8-2016

Learning Program Specifications from Sample Runs

He Zhu
Purdue University

Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By He Zhu

Entitled
Learning Program Specifications from Sample Runs

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Suresh Jagannathan
Chair

Xiangyu Zhang

Tiark Rompf

Dongyan Xu

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Suresh Jagannathan

Approved by: Sunil Prabhakar / William J. Gorman 07/21/2016

Head of the Departmental Graduate Program Date

LEARNING PROGRAM SPECIFICATIONS FROM SAMPLE RUNS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

He Zhu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2016

Purdue University

West Lafayette, Indiana

For Mom and Dad.

ACKNOWLEDGMENTS

I owe a huge thanks to my advisor, Suresh Jagannathan, for all the generous and patient guidance and support as well as the insight and inspiration he has provided over the years.

Thanks to my committee members Aditya Nori, Xiangyu Zhang, Tiark Rompf and Dongyan Xu for showing a keen interest in the work and firming up my efforts with their questions and insights.

I felt very fortunate to have spent the last few years with the incredibly talented and driven Purdue Secure Software Systems Group. Thanks to all of you for hearing out my half-baked ideas, reading my half-written drafts, and sitting through my half-cocked talks. Particular thanks to Gustavo Petri, who was always ready to dole out advice or lend an ear as needed.

I have been especially lucky to have Nicholas Kidd, Armand Navabi, K.C. Sivaramakrishnan, Gowtham Kaki, Xuankang Lin, Kia Rahmani and Suyash Gupta as collaborators and friends. Trying to keep up with them has always pushed me to go further and faster.

I am lucky to have made a large number of friends at Purdue who have changed my life for the better in countless ways. I will not attempt an exhaustive list, for fear of missing someone; you know who you are. Thanks for everything!

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Zhu, He PhD, Purdue University, August 2016. Learning Program Specifications from Sample Runs. Major Professor: Suresh Jagannathan.

With science fiction of yore being reality recently with self-driving cars, wearable computers and autonomous robots, software reliability is growing increasingly important. A critical pre-requisite to ensure the software that controls such systems is correct is the availability of precise specifications that describe a program's intended behaviors. Generating these specifications manually is a challenging, often unsuccessful, exercise; unfortunately, existing static analysis techniques often produce poor quality specifications that are ineffective in aiding program verification tasks.

In this dissertation, we present a recent line of work on automated synthesis of specifications that overcome many of the deficiencies that plague existing specification inference methods. Our main contribution is a formulation of the problem as a sample driven one, in which specifications, represented as terms in a decidable refinement type representation, are discovered from observing a program's sample runs in terms of either program execution paths or input-output values, and automatically verified through the use of expressive refinement type systems.

Our approach is realized as a series of inductive synthesis frameworks, which use various logic-based or classification-based learning algorithms to provide sound and precise machine-checked specifications. Experimental results indicate that the learning algorithms are both efficient and effective, capable of automatically producing sophisticated specifications in nontrivial hypothesis domains over a range of complex real-world programs, going well beyond the capabilities of existing solutions.

1 INTRODUCTION

1.1 The Problem

One of the great things about type systems is that they allow one to enforce a variety of invariants at compile time, thereby nipping in the bud a large swathe of run-time errors. However, well-typed programs do go wrong in a variety of ways: a function that computes the average of a list can encounter a crash if the programmer fails to deal with the corner case that the list is empty; on web services built using a high-performance string processing Haskell library *text*, a cunning adversary can use well-crafted inputs to read extra bytes beyond the boundary of a legal *text*.

To avoid such calamities at run-time, *refinement type systems* [1–3] enrich simple type systems with predicates that precisely describe the sets of valid inputs and outputs of functions. Refinement type predicates can be thought as Boolean-valued expressions that constrain the set of values described by the types and specify *invariants* of the underlying values. The predicates are carefully chosen from expressive yet decidable logics for which there exists fast decision procedures (*e.g.*, SMT solvers). For example, $\texttt{type Nat} = \{\nu : \texttt{int} \mid \nu \geq \texttt{0}\}$ describes the set of `int` values that are non-negative and is a refined type of `int` (the value variable ν denotes the set of valid inhabitants of the refinement type). Consider the simple program shown in Fig. 1.1. Intuitive program invariants for `max` and `f` can be expressed in terms of the following refinement types:

$$
\begin{aligned}
&\texttt{max} :: (\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \texttt{z} : \texttt{int} \rightarrow \\
&\quad \texttt{m} : (\texttt{m}_0 : \texttt{int} \rightarrow \texttt{m}_1 : \texttt{int} \rightarrow \{\texttt{int} \mid \nu \geq \texttt{m}_0 \wedge \nu \geq \texttt{m}_1\}) \\
&\quad \rightarrow \{\texttt{int} \mid \nu \geq \texttt{x} \wedge \nu \geq \texttt{y} \wedge \nu \geq \texttt{z}\}) \\
&\texttt{f} :: (\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \{\texttt{int} \mid \nu \geq \texttt{x} \wedge \nu \geq \texttt{y}\})
\end{aligned}
$$

```
let max x y z m = m (m x y) z      let main x y z =
let f x y =                          let result = max x y z f in
  if x >= y then x else y            assert (f x result = result)
```

Figure 1.1.: A simple higher-order program.

The types specify that both `max` and `f` produce an integer that is no less than the value of their parameters. However, these types are not sufficient to prove the assertion in `main`; to do so, requires specifying more precise invariants.

The above example shows that refinement types offer a promising way to express rich invariants that can go beyond the capabilities of traditional type systems [4] or control-flow analyses [5], albeit at the price of automatic inference. Recently, there has been substantial progress in reducing this annotation burden [6–14] using techniques adopted from model-checking and verification of first-order imperative programs [15,16]. These solutions, however, either (a) involve a complex reformulation of the intuitions underlying invariant detection and verification from a first-order context to a higher-order one [11, 13], making it difficult to directly reuse existing tools and methodologies, (b) infer refinement types by solving a set of constraints collected by a whole-program analysis [6, 7], additionally seeded with programmer-specified qualifiers, that can impact compositionality and usability, or (c) entail a non-trivial translation to a first-order setting [9], making it more complicated to relate the inferences deduced in the translated first-order representation back to the original higher-order source when there is a failure.

1.2 My Thesis

This dissertation addressed the above problems. The first key component of our solution is an efficient symbolic execution over higher-order functional programs. Refinement type systems prove the absence of errors but may give false alarms. We

developed a symbolic execution tool for higher-order programs that aims to discover concrete and real counterexamples to faulty programs. The tool can be viewed as a complement to refinement type based verification. The goal is to synthesize test inputs that lead to an assertion violation in higher-order programs. To this end, our analysis pushes up the negation of assertions backwards. Our symbolic execution is extended to deal with unknown functions which are functional arguments or returns in a higher-order function. The key idea is to encode unknown functions into uninterpreted functions. As a result, we can generate constraints over the input/output behaviors of unknown functions in higher-order functions. The symbolic analysis for the actual function represented by the unknown function is deferred until it becomes known at call-sites of higher-order functions. A query for satisfiable solution to the result of this analysis via SMT solvers (with supports for background theories of the logic used to encode assertions) returns desired inputs leading to errors.

The symbolic executer has tremendous uses and proves effective in our subsequent research in proof-directed refinement type inference, which is capable of searching specifications to prove user-provided program assertions. Specifically, we use *logic*-based learning algorithms, *e.g.* weakest precondition generation, to glean important and necessary information, *e.g.* path conditions, from symbolic execution paths, filtering out spurious assertion violations.

In addition to *logic*-based learning algorithms, the second key component of our solution focuses on sound and relatively complete specification synthesis procedures that can automatically learn sophisticated program specifications, using *classification*-based learning algorithms. The idea is based on the well-understood intuition that useful, but difficult to infer, program properties can often be observed from concrete program states generated by tests; these properties act as *likely* invariants, which if used to refine simple types, can have their validity checked by an underlying refinement type checker. We present efficient *classification*-based learning algorithms to automatically discover and verify expressive function specifications from sample

program runs, which are classified into different groups with respect to certain program properties (*e.g.* program assertions).

An immediate question is *can we ensure discovered specifications from sample program runs not only hold in observed samples but also generalize well in unobserved runs?* To address this concern, the design of our learning algorithm follows the well-known Occam's razor principle. It flavors a simple program specification to a complex one.

1.3 Contributions

In this section, we provide a brief overview of the contributions made by this dissertation.

1.3.1 Compositional and Lightweight Refinement Type Inference

We consider the problem of inferring expressive safety properties of higher-order functional programs using first-order decision procedures. Our approach encodes higher-order features into first-order logic formula whose solution can be derived using a lightweight counterexample guided refinement loop. To do so, we extract initial verification conditions from refinement type checking rules derived by a syntactic scan of the program. Subsequent type-checking and type-refinement phases infer and propagate specifications of higher order functions, which are treated as *uninterpreted* first-order constructs, via subtyping chains and counterexample paths over which our symbolic execution procedure is invoked by a *logic*-based learning algorithm.

Our technique provides several benefits not found in existing systems: (1) it enables *compositional* verification and inference of useful safety properties for functional programs; (2) additionally provides counterexamples that serve as witnesses of unsound assertions: (3) does not entail a complex translation or encoding of the original source program into a first-order representation; and, (4) most importantly, profitably employs the large body of existing work on verification of first-order imperative pro-

grams to enable efficient analysis of higher-order ones. We have implemented the technique as part of the MLton SML compiler toolchain, where it has shown to be effective in discovering useful invariants with low annotation burden.

1.3.2 Learning Based Refinement Type Inference

The above technique learns a program specification solely from a refinement type system. Its expressivity and ability are inherited from that of static analyses. We propose the integration of a random test generation system (capable of discovering program bugs) and a refinement type system (capable of expressing and verifying program invariants), for higher-order functional programs, using a novel lightweight learning algorithm as an effective intermediary between the two.

Applying this intuition to yield an automatic verification strategy for higher-order programs is challenging, however. To overcome the difficulty of translating the output of test cases to a dependent type, we employ *classification*-based learning algorithms that classify positive samples collected from test runs and negative samples generated from our symbolic execution analysis. The key insight is that the result of the symbolic analysis provides an assertion violation condition that must be respected by any candidate program specification. The structure of these samples enables the construction of a likely invariant.

Failure to type check the invariant results in the generation of new tests designed to explore execution paths not previously encountered to strengthen inferred types, and provide additional inputs for classification. Notably, this iterative testing-learning-checking framework does not sacrifice precision, and is capable of inferring fine-grained context-sensitive invariants.

We describe an implementation of our technique for a variety of benchmarks written in ML, and demonstrate its effectiveness in inferring and proving useful invariants for programs that express complex higher-order control and dataflow that confound existing static verification and inference tools.

1.3.3 Automatically Learning Shape Specifications

Understanding, discovering, and proving useful *invariant-based specifications* of sophisticated data structure functions are central problems in program analysis and verification. A particularly challenging exercise for shape analyses, and the focus of this dissertation, involves reasoning about sophisticated ordering specifications that relate the shape of a data structure (*e.g.*, a binary tree data structure) with the values contained therein (*e.g.*, the *in-order* relation of the elements of a binary tree).

Given that interesting properties of inductive data structures are typically related to the way in which constructors are composed, our approach extracts potential shape predicates based on the definition of constructors of arbitrary user-defined inductive data types, that state general ordering properties about the elements contained in a data structure with respect to its shape, and combines these predicates within an expressive first-order specification language using a lightweight data-driven learning procedure.

For example, consider a tree data structure. Because a tree element u in a binary tree t has two subtrees, we obtain a tree-parent-left-child atomic predicate $\texttt{treeleft}\ (t, u, v)$ relating u with another tree element v in the left subtree of u. The synthesis procedure simultaneously outputs the inductive definition of this predicate based on the inductive structure of t. Similarly, we can obtain tree-parent-right-child atomic predicate $\texttt{treeright}\ (t, u, v)$ and tree-left-child-right-child atomic predicate $\texttt{treeleftright}\ (t, u, v)$ which predicates that u comes from a left subtree and v comes from a right subtree of tree element contained in t. An in-ordering relation between two elements u and v of t, notated by $\mathsf{inorder}\ (t, u, v)$, is trivially $\forall u, v.\ \texttt{treeleft}\ (t, v, u) \vee \texttt{treeright}\ (t, u, v) \vee \texttt{treeleftright}\ (t, u, v)$.

Suppose that a balanced binary search tree `insert` function takes a tree t as input and outputs a tree t'. The specification, encoded as a refinement type,

$$\texttt{insert} :: (t : \texttt{tree} \rightarrow \{t' : \texttt{tree} \mid \forall u, v.\ \mathsf{inorder}\ (t) \Rightarrow \mathsf{inorder}\ (t')\})$$

ensures that the output tree t' preserves the in-order of the input tree t. We developed a refinement type system to verify specifications of such kind, which unfolds the inductive definitions of the synthesized atomic predicates when necessary. Our type system is decidable because we can encode subtype checking in our system using decidable effectively propositional logic (with first-order axiomatizations of transitive closures [17, 18] to bound the shape of list or tree like data structures).

We present a novel automated procedure, called DOrder, for discovering expressive shape specifications for sophisticated functional data structures. The heart of DOrder is a relational learning algorithm that can effectively search propositional relations over a hypothesis domain Ω of atomic predicates. The hypothesis domain Ω defines the solution space for the learning algorithm to draw candidate specifications. For example, if Ω is chosen as the set of ordering atomic predicates synthesized for data structure programs, the solution space contains ordering specifications for data structure functions, composed of predicates from Ω. We gave a concrete instantiation of DOrder, which is a sound, relatively complete and scalable learning algorithm to synthesize input-output specifications for recursive functions. To the best of our knowledge, this is the first learning algorithm to automatically synthesize expressive function specifications from test data, that operates without assuming any predefined templates, assertions or post-conditions in program sources, yet which is nonetheless able to learn the strongest inductive invariant in the solution space from which specifications are drawn.

Notably, this technique requires *no* programmer annotations, and is equipped with a type based decision procedure to verify the correctness of discovered specifications. If verification succeeds, we ensure that synthesized specifications correspond to the strongest inductive invariant in the solution space; otherwise, we ensure that there exists a test input to the function f which yields a concrete input-output sample that invalidates the candidate specifications. In fact, we can reconstruct such a test input from verification failures. In turn, running the learning algorithm again using the new program samples from the new input, necessarily produces a more refined

specification. This strategy, which is implemented via a CEGIS (counterexample guided inductive synthesis) loop, ensures that *we can construct a finite number of test cases to guarantee convergence in the presumed solution space.*

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. Concretely, we were able to use DOrder to synthesize useful data structure specifications for practical linked list programs, priority heap data structures (*e.g.*, Skew and Binomial heaps) and balanced binary search tree data structures (*e.g.*, AVL, Redblack and Splay trees). From the programmer's perspective, the approach is lightweight and requires no custom annotation to get started. The prototype also has the ability to direct test generations, capable of searching counterexamples. Our experiments demonstrated that it is possible to construct a tool that can automatically guarantee correctness of programs and simultaneously ease the understanding of faulty programs, speeding up the development of reliable software.

1.4 Road Map

The rest of the dissertation is organized as follows. Chapter 2 describes a basic refinement type system with a powerful verification condition generation algorithm, which serves as the verification vehicle for the upcoming three specification synthesis techniques. Chapter 3 presents Popeye, a compositional and lightweight refinement type inference engine for ML. Chapter 4 extends Popeye, providing a classification-based algorithm for learning refinement types. Chapter 5 presents DOrder, a learning system that can synthesize the strongest specifications from a hypothesis domain even for annotation-free programs. For exposition purposes, this chapter focuses on inductive user-defined data structure programs; but the core technique can be generalized to any program domains. Related work is presented at the end of each chapter. Concluding remarks and future direction of this research are given in Chapter 6.

2 TYPE BASED PROGRAM ANALYSIS AND VERIFICATION

This chapter illustrates an expressive refinement type system, which serves as a sound verification vehicle for the whole dissertation. Our system combines *Hindley-Milner* type inference with *Predicate Abstraction* to automatically infer and verify refinement types to prove a variety of safety properties. The system allows programmers to specify function specifications and uses static verification to validate them, thereby eliminating expensive run-time checks.

2.1 Refinement Type Checking

Language.

For exposition purposes, we formalize our ideas in the context of an idealized language: a call-by-value variant of the λ-calculus, shown in Fig. 2.1, with support for refinement types.

Typically, x and y are bound to variables; f is bound to function symbol. By convention, d represents a variable with an inductive data type. (We cover inductive data structure program analysis and verification mainly in Sec. 4.5 and Chapter 6.) 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`). The special variable ν is used to denote the value of a term in its corresponding type refinement predicate. Primitive operators are encoded with the metaoperator $\oplus$ (where unary operators ignore the second argument).

We additionally provide the syntactic sugar form `let rec` defined in terms of `fix` in the usual way: $\texttt{let rec}\, f\ \vec{x} = e\ \texttt{in}\ e'$ is converted from $\texttt{let}\, f = \texttt{fix}\,(\texttt{fun}\, f \rightarrow \lambda\vec{x}.e)\,\texttt{in}\, e'$. The length of $\vec{x}$ is called the *arity* of f.

To simplify the technical development, we assume our language is A-normalized, ensuring every abstraction and function argument is associated with a program variable. For example, in function applications $f\ \vec{y}$, we ensure every function and its arguments are associated with a program variable. When the length of $\vec{y}$ is smaller than the arity of f, $f\ \vec{y}$ is a partial application. For any expression of the form $\texttt{let}\, f = \lambda\vec{x}.e\, \texttt{in}\, e'$, we say that the function f is *known* in the expression e'. Functional arguments and return values of higher-order functions are *unknown* (e.g., in $\texttt{let}\ g = f\ v\, \texttt{in}\, e'$ if the symbol g is used as a function in e', it is an unknown function in e'; in $\lambda x.\ e'$ if x is used as a function in e', x is an unknown function in e').

An assert statement of the form "$\texttt{assert}\ \ v$" evaluates expression v and returns the special value `fail` if v is `false`. Program executions resulting in assertion failures immediately terminate.

Finally, we allow polymorphic type abstraction and type instantiation.

The language supports a small set of base types (B), monotypes (τ). We allow polymorphic types via type variables that are universally quantified. Refinement types (P) include refinement base (data) types and refinement function types.

