$\Pi_6 \equiv t \dashrightarrow u$	$\Pi_7 \equiv t \dashrightarrow v$	
	$\Pi_8 \equiv u = x$	$\Pi_9 \equiv v = x$	
Π_O	$\Pi_{10} \equiv \nu : u \swarrow v$	$\cdots$	

Table 3: Hypothesis domain for the `insert` function.

	Π_0	Π_1	Π_2	Π_3	Π_4	Π_5	Π_6	Π_7	Π_8	Π_9	Π_{10}
S	0	0	0	0	0	0	1	0	0	1	1
	1	0	0	0	0	0	1	1	0	0	1
U	0	0	0	0	0	0	1	0	0	1	0
	0	0	0	0	0	0	0	1	1	0	0
	0	0	0	0	1	0	1	1	0	0	0
	0	0	0	1	0	0	1	1	0	0	0
	0	1	0	0	0	0	1	1	0	0	0
	0	0	0	0	0	1	1	1	0	0	0
	0	0	1	0	0	0	1	1	0	0	0

Table 4: Partition $V^b_{\texttt{insert}}$ evaluated from atomic predicates in Tab. 3 using the predicate Π_{10}.

first sample in `U` (except for the value of Π_{10}), meaning that no classifier can be generated from Tab. 4 to separate the samples precisely according to Π_{10}, since their intersection is not empty. To see why this could occur, consider the evaluation of

```
insert 3(Node (Node (Leaf, 2, Leaf), 5, Leaf))
```

Here, the input tree rooted at 5 has a non-empty left subtree rooted at 2. Based on the recursive definition of `insert`, 3 is inserted into the right subtree of 2 and is still in the left subtree of 5. Thus, abstracting the input-output behavior of `insert` with a pair of elements ($u = 5$ and $v = 3$) in the sample would correspond to the first row of `S` in Tab. 4 while the first row of `U` corresponds to an abstraction of a pair of elements ($u = 2$ and $v = 3$). Clearly, the latter pair does not satisfy Π_{10} while the former does.

To succeed in this case we need to relax the condition of obtaining exact "$\iff$" specifications by removing the samples that coincide in `S` and `U` for Π_{10} from `S` in $V^b_{\texttt{insert}}$. By doing so, upon inferring a classifier φ, we can conclude that $\varphi \Rightarrow \Pi_{10}$ is a likely specification for `insert`, since the set `S` has been generalized. Conversely, if the coinciding samples are removed from `U`, we can learn another classifier φ' and output a specification of the form $\Pi_{10} \Rightarrow \varphi'$.

Adopting this relaxation, our approach infers the following specification for `insert`:

$$\begin{aligned}&\big(\forall u\, v,\ t : u \swarrow v \Rightarrow \nu : u \swarrow v\big) \wedge \\ &\quad \big(\forall u\ v,\ \nu : u \swarrow v \Rightarrow \left(\begin{array}{c}(t \dashrightarrow u \wedge v = x) \vee \\ t : u \swarrow v\end{array}\right)\big)\end{aligned}$$

which asserts that x is added only in the bottom layer of the tree and the order of elements of the input tree is preserved in the output tree.

```
Algorithm 1: Synthesize ($f$)
let $V_f$ = test($f$) in;
let $V^b_f$ = $\alpha(V_f, \Omega(f))$ in;
let $\xi$ = Learn ($V^b_f$, $\Pi_I(\Omega(f))$, $\Pi_O(\Omega(f))$) in ($\forall u\ v,\ \xi$)
```

3.3 Formalization of Learning System

We now formalize the learning algorithm discussed in Sec. 3.2. Given a function f and the hypothesis domain Ω (Sec. 2), the problem of inferring an input-output specification for f reduces to a search problem in the solution space of $Spec(\Omega, f)$, driven by the samples of f.

For the remainder of the paper, we assume that a program sample σ is a mapping that binds program variables to values. These mappings are obtained from the log-file that records the execution trace. To relate the hypothesis domain $\Omega(f)$ to a set of samples V_f, we formally define a predicate-abstraction [19] function α on a sample $\sigma \in V_f$ as follows:

$$\alpha(\sigma, \Omega(f)) = \big\{ \langle \Pi_0(\sigma, u, v), \cdots, \Pi_n(\sigma, u, v) \rangle \,\big|\, u, v \in \texttt{Val}(\sigma) \cup \{\bot\} \text{ and } \Pi_0, \cdots, \Pi_n \in \Omega(f) \big\}$$

where we assume that $\texttt{Val}(\sigma)$ returns all values appearing in data structures within σ. This definition is trivially extended to a set of samples, for which we overload the notation as $\alpha(V_f, \Omega(f))$. As can be seen in the definition above, we consider the symbolic value $\bot$ (unreachable from f c.f. Sec. 3.1) when sampling the quantified variables u and v. The evaluation of predicates in $\Omega(f)$ is extended to the abstract value $\bot$ with the following set of equations:

$$(d \dashrightarrow \bot) = (d : u \,\mathcal{R}\, \bot) = (d : \bot \,\mathcal{R}\, v) = 0 \quad (x = \bot) = *$$

for all $\mathcal{R} \in \{\swarrow, \searrow, \curvearrowright\ \rightarrow\}$, $u, v \in \texttt{Val}(\sigma)$ and $x \in \theta_B(f)$. Notice that by the semantics of $\bot$, we do not need to consider the data structure $d \in \theta_D(f)$ in the equations above. In the first and the second cases, since $\bot$ is assumed to be unrelated to d we can safely deduce that the predicate must evaluate to 0. In the final case, any valuation of the predicate is possible, since we do not know the value of $\bot$; in that case, the evaluation results in $*$ representing either 0 or 1.

Algorithm 1 defines the main synthesis procedure. The first step is to obtain a set of samples V_f for the function f as described in the previous section. These samples are then evaluated according to $\Omega(f)$ using the abstraction function α (deriving V^b_f). For any valuation with a predicate Π_j resulting in a value $*$ the full vector is duplicated to consider both possible valuations of Π_j.

We then call the Learn algorithm (Algorithm 2 described below) to synthesize a candidate specification for f, which efficiently searches over the hypothesis domain of f, based on the valuation V^b_f. The resulting specification is returned after universally quantifying the free variables u and v.

Algorithm 2 takes as input a set of abstract samples (Boolean vectors) V^b, each of which is an assignment to

```
Algorithm 2: Learn ($V^b$, $\Pi_I$, $\Pi_O$)
$\bigwedge_{\Pi \in \Pi_O} \Big($ let $(V^b_S, V^b_U)$ = partition($\Pi$,$V^b$) in
        let $\Pi_C = \Pi_I \cup \Pi_O \backslash \{\Pi\}$ in
        if $V^b_S = \emptyset$ then $\neg\Pi$          else if $V^b_U = \emptyset$ then $\Pi$
else  $(\Pi \Rightarrow L(V^b_S, (V^b_U \backslash V^b_S), \Pi_C)) \wedge$
              $(L((V^b_S \backslash V^b_U), V^b_U, \Pi_C) \Rightarrow \Pi)\Big)$
```

the predicates in $\Pi_I \cup \Pi_O$; it aims to learn relations expressed in propositional logic between the predicates in Π_I and those in Π_O, using the structure of V^b.

For each predicate $\Pi \in \Pi_O$, the algorithm partitions V^b into the abstract *sat* samples V^b_S which satisfy Π and the *unsat* samples V^b_U which do not. Each abstract sample $\sigma^b \in V^b_S \cup V^b_U$ is a Boolean vector over the predicates $\Pi_C \equiv \Pi_I \cup \Pi_O \backslash \{\Pi\}$.

If V^b_S is empty, we conclude that $\neg\Pi$ is a candidate specification. The case when V^b_U is empty is symmetric. Otherwise the learning algorithm L aims to produce a *consistent* binary classifier φ with respect to V^b_S and V^b_U, that is, it must satisfy the following requirement:

$$\left(\forall \sigma^b \in V^b_S,\ \varphi(\sigma^b)\right) \quad \& \quad \left(\forall \sigma^b \in V^b_U,\ \neg\varphi(\sigma^b)\right)$$

In other words, the result of $L(V^b_S, V^b_U, \Pi_C)$ should be an interpolant [52] separating the *sat* samples (V^b_S) from the *unsat* samples (V^b_U). If this classification algorithm succeeds, $\Pi \iff L(V^b_S, V^b_U, \Pi_C)$ captures the *iff relation* between Π and the rest of the predicates in $\Pi_I \cup \Pi_O$ (c.f. Π_C).