In a refinement type system, a *base type* (*data type*) such as `int` (data structure) is specified into a *refinement base* (*data*) *type* written $\{\texttt{int}|\, \psi\}$ where ψ (a *type refinement*) is a Boolean-valued expression constraining the value of the term defined by the type. For example, $\{\texttt{int}|\, \nu > 0\}$ defines the type of positive integers where the special variable ν denotes the value of the term. Refinement types naturally generalize to function types. A *refinement function type*, written $\{x : P_x \to P\}$, constrains the argument x by the refinement type P_x, and produces a result whose type is specified by P. In this dissertation, ψ is chosen from a specification space parameterized by a hypothesis domain Ω of atomic predates and closed by standard propositional logic connectives. For example, Ω can be set to standard abstract interpretation domains including octagon domain, polyhedra domains and so on and so forth.

To encode program specifications into refinement types, we present the general `specType` function below. Assume a specification ψ is for a function f. The `specType`

$$x, y, d, f, \nu \in \texttt{Var} \qquad c \in \texttt{Constant} \qquad \texttt{'a} \in \texttt{TyVar}$$

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

$$\psi \in \text{Specification Space}(\Omega)$$

$$\begin{array}{rcl}
P \in \texttt{RType} & ::= & \{\nu : B \mid \psi\} \\
& \mid & \{\nu : D \mid \psi\} \\
& \mid & x : P \to P
\end{array}$$

$$B \in \texttt{Base} ::= \texttt{'a} \mid \texttt{int} \mid \texttt{bool}$$

$$D \in \texttt{DType} ::= \mu t\ \Sigma_i\ \mathcal{C}_i\langle \vec{\texttt{'a}}, \vec{D_i} \rangle$$

$$\tau ::= B \mid D \mid x : \tau \to \tau$$

Figure 2.1.: Core language syntax and types.

definition takes the f's unrefined type as input and constructs a refinement type for f encoding ψ:

$$\begin{array}{l}
\texttt{spec}(B, \varpi, \psi) = \{\nu : B \mid \psi\} \\
\texttt{spec}(D, \varpi, \psi) = \{\nu : D \mid \psi\} \\
\texttt{spec}(\{x : \tau_x \to \tau\}, \varpi, \psi) = \\
\qquad \begin{cases}
\{x : \texttt{spec}(\tau_x, \varpi, [\nu/x]\psi) \to \tau\} & free(\psi) \subseteq \varpi \cup \{x\} \\
\{x : \tau_x \to \texttt{spec}(\tau, \varpi \cup \{x\}, \psi)\} & \texttt{otherwise}
\end{cases}
\end{array}$$

In this function, $free(\psi)$ represents the free (program) variables of the predicate ψ. Essentially, `specType` pushes the refinement predicate ψ to the first refinement place in the type where all the free variables of ψ are in scope. The second argument to the function `specType` serves to record the variables in scope at each place, and therefore the top level call from $\texttt{specType}(\Gamma_f, f, \psi)$ to `spec` for a certain function f is of the form:

$$\texttt{specType}(\Gamma_f, f, \psi) = \texttt{spec}(\texttt{HM}(\Gamma_f, f), dom(\Gamma_f), \psi)$$

$$E(\texttt{eval.ctx.}) \;::=\; [\,] \mid v\ E \mid \texttt{let}\ x\ = E\ \texttt{in}\ e$$

$$E[(\texttt{fix}\,(\texttt{fun}\ f \to \lambda x.\ e))\ v] \hookrightarrow$$
$$\quad E[e[f \mapsto (\texttt{fix}\,(\texttt{fun}\ f \to \lambda x.\ e))][x \mapsto v]]$$
$$E[\,\texttt{op}\,(v_0, \cdots, v_n)] \hookrightarrow E[[\![\,\texttt{op}\,]\!](v_0, \cdots, v_n)]$$
$$E[\,\texttt{if true then}\ e_t\ \texttt{else}\ e_f] \hookrightarrow E[e_t]$$
$$E[\,\texttt{if}\ v\ \texttt{then}\ e_t\ \texttt{else}\ e_f] \hookrightarrow E[e_f](v \neq \texttt{true})$$
$$E[\,\texttt{let}\ x\ =\ v\ \texttt{in}\ e] \hookrightarrow E[e[x \mapsto v]]$$
$$E[\,\texttt{assert true}\,] \hookrightarrow\ E[()]$$
$$E[\texttt{assert}\ v] \hookrightarrow\ \texttt{fail}\,(v \neq \texttt{true})$$

Figure 2.2.: Core language small-step semantics.

where we assume the existence a Hindley-Milner type checking oracle $\texttt{HM}(\Gamma_f, f)$, which returns the unrefined type of a function f, and Γ_f is the type environment for the definition of f. The call $dom(\Gamma_f)$ returns all the in-scope variables embedded in Γ_f.

Semantics.

Fig. 2.2 defines a call-by-value semantics for our language in terms of a small-step evaluation relation ($\hookrightarrow$). The semantics is standard. Note that an assertion failure results in program `fail`.

Fig. 2.3 defines salient refinement type inference rules for our core language; these rules are essentially extended from [7]. Syntactically, $\Gamma \vdash e : P$ states that expression e has refinement type P under ordered *type environment* Γ that consists of a sequence of refinement type bindings $x : P_x$ along with guard expressions drawn from conditional expression predicates. The use of these guard expressions makes the type system path-sensitive since the refinement types inferred for a term are computed using the

guard expressions that encode the program path taken to reach this term. We define the *shape* of a refinement type as its corresponding ML type; thus, for a refinement type P, its shape $\|P\|$ is obtained by replacing all refinements in P with $\texttt{true}$, effectively erasing the refinement to yield an unrefined type. For function types, erasure is defined recursively:

$$\begin{aligned} \|\{\nu : B \mid \psi\}\| &= B \\ \|\{\nu : D \mid \psi\}\| &= D \\ \|x : P \to P\| &= x : \|P\| \to \|P\| \end{aligned}$$

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

$$\begin{aligned} \|\Gamma,\, x : P\| &= \|\Gamma\|\,,\, x : \|P\| \\ \|\Gamma,\, P\| &= \|\Gamma\| \\ \|\emptyset\| &= \emptyset \end{aligned}$$

Hence, $\|\Gamma\|$ consists only of bindings that relate variables to ML types, with all refinements replaced with $\texttt{true}$ and guard expressions found in Γ removed.

Constants. The basic units of computation are the constants c built into our programming language, each of which has a refinement type $ty(c)$ that precisely captures the semantics of the constants. These include *basic constants*, corresponding to integers and boolean values, and *primitive functions*, which encode various operations. For example,

$$\begin{aligned} \texttt{true} &:: \{\nu : \texttt{bool} \mid \nu \iff \texttt{true}\} \\ \texttt{false} &:: \{\nu : \texttt{bool} \mid \nu \iff \texttt{false}\} \\ = &:: x : \texttt{'a} \to y : \texttt{'a} \to \{\nu : \texttt{bool} \mid \nu \iff (x = y)\} \\ > &:: x : \texttt{'a} \to y : \texttt{'a} \to \{\nu : \texttt{bool} \mid \nu \iff (x > y)\} \\ < &:: x : \texttt{'a} \to y : \texttt{'a} \to \{\nu : \texttt{bool} \mid \nu \iff (x < y)\} \\ + &:: x : \texttt{int} \to y : \texttt{int} \to \{\nu : \texttt{int} \mid \nu = x + y\} \\ - &:: x : \texttt{int} \to y : \texttt{int} \to \{\nu : \texttt{int} \mid \nu = x - y\} \end{aligned}$$

Well-formedness Judgement. These rules are of the form $\Gamma \vdash P$, and check if refinement type P is well-defined under type environment Γ. The WF-Base rule checks that the refinement ψ of a refinement base type does not refer to program variables that escape from its type environment Γ, i.e., ψ is a well-defined predicate.

It is important to note that the WF-Base rule makes use of an unrefined typing judgment ($\Vdash$) under a *refinement erased* Γ (denoted $\|\Gamma\|$) for this purpose. Rules for unrefined typing judgments are straightforward and can be obtained from the Hindley-Milner type system. The WF-Fun rule defines well-formedness conditions for functions.

Type Judgement. The typing rules state how an expression e can be typed. Our typing rules are refinements of the ML typing rules. If $\Gamma \vdash e : P$ then $\|\Gamma\| \vdash e : \|P\|$. $\Gamma; \mathtt{x} : P$ defines the type environment that extends the sequence Γ with a binding for $\mathtt{x}$ to P. The rules for variables, constants, let-expressions and if-conditions are standard. Rule T-Abs defines recursive functions in the obvious way. Rule T-App establishes a subtyping relation between the actual and formal parameters in the application. The subtype judgment in rule T-Assert enforces that the assertion expression v hold. Polymorphic instantiation and generalization are defined in the standard way.

Subtype Judgement. This important class of rules check, at each call-site, that the actual arguments satisfy the precondition of the function called, and verify at each definition site, that the return value establishes the desired postcondition. The Subt-Base1 and Subt-Base2 rule checks whether a refinement type subtypes another refinement type.

The premise check requires the conjunction of environment formula $\langle\Gamma\rangle$ and $\langle\psi_1\rangle$ implies $\langle\psi_2\rangle$. The encoding $\langle\psi\rangle$ translates a predicate ψ into a (decidable) logic formula for satisfiability checking. Our encoding of $\langle\Gamma\rangle$ is adapted from [7]:

$$\bigwedge\{v \mid v \Leftrightarrow \mathtt{true} \in \Gamma\} \wedge \bigwedge\{\neg v \mid v \Leftrightarrow \mathtt{false} \in \Gamma\} \wedge \bigwedge\left\{ \langle[x/\nu]\psi\rangle \;\middle|\; (x : \{\tau \mid \psi\}) \in \Gamma \wedge \tau \in B \cup D \right\}$$

WF-Base1
$$\frac{\|\Gamma; \nu : B\| \Vdash \psi \ : \ \texttt{bool}}{\Gamma \vdash \{\nu : B \mid \psi\}}$$

WF-Base2
$$\frac{\|\Gamma; \nu : D\| \Vdash \psi \ : \ \texttt{bool}}{\Gamma \vdash \{\nu : D \mid \psi\}}$$

WF-Fun
$$\frac{\Gamma; x : P_x \vdash P}{\Gamma \vdash x : P_x \to P}$$

T-Var
$$\frac{x \text{ is in the domain of } \Gamma}{\Gamma \vdash x : \Gamma(x)}$$

T-Const
$$\frac{}{\Gamma \vdash c : ty(c)}$$

T-Sub
$$\frac{\Gamma \vdash e : P' \qquad \Gamma \vdash P' <: P}{\Gamma \vdash e : P}$$

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

T-Abs-1
$$\frac{\Gamma; x : P_x \vdash e : P_e \qquad \Gamma; x : P_x \vdash P_e <: P}{\Gamma \vdash \lambda x.\ e : \{x : P_x \to P\}}$$

T-Assert
$$\frac{\Gamma \vdash \{\,\texttt{bool} \mid \texttt{true}\,\} <: \{\,\texttt{bool} \mid v\}}{\Gamma \vdash \texttt{assert}\ v : ()}$$

T-App
$$\frac{\Gamma \vdash e : \{x : P_x \to P\} \quad \Gamma \vdash v : P_v \quad \Gamma \vdash P_v <: P_x}{\Gamma \vdash e\ v : [v/x]P}$$

T-Let
$$\frac{\Gamma \vdash e_1 : P' \quad \Gamma; x : P' \vdash e_2 : P \quad \Gamma \vdash P}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : P}$$

T-If
$$\frac{\|\Gamma\| \Vdash v \ : \ \texttt{bool} \quad \Gamma \vdash P \qquad \Gamma; v \Leftrightarrow \texttt{true} \vdash e_2 : P \quad \Gamma; v \Leftrightarrow \texttt{false} \vdash e_3 : P}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : P}$$

T-Gen
$$\frac{\Gamma \vdash e : P \qquad \texttt{'a} \notin \Gamma}{\Gamma \vdash \forall \texttt{'a} \cdot e : \forall \texttt{'a}.P}$$

T-Inst
$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash e : \forall \texttt{'a}.P}{\Gamma \vdash \tau\ e : P[\tau/\texttt{'a}]}$$

Subt-Base1
$$\frac{\texttt{Valid}(\langle\Gamma\rangle \wedge \langle\psi_1\rangle \Rightarrow \langle\psi_2\rangle)}{\Gamma \vdash \{B|\psi_1\} <: \{B|\psi_2\}}$$

Subt-Base2
$$\frac{\texttt{Valid}(\langle\Gamma\rangle \wedge \langle\psi_1\rangle \Rightarrow \langle\psi_2\rangle)}{\Gamma \vdash \{D \mid \psi_1\} <: \{D \mid \psi_2\}}$$

Subt-Arrow
$$\frac{\Gamma \vdash P'_x <: P_x \quad \Gamma; x : P'_x \vdash P <: P'}{\Gamma \vdash \{x : P_x \to P\} <: \{x : P'_x \to P'\}}$$

Figure 2.3.: Core language refinement type system.

We encode all the path-sensitive conditions and refinement types of base typed or inductive data typed variables found from type environment into the choice of a logic. This kind of embedding aims to strengthen the antecedent of the implication and is conservative [7].

The SUBT-ARROW rule refines the simple subtype rules for function subtyping.

We state that our refinement type system is sound: our core language enjoys the usual progress and preservation properties; evaluation preserves types, and well-typed programs do not get stuck. (An assertion violation causes the program to `fail`.)

Theorem 2.1.1 [*Refinement Type Safety*]

1. (*Preservation*) *If* $\Gamma \vdash e : P$ *and* $e \hookrightarrow e'$ *then* $\Gamma \vdash e' : P$
2. (*Progress*) *If* $\emptyset \vdash e : P$ *and* e *is not a value then there exists an* e' *such that* $e \hookrightarrow e'$.

Since our type system is adapted from [7], we refer readers to [7] for the proof of correctness of refinement type checking.

2.2 Refinement Type Based Verification Algorithm

After lifting presumed invariants into refinement types, we need to subsequently validate those types through the type system introduced in Fig. 2.3. Following [7], our refinement types are based on unrefined types, in which each unknown type refinement is represented by an *unknown refinement variable* κ. In our system, the type refinements for functional types are automatically inferred from test data and are initially associated with unknown refinement variables for the corresponding functions, by the `specType` procedure. Solving the well-formedness judgements is standard, as described in e.g., [7].

However, other unknown refinement variables, associated with local expressions inside a function's definitions, are still undefined, which prohibits us from directly

reusing the type checking infrastructure in [7]. In this section, we describe an algorithm that extracts path-sensitive verification conditions (VC) from refinement subtyping relations. We generate the type refinements for the unknown refinement variables for local expressions as part of VC generation.

First, type constraints `cs` over unknown refinement variables that capture the subtyping relations between the types of various subexpressions are generated by traversing the program expression in a syntax-directed manner, applying the typing rules in Fig. 2.3. We present the type constraint generation algorithm, adapted from [7], in Fig. 2.4. Procedure InferExp takes typing environment Γ and an expression e as inputs and generate subtyping relations between the types of various subexpressions by traversing the syntax of e. The template function takes a HM (Hindley-Milner) based unrefined type as input and outputs a refinement type in which each type refinement predicate is represented as an unknown refinement variable κ, which is then instantiated by the `specType` procedure using some candidate specifications obtained somehow.

There are three verification conditions generated from the type checking rules. First, a subtyping constraint introduced by an `assert` expression:

$$\Gamma \vdash \{\, \texttt{bool} \mid \texttt{true} \,\} <: \{\, \texttt{bool} \mid v \}$$

entails a verification condition that checks the validity of v under the path constraints and type bindings defined by Γ. Second, the subtyping constraint associated with function abstraction:

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

establishes a verification condition on the post-condition of this abstraction that requires it be consistent with the invariants inferred for its body. Second, the subtyping constraint associated with function application:

$$\Gamma \vdash P_v <: P_x$$

entails a verification condition that checks that the specification of the function's pre-condition subsumes the invariants associated with the argument at the call. We

```
let InferExp Γ e =
  match e with
    | x ⟶ (Γ(x), ∅)
    | c ⟶ (ty(c), ∅)
    | fix (fun f → λx. e) ⟶
      let (x : P_x → P) = template (HM (‖Γ‖, fix (fun f → λx. e))) in
      let (P_e, C) = InferExp (Γ; f : {x : P_x → P}; x : P_x, e) in
      ((x : P_x → P), {Γ; x : P_x ⊢ P_e <: P} ∪ C)
    | e v ⟶
      let (x : P_x → P, C_1) = InferExp (Γ, e) in
      let (P_v, C_2) = InferExp (Γ, v) in
      ([v/x]P, {Γ ⊢ P_v <: P_x} ∪ C_1 ∪ C_2)
    | if v then e_2 else e_3 ⟶
      let P = template (HM (‖Γ‖ , e)) in
      let (P_2, C_2) = InferExp (Γ; v ⇔ true, e_2) in
      let (P_3, C_3) = InferExp (Γ; v ⇔ false, e_3) in
      (P, C_2 ∪ C_3 ∪ {Γ; v ⇔ true ⊢ P_2 <: P} ∪ {Γ; v ⇔ false ⊢ P_3 <: P})
    | let x = e_1 in e_2 ⟶
      let (P_1, C_1) = InferExp (Γ, e_1) in
      let (P_2, C_2) = InferExp (Γ; x : P_1, e_2) in
      ((P_2, C_1 ∪ C_2)
    | assert v ⟶
      let (P, C) = InferExp (Γ, v) in
      (P, {Γ ⊢ {bool | true} <: {bool | v}} ∪ C)
```

Figure 2.4.: Refinement typing constraint generation.

convert all the subtyping constraints over function types into base subtyping constraints by using the SUBT-ARROW rule in Fig. 2.3.

1: let `Sol` cs $\mathcal{A}$ κ =

2: if $\mathcal{A}[\kappa]$ is set then $\mathcal{A}[\kappa]$

3: else

4: $\bigvee$ {(`Sol` cs $\mathcal{A}$ Γ) $\wedge$ (`Sol` cs $\mathcal{A}$ r_1) |

5: $\{\Gamma \vdash \{\tau|r_1\} <: \{\tau|\kappa\}\} \in$ cs $\wedge$ $\tau \in B \cup D\}$

6:

7: and `Sol` cs $\mathcal{A}$ Γ =

8: $\bigwedge$ $\{[x/\nu](\texttt{Sol}$ cs $\mathcal{A}$ $r) \mid \{x{:}\{\tau|r\}\}\in\Gamma \wedge \tau \in B \cup D\}$

9:

10: and `Sol` cs $\mathcal{A}$ r = match r with

11: $\mid \kappa' \rightarrow$ `Sol` cs $\mathcal{A}$ κ'

12: $\mid \psi \rightarrow \psi$

13:

14: let `Verify` cs $\mathcal{A}$ $\{\Gamma \vdash \{\tau|r_1\} <: \{\tau|r_2\}\}$ =

15: smt_query ((`Sol` cs $\mathcal{A}$ Γ)$\wedge$(`Sol` cs $\mathcal{A}$ r_1)$\Rightarrow$(Sol cs $\mathcal{A}$ r_2)

16:

17: let `TyCheck` cs $\mathcal{A}$ =

18: if ($\exists c \in$ cs. $c = \{\Gamma \vdash \{\tau|r_1\} <: \{\tau|\kappa_2\}\}$ and

19: κ_2 is with a type refinement in functional type and

20: `Verify` cs $\mathcal{A}$ c = `false`) then

21: `TyCheck` cs $\mathcal{A}[\kappa_2 \leftarrow \{\psi \mid \psi \in A[\kappa_2] \wedge$ `Verify` cs $\mathcal{A}$ $(\Gamma \vdash \{\tau|r_1\} <: \{\tau|\psi\})\}]$

22: else

23: $\bigwedge$ {`Verify` cs $\mathcal{A}$ $c \mid c = \{\Gamma \vdash \{\tau|r_1\} <: \{\tau|\psi_2\}\} \in$ cs}

Figure 2.5.: Refinement type based verification algorithm.

Our specific type checking algorithm is summarized in Fig. 2.5. In `TyCheck`, all the base or data type subtyping constraints, inherited from the result of calling `InferExp`, are organized into a list `cs`, which are all in the form of $\Gamma \vdash \{\tau|r_1\} <: \{\tau|r_2\}$

where $\tau \in B \cup D$ and r_1 or r_2 may be either an unknown refinement variable or a concrete predicate. $\mathcal{A}$ is a solution map from unknown refinement variable κ to a set of likely type refinements. As mentioned before, type refinements are only associated to unknown refinement variables for function types. We solve all VC constraints over function abstraction and application by iteratively removing type refinements that prevent a constraint from being satisfied using an SMT solver (line 21), querying the implication check in the SUBT-BASE1 and SUBT-BASE2 rules shown in Fig. 2.3, which is already encoded into the `Verify` procedure (line 14). `Sol` is a procedure that retrieves type refinements for an unknown refinement variable, which is also capable of inferring refinement types for local expressions. If an unknown refinement variable κ is hosted in $\mathcal{A}$, κ is for function type and we directly return (line 2). Otherwise, we repeatedly find the other constraints in `cs` which are in the form of $\Gamma \vdash \{\tau|r_1\} <: \{\tau|\kappa\}$ and recursively encode them (line 4 and 5). Such constraints constrain the solution for κ in the type derivation and are turned into a disjunction with appropriate path constraints defined in their typing environments (line 4). Thus, our VC encoding is a path-sensitive analysis. `Sol` terminates because unknown refinement variables for function types are explicitly set in $\mathcal{A}$. After solving all refinement type variables that sit in refinement function types, by the end of `TyCheck`, we check the validity of assert expressions (line 23).

We state that the VC generation algorithm, `TyCheck`, is sound. To ease our proof, we use the notation $\mathcal{A}\kappa$ to abbreviate the procedure `Sol` cs $\mathcal{A}$ κ in Fig. 2.5. We neglect cs here since cs is finalized when we come to the `TyCheck` algorithm. Similarly we use $\mathcal{A}\Gamma$ to abbreviate `Sol` cs $\mathcal{A}$ Γ. We also lift this definition from unknown type variables to refinement types with unknown type variables and subtyping constraints with unknown type variables in the obvious way.

Lemma 1 *For any type environment* Γ*, expression* e*, and solution map* $\mathcal{A}$*, if* `InferExp` $(\Gamma, e) = (P, C)$ *then* `InferExp` $(\mathcal{A}\Gamma, e) = (\mathcal{A}P, \mathcal{A}C)$.

Proof Induction on the structure of e. ■

Assume a refinement type P is with unknown type variables (κ), we use ($\mathcal{A}(P \mapsto P')$) to denote that all the unknown type variables in P are updated to the corresponding type refinements in P'. For example, $\mathcal{A}(\{x : \kappa_1 \to \kappa_2\} \mapsto \{x : \{\nu = 0\} \to \{\nu = 1\}\})$ indicates $\mathcal{A}[\kappa_1] = \{\nu = 0\}$ and $\mathcal{A}[\kappa_2] = \{\nu = 1\}$. We also use $\mathcal{A}(P)$ to concretize a refinement type whose unknown type variables are instantiated by the solution map $\mathcal{A}$. For example, $\mathcal{A}(\{x : \kappa_1 \to \kappa_2\}) = \{x : \{\nu = 0\} \to \{\nu = 1\}\}$.

Lemma 2 *For any solution map $\mathcal{A}$, and refinement type P and P' such that $\|P\| = \|P'\|$,*

1. $(\mathcal{A}(P \mapsto P'))(P) = P'$,
2. *For any P'' such that all unknown type variables in P'' are in* Dom *($\mathcal{A}$) then* $(\mathcal{A}(P \mapsto P'))(P'') = \mathcal{A}P''$.

Proof Induction on the structure of P. ■

The following theorem states the correctness of our VC generation. Recall that the solution map $\mathcal{A}$ associates an unknown refinement type variable to a type refinement predicate only if the unknown sits in a refinement function type template. Otherwise, unknowns for local expressions are undefined in $\mathcal{A}$.

Theorem 2.2.1 *For every type environment Γ and expression e such that* `InferExp` *(Γ, e) = (P, C), $\Gamma \vdash e : P'$ iff there exists a solution $\mathcal{A}$ such that $\mathcal{A}P = P'$ when $e \equiv e_1\ y$ and $e \equiv \lambda x.e'$ and, otherwise, $\Gamma \vdash \mathcal{A}P <: P'$ and $\mathcal{A}C$ is valid.*

Proof First prove $\Rightarrow$.

By induction on the structure of e.

1. case $e \equiv c$ or $e \equiv x$:

   Immediate hold since any $\mathcal{A}$ is correct.

2. case $e \equiv \texttt{fix}\,(\,\texttt{fun}\ \ f \to \lambda x.\ e')$:

$$
\begin{aligned}
P &= x : P_x \to P_1 \\
C &= C_1 \cup \{\Gamma; x : P_x \vdash P_e <: P_1\} \\
P' &= x : P'_x \to P'_1 \\
x : P_x \to P_1 &= \mathsf{template}(\mathsf{Shape}(x : P'_x \to P'_1)) \\
(P_e, C_1) &= \texttt{InferExp}(\Gamma; f : \{x : P_x \to P\}; x : P_x, e')
\end{aligned}
$$

Let $\mathcal{A}_0 = \emptyset(P \mapsto P')$

$$
\begin{aligned}
(\mathcal{A}_0 P_e, \mathcal{A}_0 C_1) &= \texttt{InferExp}(\Gamma; f : \{x : P_x \to P\}; x : \mathcal{A}_0 P_x, e') \\
&= \texttt{InferExp}(\Gamma; f : \{x : P'_x \to P'_1\}; x : P'_x, e')
\end{aligned}
$$

By inversion, there exists a refinement type S such that

$$\Gamma; f : \{x : P'_x \to P'_1\}; x : P'_x \vdash e' : S \tag{a}$$

$$\Gamma; f : \{x : P'_x \to P'_1\}; x : P'_x \vdash S <: P'_1 \tag{b}$$

Thus, from (a), there exists $\mathcal{A}_1$ such that:

$$\mathcal{A}_1(\mathcal{A}_0 P_e) = S \tag{c}$$

$$\mathcal{A}_1(\mathcal{A}_0 C_1) \text{ is valid} \tag{d}$$

Let $\mathcal{A} = \mathcal{A}_1; \mathcal{A}_0$ then:

$$
\begin{aligned}
\mathcal{A}P &= \mathcal{A}_1(\mathcal{A}_0 P) \\
&= \mathcal{A}_1(P') \\
&= P'
\end{aligned}
$$

Also, from (1), obtain

$$
\begin{aligned}
\mathcal{A}C &= (\mathcal{A}_1; \mathcal{A}_0) C_1 \\
&\quad \cup \{\Gamma; f : \{x : (\mathcal{A}_1; \mathcal{A}_0) P_x \to (\mathcal{A}_1; \mathcal{A}_0) P\}; x : (\mathcal{A}_1; \mathcal{A}_0) P_x \\
&\qquad \vdash (\mathcal{A}_1; \mathcal{A}_0) P_e <: (\mathcal{A}_1; \mathcal{A}_0) P_1\} \\
&= (\mathcal{A}_1; \mathcal{A}_0) C_1 \cup \{\Gamma; f : \{x : P'_x \to P'_1\}; x : P'_x \vdash S <: P'_1\}
\end{aligned}
$$

which by (b),(c) and (d) is valids.

3. case $e \equiv e_1\ v$:

$$P = [v/x]P''$$
$$C = C_1 \cup C_2 \cup \{\Gamma \vdash P_2' <: P_2''\}$$
$$(x : P_2'' \to P'', C_1) = \texttt{InferExp}(\Gamma, e_1)$$
$$(P_2', C_2) = \texttt{InferExp}(\Gamma, v)$$

By inversion, there exist T_2', T_2 and T:

$$\Gamma \vdash e_1 : x : T_2 \to T \tag{a}$$
$$\Gamma \vdash v : T_2' \tag{b}$$
$$\Gamma \vdash T_2' <: T_2 \tag{c}$$
$$P' = [v/x]T \tag{d}$$

By IH and (a), there exists $\mathcal{A}_1$:

$$\mathcal{A}_1 P_2'' = T_2' \tag{e}$$
$$\mathcal{A}_1 P'' = T \tag{f}$$
$$\mathcal{A}_1 C_1 \text{ is valid} \tag{g}$$

By IH and (b), there exists $\mathcal{A}_2$:

$$\mathcal{A}_2 P_2' = T_2 \tag{h}$$
$$\mathcal{A}_2 C_2 \text{ is valid} \tag{i}$$

Let $\mathcal{A} = \mathcal{A}_1; \mathcal{A}_2$

$$\begin{aligned}\mathcal{A}P &= (\mathcal{A}_1; \mathcal{A}_2)[v/x]P'' \\ &= [v/x](\mathcal{A}_1; \mathcal{A}_2)P'' \\ &= [v/x]\mathcal{A}_1 P'' \\ &= [v/x]T \\ &= P'\end{aligned}$$

Also,

$$\begin{aligned}\mathcal{A}C &= (\mathcal{A}_1; \mathcal{A}_2)C_1 \cup (\mathcal{A}_1; \mathcal{A}_2)C_2 \\ &\quad \cup \{\Gamma \vdash (\mathcal{A}_1; \mathcal{A}_2)P_2' <: (\mathcal{A}_1; \mathcal{A}_2)P_2''\} \\ &= \mathcal{A}_1 C_1 \cup \mathcal{A}_2 C_2 \cup \{\Gamma \vdash \mathcal{A}_2 P_2' <: \mathcal{A}_1 P_2''\} \\ &= \mathcal{A}_1 C_1 \cup \mathcal{A}_2 C_2 \cup \{\Gamma \vdash T_2' <: T_2\}\end{aligned}$$

which by (c),(e),(f),(g),(h),(i) is valid.

4. case $e \equiv$ if v then e_2 else e_3:

$$\begin{aligned}C &= C_2 \cup C_3 \cup \{\Gamma; v \Leftrightarrow \texttt{true} \vdash P_2 <: P\} \\ &\quad \cup \{\Gamma; v \Leftrightarrow \texttt{false} \vdash P_3 <: P\} \\ (P_2, C_2) &= \texttt{InferExp}(\Gamma; v \Leftrightarrow \texttt{true}, e_2) \\ (P_3, C_3) &= \texttt{InferExp}(\Gamma; v \Leftrightarrow \texttt{false}, e_3)\end{aligned}$$

By inversion and applying IH, there exists $\mathcal{A}_2$, $\mathcal{A}_3$, S_2 and S_3 such that $\Gamma_2; v \Leftrightarrow \texttt{true} \vdash e_2 : S_2$ and $\Gamma_3; v \Leftrightarrow \texttt{false} \vdash e_3 S_3$

$$\begin{aligned}&\mathcal{A}_2 C_2 \text{ is valid} \\ &\mathcal{A}_3 C_3 \text{ is valid} \\ &\mathcal{A}_2 P_2 = S_2 \\ &\mathcal{A}_3 P_3 = S_3 \\ &\Gamma; v \Leftrightarrow \texttt{true} \vdash S_2 <: P' \\ &\Gamma; v \Leftrightarrow \texttt{false} \vdash S_3 <: P'\end{aligned}$$

By Fig. 2.5, $\mathcal{A}P = (v \Leftrightarrow \texttt{true} \wedge \mathcal{A}P_2) \vee (v \Leftrightarrow \texttt{false} \wedge \mathcal{A}P_3)$.

Let $\mathcal{A} = (\mathcal{A}_2; \mathcal{A}_3)$.

$$\begin{aligned}
&\Gamma \vdash \mathcal{A}P <: P' \\
&\mathcal{A}C = \mathcal{A}_2 C_2 \cup \mathcal{A}_3 C_3 \\
&\qquad \cup \{\Gamma; v \Leftrightarrow \texttt{true} \vdash \mathcal{A}P_2 <: \mathcal{A}P\} \\
&\qquad \cup \{\Gamma; v \Leftrightarrow \texttt{false} \vdash \mathcal{A}P_3 <: \mathcal{A}P\}
\end{aligned}$$

which is valid immediately.

5. case $e \equiv \text{let } x = e_1 \text{ in } e_2$:

$$C = C_1 \cup C_2 \tag{a}$$

$$(P_1, C_1) = \texttt{InferExp}(\Gamma, e_1) \tag{b}$$

$$(P, C_2) = \texttt{InferExp}(\Gamma; x : P_1, e_2) \tag{c}$$

By inversion, there exists S_1 such that:

$$\Gamma \vdash e_1 : S_1 \tag{d}$$

$$\Gamma; x : S_1 \vdash e_2 : P' \tag{e}$$

By (d) and IH there exists $\mathcal{A}_1$ such that:

$$\mathcal{A}_1 C_1 \text{ is valid} \tag{f}$$

$$\Gamma \vdash \mathcal{A}_1 P_1 <: S_1 \tag{g}$$

By (c),

$$\begin{aligned}
(\mathcal{A}_1 P, \mathcal{A}_1 C_2) &= \texttt{InferExp}(\Gamma; x : \mathcal{A}_1 P_1, e_2) \\
&= \texttt{InferExp}(\Gamma; x : S_1, e_2)
\end{aligned}$$

By (e) and IH there exists $\mathcal{A}_2$ such that:

$$\begin{aligned}
&\mathcal{A}_2(\mathcal{A}_1 C_2) \text{ is valid} \\
&\Gamma; x : S_1 \vdash \mathcal{A}_2(\mathcal{A}_1 P) <: P'
\end{aligned}$$

Let $\mathcal{A} = (\mathcal{A}_2; \mathcal{A}_1)$. According to the well-formed constraints of `Let`,

$$\Gamma \vdash \mathcal{A}P <: P'$$
$$\mathcal{A}C = \mathcal{A}_1 C_1 \cup (\mathcal{A}_2; \mathcal{A}_1) C_2$$

is valid.

6. case $e \equiv$ assert v:

$$C = \{\Gamma \vdash \{\, \texttt{bool} \mid \texttt{true} \,\} <: \{\, \texttt{bool} \mid v \}\}$$

By inversion,

$$\Gamma \vdash \{\, \texttt{bool} \mid \texttt{true} \,\} <: \{\, \texttt{bool} \mid v \}$$

Pick any $\mathcal{A}$.

$$\mathcal{A}C = \mathcal{A}(\Gamma \vdash \{\, \texttt{bool} \mid \texttt{true} \,\} <: \{\, \texttt{bool} \mid v \})$$

is valid.

Then prove $\Leftarrow$.

By induction on the structure of e.

1. case $e \equiv c$ or $e \equiv x$: Immediate since $C = \emptyset$ and and since there is no template for e $\Gamma \vdash e : P$ holds.

2. case $e \equiv \texttt{fix}\,(\texttt{fun}\ f \to \lambda x.\ e')$:

$$P = x : P_x \to P_1$$
$$C = C_1 \cup \{\Gamma; f : \{x : P_x \to P_1\}; x : P_x \vdash P_e <: P_1\}$$
$$P' = x : P'_x \to P'_1$$
$$x : P_x \to P_1 = \mathsf{template}(\mathsf{Shape}(x : P'_x \to P'_1))$$
$$(P_e, C_1) = \texttt{InferExp}(\Gamma; f : \{x : P_x \to P_1\}; x : P_x, e')$$

As $\mathcal{A}C$ is valid,

$$\mathcal{A}C_1 \text{ is valid} \tag{a}$$

$$\Gamma; f : \{x : \mathcal{A}P_x \to \mathcal{A}P_1\}; x : \mathcal{A}P_x \vdash \mathcal{A}P_e <: \mathcal{A}P_1 \tag{b}$$

Obtain

$$(\mathcal{A}P_e, \mathcal{A}C_1) = \texttt{InferExp}(\Gamma; f : \{x : \mathcal{A}P_x \to \mathcal{A}P_1\}; x : \mathcal{A}P_x, e')$$

By (a) and IH,

$$\Gamma; f : \{x : \mathcal{A}P_x \to \mathcal{A}P_1\}; x : \mathcal{A}P_x \vdash e' : \mathcal{A}P_e \tag{c}$$

From (b), (c),

$$\Gamma \vdash \texttt{fix}\,(\texttt{fun}\ f \to \lambda x.e') : x : \mathcal{A}P_x \to \mathcal{A}P_1$$

hence

$$\Gamma \vdash \texttt{fix}\,(\texttt{fun}\ f \to \lambda x.e') : \mathcal{A}(x : P_x \to P_1)$$

Finally, $\Gamma \vdash \texttt{fix}\,(\texttt{fun}\ f \to \lambda x.e') : \mathcal{A}P$

3. case $e \equiv e_1\ v$:

$$P = [v/x]P''$$

$$C = C_1 \cup C_2 \cup \{\Gamma \vdash P_2' <: P_2''\}$$

$$(x : P_2'' \to P'', C_1) = \texttt{InferExp}(\Gamma, e_1)$$

$$(P_2', C_2) = \texttt{InferExp}(\Gamma, v)$$

As $\mathcal{A}C$ is valid,

$$\mathcal{A}C_1 \text{ is valid}$$

$$\mathcal{A}C_2 \text{ is valid}$$

$$\Gamma \vdash \mathcal{A}P_2' <: \mathcal{A}P_2'' \tag{a}$$

By IH,

$$\Gamma \vdash e : x : \mathcal{A}P_2'' \to \mathcal{A}P'' \tag{b}$$

$$\Gamma \vdash v_1 : \mathcal{A}P_2' \tag{c}$$

From (b), (c),

$$\Gamma \vdash e_1\ v : [v/x]\mathcal{A}P''$$

$$\Gamma \vdash e_1\ v : \mathcal{A}[v/x]P''$$

$$\Gamma \vdash e_1\ v : \mathcal{A}P$$

4. case $e \equiv \text{if } v \text{ then } e_2 \text{ else } e_3$:

$$C = C_2 \cup C_3 \cup \{\Gamma; v \Leftrightarrow \texttt{true} \vdash P_2 <: P\}$$

$$\cup \{\Gamma; v \Leftrightarrow \texttt{false} \vdash P_3 <: P\}$$

$$(P_2, C_2) = \texttt{InferExp}(\Gamma; v \Leftrightarrow \texttt{true}, e_2)$$

$$(P_3, C_3) = \texttt{InferExp}(\Gamma; v \Leftrightarrow \texttt{false}, e_3)$$

AS $\mathcal{A}C$ is valid,

$$\mathcal{A}C_1 \text{ is valid} \tag{a}$$

$$\mathcal{A}C_2 \text{ is valid} \tag{b}$$

By (a), (b), IH,

$$\Gamma; v \Leftrightarrow \texttt{true} \vdash e_2 : \mathcal{A}P_2$$

$$\Gamma; v \Leftrightarrow \texttt{false} \vdash e_3 : \mathcal{A}P_3$$

By Fig. 2.5, $\mathcal{A}P = (v \Leftrightarrow \texttt{true} \wedge \mathcal{A}P_2) \vee (v \Leftrightarrow \texttt{false} \wedge \mathcal{A}P_3)$.