However, there is no guarantee that V^b_S and V^b_U must be separable because there could be coinciding samples in V^b_S and V^b_U. To address this possibility, we first remove coinciding samples from V^b_U and infer $\Pi \Rightarrow L(V^b_S, V^b_U \backslash V^b_S, \Pi_C)$, and similarly remove them from V^b_S, resulting in the specification $L(V^b_S \backslash V^b_U, V^b_U, \Pi_C) \Rightarrow \Pi$. Algorithm 2 does not list the cases when $V^b_U \backslash V^b_S$ or $V^b_S \backslash V^b_U$ are empty. In such cases, it is impossible for L to find a classifier, indicating that the hypothesis domain is insufficient to find a corresponding relation between Π and Π_C.

The implementation of $L(V^b_S, V^b_U, \Pi_C)$ reduces to the well-studied problem of inferring a classifier separating some samples V^b_S from the other samples V^b_U using predicates form Π_C [54, 70]. To generalize, we attempt to find the solution which uses the *minimal number of predicates from the hypothesis domain* to classify the samples, as exemplified in Tab. 2d. A number of off-the-shelf solvers can be used to solve this constraint optimization problem [4, 57]. We employ the simple classifier described in [70] to implement L. Details are provided in our technical report [71].

4. Verification

This section presents the full verification procedure of our technique, in the context of an idealized functional language.

$$\begin{array}{c}
x, y, d, f, \nu \in \texttt{Var} \qquad c \in \texttt{Constant} \qquad \texttt{'a} \in \texttt{TyVar} \\
\begin{array}{rcl}
v \in \texttt{Val} & ::= & c \mid x \mid \lambda\, x\, e \mid \texttt{fix}\,(\texttt{fun}\ f \rightarrow \lambda\, x\, e) \\
e \in \texttt{Exp} & ::= & v \mid e_0 \oplus e_1 \mid e\ v \mid \mathcal{C}\langle \vec{x}, \vec{d} \rangle \mid \forall \texttt{'a} \cdot e \mid \tau\ e \\
& \mid & \texttt{if}\ v\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \mid \texttt{let}\ x = e_0\ \texttt{in}\ e_1 \\
& \mid & \texttt{match}\ v\ \texttt{with}\ \big|_i\ \mathcal{C}_i\langle \vec{x_i}, \vec{d_i} \rangle \rightarrow e_i
\end{array}
\end{array}$$

$$\begin{array}{rl}
\psi \in & \text{Specification Space}(\Omega) \\
P \in \texttt{RType} ::= & \{\nu : B \mid \psi\} \\
\mid & \{\nu : D \mid \psi\} \\
\mid & x : P \rightarrow P
\end{array}
\qquad
\begin{array}{r}
B \in \texttt{Base} ::= \texttt{'a} \mid \texttt{int} \mid \texttt{bool} \\
D \in \texttt{DType} ::= \mu t\ \Sigma_i\ \mathcal{C}_i\langle \vec{\texttt{'a}}, \vec{D_i} \rangle \\
\tau ::= B \mid D \mid x : \tau \rightarrow \tau
\end{array}$$

Figure 6: Syntax and Types.

4.1 Programming Language

Our language is a core-ML call-by-value variant of the λ-calculus with support for polymorphism and refinement types. Fig. 6 provides the syntax of our language in A-normal form. Primitive operators are encoded with the meta-operator $\oplus$ (where unary operators ignore the second argument). By convention, metavariables x and y represent program variables, d represents a variable with an inductive data type, and f represents a function. We denote by $\vec{x}$ a sequence of program variables, and similarly for the syntactic categories of values, type variables (TyVar) and data types (DType). We additionally provide the syntactic sugar form let rec defined in terms of fix in the usual way.

At the type level, the language supports base types B and user-defined inductive data types D. We use $\mathcal{C}$ to represent data type constructors. To simplify the presentation, we only consider polymorphic inductive data type definitions, and require all type variables ('a) to appear before all the data types in constructor expressions.

To formally verify program specifications, we encode them into refinement types (RType) and employ a refinement type system. A data type such as list is specified into a *refinement data type* written $\{\nu : \texttt{list} \mid \psi\}$ where ψ (a *type refinement predicate*) is a Boolean-valued expression. This expression constraints the value of the term (defined as the special variable ν) associated with the type. In this paper, ψ is drawn from the specification space parameterized by a hypothesis domain Ω. For expository purposes, we assume Ω is instantiated to the domain defined in Sec. 2.

A *refinement function type*, written $\{x : P_x \rightarrow P\}$, constrains the argument x by the refinement type P_x, and produces a result whose type is specified by P. For example, the specification (3) is encoded as the following type:

$$\begin{array}{l}
\texttt{flat} : accu : \texttt{'a list} \rightarrow t : \texttt{'a tree} \rightarrow \\
\quad \Big\{\nu : \texttt{'a list} \mid \big(\forall u,\ \nu \dashrightarrow u \iff (t \dashrightarrow u \vee accu \dashrightarrow u)\big)\Big\}
\end{array}$$

The encoding of a candidate specification ψ obtained from the learning algorithm (as in Sec. 3.3) into a refinement type is given via the specType(Γ_f, f, ψ) definition below where Γ_f is the type environment for the definition of f.

$$\begin{array}{rcl}
\texttt{spec}(B,\ \psi) & = & \{\nu : B \mid \psi\} \\
\texttt{spec}(D,\ \psi) & = & \{\nu : D \mid \psi\} \\
\texttt{spec}(\{x:\tau_1 \to \tau_2\},\ \psi) & = & \{x:\tau_1 \to \texttt{spec}(\tau_2,\psi)\} \\
\texttt{specType}(\Gamma_f, f, \psi) & = & \texttt{spec}(\texttt{HM}(\Gamma_f, f), \psi)
\end{array}$$

Here we assume the existence a Hindley-Milner type checking oracle $\texttt{HM}(\Gamma_f, f)$ which returns the unrefined type of a function f. The auxiliary function `spec` pushes a specification ψ of f into the result type of f because ψ is assumed to capture the input-output relations of f.

4.2 Refinement Type System

An excerpt of our refinement type system is given in Fig. 7. The type system is an extension of LIQUID TYPES [30, 50]. The basic typing judgment is of the form $\Gamma \vdash e : P$, where the typing environment Γ comprises type bindings mapping program variables to refinement types (eg. $x : P$), and refinement predicates constraining the variables bound in Γ. The judgment means that under the environment Γ, where the values in the bound variables are assumed to satisfy the constraints contained in Γ, the expression e has the refinement type P. To ease the exposition, we show only the most salient rules, and in particular, we only show *instances* of the general rules for the list data structure.[8]

The LIST MATCH rule stipulates that the entire expression has type P if the body of each of the match cases has type P under the type environment extended with the variables bound by the matched pattern, where the variables bound assume types as defined by the constructor definition. Moreover, *we unfold the inductive definitions of the atomic predicates* from our hypothesis domain Ω in the environment, exploiting the fact that we know the structure of the matched pattern (c.f. the case considered), thus allowing us to use the variables bound in the matched pattern to instantiate the variables of the recursive unfolding of the predicate. For instance, in the $\texttt{Cons}\ (x, xs)$ case, we use x and xs to stand for the existential variables u' and l' in the definition of Tab. 1. In summary, the guard predicates unfold the inductive definitions introduced in Tab. 1.

The FUNCTION rule for recursive functions has a subtyping constraint associated with function abstractions:

$$\Gamma; x : P_x \vdash P_e <: P$$

which establishes a constraint on the post-condition P of the abstraction (in our case encoding the synthesized candidate specifications) and it is required to be consistent with P_e inferred for the function body using the type checking rules.