Immediate obtain

$$\Gamma; v \Leftrightarrow \texttt{true} \vdash \mathcal{A}P_2 <: \mathcal{A}P \tag{c}$$

$$\Gamma; v \Leftrightarrow \texttt{false} \vdash \mathcal{A}P_3 <: \mathcal{A}P \tag{d}$$

By (c), (d),

$$\Gamma \vdash \text{if } v \text{ then } e_2 \text{ else } e_3 : \mathcal{A}P <: P'$$

5. case $e \equiv \text{let } x = e_1 \text{ in } e_2$:

$$C = C_1 \cup C_2$$
$$(P_1, C_1) = \texttt{InferExp}(\Gamma, e_1)$$
$$(P, C_2) = \texttt{InferExp}(\Gamma; x : P_1, e_2)$$

As $\mathcal{A}C$ is valid

$$\mathcal{A}C_1 \text{ is valid} \tag{a}$$
$$\mathcal{A}C_2 \text{ is valid} \tag{b}$$

By (a), (b), IH,

$$\Gamma \vdash e_1 : \mathcal{A}P_1 \tag{c}$$
$$\Gamma; x : \mathcal{A}P_1 \vdash e_2 : \mathcal{A}P \tag{d}$$

By (c), (d),

$$\Gamma' \vdash \text{let } x = e_1 \text{ in } e_2 : \mathcal{A}P <: P'$$

6. case $e \equiv \text{assert } v$:

$$C = \{\Gamma \vdash \{\, \texttt{bool} \mid \texttt{true} \,\} <: \{\, \texttt{bool} \mid v\}\}$$

As $\mathcal{A}C$ is valid,

$$\Gamma \vdash \{\, \texttt{bool} \mid \texttt{true} \,\} <: \{\, \texttt{bool} \mid v\} \text{ is valid}$$

Immediately,

$$\Gamma \vdash \text{assert p} : () <: ()$$

■

We show the soundness proof of the refinement type based verification algorithm. Recall that in type checking phase (Sec. 2.2) for any program e, we infer its subtyping constraints into a list `cs`. Let $\mathcal{A}$ denote the solution map for all unknown type

variables in `cs`. For a constraint $c = \{\Gamma \vdash \{B|r_1\} <: \{B|r_2\}\} \in$ `cs`, we claim $\mathcal{A}c$ is valid if `Verify cs` $\mathcal{A}$ c = `true` and $\mathcal{A}$ `cs` is valid if all of the constraints in `cs` are valid.

Theorem 2.2.2 [*Soundness*] *For every program e annotated with some program assertion ψ, assume cs is the list of subtyping constraints inferred for e and $\mathcal{A}$ is the solution map for cs. If* `TyCheck` *return* `true`*, then e is a well-typed program and ψ is a valid specification for e.*

Proof `TyCheck` *cs* $\mathcal{A}$ returns `true` if $\mathcal{A}$ *cs* is valid. From Theorem 2.2.1, the generated subtyping constraints (in `cs`) are solvable if and only if a valid type derivation (for e) exists. ψ is hence valid for e. ■

It is important to note that when `TyCheck` *cs* $\mathcal{A}$ returns `false`, we cannot claim the program is buggy because our analysis is incomplete in general and $\mathcal{A}$ might not be strong enough to complete a verification proof.

3 COMPOSITIONAL AND LIGHTWEIGHT REFINEMENT TYPE INFERENCE

In this chapter, we present POPEYE, a compositional verification system that integrates a first-order verification engine, unaware of higher-order control- and dataflow, into a path- and context-sensitive refinement type inference framework for Standard ML. Notably, our solution treats uses of unknown functions as *uninterpreted* terms. In this way, we are able to directly exploit the scalability and efficiency characteristics of first-order verification tools *without* having to either consider a sophisticated translation or encoding of our functional source program into a first-order one [9], or to re-engineer these tools for a higher-order setting [11]. Our verification strategy is based on a counterexample-guided refinement loop that systematically strengthens a function's inferred refinement type based on new predicates discovered during examination of a derived counterexample path. Moreover, our strategy allows us not to only verify the validity of complex assertions, but can also be used to directly provide counterexample witnesses that disprove the validity of presumed invariants that are incorrect.

Our technique is *compositional* because it lazily propagates refinements computed at call-sites to procedures and *vise versa*, allowing procedure specifications to be strengthened incrementally. It is *lightweight* because it directly operates on source programs without the need to generate arbitrary program slices [19], translate the source to a first-order program [9], or abstract the source to a Boolean program [13]. POPEYE's design consists of two distinct parts:

1. ***Refinement Type Checking***. Initially, we infer coarse refinement types for all local expressions within a procedure using refinement type rules that encode intraprocedural path information in terms of first-order logic formulae

that range over linear arithmetic and uninterpreted functions, the latter used to abstract a program's higher-order control-flow. We build verification conditions that exploit the refinement types and which are subsequently supplied into a first-order decision procedure. Verification failure yields an intraprocedural counterexample path.

2. ***Refinement Type Refinement***. The counterexample path can be used by existing predicate discovery algorithms to appropriately strengthen pre- and post-conditions at function calls. Newly discovered refinement predicates are propagated along subtyping chains that capture interprocedural dependencies to strengthen the refinement type signatures of the procedures used at these call-sites.

The remainder of the chapter is organized as follows. In the next section, we present an informal overview of our approach. Sec. 3.2 defines a context-sensitive refinement type system for our core language. We formalize our verification strategy for this language in Sec. 3.3. Sec. 3.4 discusses the implementation and experimental results. Related work is given in Sec. 3.5.

3.1 Overview and Preliminaries

Refinement types for context-sensitivity. We consider two kinds of refinement type expressions:

1. a *refinement base type* written $\{\nu : B \mid r\}$, where ν is a special value variable undefined in the program whose scope is limited to r, B is a base type, such as `int` or `bool`, and r is a boolean-valued expression (called a *refinement*). For instance, $\{\nu : \texttt{int} \mid \nu > 0\}$ defines a refinement type that represents the set of positive integers.

2. a *refinement function type* written:

$$\{x : P_{1_x} \to P_1\} \oplus \{x : P_{2_x} \to P_2\} \oplus \ldots \oplus \{x : P_{n_x} \to P_n\}$$

abbreviated as $\oplus_i\{x : P_{ix} \to P_{io}\}$, where each $\{x : P_{i_x} \to P_i\}$ defines a function type whose argument x is constrained by refinement type P_{ix} and whose result type is specified by P_i. The different components of a refinement function type distinguish different contexts in which the function may be used. For instance,

$$\{x : \{\nu : \mathsf{int} \mid \nu > 0\} \to \{\nu \ : \mathsf{int} \mid \nu > x\}\}\oplus$$
$$\{x : \{\nu : \mathsf{int}\mid \nu < 0\} \to \{\nu : \mathsf{int}\mid \nu < x\}\}$$

specifies the function that, in one call-site, given a positive integer returns an integer greater than x, while in another, given a negative integer returns an integer less than x. Components in a refinement function type are indexed by an implicit label, e.g., a finite call-string used in polyvariant control-flow analyses [20, 21].

As shorthand, we sometimes write only the refinement predicate to represent the refinement type, omitting its type constructor. Thus, in the following, we sometimes write $\{r\}$ as shorthand for $\{\nu : B \mid r\}$. For example, $\{\nu > 0\}$ represents $\{\nu : \mathsf{int} \mid \nu > 0\}$. We also write B as shorthand for $\{\nu : B \mid \mathsf{true}\}$. For perspicuity, we use syntactic sugar to allow the $\oplus$ operator to be "pushed into" refinements:

$$\{\nu : B \mid r_1\} \oplus \{\nu : B \mid r_2\} = \{\nu : B \mid r_1 \oplus r_2\}$$
$$\{x : P_1 \to P_{r_1}\} \oplus \{x : P_2 \to P_{r_2}\} = \{x : P_1 \oplus P_2 \to P_{r_1} \oplus P_{r_2}\}$$

As a result, context-sensitive refinement types reuse the shape of ML types (Sec. 2.1). Additionally, we define $P.i$ to return the refinement type indexed by label i. When a function is used in a single context, we simply write $\{x : P_x \to P\}$.

Procedure specifications. A procedure specification is given in terms of a pre- and post-condition of a procedure; we express these conditions in terms of a refinement function type where the type of the function's domain can be thought of as the function's pre-condition, and where the type of the function's range defines its post-condition.

```
fun f g x =
  if x>=0 then
    let r = g x in r
  else
    let p = f g
        q = compute x
        s = f p q
    in s

fun main h n =
  let r = f h n
  in assert (r >= 0)
```

Figure 3.1.: Higher-order functions challenge compositional refinement type inference.

3.1.1 Example

Consider the program shown in Fig. 3.1. This program exhibits complex dataflow (e.g., it can create an arbitrary number of closures via the partial application of `f`) and makes heavy use of higher-order procedures (e.g., the formal parameter `g` in function `f`). We wish to infer a useful specification for `f` without having to (a) supply candidate qualifiers used in the refinement types that define the specification, (b) know the possible concrete arguments that can be supplied to `g`, or (c) require details about `compute`'s definition. In spite of these constraints, our technique nonetheless associates the following non-trivial type to `f`:

$$\mathtt{f} : \{\mathtt{g} : \{\mathtt{g}_{\mathtt{arg}} : \{\nu \geq 0\}\} \rightarrow \{\nu \geq 0\}\} \rightarrow \mathtt{x} : \{\,\mathtt{true}\,\} \rightarrow \{\nu \geq 0\}\}$$

This type ascribes an invariant to `g` that asserts that `g` must take a non-negative number as an argument (as a consequence of the path constraint `(x >= 0)` within which it is applied) and returns a non-negative number as a result (as a consequence of the assertion made in `main`).

The utility of context-sensitive refinement types arises when a function is called in different (potentially inconsistent) contexts. Consider the program shown in Fig. 3.2. Here, function `f` (which is supplied the argument `neg` in `main`) is called in two different contexts in the procedure `twice`. The first argument to `f` is a higher-

```
fun g x y = x
fun twice f x y =
  let p = f x
  in f p y
fun neg x y = -(x ())
```

```
fun main n =
  if n >= 0 then
    assert(twice neg (g n) () >= 0)
  else ()
```

Figure 3.2.: A function's specification can be refined based on its context.

$$\text{T-Fun}\quad \frac{\forall\, i.\ \Gamma_i; x : P_{ix} \vdash e : P_{ie} \quad \Gamma_i; x : P_{ix} \vdash P_{ie} <: P_i}{\oplus_i \Gamma_i \vdash \lambda x.e : \oplus_i \{x : P_{ix} \to P_i\}}$$

$$\text{T-Pick}\quad \frac{\Gamma \vdash f : \oplus_i \{x : P_{ix} \to P_i\}}{\Gamma \vdash f_j : \{x : P_{j_x} \to P_j\}} \qquad \text{T-Conc}\quad \frac{\forall\, i.\ \Gamma_i \vdash f_i : \{x : P_{ix} \to P_i\}}{\oplus_i \Gamma_i \vdash f : \oplus_i \{x : P_{ix} \to P_i\}}$$

$$\text{T-ConcFunSub}\quad \frac{\forall\, i.\ \Gamma \vdash \{x : P_{ix} \to P_i\} <: \{x : P'_{i\,x} \to P'_i\}}{\Gamma \vdash \oplus_i \{x : P_{ix} \to P_i\} <: \oplus_i \{x : P'_{i\,x} \to P'\}}$$

Figure 3.3.: Context-sensitive refinement typing rules.

order procedure - in the first call, this procedure is bound to the result of evaluating `g n`; in the second call, the procedure (bound to `p`) is the result of the first partial application. Since `f` negates the value yielded by applying its procedure argument to `()`, we thus infer the following specification:

$$\mathtt{f}_{\mathtt{arg}_1} : \{\{\mathsf{true} \oplus \mathsf{true}\} \to \{\nu \geq 0 \oplus \nu \leq 0\}\} \to \mathtt{f}_{\mathtt{arg}_2} : \{\mathsf{true} \oplus \mathsf{true}\} \to \{\nu \leq 0 \oplus \nu \geq 0\}$$

3.2 Context-sensitive Refinement Type System

This section refines the refinement type system in Fig. 2.3 for adding context-sensitivity. Fig. 3.3 refines only the refinement type inference rules related with

context-sensitivity; the other rules are shown in Fig. 2.3. Rule T-FUN associates a context-sensitive refinement function type with an abstraction. The structure of this type is determined by the different contexts in which the abstraction is applied (Γ_i) generated from rule T-CONC described below. The first judgment in the antecedent considers the type of the abstraction body in all type environments Γ_i enriched by a type binding of bound variable x with refinement type P_{i_x}. The second judgment asserts that P_{i_e}, the type associated with the body of the abstraction, be a subtype of the return type of the abstraction. The abstract labels that subscript function identifiers in the rules are used to express context-sensitivity but are not part of the program syntax, and are constructed during the interprocedural type refinement phase.

There are two rules for extracting and generating context-sensitive refinement type functions. A term f with type $\oplus_i \{x : P_{ix} \rightarrow P_i\}$ reflects the type of all uses of f in different contexts; the type at a given context can be indexed by the label at the use (rule T-PICK). Conversely, we can construct the concatanation of the types at each context to yield the actual type of the function (rule T-CONC).

Rule T-CONCFUN generalizes the usual subtyping rule on functions to deal with context-sensitivity. The function subtyping rules implicitly encode subtyping chains, allowing specifications to be propagated across function boundaries.

Our semantics enjoys the usual progress and preservation properties; evaluation preserves types, and well-typed programs do not get stuck. (An assertion violation causes the program to halt with the special value `fail`.)

Theorem 3.2.1 (Refinement Type Safety)

1. (*Preservation*) *If* $\Gamma \vdash e : P$ *and* $e \hookrightarrow e'$ *then* $\Gamma \vdash e' : P$
2. (*Progress*) *If* $\Gamma \vdash e : P$, *where* $e \neq \mathsf{fail}$ *then* e *is either a constant or an abstraction, or there exists an* e' *such that* $e \hookrightarrow e'$.

Proof Proof is immediate from Theorem 2.1.1. ■

3.3 Verification Procedure

Our verification system consists of (a) a type-checking algorithm that encodes intra-procedural path constraints and generates verification conditions whose validity can be checked by a first-order decision procedure, and (b) a counterexample guided refinement type refinement loop that uses the counterexample yielded by a verification failure to strengthen existing invariants, and propagate new ones inter-procedurally via refinement subtyping chains.

3.3.1 Refinement Type Checking

To support refinement type inference, we introduce refinement type templates (P_T), which are refinement types whose refinement expressions are only refinement variables (κ). The pick or selection operator $\kappa.i$ on refinement variable allows $\oplus$ to be pushed into refinements (as described in Sec. 3.1), and hence omitted in template definitions. Instantiation of the refinement variables to concrete predicates takes place through the type refinement algorithm described in Sec. 3.3.2.

$$\kappa \in \mathsf{RefinementVar} ::= \kappa \mid \kappa.i \quad P_T \in \mathsf{Template} ::= \{\nu : B \,|\, \kappa\} \mid \{x : P_T \rightarrow P_T\}$$

Figure 3.4.: Syntax.

The first step of our verification procedure is to assign each function a refinement type *template* as described earlier. By applying our inference rules, with the type template, given a type environment Γ and expression e, we can construct refinement types for local expressions and derive a set of subtyping constraints, which will be subsequently used to generate verification conditions (VC) as in Sec. 2.2.

A solution in our system is defined by a refinement environment $\mathcal{A}$ that maps refinement variables κ to refinements. We lift this notion to refinement types $\mathcal{A}(P_T)$

and type environment $\mathcal{A}(\Gamma)$ by substituting each place holder κ with $\mathcal{A}(\kappa)$ appearing in P_T and Γ. A verification condition c is valid if $\mathcal{A}(c)$ is valid. We say $\mathcal{A}$ satisfies a subtyping constraint $\Gamma \vdash P_{T1} <: P_{T2}$ if $\mathcal{A}(\Gamma) \vdash \mathcal{A}(P_{T1}) <: \mathcal{A}(P_{T2})$. $\mathcal{A}$ is a valid solution if it satisfies all subtype constraints.

We deconstruct arbitrary subtyping constraints to base subtyping constraints (Fig. 3.3). According to the SUBT-BASE rule (Sec. 2.2), the verification condition formula is generated as

$$\langle \mathcal{A}(\Gamma) \rangle \wedge \langle \mathcal{A}(r_1) \rangle \Rightarrow \langle \mathcal{A}(r_2) \rangle$$

To allow our verification engine to deal with unknown higher-order functions, we encode higher-order functions into an uninterpreted form. Suppose the type of function f is $x_0 : P_{x_0} \rightarrow \cdots \rightarrow x_n : P_{x_n} \rightarrow P_f$. We encode P_f to be $\{[f/\mathcal{F}]\nu = \mathcal{F}(x_0, x_1, \cdots, x_n)\}$; here, $\mathcal{F}(x_0, x_1, \cdots, x_n)$ is an uninterpreted term representing the result of function. Applications of unknown function f are encoded by substituting actuals for the appropriate (suitably encoded) formal. This gives us the ability to verify a function modularly without having to know the set of definitions referenced by a functional argument or result. For example, for the program shown in Fig. 3.1, the variable `r` in the **let**-binding, `r = g x`, is encoded as $[\mathtt{x}/x_0][\mathtt{g}/\mathcal{F}](\mathcal{F}(x_0))$, which is simply `g x`. The subtyping constraint built for checking the post-condition during the verification of `f`, leads to the construction of the verification condition:

$$((\mathtt{x} \geq 0 \wedge \mathtt{r} = \mathtt{g\ x}) \Rightarrow \nu = \mathtt{r}) \wedge ((\neg(\mathtt{x} \geq 0) \wedge \mathtt{s} \geq 0) \Rightarrow \nu = \mathtt{s}) \Rightarrow (\nu \geq 0)$$

3.3.2 Counterexample-guided Refinement Type Inference

The heart of our counterexample-guided type refinement loop is given in Fig. 3.5. Our refinement algorithm exploits the refinement type template and subtyping constraints generated from type inference rules and finally returns solution $\mathcal{A}$. In `Solve`, our method iteratively type checks each procedure of the given program using the subtyping rules listed in Fig. 2.3 until a fix-point is reached. When a procedure cannot be typed with the set of current refinements, our method supplies the un-

`Refine` $(\Gamma, \mathcal{A}, \{x : P_{T_x} \to P_T\}, \lambda x.e, C) =$

if exists $c \in C$ such that $\mathcal{A}(c)$ is not valid with a witness of ce

then

let $\mathcal{A}' =$ case c of

$\mid \Gamma \vdash \{\nu : B | p_1\} <: \{\nu : B | r_2\} \Rightarrow$

let pred = case r_2 of $\mid p_2 \Rightarrow r_2 \mid \_ \Rightarrow \mathcal{A}[r_2]$

in Strengthen $(\{x : P_{T_x} \to P_T\}, \mathcal{A},$ `wp` $(2,$ ce, pred$), r_2)$

$\mid \Gamma \vdash \{\nu : B \mid \kappa_1\} <: \{\nu : B \mid \kappa_2\} \Rightarrow$

$\mathcal{A}[\kappa_1 \mapsto (\mathcal{A}[\kappa_1]) \wedge (\mathcal{A}[\kappa_2])]$

in `Refine` $(\Gamma, \mathcal{A}', \{x : P_{T_x} \to P_T\}, \lambda x.e, C)$

else $\mathcal{A}$

`Solve` (procedures as List$[\Gamma, \{x : P_{T_x} \to P_T\}, \lambda x.e, C], \mathcal{A}) =$

if exists $(\Gamma, \{x : P_{T_x} \to P_T\}, \lambda x.e, C)$ for a procedure needs to be checked

then

`Solve` (procedures, `Refine` $(\Gamma, \mathcal{A}, \{x : P_{T_x} \to P_T\}, \lambda x.e, C)$)

else $\mathcal{A}$

Figure 3.5.: Counterexample guided type refinement algorithm.

verified procedure's type environment Γ, the current refinement map $\mathcal{A}$, its type template $x : P_{T_x} \to P_T$, the unverified function $\lambda x.e$, and the verification conditions C constructed for the function to `Refine` which can then proceed to strengthen the function's refinement type.

Counterexample generation. Our refinement algorithm first constructs a counterexample *ce* for an unverified verification condition. The counterexample is derived by solving the negation of the desired verification condition:

$$\langle \mathcal{A}(\Gamma) \rangle \wedge \langle \mathcal{A}(r_1) \rangle \wedge \neg \langle \mathcal{A}(r_2) \rangle$$

The encoding of Γ and r_1 reflects path information; by the structure of the rules in Fig. 2.3, the encoding of refinement r_2, on the other hand, reflects a safety property that is implied by $\langle\Gamma\rangle \wedge \langle r_1\rangle$. Thus, an assignment to this formula leads to a counterexample of a possible safety violation; this counterexample path is represented as a straight-line program.

A path expression of the form: "`if` p `then` e_t `else` e_f" is translated to: "`assume` p; e_t" if an assignment from the VC evaluates p to `true` and "`assume` $\neg p$; e_f" otherwise. Consider our example from Fig. 3.1. A first-order decision procedure would find an assignment to the the negation of the VC as an error witness, e.g., `r = -1` and `x = 1`. The representation of the counterexample path of procedure `f` given in Fig. 3.1 is thus:

```
fun f g x = assume (x >= 0); let r = g x in r
```

According to the two different forms of subtyping constraints generated, refinement types can be refined from the counterexample path in one of two ways: weakest precondition generation or procedure specification propagation.

Weakest precondition generation. In this setting, the constraint is of the form: $\Gamma \vdash \{\nu : B \mid p_1\} <: \{\nu : B \mid r_2\}$, corresponding to the first case in `Refine` in Fig. 3.5, where p_1 is a concrete predicate and r_2 is either a concrete predicate or a refinement variable or a selection of refinement variable. This constraint is generated when based typed expression is supplied as function argument or return or establishing assertions. Our type refinement in this case can be implemented by a backward symbolic analysis analogous to weakest precondition generation, operating over a counterexample. Given an arbitrary program construct e, our analysis wp simply pushes up the postcondition δ backwards, substituting terms for values in δ based on the structure of the term e. To ensure termination, recursive functions are unrolled a fixed number of times, defined by the parameter i. As is typical for weakest precondition generation, `wp` ensures that the execution of e, from a state satisfying $\texttt{wp}(i, e, \delta)$, terminates in a state satisfying δ. The definition of `wp` is given in Fig. 3.6.

$\texttt{wp}(i, e, \delta) =$

let $\delta =$ match e with

| $v \rightarrow [e/\nu]\delta$

| $\texttt{op}\,(v_0, \cdots, v_n) \rightarrow [e/\nu]\delta$

| $\texttt{assume}\ v;\ e \rightarrow$

$(v \Rightarrow \texttt{wp}(i, e,\ \delta))$

| $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow$

$((v \wedge \texttt{wp}(i, e_1,\ \delta)) \vee (\neg v \wedge \texttt{wp}(i, e_2,\ \delta)))$

| $\texttt{let}\ x\ =\ e_1$ in $e_2 \rightarrow$

$\texttt{wp}(i, e_1, [\nu/x]\texttt{wp}(i, e_2, \delta)))$

| $f\ \vec{y} \rightarrow$

(match f with

| unknown fun or partial application $\rightarrow [(f\ \vec{y})/\nu]\delta$

| known fun (when $\texttt{let}\ f = \lambda\vec{x}.\ e$) $\rightarrow [\vec{y}/\vec{x}]\texttt{wp}(i, e, \delta)$

| known fun (when $\texttt{let}\ f = \texttt{fix}\,(\,\texttt{fun}\ f \rightarrow \lambda\vec{x}.e)) \rightarrow$

if $i > 0$ then $[\vec{y}/\vec{x}]\texttt{wp}(i-1, e, \delta)$ else $\texttt{false}\,)$

in

if exists $f\ \vec{y}$ in δ and f is a known fun

then $\texttt{wp}(i, f\ \vec{y}, [\nu/(f\ \vec{y})]\delta)$ else δ

Figure 3.6.: Weakest precondition generation definition.

Our `wp` function is standard, extended to deal with unknown function calls but for which context information constraining their arguments or results is available, reflected in the $f\ \vec{y}$ case for an application expression when f is an unknown function

with a list of argument $\vec{y}$ or $f\ \vec{y}$ is a partial application. (The concept of known function and unknown function is defined in Sec. 2.1.) Here, we can only strengthen relevant signatures, deferring the re-verification of the procedure being invoked until it becomes known. The called function's post-condition will be eventually propagated via refinement subtyping chains back to the procedures that flow into this call-site; in doing so, pre-conditions of these functions could be strengthened, requiring re-verification of the calling contexts in which they occur to ensure that these contexts imply the pre-condition. Such flows are handled directly by the subtyping chains analyzed by the refinement phase.

When a function call `f(x)` is encountered and the abstraction to which `f` is bound is known precisely (e.g., based on a syntactic or control-flow analysis pre-processing phase), our method strengthens the post-condition of the function's body of `f` to that available at the call reflected in the last two lines of the definition—if there exists a known function f that has substituted an unknown function in δ (e.g. at a call-site), and $f\ \vec{y}\ \in \delta$ where $\vec{y}$ is a list of arguments, we perform $\texttt{wp}(i, f\ \vec{y}, [\nu/(f\ \vec{y})]\delta)$.

Essentially, `wp` recursively applies our verification technique to refine the function's precondition based on the post-condition defined by the context in which it is called. `wp` can then be executed from this call site operating on the rest of statements of the counterexample beyond the call site and the newly strengthened precondition.

In the $f\ \vec{y}$ case, if f is bound to a known recursive function, since we restrict the number of times a recursive function is unrolled, when $i = 0$, we simply return `false` to avoid considering further unrolling of f; otherwise, the bad-condition δ is directly pushed back to the definition of f in order to drive the sampling for f. In the latter case, the value of i is accordingly decremented. In this chapter, because any inferred specification needs to be verified by the verification algorithm, it suffices to set $i = 2$ (in Fig. 3.5), meaning that recursive functions are unrolled at most two times. If 2 is insufficient, further unrollings are implicitly embedded into the refinement phase for validating specifications inferred when setting i to 2 initially and so on and so forth. When the context is clear, we often write $\texttt{wp}(i, e, \delta)$ as $\texttt{wp}(e, \delta)$ as shorthand.

Consider the example in Fig. 3.1. The post-condition inferred is $\nu \geq 0$. We can infer the precondition shown in Sec. 3.1 by applying our wp rules as follows:

$$\mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0); \mathsf{let}\ \mathtt{r}\ =\ \mathtt{g}\ \ \mathtt{x}\ \mathsf{in}\ \mathtt{r}\,), \nu \geq 0) =$$

$$\mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0); \mathsf{wp}(\mathsf{let}\ \mathtt{r}\ =\ \mathtt{g}\ \ \mathtt{x}\ \mathsf{in}\ \mathtt{r}\,, \nu \geq 0)) =$$

$$\mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0); \mathsf{wp}(\,\mathtt{r}\ =\ \mathtt{g}\ \ \mathtt{x}\,, (\mathsf{wp}(\nu =\ \mathtt{r}\,, \nu \geq 0)))) =$$

$$\mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0); \mathsf{wp}(\,\mathtt{r}\ =\ \mathtt{g}\ \ \mathtt{x}\,, \mathtt{r} \geq 0)) =$$

$$\mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0);\ \mathtt{g}\ \ \mathtt{x} \geq 0) =$$

$$\mathtt{x} \geq 0 \Rightarrow\ \mathtt{g}\ \ \mathtt{x} \geq 0$$

Thus g 's specification is strengthened to $\mathtt{g} : \{\{\nu \geq 0\} \rightarrow \{\nu \geq 0\}\}$.

Procedure specification propagation. In this setting, the subtyping constraint is of the form: $\Gamma \vdash \{\nu\ :\ B \mid \kappa_1\} <: \{\nu\ :\ B \mid \kappa_2\}$, corresponding to the second case in Refine in Fig. 3.5, Refinement variables are introduced when defining refinement type templates; this occurs during inference of function abstraction and fix expressions. Ensuring the subtyping constraint holds requires that any instantiation of κ_2 be propagated to κ_1. This enables refinements associated with the post-condition of a higher order function to be propagated into the real function body, and conversely to propagate refinements associated with a function's pre-condition back to the parameters of higher order function.

Consider how we might verify the program shown in Fig. 3.2. Our method initially infers a refinement type template for f as $\{\{\kappa_{1_1} \rightarrow \kappa_{1_2}\} \rightarrow \kappa_2 \rightarrow \kappa_{\mathtt{f}}\}$. The assertion in main drives a new post-condition $\{\nu \geq 0\}$ for twice, and hence $\mathtt{f}_2$ which is the second the call to f, instantiating $\kappa_{\mathtt{f}}$ to $\{\mathsf{true}\ \oplus\ \nu \geq 0\}$. This constraint is then propagated to the post-condition of neg since neg subtypes to f at the call site of twice in main. The weakest pre-condition backward analysis of our system then strengthens the pre-condition for neg and propagates it back to f, instantiating $\{\kappa_{1_2}\}$ to $\{\mathsf{true}\ \oplus\ \nu\ \leq 0\}$. In twice, our technique needs to ensure, at the second call site of $\mathtt{f}_2$, the actual higher-order function p subtypes to the first argument of f where p is derived from the first call to f notated as $\mathtt{f}_1$. The subtyping relation can

then be expressed as $\Gamma \vdash \{\kappa_2.1 \rightarrow \kappa_{\mathtt{f}}.1\} <: \{\kappa_{1_1}.2 \rightarrow \kappa_{1_2}.2\}$. The post-condition in $\kappa_{1_2}.2$ ($\{\nu \leq 0\}$) is then propagated to $\kappa_{\mathtt{f}}.1$, which becomes $\{\nu \leq 0 \oplus \nu \geq 0\}$. Finally, the context-sensitive type for `f` is derived as

$$\mathtt{f}_{\mathtt{arg}_1} : \{\{\mathsf{true} \oplus \mathsf{true}\} \rightarrow \{\nu \geq 0 \oplus \nu \leq 0\}\} \rightarrow \mathtt{f}_{\mathtt{arg}_2} : \{\mathsf{true} \oplus \mathsf{true}\} \rightarrow \{\nu \leq 0 \oplus \nu \geq 0\}$$

3.3.3 Correctness

We prove that our counterexample-guided refinement algorithm is sound and able to return the weakest solution to discharge all subtyping constraints and verification goals. We firstly relate `wp` computation with the big step semantics of our idealized language, which is given in Fig. 3.7. We define Σ as whole program state space. For a program state $\sigma \in \Sigma$, it is a map from program variables to values. The meaning of $[e, \sigma] \Downarrow \sigma'$ is that the expression e takes state σ to state σ'. We use ν to denote evaluation result of e in σ' and $[e, \sigma] \Downarrow e'$ is a shorthand of $[e, \sigma] \Downarrow (\nu = e')$. Rule $\mathsf{Eval-App1}$ deals with full function application while $\mathsf{Eval-App2}$ deals with partial application.

Definition 3.3.1 (*Partial Correctness Assertion*) *we say $\sigma \models P$ if assertion P is evaluated to* true *by σ. We say $\models \{Q_1\}e\{Q_2\}$ if $\forall \sigma \in \Sigma, \forall \sigma' \in \Sigma, (\sigma \models Q_1 \wedge ([e, \sigma] \Downarrow \sigma')) \Rightarrow (\sigma' \models Q_2)$. Q_1 and Q_2 is the pre- and post-condition of e respectively.*

The following lemma states the validity of `wp` computation. Formally, `wp` is defined recursively over the abstract syntax of statements. Actually, `wp` semantics is a Continuation-passing style semantics of state transformers where the predicate in parameter is a continuation.

Lemma 3 (`wp` *is valid*) *For any expression e, post-condition Q, and state σ and σ', If $[\sigma, e] \Downarrow \sigma'$ and $\sigma' \models Q$ then we have $\sigma \models \mathtt{wp}(e, Q)$.*

Proof Induction on the evaluation judgment $\Downarrow$. ■

$$\frac{[e', \sigma] \Downarrow \sigma' \qquad \mathsf{Eval} - \mathsf{Fun}}{[\texttt{let}\ \ f = \texttt{fix}\,(f, \lambda\vec{x}.\ e)\,\texttt{in}\,e', \sigma] \Downarrow (\sigma'; f \mapsto \lambda\,\texttt{fix}\,(f, \vec{x}.\ e))}$$

$$\frac{x \in dom(\sigma)\ x \text{ is not a function}}{[x, \sigma] \Downarrow (\sigma; \nu \mapsto x)}\ \mathsf{Eval} - \mathsf{Var1}$$

$$\frac{x \in dom(\sigma)\ x \text{ is a function}}{[x, \sigma] \Downarrow (\sigma(x))}\ \mathsf{Eval} - \mathsf{Var2}$$

$$\frac{[f, \sigma] \Downarrow \lambda\vec{x}.e \quad \texttt{arity}\,(\vec{x}) = \texttt{arity}\,(\vec{v}) \quad [e, \sigma; \vec{x} \mapsto \vec{v}] \Downarrow \sigma'}{[f\ \vec{v}, \sigma] \Downarrow \sigma' \quad \mathsf{Eval} - \mathsf{App1}}$$

$$\frac{[f, \sigma] \Downarrow \lambda\vec{x}\vec{y}.\ e \quad \texttt{arity}\,(\vec{x}) = \texttt{arity}\,(\vec{v}) \quad \texttt{arity}\,(\vec{y}) > 0}{[f\ \vec{v}, \sigma] \Downarrow (\sigma; \vec{x} = \vec{v}; \nu \mapsto \lambda\vec{y}.\ e) \quad \mathsf{Eval} - \mathsf{App2}}$$

$$\frac{[v, \sigma] \Downarrow \texttt{true} \quad [e_2, \sigma; v \mapsto \texttt{true}] \Downarrow \sigma'}{[\sigma, \text{if } v \text{ then } e_2 \text{ else } e_3] \Downarrow \sigma'}\ \mathsf{Eval} - \mathsf{If} - \mathsf{True}$$

$$\frac{[v, \sigma] \Downarrow \texttt{false} \quad [e_3, \sigma; v \mapsto \texttt{false}] \Downarrow \sigma'}{[\sigma, \text{if } v \text{ then } e_2 \text{ else } e_3] \Downarrow \sigma'}\ \mathsf{Eval} - \mathsf{If} - \mathsf{False}$$

$$\frac{[e', \sigma] \Downarrow \sigma' \quad [\sigma; x \mapsto \sigma'(\nu), e] \Downarrow \sigma'}{[\text{let } x = e' \text{ in } e, \sigma] \Downarrow \sigma'}\ \mathsf{Eval} - \mathsf{Let}$$

$$\frac{[v, \sigma] \Downarrow \texttt{true}}{[\text{assert } v, \sigma] \Downarrow \sigma}\ \mathsf{Eval} - \mathsf{assertion1}$$

$$\frac{[v, \sigma] \Downarrow \texttt{false}}{[\text{assert } v, \sigma] \Downarrow \bot}\ \mathsf{Eval} - \mathsf{assertion2}$$

Figure 3.7.: Big-step semantics for our iealized language.