Finally, the rule SUBTYPE DTYPE checks whether a refinement type subtypes another by issuing an implication verification condition over the refinement predicates of the types involved. We use the notation $\langle\psi\rangle$ to denote the encoding of refinement predicates ψ into terms of (decidable) BSR logic. Our encoding translates the containment and ordering predicates in ψ into uninterpreted relations.

The validity check in the premise of the rule SUBTYPE DTYPE requires that the conjunction of the environment formula $\langle\Gamma\rangle$ and $\langle\psi_1\rangle$ implies $\langle\psi_2\rangle$. Our encoding of $\langle\Gamma\rangle$ is adapted from [30, 50]:

$$\langle\Gamma\rangle = \bigwedge \{\, \langle[x/\nu]\psi\rangle \,|\, (x : \{\tau \mid \psi\}) \in \Gamma \,\wedge\, \tau \in B \cup D\}$$

Recall that for a function f, the set of specifications allowed in the specification space of containment and ordering formulae are restricted to the form:

$$\psi \in \{(\forall u\ v, \xi) \mid \xi \in BF(\Omega(f))\}$$

The prenex normal form of the encoding of the premise in the rule SUBTYPE DTYPE, $\texttt{Valid}(\langle\Gamma\rangle \wedge \langle\psi_1\rangle \Rightarrow \langle\psi_2\rangle$, therefore results in a $\exists^*\forall^*$ quantifier prefix, with no functions. As a result, subtype checking in our system is decidable and can be handled by a BSR solver [45].

The soundness of the refinement type system is defined with respect to a reduction relation ($\hookrightarrow$) that encodes the language's operational semantics, which is standard:

Theorem 1. *If $\emptyset \vdash e : P$, then either e is a value, or there exists an e' such that $e \hookrightarrow e'$ and $\emptyset \vdash e' : P$.*

The completeness of subtype checking reduces to the completeness of the underlying solver for inductive data types. For lists or trees, we use additional axioms (as local theory extensions [23]) based on first-order axiomatizations of transitive closures found in [31, 47] to bound the shape of list or tree data structures in BSR models to ensure completeness.

4.3 Progress

For a candidate specification ψ inferred for the recursive function f, our verification algorithm encodes ψ into the refinement type of f and checks the following judgment

$$\Gamma_f \vdash \texttt{fix}\,(\texttt{fun}\ f \to \lambda x.\, e) : \texttt{specType}(\Gamma_f, f, \psi)$$

where Γ_f is the type environment under which f is defined. We call a specification ψ which can be type-checked as shown above an *inductive invariant* of f. We call ψ the *strongest inductive invariant* of f in $Spec(\Omega, f)$, if for any other inductive invariant ψ_f of f in $Spec(\Omega, f)$, $\Gamma_f \vdash \texttt{specType}(\Gamma_f, f, \psi) <: \texttt{specType}(\Gamma_f, f, \psi_f)$ holds.

Importantly, our technique is *progressive*. This means that it is always possible to add new tests to refine ψ whenever ψ fails to be inductive, provided that one inductive invariant exists in the specification space. We formalize the progressive property in Theorem 2 under the assumption that the underlying solver is complete (c.f. Sec. 4.2).

Theorem 2. *Given a function f with a hypothesis domain Ω, and assuming that an inductive invariant of f exists in $Spec(\Omega, f)$, if $\Gamma_f \not\vdash \texttt{fix}\,(\texttt{fun}\ f \to \lambda x.\, e) : \texttt{specType}(\Gamma_f, f, \psi)$ where $\psi =$ Synthesize(f), then there exists a test input for f which leads to an unseen sample σ of f, for which $\psi(\sigma)$ does* not *hold; otherwise ψ is the strongest inductive invariant of f in $Spec(\Omega, f)$.*

[8] The full type system provides general rules for arbitrary inductive data types and is presented in our technical report [71].

$$\textsc{List Match}\quad \frac{\Gamma \vdash v : \texttt{'a list} \qquad \left[\ \Gamma;\ (\forall u\, v,\ v : u \to v \iff \texttt{false} \wedge \forall u,\ v \dashrightarrow u \iff \texttt{false})\ \right] \vdash e_1 : P \qquad \left[\begin{array}{c} \Gamma;\ x : \texttt{'a};\ xs : \texttt{'a list};\ (\forall u,\ v \dashrightarrow u \iff (u = x \vee xs \dashrightarrow u) \\ \wedge \forall u\, v,\ v : u \to v \iff ((u = x \wedge xs \dashrightarrow v) \vee xs : u \to v)) \end{array}\right] \vdash e_2 : P}{\Gamma \vdash \left(\ \texttt{match}\ v\ \texttt{with} \mid \texttt{Nil} \to e_1 \mid \texttt{Cons}\,(x, xs) \to e_2\ \right) : P}$$

$$\textsc{Function}\quad \frac{\Gamma; f : \{x : P_x \to P\}; x : P_x \vdash e : P_e \qquad \Gamma; x : P_x \vdash P_e <: P}{\Gamma \vdash \texttt{fix}\,(\texttt{fun}\ f \to \lambda x.\, e) : \{x : P_x \to P\}}$$

$$\textsc{Subtype DType}\quad \frac{\texttt{Valid}(\langle \Gamma \rangle \wedge \langle \psi_1 \rangle \Rightarrow \langle \psi_2 \rangle)}{\Gamma \vdash \{D \mid \psi_1\} <: \{D \mid \psi_2\}}$$

Figure 7: Representative Typing Rules (list instantiation excerpt).

The theorem states that if an inductive invariant of f exists in the specification space parameterized by Ω (*i.e.*, in $Spec(\Omega, f)$), then for any candidate specification ψ inferred for f, either ψ is such an invariant (*i.e.*, refinement type checking succeeds) and is the strongest one in the specification space, or there exists a test input which yields a concrete program sample that invalidates ψ. We remark that finding such a test input reduces to the well-studied problem of generating inputs for a program (function f) causing it to violate its specifications (safety property ψ). In our setting, we can harness techniques such as [41], which provides a relatively complete method for counterexample generation in functional (data structure) programs, to derive test inputs that violate ψ. In fact, because ψ is an input-output specification, we can directly reconstruct a new test input from SMT models of subtype checking failures. In turn, running the learning algorithm using the new program samples from the new input, necessarily produces a more refined invariant. This strategy, which can be implemented via a CEGIS (counterexample guided inductive synthesis) loop [2, 55], ensures that *we can construct a finite number of test cases to guarantee convergence in the presumed specification space*.

Details about the proof of Theorem 2 are provided in our technical report [71]. The key idea is that our learning algorithm ensures that ψ will *never produce an invariant that is true for all possible function input/output pairs*, but *which is not inductive*. This is a fundamental property, since an invariant that is true which fails to be inductive (*i.e.*, fails type checking) cannot be invalidated by adding tests, since the true invariant is guaranteed to be satisfied in every test run. Without such a property, we might never find a typable specification.

Consider the `flat` function in Fig. 1. If our only goal was to use the smallest number of atomic predicates from the hypothesis domain to construct a specification (satisfied by all the samples of `flat`), we obtain the following result:

$$\big(\forall u\, v,\ \nu : u \to v \Rightarrow \\ \left(\begin{array}{l} (t \dashrightarrow u \wedge accu \dashrightarrow v) \vee \\ (t \dashrightarrow u \wedge t \dashrightarrow v) \vee\ (accu \dashrightarrow u \wedge accu \dashrightarrow v) \end{array}\right)$$

Compared to the specification (2), the above specification is simpler (comprising fewer atomic predicates) and is always true for the program above. But it is *not* an inductive invariant, and cannot be verified using our type checking rules, especially the FUNCTION rule in Fig. 7. In particular, the failure stems from the predicate $(t \dashrightarrow u\ \wedge\ t \dashrightarrow v)$ in the last line of the specification, which is too over-approximative. It does not specify an order between u and v if they both come from t, which is necessary to discharge the subtype constraint in the FUNCTION rule. Adding more tests would not refine the resulting specification, since it is a true invariant, albeit not an inductive one.

Our learning algorithm rules out this problem by guaranteeing that any candidate specification *rejects* a Boolean assignment to the selected atomic predicates that are *not observed or inconsistent with the samples*. This means that for any two elements u, v from t, if u occurs before v in the output list (ν), any learnt specification must ensure that u and v respect the in-order property of t, since such a property would be observed in every sample. More generally, for any two elements u, v from t that do *not* respect the in-order of t, they are classified into the $\texttt{U}(\texttt{nsat})$ samples of $\nu : u \to v$.

5. Extensions

Previous sections focused on list and tree data structures to illustrate our technique. But, as we elaborate below, DORDER supports complex functional data structures beyond lists and trees, including nested and composite structures.

We also discuss the extension of our algorithm to synthesize specifications relating data constraints to values contained within inductive data structures. Surprisingly, the expressive power of our learning procedure is not constrained by the underlying hypothesis domain on which it is parameterized. In this sense, we claim that DORDER defines a *general* framework to perform specification synthesis.

5.1 Extensions for Arbitrary User-defined Inductive Data Structures

Our technique discovers "templates" of atomic-predicates on a per-data-structure basis. We are able to discover customized ordering predicates for nested datatypes (*e.g.*, multiway-trees), and composite datatypes that have significantly different structure than the predicates discovered for simple trees and lists.

We first present the general definitions for ordering and containment predicates. Given a data structure $\mathcal{C}_h\langle\vec{x},\vec{d}\rangle$, the definition of the *containment* predicate ($\mathcal{C}_h\langle\vec{x},\vec{d}\rangle \dashrightarrow u$) simply states that value u can be found in the data structure, and can be defined generically as follows:

$$\mathcal{C}_h\langle\vec{x},\vec{d}\rangle \dashrightarrow u \equiv \bigvee_{i=1}^{|\vec{x}|} x_i = u \;\vee\; \bigvee_{j=1}^{|\vec{d}|} d_j \dashrightarrow u$$

where $|\vec{d}|$ (*resp.* $|\vec{x}|$) denotes number of inductive data type (*resp.* base type or type variable) valued arguments of the constructor $\mathcal{C}_h$. This definition, when applied to a list or tree data type, renders the definitions shown in Sec. 2.

The definition of predicates that expose ordering relations must take into account: (i) the constructor of the data structure, and (ii) which arguments of the constructor need to be considered. All these arguments are provided in the generic version of the *order* predicate; we express it using the notation $\mathcal{C}_h\langle\vec{x},\vec{d}\rangle : u@n \xrightarrow{\mathcal{C}} v@m$. This predicate asserts that, in the data structure $\mathcal{C}_h\langle\vec{x},\vec{d}\rangle$, there exists an ordering relation between the values u and v in a substructure of the data structure (including itself), constructed from the $\mathcal{C}$ constructor, and u and v relate to the n^{th} and m^{th} arguments of $\mathcal{C}$. Formally, the predicate is satisfied in two cases: (a) if $\mathcal{C}_h$ is $\mathcal{C}$ and the n^{th} argument on the application of $\mathcal{C}$ is of a base type, then it must equal u, otherwise, if it is of an inductive data type, it must contain the value u, and similarly for the m^{th} argument, using value v; (b) or is recursively established in the substructures of d. The full recursive definition is given below.

$$\mathcal{C}_h\langle\vec{x},\vec{d}\rangle : u@n \xrightarrow{\mathcal{C}} v@m \equiv \Big(\bigvee_{j=1}^{|\vec{d}|} d_j : u@n \xrightarrow{\mathcal{C}} v@m\Big) \vee$$
$$\begin{cases} x_n = u \wedge x_m = v & \text{if } \mathcal{C}_h = \mathcal{C} \text{ and } n, m \leq |\vec{x}| \\ x_n = u \wedge d_{m-|\vec{x}|} \dashrightarrow v & \text{if } \mathcal{C}_h = \mathcal{C} \text{ and } n \leq |\vec{x}| \\ d_{n-|\vec{x}|} \dashrightarrow u \;\wedge d_{m-|\vec{x}|} \dashrightarrow v & \text{if } \mathcal{C}_h = \mathcal{C} \\ \texttt{false} & \text{otherwise} \end{cases}$$

Recall that we assume that all variables in $\vec{x}$ are of base type, and all the ones in $\vec{d}$ are of inductive data types in constructors. Then, the first disjunct represents the recursive definition to the substructures of d, and the other cases correspond to the description given above. Our approach considers all constructors and their arguments of a data type definition to export all such order predicates. With this generic definition we can de-sugar the definitions we provided in Sec. 2 as shown in Tab. 5.

5.2 Specifications over Shapes and Data

We now enrich our inference algorithm to infer specifications relating data constraints (binary predicates) to values contained within inductive data structures. We extend our hypothesis domain to include binary data predicates, which are restricted to range over relational data ordering properties. Given a function f, the data domain (denoted by $\Pi_{\texttt{data}}$), is constructed from the atomic predicates:

$$\Pi_{\texttt{data}}(f) = \{u \leq v, v \leq u\} \cup \{u \leq x, x \leq u \mid x \in \theta_B(f)\}$$

$l : u \rightarrow v$	$l : u@1 \xrightarrow{\texttt{Cons}} v@2$
$t : u \searrow v$	$t : u@1 \xrightarrow{\texttt{Node}} v@3$
$t : u \swarrow v$	$t : u@1 \xrightarrow{\texttt{Node}} v@2$
$t : u \circlearrowright v$	$t : u@2 \xrightarrow{\texttt{Node}} v@3$

Table 5: Definitions of shape predicates for list and tree.

$$\Pi_{\texttt{data}} \begin{cases} \Pi_0 \equiv u \leq v & \Pi_1 \equiv v \leq u \\ \Pi_2 \equiv u \leq x & \Pi_3 \equiv x \leq u \end{cases}$$

$$\Pi_{\texttt{shape}} \begin{cases} \Pi_4 \equiv t : u \swarrow v \quad \Pi_5 \equiv t : u \searrow v \quad \Pi_6 \equiv t : u \circlearrowright v \\ \Pi_7 \equiv \nu : u \swarrow v \quad \Pi_8 \equiv \nu : u \searrow v \quad \Pi_9 \equiv \nu : u \circlearrowright v \\ \qquad \Pi_{10} \equiv t \dashrightarrow u \quad \Pi_{11} \equiv \nu \dashrightarrow u \end{cases}$$

Table 6: Hypothesis domain for synthesizing shape and data specifications of the `insert` function (Fig. 5).

While the domain $\Pi_{\texttt{data}}$ is small in the number of permissible predicates, our experiments show that it is sufficient to synthesize sophisticated properties such as BST, heap- and list-sortedness, etc. Recall that we also admit the following set of of predicates over the shapes of the data structures:

$$\Pi_{\texttt{shape}}(f) = \{d \dashrightarrow u, d \dashrightarrow v, d : u \rightarrow v,$$
$$d : u \swarrow v, d : u \searrow v, d : u \circlearrowright v \mid d \in \theta_D(f)\}$$

where only well-typed predicates are considered (depending on the type of d). To learn shape-data properties, for a given set of samples V_f of f we use Algorithm 2 to compute:

$$\forall u\ v,\ \text{Learn}\ (V_f^b,\ \Pi_{\texttt{data}}(f),\ \Pi_{\texttt{shape}}(f))$$

where V_f^b is evaluated from V_f using the α abstraction function defined in Sec. 3.3 based on the predicates from $\Pi_{\texttt{data}}(f) \cup \Pi_{\texttt{shape}}(f)$.

To discharge the candidate specifications produced by this domain, following [24, 25], we encode binary predicates in $\Pi_{\texttt{data}}$ as ordering relations, and feed the resulting formula to an SMT solver, which permits multiple relation symbols.

Consider the binary search tree `insert` function in Fig. 5. Tab. 6 shows the atomic predicates in the hypothesis domain that allows the inference of shape and data invariants.