Definition 3.3.2 (*Weakest Solution*) *Under a set of verification conditions C, for two solutions $\mathcal{A}$ and $\mathcal{A}'$, we define $\mathcal{A} \leq_C \mathcal{A}'$ if foreach c as $\{\Gamma \vdash \{\nu : B|r_1\} <: \{\nu : B|r_2\}\} \in C$, foreach $\{\nu : B|\kappa\} \in \{R(\Gamma) \cup \{\nu : B|r_1\}\}$, $\mathcal{A}'(\Gamma) \vdash \{\nu : B|\mathcal{A}'(\kappa)\} <: \{\nu : B|\mathcal{A}(\kappa)\}$. $\mathcal{A}^*$ is the weakest solution if*

1. *$\mathcal{A}^*C$ is valid.*

2. *For each valid solution $\mathcal{A}$, $\mathcal{A}^* \leq_C \mathcal{A}$*

Lemma 4 (*Refinement*) *For a given* procedure *as four-tuple* (Γ, $\mathcal{A}$, $\{x : P_x \to P\}$, $\lambda x.e$, C), *if* $\mathcal{A}' =$ `Refine` (Γ, $\mathcal{A}$, $\{x : P_x \to P\}$, $\lambda x.e$, C) *then,*

1. $\mathcal{A} \leq_C \mathcal{A}'$
2. *if* $\mathcal{A}$ *is not valid for* C*, then* $\mathcal{A} \neq \mathcal{A}'$
3. *if* $\mathcal{A}''$ *is valid for* C *and* $\mathcal{A} \leq_C \mathcal{A}''$ *then* $\mathcal{A}' \leq_C \mathcal{A}''$

Proof

1. According to the definition of `Refine`,

$$\mathcal{A}' \equiv \mathcal{A}[\kappa_c \mapsto \mathcal{A}(\kappa) \wedge r_c(\kappa_c)]$$

   i.e., some placeholder κ_c is strengthened with additional refinement $r_c(\kappa_c)$.

   Foreach c as $\{\Gamma \vdash \{\nu : B|r_1\} <: \{\nu : B|r_2\}\} \in C$, foreach $\{\nu : B|\kappa\} \in \{\Gamma \cup \{\nu : B|r_1\}\}$,

   - $\kappa \notin \kappa_c$. $\mathcal{A}'(\kappa) = \mathcal{A}(\kappa)$ so $\mathcal{A}'(\Gamma) \vdash \{\nu : B|\mathcal{A}'(\kappa)\} <: \{\nu : B|\mathcal{A}(\kappa)\}$ holds naturely.
   - $\kappa \in \kappa_c$. $\mathcal{A}'(\Gamma) \vdash \{\nu : B|\mathcal{A}'(\kappa)\} <: \{\nu : B|\mathcal{A}(\kappa)\}$ since $\langle \mathcal{A}(\Gamma) \rangle \wedge \langle \mathcal{A}(\kappa) \wedge r_c(\kappa) \rangle \Rightarrow \langle \mathcal{A}(\kappa) \rangle$ holds.

2. Since $\mathcal{A}$ is not valid for C, there exists c as $(\Gamma \vdash \{\nu : B|r_1\} <: \{\nu : B|r_2\}) \in C$ such that $\mathcal{A}c$ is not valid. From the definition of `Refine`,

$$\mathcal{A}'\Gamma \vdash \mathcal{A}'(r_1) <: \mathcal{A}'(r_2)$$

   From 1,

$$\mathcal{A}'\Gamma \vdash \mathcal{A}'(r_2) <: \mathcal{A}(r_2)$$

   Thus, $\mathcal{A}'\Gamma \vdash \mathcal{A}'(r_1) <: \mathcal{A}(r_2)$.

   If $\mathcal{A} = \mathcal{A}'$ then,

$$\mathcal{A}\Gamma \vdash \mathcal{A}(r_1) <: \mathcal{A}(r_2)$$

   which contradicts the assumption that c is not valid.

3. Since $\mathcal{A}''C$ is valid and $\mathcal{A} \leq \mathcal{A}''$,

   $\forall\ \kappa$, for each c as $\{\Gamma \vdash \{\nu : B|r_1\} <: \{\nu : B|r_2\}\} \in C$,

   $$\mathcal{A}''\Gamma \vdash \{\nu : B|\mathcal{A}''r_1\} <: \{\nu : B|\mathcal{A}''r_2\} \tag{a}$$

   And, for each $\{\nu : B|\kappa\} \in \{R(\Gamma) \cup \{\nu : B|r_1\}\}$,

   $$\mathcal{A}''(\Gamma) \vdash \{\nu : B|\mathcal{A}''(\kappa)\} <: \{\nu : B|\mathcal{A}(\kappa)\} \tag{b}$$

   According to the definition of `Refine`,

   $$\mathcal{A}' \equiv \mathcal{A}[\kappa_c \mapsto \mathcal{A}(\kappa) \wedge r_c(\kappa_c)]$$

   i.e., some placeholder κ_c is strengthened with additional refinement $r_c(\kappa_c)$.

   - $\kappa \notin \kappa_c$. $\mathcal{A}'(\kappa) = \mathcal{A}(\kappa)$ must hold. By (a), so $\mathcal{A}''(\Gamma) \vdash \{\nu : B|\mathcal{A}''(\kappa)\} <: \{\nu : B|\mathcal{A}'(\kappa)\}$
   - $\kappa \in \kappa_c$. In this case $\mathcal{A}'(\kappa) = \mathcal{A}(\kappa) \wedge r_c(\kappa)$
     - Both r_1 and r_2 are in the form of place holder with substitution. Suppose c is $\Gamma \vdash \kappa <: \kappa'$ and $r_c(\kappa) = \mathcal{A}(\kappa')$.

       Since $\mathcal{A}''$ is valid.

       $$\mathcal{A}''\Gamma \vdash \mathcal{A}''\kappa <: \mathcal{A}''\kappa'$$

       Based on (b),

       $$\mathcal{A}''\Gamma \vdash \mathcal{A}''\kappa <: \mathcal{A}\kappa'$$

       Thus,

       $$\mathcal{A}''\Gamma \vdash \mathcal{A}''\kappa <: r_c(\kappa)$$

       Based on,

       $$\mathcal{A}''\Gamma \vdash \mathcal{A}''\kappa <: \mathcal{A}\kappa \wedge r_c(\kappa)$$

       Finally,

       $$\mathcal{A}''\Gamma \vdash \mathcal{A}''\kappa <: \mathcal{A}'\kappa$$

       $\mathcal{A}''(\Gamma) \vdash \{\nu : B|\mathcal{A}''(\kappa)\} <: \{\nu : B|\mathcal{A}'(\kappa)\}$ holds immediately.

– Otherwise $r_c(\kappa)$ is from weakest precondition generation due to invalidity of c. Allow r_2 to be either κ_2 or a simple predicate e_2, by (b),

$$\mathcal{A}''(\Gamma) \vdash \{\nu : B | \mathcal{A}'' r_2\} <: \{\nu : B | \mathcal{A} r_2\} \quad \text{(c)}$$

By (a) and (c),

$$\mathcal{A}''\Gamma \vdash \{\nu : B | \mathcal{A}'' r_1\} <: \{\nu : B | \mathcal{A} r_2\} \quad \text{(d)}$$

According to (d), the semantics of `wp` and Lemma 3,

$$\mathcal{A}''(\Gamma) \vdash \{\nu : B | \mathcal{A}''(\kappa)\} <: \{\nu : B | r_c(\kappa)\} \quad \text{(e)}$$

By (b), (e),

$$\mathcal{A}''(\Gamma) \vdash \{\nu : B | \mathcal{A}''(\kappa)\} <: \{\nu : B | \mathcal{A}(\kappa) \wedge r_c(\kappa)\}$$

So $\mathcal{A}''(\Gamma) \vdash \{\nu : B | \mathcal{A}''(\kappa)\} <: \{\nu : B | \mathcal{A}'(\kappa)\}$ holds.

Finally, $\mathcal{A}' \leq_C \mathcal{A}''$.

■

Theorem 3.3.1 (Iterative Solve) *For a given list of* procedures *each as a four-tuple* (Γ, $\mathcal{A}$, $\{x : P_x \rightarrow P\}$, $\lambda x.e$, C) *with an arbitrary coarse solution* $\mathcal{A}_{any}$, *if* `Solve` *terminates and* `Solve` *returns* $\mathcal{A}$ *then* $\mathcal{A}$ *is the weakest solution.*

Proof The termination of `Solve` means a solution $\mathcal{A}$. Let $\mathcal{A}^*$ be the weakest solution. We prove by induction over n that after n iterations of the loop in `Solve`, $\mathcal{A} \leq \mathcal{A}^*$. In the base case, $\mathcal{A}$ maps each place holder to *true* and thus it is obvious $\mathcal{A} \leq \mathcal{A}^*$. Assume that after n iterations, $\mathcal{A} \leq \mathcal{A}^*$. In the $n+1$ iteration, obtain $\mathcal{A}'$ = `Refine` (Γ, $\mathcal{A}$, $\lambda x.e$, $\{x : P_x \rightarrow P\}$, C). Due to that $\mathcal{A}^*$ is valid and $\mathcal{A} \leq \mathcal{A}^*$, $\mathcal{A}' \leq_C \mathcal{A}^*$ hold based on Lemma 4. As the $n+1$ iteration only update place holders involved in C, $\mathcal{A}' \leq \mathcal{A}^*$. Finally, the returned solution $\mathcal{A}$ from `Solve` satisfies $\mathcal{A} \leq \mathcal{A}^*$. However, as $\mathcal{A}^*$ is the weakest solution, so by definition $\mathcal{A}^* \leq \mathcal{A}$. Thus, $\mathcal{A}$ is a valid solution such that $\mathcal{A} = \mathcal{A}^*$ ■

We provide two correctness results for our verification algorithm `Solve` (procedures as List[Γ, $\{x : P_{Tx} \rightarrow P_T\}$, $\lambda x.e$, C], $\mathcal{A}_{init}$) where $\mathcal{A}_{init}$ is an initial solution for unknown refinement type variables in which the unknowns in function types are mapped to `true`. The first (*Soundness*) states that the refinement types inferred by our verification procedure are consistent with our type rules. The second (*Weak*) states that our procedure generates the least type necessary to discharge the subtyping constraints collected by the inference algorithm. In the following, $R(\Gamma)$ recursively extracts refinement base types $\{\nu : B|\kappa\}$ from the domain of Γ.

Theorem 3.3.2 (Verification Algorithm)

1. (*Soundness*) *Let* $\mathcal{A}$ *be the result of* `Solve` (procedures as List[Γ, $\{x : P_{Tx} \rightarrow P_T\}$, $\lambda x.e$, C], $\mathcal{A}_{init}$). *Then, provided* `Solve` *terminates,* $\mathcal{A}(\Gamma) \vdash \lambda x.e : \{x : \mathcal{A}(P_{Tx}) \rightarrow \mathcal{A}(P_T)\}$.

2. (*Weak*) *And, for all other valid solution* $\mathcal{A}'$, *the algorithm generates the weakest solution:* $\forall c$ *as* $\{\Gamma \vdash \{\nu : B|r_1\} <: \{\nu : B|r_2\}\} \in C$, *and* $\forall \{\nu : B|\kappa\} \in \{R(\Gamma) \cup \{\nu : B|r_1\}\}$, $\mathcal{A}'(\Gamma) \vdash \{\nu : B|\mathcal{A}'(\kappa)\} <: \{\nu : B|\mathcal{A}(\kappa)\}$.

Proof

1. Immediate from Theorem 2.2.1.

2. Immediate from Theorem 3.3.1.

■

3.3.4 Invariant Generation

Because our technique does not guarantee termination given the undecidability of automatically synthesizing loop invariants, the size of a refinement function type may grow into an infinite representation, and a fixed-point may never be reached. Consider the ML program fragment shown in Fig. 3.8 adapted from [9]. The procedure `iteri`

```
fun iteri i xs f =
  case xs of
    [ ] => ()
  | x :: xs' =>  (f i x; iteri (i+1) xs' f)

fun mask a xs =
  let g j y = ··· y ··· Array.sub (a, j) ··· in
    if Array.length a = List.length xs then
      iteri 0 xs g
    else () end
```

Figure 3.8.: A program that has a non-trivial loop invariant.

visits the elements of a list `xs`, applying function `f` to each element and its index in the list. Procedure `mask` calls `iteri` when the length of its array and list arguments are the same. It supplies function `g` as the higher-order argument to `iteri` which performs some computation involving a list and array element at the same index. We desire to verify the array bound safety property $\texttt{j} < \mathsf{len}(\texttt{a})$ for the array access in procedure `g` (Note $\texttt{j} \geq 0$ can be directly proved by our method introduced in Sec. 3.3.2).

During the course of verifying this program, we would need to discharge a specification that forms a pre-condition for `iteri` asserting that $\mathsf{len}(\texttt{xs}) \neq 0 \Rightarrow \texttt{i} < \mathsf{len}(\texttt{a})$. However, verifying this specification requires a theorem prover to conclude that $\mathsf{len}(\texttt{xs})\text{-}1 \neq 0 \Rightarrow \texttt{i}+1 < \mathsf{len}(\texttt{a})$ as precondition for the recursive call to `iteri (i+1) xs'`. In trying to discover a counter-example to this claim, a theorem prover would likely generate an infinite number of pre-conditions, $\mathsf{len}(\texttt{xs}) - k \neq 0 \Rightarrow \texttt{i} + k < \mathsf{len}(\texttt{a})$ where $k = 0, 1, 2 \cdots$ What is required is a sufficiently strong invariant that can be used to validate the required safety properties. While programmers could certainly write such specifications if necessary, we follow the idea of interpolation-based model-checking [22] to automatically infer them when possible.

```
fun iteri0 i0 xs0 f0 =
  case xs0 of
    [ ] => ()
  | x0 :: xs0' =>  (f0 i0 x0; iteri1 (i0+1) xs0' f0)

fun iteri1 i1 xs1 f1 =
  case xs1 of
    [ ] => ()
  | x1 :: xs1' =>  (f1 i1 x1; iteri2 (i1+1) xs1' f1)

fun iteri2 i2 xs2 f2 = halt
```

Figure 3.9.: Unrolling a recursive function for invariant discovery using interpolation.

When our mainline verification algorithm diverges or reaches a pre-determined timebound during the analysis of a recursive procedure, it is unrolled incrementally together with its calling context. Our method then infers refinement type templates and generates subtyping constraints for the k-unrolled procedures. Pre-conditions of the higher order functions used in recursive procedure are propagated via subtyping chain from that of the real function they represent for. Post-conditions of the higher order functions are also propagated from that of the real function which can be obtained from our type inference algorithm. We then exploit a technique described in [23] to infer refinement types from the collected base subyping constraints. The basic idea is to use the interpolation of the first-order formulas derived from the subtyping constraints to deduce an instantiation for a given type refinement variable κ. We desire that the prover returns a more suitable refinement beyond that yielded by a weakest precondition generator. Refinements synthesized from k-unrolled non-recursive procedures are folded back to the original procedure as candidates.

For example, suppose our method discovers that it must unroll the recursive procedure `iteri` two times, obtaining the program shown below:

Here, `halt` is a special term, representing a termination point. Because we maintain the original calling context of `iteri`, we have $\mathsf{len}\ \mathtt{a} = \mathsf{len}\ \mathtt{xs}$ in the typing environment and leverage subtyping constraints to establish that the actual `g` subtypes to the formal `f0`. We infer refinements for this unrolled excerpt using the obtained base subtyping constraints. We thus have the following subtyping constraint:

$$\mathtt{i1} : \kappa_{\mathtt{i1}}, \mathtt{i0} : \kappa_{\mathtt{i0}}, \mathtt{xs0} : \kappa_{\mathtt{xs0}}, \mathsf{len}\,(\mathtt{xs0}) = \mathsf{len}\,(\mathtt{xs0'}) + 1, \mathsf{len}\,(\mathtt{xs}) = \mathsf{len}\,(\mathtt{a})$$
$$\vdash \{\nu = \mathtt{xs0'}\} <: \kappa_{\mathtt{xs1}}$$

that establishes that the actual `xs0'` given to `iteri1` subtypes to the formal `xs1`. In the body of `iteri1`, there is another constraint for the call to `f1 i1`:

$$\mathtt{i1} : \kappa_{\mathtt{i1}}, \mathtt{xs1} : \kappa_{\mathtt{xs1}}, \mathsf{len}\,(\mathtt{xs1}) = \mathsf{len}\,(\mathtt{xs1'}) + 1 \vdash \{\nu' = i1\} <: \{\nu' < \mathsf{len}\,(\mathtt{a})\}$$

Because we have already inferred the refinement type for procedure `g` before typing `iteri` and obtained precondition $\nu' <$ `len(a)` for its first argument, we can use it to also serve as the precondition of the first argument of `f1` propagated through the subtyping chains.

We extend the above constraints into first order logic formulas:

$$\{\mathtt{i1} = \mathtt{i0} + 1 \ \wedge\ \mathtt{i0} = 0 \ \wedge\ \mathtt{xs0} = \mathtt{xs} \ \wedge\ \mathsf{len}\,(\mathtt{xs0}) = \mathsf{len}\,(\mathtt{xs0'}) + 1 \wedge$$
$$\mathsf{len}\,(\mathtt{xs}) = \mathsf{len}\,(\mathtt{a}) \ \wedge\ \nu = \mathtt{xs0'}\}^{(a)} \Rightarrow \kappa_{xs1}$$
$$\kappa_{xs1} \Rightarrow \{\mathtt{i1} = \mathtt{i0} + 1 \ \wedge\ \nu = \mathtt{xs1} \ \wedge\ \mathsf{len}\,(\mathtt{xs1}) = \mathsf{len}\,(\mathtt{xs1'}) + 1 \wedge$$
$$\nu' = \mathtt{i1} \Rightarrow \nu' < \mathsf{len}\,(\mathtt{a})\}^{(b)}$$

The unknown refinement represented by $\kappa_{\mathtt{xs1}}$ is indeed an interpolation of formula (a) and formula (b) and can be inferred by feeding them into an appropriate interpolation theorem prover [22] which may return $\mathsf{len}(\nu)$ + `i1` = len `(a)` as result. Our method then yields $\mathsf{len}(\nu)$ + `i` = len `(a)` (discarding subscript) as a refinement candidate of the second argument `xs` of procedure `iteri`.