With this extended hypothesis domain, we derive the following specification for the `insert` function:

$$(\forall u\ v, t : u \swarrow v \Rightarrow (\neg u \leq v)) \wedge (\forall u\ v,\ t : u \searrow v \Rightarrow (\neg v \leq u))$$

essentially specifying that t is a BST, abbreviated as BST(t). This specification over the input t, in conjunction with the specification learnt over the output variable ν, makes it possible to infer the following refinement type for `insert` :

$$x : \texttt{'a} \rightarrow t : \{\nu : \texttt{'a tree} \mid \texttt{BST}(\nu)\} \rightarrow \{\nu : \texttt{'a tree} \mid \texttt{BST}(\nu)\}$$

5.3 Specifications over Numeric Properties

Another important class of data structure invariants uses common *measures* of data types, which maps a data structure to a numeric value, such as the *length* of a list or *height* of a tree. Such measures are needed, for instance, to prove that a binary tree respects a tree *balance* specification. DORDER integrates measure definitions used in the source code into a hypothesis domain that can be leveraged by the learning algorithm to enrich data structure specifications. For instance, consider the following code snippet adapted from a recursive tree balance function `bal l v r` $= \nu$ from the OCaml set library implementation.

```
let rec bal l v r =
  let hl = ht l in
  let hr = ht r in
  if hl > hr + 2 then ... /*call bal on l*/
  else if hr > hl + 2 then ... /*call bal on r*/
  else Node (v, l, r)
```

Here, two input trees `l` and `r` with arbitrary heights and a single value `v` are merged into one output balanced tree ν. Observe the function uses a `ht` measure, which returns the height of a tree. The definition of `ht` is standard and elided, but would presumably be provided as a useful measure that should appear in specifications.

To simplify the presentation, assume that a certain function f manipulates only one data structure, and furthermore that a single measure m is associated with that data structure. We consider the hypothesis domain for numeric properties over the hypothesis domain which we denote by $\Omega_{\texttt{num}}$:

$$\Omega_{\texttt{num}}(f) = \Big\{ \pm m(x) \pm m(y) \leq l \;\Big|\; x, y \in \theta_D(f) \wedge \\ 0 \leq l \leq C \text{ where } C \text{ is the maximum constant in } f \Big\}$$

Predicates drawn from this domain allow us, for example, to compare the height of different input subtrees or sublists or compare the height of an input tree with an output tree, or the length of an input list with the length of an output list. It suffices to use Algorithm 2 to compute:

$$\text{Learn}(V_f^b, \Pi_I(\Omega_{\texttt{num}}(f)), \Pi_O(\Omega_{\texttt{num}}(f)))$$

for synthesizing numeric input-output specifications for f (without generating quantifiers) where V_f^b is evaluated from a number of input-output samples of f using predicates from $\Omega_{\texttt{num}}(f)$.

Because we allow integer constants in $\Omega_{\texttt{num}}(f)$, it is possible to synthesize specifications that are vacuous. For example, if $m(x) - m(y) \leq l_1$ is chosen as an output predicate in the learning algorithm, we may synthesize a formula $m(x) - m(y) \leq l_1 \Rightarrow m(x) - m(y) \leq l_2$ where $l_1 \leq l_2$; this formula, while logically true, is semantically useless. We detect such invariants using an SMT solver, and restart synthesis, filtering $m(x) - m(y) \leq l_2$ out of the hypothesis domain.

Consider now how we might synthesize a specification for the *recursive*-balance function. We show a subset of input predicates Π_I from $\Omega_{\texttt{num}}(\texttt{bal})$ for expository purposes:

$$\texttt{ht}\ l \leq \texttt{ht}\ r,\ \texttt{ht}\ l \leq \texttt{ht}\ r + 2,\ \texttt{ht}\ r \leq \texttt{ht}\ l + 2, \cdots$$

Similarly the output predicates Π_O contains:

$$\texttt{ht}\ l \leq \texttt{ht}\ \nu,\ \texttt{ht}\ \nu \leq 1 + \texttt{ht}\ l,\ \texttt{ht}\ \nu \leq 1 + \texttt{ht}\ r, \cdots$$

By applying Learn$(V_{\texttt{bal}}^b, \Pi_I, \Pi_O)$ where $V_{\texttt{bal}}^b$ is evaluated from a number of input-output samples of `bal` using predicates from $\Omega_{\texttt{num}}(\texttt{bal})$, our technique automatically synthesizes the following specification:

$$\begin{array}{rcl}
\texttt{ht}\ l \leq \texttt{ht}\ \nu & \wedge & \texttt{ht}\ r \leq \texttt{ht}\ \nu\ \wedge \\
\texttt{ht}\ r \leq \texttt{ht}\ l & \Rightarrow & \texttt{ht}\ \nu \leq 1 + \texttt{ht}\ l\ \wedge \\
\neg(\texttt{ht}\ r \leq \texttt{ht}\ l) & \Rightarrow & \texttt{ht}\ \nu \leq 1 + \texttt{ht}\ r\ \wedge \\
\neg(\texttt{ht}\ r \leq \texttt{ht}\ l + 2) & \Rightarrow & 1 + \texttt{ht}\ r \leq \texttt{ht}\ \nu\ \wedge \\
(\texttt{ht}\ l \leq \texttt{ht}\ r + 2) & \Rightarrow & 1 + \texttt{ht}\ l \leq \texttt{ht}\ \nu
\end{array}$$

which precisely specifies that the height of the returned tree is either $\texttt{max}(\texttt{ht}\,l, \texttt{ht}\,r)$ or $\texttt{max}(\texttt{ht}\,l, \texttt{ht}\,r) + 1$ and is always the latter when $|\texttt{ht}\,l - \texttt{ht}\,r| \leq 2$. Handcrafting this specification by the programmer is challenging. Yet, the specification turns out to be key to proving that `bal` is guaranteed to return a balanced tree.

6. Experiments

DORDER is an implementation of our learning procedure and type-based verification technique.[9] We use the Z3 SMT solver [10] to discharge our verification conditions. DORDER takes as input an inductive data structure program, written in OCaml, and produces as output the list of specifications (as refinement types) for the functions in the program.

Random Testing. While the progressive property of Theorem 2 guarantees that the learning algorithm can be equipped with a directed and automated test synthesis procedure, our implementation simply uses a lightweight random testing strategy based on QUICKCHECK [8]. Concretely, DORDER synthesizes the specifications for a data structure program using the test data obtained from executing the program by a random sequence of method calls to the data structure's interface functions. In our experience, the length of such call sequences can be relatively small; setting it to 100 suffices to yield desired specifications for the benchmarks we consider.

Benchmarks. Our benchmarks (shown in Tab. 7) are classified into four groups: (a) **Stack and Queue**: implementations of Okasaki's functional stack and queue. (b) **List**: a list library, including list manipulating functions such as: *delete, filter, merge, reverse*, etc.; a ListSet implementation of set interface represented as lists; and, various classic list sorting algorithms. (c) **Heap**: various classic heap implementations and two implementations, Heap$_1$ and Heap$_2$, searched

[9] Our implementation and benchmarks are provided via the URL https://github.com/rowangithub/DOrder.

Program	Loc	H	I	LT	T	Inferred Spec
List Stack	29	64	8	1s	1s	$\mathtt{O_{rd}}$
Lazy Queue	28	91	14	4s	4s	$\mathtt{O_{rd}}$
List Lib	133	306	54	7s	10s	$\mathtt{O_{rd}}$
List Set	51	96	50	11s	17s	$\mathtt{O_{rd}}$, Set
Quicksort	19	49	25	1s	5s	$\mathtt{O_{rd}}$, Sorted
Mergesort	30	32	11	1s	5s	$\mathtt{O_{rd}}$, Sorted
Insertionsort	12	22	8	1s	1s	$\mathtt{O_{rd}}$, Sorted
Selectionsort	22	32	11	1s	2s	$\mathtt{O_{rd}}$, Sorted
Heap$_1$	85	139	48	37s	133s	$\mathtt{O_{rd}}$, Min, Heap
Heap$_2$	77	70	23	5s	28s	$\mathtt{O_{rd}}$, Min, Heap
Heapsort	37	81	28	9s	29s	$\mathtt{O_{rd}}$, Sorted, Heap
Leftist Heap	43	106	32	12s	18s	$\mathtt{O_{rd}}$, Min, Heap
Skew Heap	32	71	25	16s	22s	$\mathtt{O_{rd}}$, Min, Heap
Splay Heap	58	98	44	9s	38s	$\mathtt{O_{rd}}$, Min, BST
Pairing Heap	42	49	20	1s	7s	$\mathtt{O_{rd}}$, Min, Heap
Binomial Heap	70	107	34	5s	26s	$\mathtt{O_{rd}}$, Min, Heap
Treap	107	95	17	20s	39s	$\mathtt{O_{rd}}$, BST
AVL Tree	176	127	39	27s	56s	$\mathtt{O_{rd}}$, BST
Splay Tree	127	110	56	45s	170s	$\mathtt{O_{rd}}$, BST
Braun Tree	75	111	42	19s	53s	$\mathtt{O_{rd}}$, BST
Redblack Tree	228	377	104	101s	391s	$\mathtt{O_{rd}}$, BST
OCaml Set	313	457	73	56s	134s	$\mathtt{O_{rd}}$, BST, Min, Set
Proposition	58	94	8	2s	5s	$\mathtt{O_{rd}}$
Randaccesslist	73	142	19	4s	7s	$\mathtt{O_{rd}}$

Table 7: Experimental results on inferring shape specifications: Loc describes program size, H is the number of atomic predicates in the hypothesis domain of all the functions in a data structure. I is the number of verified ordering specifications in terms of either input-output or shape-data relations. T is the total time taken (learning and verification), while LT is the time spent solely on learning (including the time spent in sampling). Inferred Spec summarizes the learnt and verified specifications by DORDER.

```
let rec merge h1 h2 =
 match h1, h2 with
  | (Leaf, h2) -> h2
  | (h1, Leaf) -> h1
  | (Node(k1, l1, r1), Node(k2,l2,r2)) ->
    if (k1<=k2) then Node(k1,(merge r1 h),l1)
                else Node(k2,(merge h1 r2),l2)
```

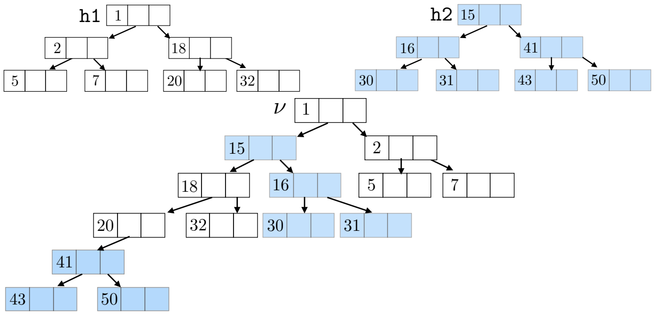


Figure 8: Skew Heap with input-output samples of `merge`.

from GitHub. (d) **Tree**: various implementations of realistic balanced tree data structures including Redblack trees with support for both insertion and deletion, a library to convert arbitrary Boolean formulae to NNF or CNF form (Proposition), a random access lists library based on trees (Randaccesslist), and the full implementation of OCaml's Set library.

Results. On these programs, DORDER inferred the following specifications[10]: (a) $\mathtt{O_{rd}}$: specifications expressed using ordering and containment predicates. For instance, the specification for a balanced tree insertion function ensures that *the output tree preserves the in-order of the input tree*. For a sorted heap merge function, DORDER discovers that *the parent-child relations of the input heap are preserved in the output-heap*. Similarly the $\mathtt{O_{rd}}$ property inferred for the Proposition benchmark ensures functional correctness: "*any logical relation* ($\wedge, \vee$) *between two Boolean variables in a given input Boolean formula is preserved in the output formula after a CNF conversion*", (b) `Set`: verifies that the structure implements a set interface, that is, *the set operations:* `union`, `diff` *and* `intersect` *are semantically correct* using the containment and ordering hypothesis domain. For example, the specification for the `diff` (t_1, t_2) function stipulates that `diff` returns a set whose elements must come from t_1 but must not be members of t_2. The following properties are obtained using the shape-data domain: (c) `Sorted`: the output list is sorted, (d) `Min`, the `findmin` function returns the smallest element of a data structure, (e) `Heap`, the output tree is heap-sorted, (f) `BST`, the output tree is a binary search tree.

Redblack tree is the most challenging benchmark in Tab. 7 given the complexity of the delete operation. The benchmark contains several complex balance functions that cooperate together to reestablish the balance property of the tree after a delete. The OCaml Set implementations also has a large code base, but the invariants it maintains are simpler. Note that most of the running time is spent in verification, and that the learning algorithm is efficient in comparison.

Case study: Skew Heap. A skew heap structure is a self-adjusting heap implemented as a binary tree. Many varieties of balanced trees are specifically designed to achieve efficiency by imposing tight balance constraints that must be maintained during updates. By relaxing such tight balance constraints, a skew heap provides better amortized running times. In particular, the left subtree of a skew heap is usually deeper than the right subtree, illustrated in Fig. 8.

Two conditions must be satisfied in a skew heap: (a) the general heap sorted order must be enforced (b) every operation (add, remove_min) on a skew heap must be done using a special skew heap merge. An implementation of the `merge` operation of skew heap is given in Fig. 8. This operation merges two input skew heaps `h1` and `h2` into one output skew heap ν.

[10] Our technical report [71] provides detailed case studies for several of these benchmarks with more complex specifications discovered.

DORDER inferred the following functional specifications for the `merge` function from the samples in Fig. 8, reflecting the functional behavior of `merge`:

(i) the output heap ν preserves the parent-child relations of `h1` and `h2`; *e.g.*, as shown in Fig. 8, 16 is a child of 15 in `h2`, and remains a child of 15 in ν, and

(ii) for any two nodes, one from `h1` and the other from `h2` (or vice-versa): they are either related by a left branch of the final tree ν ($\nu : u \swarrow v$), e.g. 18 and 41 belong to `h1` and `h2` respectively and they are related according to left branches in the output heap ν; or they are in different sub-branches ($\nu : u \circlearrowleft v$), e.g. 15 (in `h2`) and 2 (in `h1`) are located in different sub-branches of ν. Importantly, they are *not related over the right branch* ($\nu : u \searrow v$).

The inferred and verified (partial) specification formalizes (i) and (ii):

$$\begin{array}{l}
\texttt{merge} : h1 : \texttt{'a tree} \rightarrow h2 : \texttt{'a tree} \rightarrow \Big\{ \nu : \texttt{'a tree} \mid \\
\quad \big(\forall u,\ \nu \dashrightarrow u \iff (h1 \dashrightarrow u \lor h2 \dashrightarrow u)\big) \land \\
\left(\forall u\ v,\ \nu : u \swarrow v \Rightarrow \begin{pmatrix} h1 : u \swarrow v \lor h1 : u \searrow v \lor \\ h2 : u \swarrow v \lor h2 : u \searrow v \lor \\ (h1 \dashrightarrow u \land h2 \dashrightarrow v) \lor \\ (h2 \dashrightarrow u \land h1 \dashrightarrow v) \end{pmatrix}\right) \land \\
\left(\forall u\ v,\ \nu : u \searrow v \Rightarrow \begin{pmatrix} h1 : u \swarrow v \lor h1 : u \searrow v \lor \\ h2 : u \swarrow v \lor h2 : u \searrow v \end{pmatrix}\right) \land \\
\left(\forall u\ v,\ \nu : u \circlearrowleft v \Rightarrow \begin{pmatrix} h1 : u \circlearrowleft v \lor h1 : v \circlearrowleft u \lor \\ h2 : u \circlearrowleft v \lor h2 : v \circlearrowleft u \lor \\ (h1 \dashrightarrow u \land h2 \dashrightarrow v) \lor \\ (h2 \dashrightarrow u \land h1 \dashrightarrow v) \end{pmatrix}\right) \Big\}
\end{array}$$

The specification reflectes the fact that elements from `h1` and those from `h2` are only merged into the left subtree, demonstrating the intuition that the left subtree is more complex than the right subtree.

Numeric Data Structure Properties. As described earlier, DORDER can also infer measure-based specifications. To assess its effectiveness in this space, we considered benchmarks evaluated in LIQUIDTYPES [30] and compare the specifications discovered by DORDER with those inferred by [30]. Our benchmark suites additionally include `Bdd`, a binary decision diagram library, and `Vec`, a dynamic functional array library.