After candidate refinement synthesis, our method then applies an elimination procedure [7] to filter out incorrect candidates. If the original procedure is still not

typable, the process is repeated, unrolling it $k+1$ times. For this example, with the above refinement candidate, we can correctly verify the pre-condition of `f` in `iteri`. Since the theorem prover can use the case condition to know $\mathsf{length}(\texttt{xs}) > 0$ and based on the invariant $\texttt{i} + \mathsf{len}(\texttt{xs}) = \mathsf{len}(\texttt{a})$, it can determine that $\texttt{i} < \mathsf{len}(\texttt{a})$ must hold. Our method finally generates the appropriate refinement type for `iteri` as:

$$\begin{aligned}\texttt{iteri} : \texttt{i} : \mathsf{int} \rightarrow \{\texttt{xs} \ : \ '\texttt{a}\ \mathsf{list} | \ \texttt{i} + \mathsf{len}(\nu) = \mathsf{len}(\texttt{a})\} \rightarrow \\ \{\texttt{f} \ : \ \{\texttt{f}_{\texttt{arg}_1} \ : \ \mathsf{int} \mid 0 \leq \nu < \mathsf{len}(\texttt{a})\} \rightarrow '\texttt{a} \rightarrow \mathsf{unit}\} \rightarrow \mathsf{unit}\end{aligned}$$

Note the invariant generation module is only invoked when our system diverges during the verification of a recursive procedure. We differ from [23] in two respects: first, [23] does not use an elimination procedure since it tries to infer refinement types for the original program using a whole program analysis; second, we only infer refinement candidates for a non-recursive unrolled code fragment instantiated upon divergence, instead of the original whole program, greatly reducing the number of instances where interpolation computation is required.

3.4 Implementation

We have implemented our verification system in Popeye. Popeye takes as input an SML program (not necessarily closed) and outputs specifications inferred for the procedures defined by the program. We have provided specifications for built-in primitive datatypes as well as arrays, lists, tuples, and records that are used to bootstrap the inference procedure. The Yices theorem prover is used as the verification engine. CSIsat [24] is employed to generate interpolations when inferring candidate refinements for recursive procedures and loops. The implementation is incorporated within the MLton whole-program optimizing compiler toolchain and consists of roughly 14KLOC written in SML[1].

[1]The Popeye implementation is available at `http://code.google.com/p/popeye-type-checker/`

3.4.1 Case Study: Bit Vectors

To gauge POPEYE's utility, we applied it to an open-source bit vector library (BITV) [25] (version 0.6). A bit vector is represented as a record of two fields, `bits`, an array containing vector's elements, and `length`, an integer that represents the number of bits that the vector holds. Operations on bit vectors should enforce the invariant that $(\texttt{bits.length} - 1) \cdot \texttt{b} < \texttt{bits.length} \cdot \texttt{b}$, where `b` is a constant that defines the number of bits intended to be stored per array element. This invariant is assumed for all procedures. POPEYE successfully type checks the program combined with 5 manually generated preconditions (for recursive procedures as prover [24] cannot deal with mod operation heavily used in the library) by relatively longer verification time than that of DSOLVE [7] in this benchmark; however DSOLVE requires manual addition of extra 14 user-supplied qualifiers.

Bug Detection. Without any programmer annotations, POPEYE discovered an array out-of-bounds error that occurs in the `blit` function:

```
fun blit {bits=b1, length=l1} {bits=b2, length=l2}
         ofs1 ofs2 n =
  if n < 0 || ofs1 < 0 || ofs1 + n > l1
           || ofs2 < 0 || ofs2 + n > l2
  then assert false
  else unsafe_blit b1 ofs1 b2 ofs2 n
```

This function calls `unsafe_blit` only if a guard condition that checks that all offset value and the number of bits (`n`) to be copied are positive, and that the range of the copy fit within the bounds of the source and target vectors. The counterexample reported for `blit` procedure corresponds to an input as {`length(b1)`=2, `length(b2)`=0, `l1=60`, `ofs1=32`, `l2=0`, `ofs2=0`, `len=0`}. The guard holds under this assignment, but because `unsafe_blit` attempts to access the offset in the target bit-vector that is the starting point for the copy, before initiating the copy loop, an array out-of-bounds exception gets thrown. In this example, POPEYE reports

a test case that serves as a witness to the bug, and can help direct the programmer to identify the source of the error. The primary novelty of this technique in this regard is its ability to generate a precise counterexample path with concrete inputs that serve as a witness to the violation without requiring explicit user confirmation as DSOLVE.

Complex Refinement Generation. Procedure `unsafe_blit` found in this library tries to copy `n` bits starting at offset `ofs1` from bit-vector `v1` to bit-vector `v2` with target offset `ofs2`. POPEYE discovers the following precondition:

$$((\texttt{ofs2} + \texttt{n}) - 1)/\texttt{b}) < \texttt{v2.length}$$

This is a non-trivial specification comprised of refinements that we believe would be difficult, in general, for programmers to construct. Systems such as DSOLVE require users to provide these qualifiers explicitly. The ability to generate non-trivial refinements automatically only using counterexamples is an important distinguishing feature of our approach compared to e.g., LIQUIDTYPES [7].

3.4.2 Experimental Results

To test its accuracy, we have applied POPEYE to a number of synthetic SML programs from the benchmark suite used to evaluate MOCHI [13]. While these benchmarks are small (typically less than 100 LOC), they exercise complex control- and dataflow, and exploit higher-order procedures heavily, in ways intended to make refinement type inference challenging. Details of these benchmarks are provided in [13]. In the table, column num_ref denotes the number of refinements discovered by POPEYE. num_cegar shows how many iterations of the refinement loop were necessary for POPEYE to converge. prover_call gives the number of theorem prover calls; there are typically more prover calls than CEGAR loop iterations because the results of a counterexample usually entails propagation of newly discovered invariants to other contexts, thus requiring re-verification (and hence additional theorem prover calls). cegar_time shows the time spent on refinement loops. run_time gives the total running time taken.

Program	num_ref	num_cegar	prover_call	cegar_time	run_time
fhnhn	3	4	35	0s	0.014s
neg	15	20	230	0.004s	0.18s
max	10	11	175	0.005s	0.95s
r-file	11	21	205	0.012s	1.56s
r-lock	10	18	108	0.006s	0.60s
r-lock-e	13	18	113	0.01s	0.68s
repeat-e	39	18	237	0.11s	4.87s
list-zip	2	4	149	0.01s	1.55s
array-init	35	106	3617	0.03	102.3s

Figure 3.10.: POPEYE benchmark results.

The first seven benchmarks shown in Fig. 3.10 cannot be verified by DSOLVE using its default set of simple qualifiers since either context-sensitive refinement types or non-trivial invariants are required. The last two of these seven (suffixed with `-e`) are buggy, and thus cannot cannot also be automatically proved by DSOLVE. The last two benchmarks requires recursive procedure invariants which can be synthesized by our invariant generation module. Here, a single unrolling of the recursive procedure in repeat-e was sufficient to witness the error; in contrast, POPEYE required three unrollings of the recursive procedure in array-init to find a suitable set of candidate refinements. We note that MOCHI fails to verify the `array-init` program. While MOCHI can also verify the first eight benchmarks in this table, its formulation is a bit more complex than ours, and does not easily generalize to deal with data structures and user-level datatypes.

3.5 Related Work

There has been much work on the use of refinement types for checking complex safety properties of ML programs. Freeman and Pfenning [2] describe a refinement type inference scheme defined in terms of an abstract interpretation over a programmer-specified lattice of refinements for each ML type, and a restricted use of intersection types to combine these refinements that still preserves decidability of type inference. DML [3] is a conservative extension of ML's type system that supports type checking of programmer-specified refinement types; the system supports a form of partial type inference whose solution depends upon the set of refinements found in a linear constraint domain.

To reduce the annotation burden imposed by systems like DML, LiquidTypes [7, 8] requires programmers to only specify simple candidate qualifiers from which more complex refinement types defined as conjunctions of these refinements are inferred by a whole program abstract interpretation. Our approach differs from liquid types in four important respects: (1) we attempt to infer refinements, (2) a counterexample path together with a test case can be reported as a program bug witness; (3) the type refinement fixpoint loop enables compositional verification, propagating specifications via refinement subtyping chains on demand; (4) the refinement types we inferred are context-sensitive.

Broadly related to our goals, HMC [9] also borrows techniques from imperative program verification to verify functional programs. It does so by reducing the problem of checking the satisfiability of the constraints generated in a liquid type system to a safety checking problem of a simple imperative program. However, the translated imperative program loses the structure of the original source semantics. Thus, it is not obvious how we might convert a counterexample reported in the translated program into the original source for debugging.

Terauchi [19] also proposes a counterexample-guided refinement type inference scheme, albeit based on a whole-program analysis. A counterexample in his approach

is an "unwound" slice of the program that is untypable using the current set of candidate types, rather than a counterexample path. Since the unfolded program may be involved in multiple program paths, many of which may not be relevant to the verification obligation, it would appear that the size of the constraint sets that needs to be solved may become quite large.

There has been much recent interest in using higher-order recursion schemes [11, 12] to define expressive model-checkers for functional programs. In [13, 14], predicate abstraction is proposed to abstract higher-order program with infinite domains like integers to a finite data domain; the development in these papers is limited to pure functional programs without support for data structures. Model checking arbitrary μ-calculus properties of finite data programs with higher order functions and recursions can be reduced to model checking for higher-order recursion schemes [11]. Finding suitable refinements relies on a similar constraint solving to [19, 23] for a straight-line higher-order counterexample program. Such techniques involve substantial re-engineering of first-order imperative verification tools to adapt them for a higher-order setting.

One important motivation for our work is to reuse well-studied imperative program verification techniques. For example, predicate abstraction [15] has been effectively harnessed by tools such as SLAM [26] and BLAST [27] to verify complex properties of imperative programs with intricate shape and aliasing properties. Software verification tools, such as Boogie [28], ESC/Java [29] and CALYSTO [30] construct first order logic formula to encode a program's control flow. If a verification condition, expressed via programmer-specified assertions or specifications, cannot be discharged, the counterexample path can be used to refine and strengthen it.

4 LEARNING BASED REFINEMENT TYPE INFERENCE

Refinement types and random testing are two seemingly disparate approaches to build high-assurance software for higher-order functional programs. Refinement types allow the static verification of critical program properties (e.g. safe array access). Refinement type systems such as DML [3] and LiquidTypes [7] have demonstrated their utility in validating useful specifications of higher-order functional programs.

On the other hand, random testing, exemplified by systems like QuickCheck [31], can be used to define useful underapproximations, and has proven to be effective at discovering bugs. However, it is generally challenging to prove the validity of program assertions, as in the program shown above, simply by randomly executing a bounded number of tests.

Tests (which prove the existence, and provide conjectures on the absence, of bugs) and types (which prove the absence, and conjecture the presence, of bugs) are two complementary techniques for understanding program behavior. They both have well-understood limitations and strengths. It is therefore natural to ask how we might define synergistic techniques that exploit the benefits of both.

Approach. We present a strategy for automatically constructing refinement types for higher-order program verification. The input to our approach is a higher-order program $\mathcal{P}$ together with $\mathcal{P}$'s safety property ψ (e.g. annotated as program assertions). We identify $\mathcal{P}$ with a set of sampled program states. "Good" samples are collected from test runs; these are reachable states from concrete executions that do not lead to a runtime assertion failure that invalidates ψ. "Bad" samples are states generated from symbolic executions which would produce an assertion failure, and hence should be unreachable; they are synthesized from a backward symbolic execution, structured to traverse error paths not explored by good runs. The goal is to learn likely invariants φ of $\mathcal{P}$ from these samples. If refinement types encoded from

φ are admitted by a refinement type checker and ensure the property ψ, then $\mathcal{P}$ is correct with respect to ψ. Otherwise, φ is assumed as describing (or fitting) an insufficient set of "Good" and "Bad" samples. We use φ as a counterexample to drive the generation of more "Good" and "Bad" samples. This counterexample-guided abstraction refinement (CEGAR) process [16] repeats until type checking succeeds, or a bug is discovered in test runs.

There are two algorithmic challenges associated with our proof strategy: (1) how do we sample good and bad program states in the presence of complex higher-order control and dataflow? (2) how do we ensure that the refinement types deduced from observing the sampled states can generally capture both the conditions (a) sufficient to capture unseen *good* states and (b) necessary to reject unseen *bad* ones?

The essential idea for our solution to (1) is to encode the unknown functions of a higher-order function (e.g. function `m` in Fig. 1.1) as uninterpreted functions, hiding higher-order features from the sampling phase. Our solution to (2) is based on learning techniques to abstract properties classifying good states and bad states derived from the CEGAR process, without overfitting the inferred refinement types to just the samples.

Implementation. We have implemented a prototype of our framework built on top of the ML type system. The prototype can take a higher-order program over base types and polymorphic recursive data structures as input, and automatically verify whether the program satisfies programmer-supplied safety properties. We have evaluated our implementation on a set of challenging higher-order benchmarks. Our experiments suggest that the proposed technique is lightweight compared to a pure static higher-order model checker (e.g. MoCHI [13]), in producing expressive refinement types for programs with complex higher-order control and data flow. Our prototype can infer invariants comprising *arbitrary Boolean combinations* for recursive functions in a number of real-world data structure programs, useful to verify non-trivial data structure specifications. Existing approaches (e.g. LiquidTypes [7]) can verify these

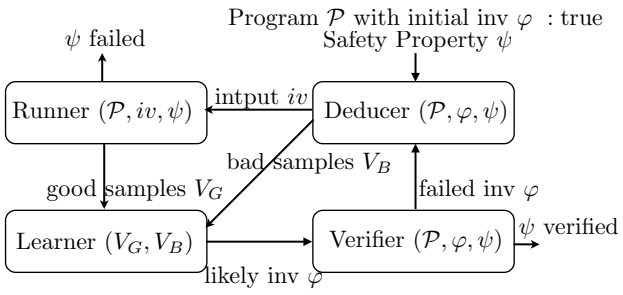


Figure 4.1.: Learning based refinement type inference.

programs only if such invariants are manually supplied which can be challenging for programmers.

Contributions. This chapter makes the following contributions:

- A CEGAR-based learning framework that combines testing with type checking, using tests to exercise error-free paths and symbolic execution to capture error-paths, to automatically infer expressive refinement types for program verification.
- An integration of a novel learning algorithm that effectively bridges the gap between the information gleaned from samples to desired refinement types. Notably, the precision of intersection types [19] are recovered in our learning approach, and allows us to infer context-sensitive invariants.
- A description of an implementation, along with experimental results over a range of complex higher-order programs, that validates the utility of our ideas.

4.1 Overview

This section describes the framework of our approach, outlined in Fig. 4.1. Our technique takes a (higher-order) program $\mathcal{P}$ and its safety property ψ as input. To bootstrap the inference process, the initial program invariant φ is assumed to be `true`. A Deducer (a) uses backward symbolic execution starting from program states

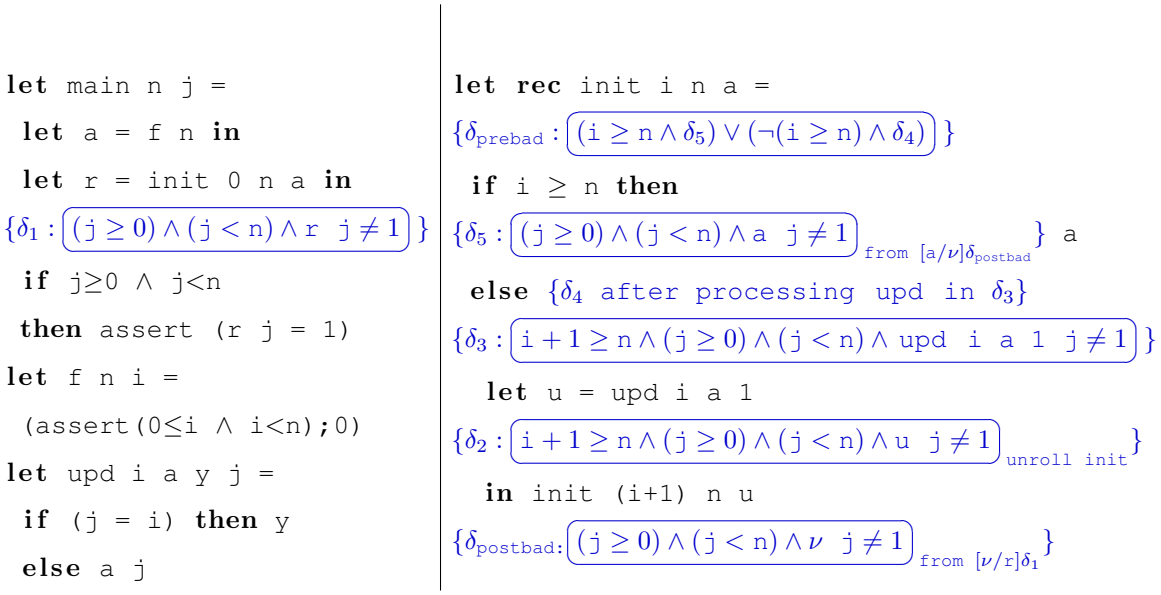


Figure 4.2.: A higher-order program and its bad-conditions (in the blue box).

that violate ψ, to supply bad sample states at all function entries and exits, i.e., those that reflect error states of $\mathcal{P}$, sufficient to trigger a failure of ψ. A Runner (b) runs $\mathcal{P}$ using randomly generated tests, and samples good states at all function entries and exits. These good and bad states are fed to a Learner (c) that builds classifiers from which likely invariants φ (for functions) are generated. Finally a Verifier (d) encodes the likely invariants into refinement types and checks whether they are sufficient to prove the provided property. If the inferred types fail type checking, the failed likely invariants φ are transferred from the Verifier to the Deducer, which then generates new sample states based on the cause of the failure. Our technique thus provides an automated CEGAR approach to lightweight refinement type inference for higher-order program verification.

In the following, we consider functional arguments and return values of higher-order functions to be *unknown functions*. All other functions are *known functions*.

Example. To illustrate, the program shown in the left column of Fig. 4.2 makes heavy use of unknown functions (e.g., the functional argument `a` of `init` is an unknown function). In the function `main`, the value for `a` supplied by `f` is a

closure over `n`, and when applied to a value `i`, it checks that `i` is non-negative but less than `n`, and returns 0. The function `init` iteratively initializes the closure `a`; in the `i`-th iteration the call to `upd` produces a new closure that is closed over `a` and yields 1 when supplied with `i`. Our system considers program safety properties annotated in the form of assertions. The assertion in `main` specifies that the result of `init` should be a function which returns 1 when supplied with an integer between [0, `n`).