To evaluate the quality of synthesized specifications, we use them to verify known data structure properties such as: `Sz` or `Ht`, functions used to alter the number of elements in a list or tree, or the height of trees; `Bal`, a property on trees that asserts they are recursively balanced (the definitions of balance in different tree implementations varies); `VOrder`, a binary decision diagram (BDD) maintains a variable order property; `Len`, the access indices of vector operations are bounded by vector length.

DORDER inferred and verified a number of measure based specifications in these programs, reflected in column I in Tab. 8, obviating the need for user-supplied invariants. The LIQUIDTYPES checker in contrast relies on the user to manually annotate function specifications in order to help verify these numeric properties.

Program	Loc	I	LT	T	Properties	LIQTYAN
AVL Tree	99	32	4s	14s	`Bal,Sz,Ht`	9
Braun Tree	49	13	2s	4s	`Bal,Sz`	3
Redblack	201	27	3s	10s	`Bal,Ht`	9
OCaml Set	110	24	5s	10s	`Bal,Ht`	10
Randaccesslist	102	15	1s	2s	`Sz, Bal`	6
Bdd lib	144	22	2s	8s	`VOrder`	14
Vec lib	211	56	46s	59s	`Bal,Len,Ht`	39

Table 8: Experimental results on inferring numeric specifications: the column interpretation is identical to Tab. 7. Properties summarizes the properties that are verified by DORDER. LIQTYAN is the number of annotations required by LIQUIDTYPES in order to prove the properties, which are now inferred by DORDER.

Limitations. The expressivity of our approach is limited by the need to ensure decidability. We cannot express and reason about ordering specifications that require interdependent shape and arithmetic constraints over data structure *indices*. For example, given a function `f (xs, low, high)` that returns only the set of elements from index `low` to `high` of a list `xs`, our technique will not be able to find a valid specification that discovers that the returned elements of `f` precisely correspond to those indexed from `low` to `high` in the input list; this is because of limitations in the theories supported by the underlying BSR solver.

7. Related Work and Conclusions

Compared to earlier sampling-based approaches [49, 60, 65, 66] which learn invariants using existing abstract interpretation transformers, our primary focus is a new specification inference technique inspired by recent advances in data-driven program analysis. These data-driven approaches can be classified into two broad categories: (1) Tools such as Daikon [12] and [18, 21, 42, 53, 69] infer invariants by summarizing properties from test data, but the structure of the constructed invariants is limited to a bounded number of disjunctions, making them unlikely to discover patterns between relations like *in-order* or *forward-order*, because it is not clear how syntax-derived templates could capture the semantics of ordering relations implicit in the construction of data structures; (2) Other tools learn unrestricted invariants but either require user-annotated post-conditions [15, 16, 36, 51, 54, 70] (in order to rule out program states not seen in normal executions) or non-commutativity conditions [17] to drive the collection of "bad samples". The quality of synthesized invariants in these systems is limited by the precision and availability of such conditions. Morevoer, these approaches learn invariants to prove given assertions, which must separate all "good" from all "bad" samples. They are not suitable for learning input-output specifications, because (1) learning fails if a sample cannot be separated by any classifier, even though a good specification might exist (e.g. Tab. 4); and (2) they only find approximate classifiers, not necessarily the strongest one needed to prove assertions. We use classification-techniques in a novel way to discover the strongest specification in a hypothesis-domain (Theorem 2). Thus, DORDER is the first annotation-free learning technique that infers high-quality (c.f. strongest) inductive shape specifications comprising unrestricted disjunctions, that can be effectively applied on realistic and complex functional data structures.

Our technique is closely related to [24, 25], which also use BSR logic to prove functional specifications for linked list structures, by relating the order of list elements and defining ordering properties on the whole memory. In contrast, our technique infers fine-grained and inductive shape predicates over concrete data structure instances. Shape specifications in terms of user-defined ordering relations are also considered in [28]. Because these systems are not equipped with an inference mechanism, they require programmers to manually write down potentially complicated and subtle program specifications. The idea of using relations to capture inductive properties of data structure programs has also been explored in [6, 27, 34, 35, 38, 67]. These non-learning based techniques differ substantially from ours, owing to the nature of pointer manipulations in their imperative program model.

There exists a number of deductive verification tools for data structure programs, which support reasoning of recursive definitions over the set of elements in the heaplets of a data structure. These systems require modular contracts to be supplied with the developed code, using pre/post-conditions, loop invariants and even proof lemmas [7, 9, 30, 31, 33, 37, 40, 43, 44, 46–48, 58, 59, 63, 68]. Our approach complements these tools with an inference procedure that can learn specifications for fully automatic data structure verification.

Following the Houdini approach [13], the LIQUID TYPES system [30, 61, 62] blends type inference for data structures with predicate abstraction, and infers refinement types from conjunctions of programmer-annotated predicates. To infer more expressive invariants, [56] infers quantified invariants for arrays and lists, limited to programmer-provided templates. To get rid of templates, automatic procedures, which can infer the Boolean structure of candidate invariants, have been proposed for linked list programs [14, 15, 26, 29]. They either lack a notion of *progress* (c.f. Sec. 4.3) [14] or require the programmer to provide nontrivial post-conditions [15, 26, 29]. Unlike other static synthesis techniques that perform shape analyses on the source code [3, 5, 11, 20, 32, 64], DORDER discovers shape specifications entirely from tests.

Conclusion. This paper presents a new specification inference framework that integrates testing with a sound type-based verification system to automatically synthesize and verify shape specifications for arbitrary inductive data structure programs. Given an arbitrary user defined inductive data structure program, our tool DORDER applies a systematic analysis on the program's data type definitions, and extracts atomic predicates stating general ordering properties about data structure values with respect to data structure shapes. These predicates are then fed to an expressive learning algorithm, which postulates potentially complex shape specifications satisfying input-output behaviors of data structure functions. The learning algorithm interacts with the verification system to ensure discovery of the strongest inductive invariant in the solution space.

Our experiments demonstrate that the approach is effective and efficient over a large class of real-world data structure programs. Using just a few number of randomly generated tests, DORDER can synthesize sophisticated and high quality shape specifications for versatile data structure manipulating functions with reasonable cost.

Acknowledgement

We thank our shepherd Martin Vechev, Gowtham Kaki, Tiark Rompf, Aditya Nori, and the anonymous reviewers for their useful comments and suggestions. This work was supported in part by the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

References

[1] A. Albarghouthi and K. L. McMillan. Beautiful Interpolants. In *CAV*, 2013. doi: 10.1007/978-3-642-39799-8_22.

[2] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, pages 1–25, 2015. doi: 10.3233/978-1-61499-495-4-1.

[3] J. Berdine, B. Cook, and S. Ishtiaq. SLAYER: Memory Safety for Systems-level Code. In *CAV*, 2011. doi: 10.1007/978-3-642-22110-1_15.

[4] M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012. doi: 10.1007/s10601-012-9123-1.

[5] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-abduction. In *POPL*, 2009. doi: 10.1145/1480881.1480917.

[6] B.-Y. E. Chang and X. Rival. Relational Inductive Shape Analysis. In *POPL*, 2008. doi: 10.1145/1328438.1328469.

[7] A. Chlipala. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In *PLDI*, 2011. doi: 10.1145/1993498.1993526.

[8] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*, 2000. doi: 10.1145/351240.351266.

[9] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs*, 2009. doi: 10.1007/978-3-642-03359-9_2.

[10] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008. doi: 10.1007/978-3-540-78800-3_24.

[11] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *PLDI*, 2011. doi: 10.1145/1993498.1993565.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. doi: 10.1016/j.scico.2007.01.015.

[13] C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME*, 2001. doi: 10.1007/3-540-45251-6_29.

[14] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning Universally Quantified Invariants of Linear Data Structures. In *CAV*, 2013. doi: 10.1007/978-3-642-39799-8_57.

[15] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A Robust Learning Framework for learning Invariants. In *CAV*, 2014. doi: 10.1007/978-3-319-08867-9_5.

[16] P. Garg, P. Madhusudan, D. Neider, and D. Roth. Learning Invariants Using Decision Trees and Implication Counterexamples. In *POPL*, 2016. doi: 10.1145/2837614.2837664.

[17] T. Gehr, D. Dimitrov, and M. Vechev. Learning Commutativity Specifications. In *CAV*, 2015. doi: 10.1007/978-3-319-21690-4_18.

[18] P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *PLDI*, 2012. doi: 10.1145/2254064.2254116.

[19] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, 1997. doi: 10.1007/3-540-63166-6_10.

[20] B. Guo, N. Vachharajani, and D. I. August. Shape Analysis with Inductive Recursion Synthesis. In *PLDI*, 2007. doi: 10.1145/1250734.1250764.

[21] A. Gupta, R. Majumdar, and A. Rybalchenko. From Tests to Proofs. In *TACAS*, 2009. doi: 10.1007/978-3-642-00768-2_24.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In *POPL*, 2004. doi: 10.1145/964001.964021.

[23] C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On Local Reasoning in Verification. In *TACAS*, 2008. doi: 10.1007/978-3-540-78800-3_19.

[24] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-Propositional Reasoning About Reachability in Linked Data Structures. In *CAV*, 2013. doi: 10.1007/978-3-642-39799-8_53.

[25] S. Itzhaky, A. Banerjee, N. Immerman, O. Lahav, A. Nanevski, and M. Sagiv. Modular Reasoning About Heap Paths via Effectively Propositional Formulas. In *POPL*, 2014. doi: 10.1145/2535838.2535854.

[26] S. Itzhaky, N. Bjørner, T. W. Reps, M. Sagiv, and A. V. Thakur. Property-Directed Shape Analysis. In *CAV*, 2014. doi: 10.1007/978-3-319-08867-9_3.

[27] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A Relational Approach to Interprocedural Shape Analysis. *ACM Trans. Program. Lang. Syst.*, 32:5:1–5:52, 2010. doi: 10.1145/1667048.1667050.

[28] G. Kaki and S. Jagannathan. A Relational Framework for Higher-order Shape Analysis. In *ICFP*, 2014. doi: 10.1145/2628136.2628159.

[29] A. Karbyshev, N. Bjorner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-Directed Inference of Universal Invariants or Proving Their Absence. In *CAV*, 2015. doi: 10.1007/978-3-319-21690-4_40.

[30] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based Data Structure Verification. In *PLDI*, 2009. doi: 10.1145/1542476.1542510.

[31] S. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In *POPL*, 2008. doi: 10.1145/1328438.1328461.

[32] Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape Analysis via Second-Order Bi-Abduction. In *CAV*, 2014. doi: 10.1007/978-3-319-08867-9_4.

[33] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR*, 2010. doi: 10.1007/978-3-642-17511-4_20.

[34] T. Lev-Ami and S. Sagiv. TVLA: A System for Implementing Static Analyses. In *SAS*, 2000. doi: 10.1007/978-3-540-45099-3_15.

[35] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating Reachability Using First-order Logic with Applications to Verification of Linked Data Structures. In *CADE*, 2005. doi: 10.1007/11532231_8.

[36] A. Loginov, T. Reps, and M. Sagiv. Abstraction Refinement via Inductive Learning. In *CAV*, 2005. doi: 10.1007/11513988_50.

[37] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive Proofs for Inductive Tree Data-structures. In *POPL*, 2012. doi: 10.1145/2103656.2103673.

[38] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-linked Lists. In *VMCAI*, 2005. doi: 10.1007/978-3-540-30579-8_13.

[39] E. J. McCluskey. Minimization of Boolean Functions. *Bell system technical Journal*, 35(6):1417–1444, 1956.

[40] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *VMCAI*, 2007. doi: 10.1007/978-3-540-69738-1_18.

[41] P. C. Nguyen and D. V. Horn. Relatively Complete Counterexamples for Higher-Order Programs. In *PLDI*, 2015. doi: 10.1145/2737924.2737971.

[42] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using Dynamic Analysis to Generate Disjunctive Invariants. In *ICSE*, 2014. doi: 10.1145/2568225.2568275.

[43] E. Pek, X. Qiu, and P. Madhusudan. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *PLDI*, 2014. doi: 10.1145/2594291.2594325.

[44] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software Verification with VeriFast: Industrial Case Studies. *Sci. Comput. Program.*, 82:77–97, 2014. doi: 10.1016/j.scico.2013.01.006.

[45] R. Piskac, L. Moura, and N. Bjørner. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. *J. Autom. Reason.*, 44:401–424, 2010. doi: 10.1007/s10817-009-9161-6.

[46] R. Piskac, T. Wies, and D. Zufferey. GRASShopper - Complete Heap Verification with Mixed Specifications. In *TACAS*, 2014. doi: 10.1007/978-3-642-54862-8_9.

[47] R. Piskac, T. Wies, and D. Zufferey. Automating Separation Logic with Trees and Data. In *CAV*, 2014. doi: 10.1007/978-3-319-08867-9_47.

[48] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural Proofs for Structure, Data, and Separation. In *PLDI*, 2013. doi: 10.1145/2462156.2462169.

[49] T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, 2004. doi: 10.1007/978-3-540-24622-0_21.

[50] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid Types. In *PLDI*, 2008. doi: 10.1145/1375581.1375602.

[51] R. Sharma and A. Aiken. From Invariant Checking to Invariant Inference Using Randomized Search. In *CAV*, 2014. doi: 10.1007/978-3-319-08867-9_6.

[52] R. Sharma, A. V. Nori, and A. Aiken. Interpolants As Classifiers. In *CAV*, 2012. doi: 10.1007/978-3-642-31424-7_11.

[53] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A Data Driven Approach for Algebraic Loop Invariants. In *ESOP*, 2013. doi: 10.1007/978-3-642-37036-6_31.

[54] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. Nori. Verification as Learning Geometric Concepts. In *SAS*, 2013. doi: 10.1007/978-3-642-38856-9_21.

[55] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, 2008.

[56] S. Srivastava and S. Gulwani. Program Verification Using Templates over Predicate Abstraction. In *PLDI*, 2009. doi: 10.1145/1542476.1542501.

[57] M. Stojadinović and F. Marić. meSAT: multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014. doi: 10.1007/s10601-014-9165-7.

[58] P. Suter, M. Dotta, and V. Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *POPL*, 2010. doi: 10.1145/1706299.1706325.

[59] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability Modulo Recursive Programs. In *SAS*, 2011. doi: 10.1007/978-3-642-23702-7_23.

[60] A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and All That: Automating Abstract Interpretation. *Electron. Notes Theor. Comput. Sci.*, 311:15–32, 2015. doi: 10.1016/j.entcs.2015.02.003.

[61] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *ICFP*, 2014. doi: 10.1145/2628136.2628161.

[62] N. Vazou, A. Bakst, and R. Jhala. Bounded Refinement Types. In *ICFP*, 2015. doi: 10.1145/2784731.2784745.

[63] H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *POPL*, 1999. doi: 10.1145/292540.292560.

[64] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable Shape Analysis for Systems Code. In *CAV*, 2008. doi: 10.1007/978-3-540-70545-1_36.

[65] G. Yorsh, T. Reps, and M. Sagiv. Symbolically Computing Most-precise Abstract Operations for Shape Analysis. In *TACAS*, 2004. doi: 10.1007/978-3-540-24730-2_39.

[66] G. Yorsh, T. Ball, and M. Sagiv. Testing, Abstraction, Theorem Proving: Better Together! In *ISSTA*, 2006. doi: 10.1145/1146238.1146255.

[67] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A Logic of Reachable Patterns in Linked Data-structures. In *FOSSACS*, 2006. doi: 10.1007/11690634_7.

[68] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *PLDI*, 2008. doi: 10.1145/1375581.1375624.

[69] H. Zhu, A. Nori, and S. Jagannathan. Dependent Array Type Inference from Tests. In *VMCAI*, 2014. doi: 10.1007/978-3-662-46081-8_23.

[70] H. Zhu, A. Nori, and S. Jagannathan. Learning Refinement Types. In *ICFP*, 2015. doi: 10.1145/2784731.2784766.

[71] H. Zhu, G. Petri, and S. Jagannathan. Learning refinement types. Technical report, Purdue Univsersity, 2015. https://www.cs.purdue.edu/homes/zhu103/pubs/draft.pdf.