Verifying this program is challenging because a proof system must account for unknown functions. The program also exhibits complex dataflow (e.g., `init` can create an arbitrary number of closures via the partial application of `upd`); thus, any useful invariant of `init` must be inductive. We wish to infer a useful refinement type for **`init`**, consistent with the assertions in `main` and `f` without having to know the concrete functions that may be bound to `a` *a priori* (note that `a` is dynamically updated in each recursive iteration of `init`).

Hypothesis domain. Assume that f is higher-order function and $\Theta(f)$ includes all the arguments (or parameters) and return variables bound in the scope of f. For each variable $x \in \Theta(f)$, if x presents an unknown function, we define $\Omega(x) = [x_0; x_1; \cdots; x_r]$ in which the sub-indexed variables are the arguments (x_0 denotes the first argument of x) and x_r denotes the return of x. Otherwise, if x is a base typed variable, $\Omega(x) = [x]$. We further define $\Omega(f) = \bigcup_{x \in \Theta(f)} \Omega(x)$. We consider refinement types of f with type refinements constructed from the variables in $\Omega(f)$. For example, $\Omega($ `init` $)$ includes variables `i`, `n`, $\mathtt{a_0}$, $\mathtt{a_r}$ where $\mathtt{a_0}$ and $\mathtt{a_r}$ denote the parameter and return of `a`.

Assume $\{y_1, \cdots, y_m\}$ are numeric variables bound in $\Omega(f)$. In this paper, following LiquidTypes [7], to ensure decidable refinement type checking, we restrict type refinements to the decidable logic of linear arithmetic. Formally, our system learns type refinements (invariants for function f) which are arbitrary Boolean combination of predicates in the form of Equation 4.1:

$$c_1 y_1 + \cdots + c_m y_m + d \ \leq \ 0 \tag{4.1}$$

where $\{c_1, \cdots, c_m\}$ are integer coefficients and d is an integer constant. Our hypothesis domain is parameterized by Equation 4.1.

To deliver a practical algorithm, we define $\mathcal{C} = \{-1, 0, 1\}$ and $\mathcal{D}$ as the set of the constants and their negations that appear in the program text of f and requires that all the coefficients $c_i \in \mathcal{C}$ and $d \in \mathcal{D}$. We define two helper functions used throughout the paper.

$$\texttt{min}(y_1, \cdots, y_m) = \min_{\forall i.c_i \in \mathcal{C}.\ d \in \mathcal{D}} \{c_1 y_1 + \cdots + c_m y_m + d\} - 1$$

$$\texttt{max}(y_1, \cdots, y_m) = \max_{\forall i.c_i \in \mathcal{C}.\ d \in \mathcal{D}} \{c_1 y_1 + \cdots + c_m y_m + d\} + 1$$

We now exemplify the execution flow presented in Fig. 4.1 by learning an invariant for function `init`.

Deducer. Any invariant inferred for `init` must be sufficiently strong to prevent assertion violations. Using assertions in the program, we perform a backward symbolic analysis (defined in Sec. 4.2), to capture *bad* runs, the pre- and post conditions of a known function sufficient to lead to an assertion failure, which we call its *pre-* and *post-bad conditions*. Bad program states are sampled as (SMT) solutions to such conditions. Program states at a function's entry and exit are called its *pre-* and *post-states*.

Consider the bad-conditions in the boxes in the program in Fig. 4.2, generated by a backward symbolic analysis from the assertion in `main` to the call to `init`. To capture bad conditions that cause failures, we negate the assertion, incorporating the path condition before the assertion in δ_1. Substituting ν (syntactic sugar for the value of an expression) for `r` in δ_1, we obtain δ_{postbad} which denotes the post-bad condition for `init`. δ_5 instantiates ν in δ_{postbad} to the real return variable `a` for the `then` branch of the `if`-expression; assume the bad-condition for the `else` branch is δ_4, we then infer the pre-bad condition for `init` as δ_{prebad}. Notably, in this process, we consider unknown functions as uninterpreted (e.g. `a` in δ_5), allowing us to generate useful constraints over their input (e.g. `j`) and output (e.g. `a j`). As a result, bad samples for `init` can be queried from δ_{prebad} and δ_{postbad}, using SMT solvers with

decidable first-order logic with uninterpreted functions [32]. Recursive functions are unrolled twice in this example as reflected by δ_2.

Consider how we might generate a useful precondition for `init`. Recall that $\mathtt{a}_0$ and $\mathtt{a}_\mathtt{r}$ denote the parameter and return values of the unknown function `a` within `init`. The *bad* pre-states, sampled from δ_{prebad} for `init`, are listed as entries under label **B** in Fig. 4.3a. Our symbolic analysis concludes, in the absence of proper constraints on `init`'s inputs, that an assertion violation in `main` occurs if the call to the closure `a` with 0 returns either `-1` or `2` when the iterator `i` is already increased to `1`.

Furthermore, the symbolic analysis for an unknown function is deferred until a known function to which it is bound (say, at a call-site) is supplied. The conditions defined for the unknown function that lead to assertion failures can eventually be propagated to the known function and used for deriving its bad samples. This is demonstrated in δ_3, where the unknown function `u` is substituted with the function `upd`, which can drive sampling for `upd`.

Deducer is also used to provide a test input for Runner based on failed invariants as counterexamples. For the initial case, we use random testing to "seed" the inference process.

Runner. Our test infrastructure instruments the entry and exit of function bodies to log values of program variables into a log-file; these values represent a coarse underapproximation of a function's pre- and post-state. For example, with a random test input, we might invoke `main` by supplying `4` as the argument for `n` and `0` as the argument for `j`. When `init` is invoked from `main`, we record the binding for its parameters, `i` to 0 and `n` to 4. The values for arguments `i` and `n` can be used to build a coarse specification. The question is how do we seed a specification for `a`, without tracking its flow to and from `upd`, which happens within a series of recursive calls to `init`? Note that the argument to the application of `a` takes place in `upd` but not `init`. To realize an efficient analysis, we sample the unknown function `a` by calling it with inputs from [`min ( i , n )`, ..., `max ( i , n )`] in the instrumented code, with the expectation that its observed input/output pairings can be subsequently

abstracted into a relation defined in terms of `i` and `n`. Note that, at run-time, the values of `i` and `n` are known. This design is related to the hypothesis domain and function `min` and `max` are exactly defined according to the hypothesis domain (see their definitions above).

In Fig. 4.3a, entries labeled under G represent *good* pre-states at the entry of `init`; these states lead to a terminating execution that does not trigger an assertion failure. In the second iteration of the function `init`, we record that function `a` returns 1 when supplied with 0. This corresponds to the good sample in the first row in the table; at this point, the closure `a` has been initialized such that $(\texttt{a}\ 0) = 1$ and $(\texttt{a}\ \texttt{a}_0) = 0$ for $0 < \texttt{a}_0 < \texttt{n}$.

Learner. A classifier that admits all good samples and prohibits all bad ones is considered a likely invariant. We rely on predicate abstraction [15] to build these classifiers. For illustration, consider a subset of the atomic predicates obtained from Equition 4.1 (simplified for readability): $\Pi_0 \equiv \texttt{a}_0 \geq 0, \Pi_1 \equiv \texttt{a}_0 < \texttt{n}, \Pi_2 \equiv \texttt{a}_\texttt{r} < \texttt{n}, \Pi_3 \equiv \texttt{a}_0 < \texttt{i}, \Pi_4 \equiv \texttt{a}_\texttt{r} < \texttt{i}, \Pi_5 \equiv \texttt{i} < \texttt{n}, \Pi_6 \equiv \texttt{a}_\texttt{r} = 1$. Our goal is to learn a sufficient invariant over such predicates. The challenge is to obtain a classifier that generalizes to unseen states. We are inspired by the observation that a simple invariant is more likely to generalize than a complex one [33]. Similar arguments have been demonstrated in machine learning and static verification techniques [27].

To learn a simple invariant, a learning algorithm should select a minimum subset of the predicates that separates all good states from all bad states. In the example, we first convert the original data sample into a Boolean table, evaluating the atomic predicates using each sample; the result is shown in Fig. 4.3b and we show the selection informally in Fig. 4.3c (Π_3 and Π_6 constitute a sufficient classifier). To compute a likely invariant, we generate its truth table Fig. 4.3d. The table rejects all (Boolean) bad samples in Fig. 4.3c and accepts all the other possible samples, including the good samples in Fig. 4.3c. Note that we generalize good states. The truth table accepts more good states than sampled. We apply standard logic minimization techniques [34]

	n	i	a_0	a_r
G	4	1	0	1
	4	1	1	0
	4	1	2	0
	4	3	2	1
	4	3	3	0
B	1	1	0	2
	2	1	0	-1

(a) samples

	Π_0	Π_1	Π_2	Π_3	Π_4	Π_5	Π_6
G	1	1	1	1	0	1	1
	1	1	1	0	1	1	0
	1	1	1	0	1	1	0
	1	1	1	1	1	1	1
	1	1	1	0	0	1	0
B	1	1	0	1	0	0	0
	1	1	1	1	1	1	0

(b) relate samples to predicates

	Π_3	Π_6
G	1	1
	0	0
B	1	0

(c) select predicates

	Π_3	Π_6
G	1	1
	0	0
	0	1
B	1	0

(d) truth table

Figure 4.3.: Classifying `init`'s good (G) and bad (B) samples.

to the truth table to generate the Boolean structure of the likely invariant. We obtain $\neg\Pi_3 \vee \Pi_6$, which in turn represents the following likely invariant:

$$\neg(a_0 < i) \vee a_r = 1$$

During the course of sampling the unknown function `a`, our system captures that certain calls to `a` may result in an assertion violation (rooted from the assertion in `f`). Consider a call to `a` that supplies an integer argument less than 0 or no less than `n`. These calls, omitted in the table, provide useful constraints on `a`'s inputs, which are also used by Learner. Indeed, comparing such calls to calls that do not lead to an assertion violation allows the Learner to deduce the invariant: $\psi_0 \equiv a_0 \geq 0 \wedge a_0 < n$,

that refines `a`'s argument. We treat ψ_0 and ψ_1 as likely invariants for the precondition for `init`. A similar strategy can be applied to also learn the post-condition of `init`.

Verifier. We encode likely program invariants into refinement types in the obvious way. For example, the following refinement types are automatically synthesized for `init`:

$$\begin{aligned}&\mathtt{i} : \mathtt{int} \to \mathtt{n} : \mathtt{int} \to \\ &\mathtt{a} : (\mathtt{a}_0 : \{\mathtt{int} | \nu \geq 0 \wedge \nu < \mathtt{n}\} \to \{\mathtt{int} | \neg(\mathtt{a}_0 < \mathtt{i}) \vee \nu = 1\}) \\ &\quad \to (\{\mathtt{int} | \nu \geq 0 \wedge \nu < \mathtt{n}\} \to \{\mathtt{int} | \nu = 1\})\end{aligned}$$

This type reflects a useful specification - it states that the argument `a` to `init` is a function that expects an argument from `0` to `n`, and produces a `1` only if the argument is less than `i`; the result of `init` is a function that given an input between 0 and `n` produces `1`. Extending [7], we have implemented a refinement type checking algorithm, which confirms the validity of the above type that is also sufficient to prove the assertions in the program.

CEGAR. Likely invariants are not guaranteed to generalize if inferred from an insufficient set of samples. We call likely invariants *failed invariants* if they fail to prove the specification. They are considered counterexamples, witnessing why the specification is refuted. Notice that, however, these could be *spurious* counterexamples. We develop a CEGAR loop that tries to refute a counterexample by sampling more states. If the counterexample is spurious, new samples prevent the occurrence of failed invariants in subsequent iterations.

Bad sample generation. Assume that Learner is only provided with the first bad sample in Fig. 4.3a. The good and bad samples are separable with a simple predicate $\Pi_2 \equiv \mathtt{a_r} < \mathtt{n}$. This predicate is not sufficiently strong since it fails to specify the input of `a`. To strengthen such an invariant, we ask for a new bad sample from the SMT solver for the condition:

$$\varphi_{\text{prebad}} \wedge (\mathtt{a_r} < \mathtt{n})$$

which was captured as the second bad sample in Fig. 4.3a. The new bad sample would invalidate the failed invariants.

Good sample generation. We exemplify our CEGAR loop in sampling good states using the program of Fig. 1.1. To bootstrap, we may run the program with arguments `1,2` and `3`, and infer the following types:

$$\begin{aligned}\texttt{max} ::(&\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \texttt{z} : \texttt{int} \rightarrow \\ &\texttt{m} : (\cdots) \rightarrow \{\texttt{int} | \nu > \texttt{x} \wedge \nu \geq \texttt{y}\})\end{aligned}$$

The refinement type of `max` is unnecessarily strong in specifying that the return value must be strictly greater than `x`. To weaken such a type, we seek to find a sample in which the return value of `max` equals `x`. To this end, we forward the failed invariant to the Deducer, which symbolically executes the negation of the post-condition of `max` ($\nu > \texttt{x} \wedge \nu \geq y$) back to `main` using our symbolic analysis. A solution to the derived symbolic condition (from an SMT solver) constitutes a new test input, e.g., a call to `main` with arguments `3`, `2` and `1`. With a new set of good samples, the program then typechecks with the desired refinement types:

$$\begin{aligned}\texttt{max} :: (&\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \texttt{z} : \texttt{int} \rightarrow \\ &\texttt{m} : (\texttt{m}_0 : \texttt{int} \rightarrow \texttt{m}_1 : \texttt{int} \rightarrow \{\texttt{int} | \ \nu \geq \texttt{m}_0 \wedge \nu \geq \texttt{m}_1\}) \\ &\rightarrow \{\texttt{int} | \nu \geq \texttt{x} \wedge \nu \geq \texttt{y} \wedge \nu \geq \texttt{z}\}) \\ \texttt{f} :: (&\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \{\texttt{int} | \ \nu \geq \texttt{x} \ \wedge \nu \geq \texttt{y} \wedge \\ &((\texttt{x} \leq \texttt{y} \wedge \nu \leq \texttt{y}) \vee (\texttt{x} > \texttt{y} \wedge \nu > \texttt{y}))\})\end{aligned}$$

The refinement type for `f` reflects the result of both the first and the second test. The proposition defined in the first disjunct, $\texttt{x} \leq \texttt{y} \wedge \nu \leq \texttt{y}$ captures the behavior of the call to `f` from `max` in the first test, with arguments `x` less than `y`; the second disjunct $\texttt{x} > \texttt{y} \wedge \nu > \texttt{y}$ captures the effect of the call to `f` in the second test in which `x` is greater than `y`.

Data Structures. Our approach naturally generalizes to richer (recursive) data structures. Important attributes of data structures can often be encoded into mea-

```
let rec iteri i xs f =
 match xs with
  | [ ] → ()
  | x::xs →
    (f i x;
    iteri (i+1) xs f)
```

```
let mask a xs =
 let g j y =
  a[j] ← a[j] && y in
 if Array.length a =
    len xs then
  iteri 0 xs g
```

Figure 4.4.: A simple data structure example.

sures (data-sorts), which are functions from a recursive structure to a base typed value (e.g. the height of a tree). Our approach verifies data structures by generating samples ranging over its measures. In this way, we can prove many data structure invariants (e.g. proving a red-black tree is a balanced tree).

Consider the example in Fig. 4.4. Function `iteri` is a higher-order list indexed-iterator that takes as arguments a starting index `i`, a list `xs`, and a function `f`. It invokes `f` on each element of `xs` and the index corresponding to the elements position in the list. Function `mask` invokes `iteri` if the lengths of a Boolean array `a` and list `xs` match. Function `g` masks the `j`-th element of the array with the `j`-th element of the list.

Our technique considers `len`, the length of list (`xs`), as an interesting measure. Suppose that we wish to verify that the array reads and writes in `g` are safe. For function `iteri`, based on our sampling strategy, we sample the unknown function `f` by calling it with inputs from [**min** (`i`, `len xs`), ..., **max** (`i`, `len xs`)] in the instrumented code. Since `f` binds to `g`, defined inside of `mask`, our system captures that some calls to `f` result in (array bound) exception, when the first argument to `f` is less than 0 or no less than `i + len xs`. Separating such calls from calls that do not raise the exception, our tool infers the following refinement type:

$$\texttt{iteri} :: (\texttt{i} : \{\texttt{int} | \nu \geq 0\} \rightarrow \texttt{xs} : \texttt{'a list} \rightarrow$$
$$\texttt{f} : (\texttt{f}_0 : \{\texttt{int} | \nu \geq 0 \wedge \nu < \texttt{i} + \texttt{len xs}\} \rightarrow \texttt{'a} \rightarrow ()) \rightarrow ())$$

This refinement type is the key to prove that all array accesses in function `mask` (and `g`) are safe.

4.2 Higher-order Program Sampling

In this section, we sketch how our system combines information gleaned from tests and (backward) symbolic analysis to prepare a set of program samples for higher-order programs.

Sampled Program States. In our approach, sampled program states, ranged over with the metavariable σ, map variables to values in the case of base types and map unknown functions to a set of input/output record known to hold for the unknown function from the tests. For example, if x is a base type variable we might have that $\sigma(x) = 5$. If f is a unary *unknown* function that was tested on with the arguments 0, 1 and 2 (such as the case of a in Fig. 4.3a), we might for instance have that $\sigma(f) = \{(f_0 : 0, f_r : 1), (f_0 : 1, f_r : 0), (f_0 : 2, f_r : 0)\}$ where we use f_0 to index the first argument of f and f_r to denote its return variable. The value of f_r is obtained by applying function f to the value of f_0. Importantly, f_r is assigned a special value "`err`" if an assertion violation is triggered in a call to f with arguments recorded in f_0.

WP Generation. "Bad" program states are captured by pre- and post-bad conditions of known functions sufficient to lead to an assertion violation. To this end, we simply use our backward symbolic analysis *wp*, defined in Fig. 3.6, with only one exception:

$$
\begin{array}{l}
\mathtt{wp}(i, e, \delta) = \\
\quad \mathsf{match}\ e\ \mathsf{with} \\
\quad\quad \mid \mathtt{assert}\ v \rightarrow \neg v \vee \delta \\
\quad\quad \mid \cdots
\end{array}
$$

For a program e, this analysis pushes up the negation of assertions inside e backwards, substituting terms for values in a bad condition δ based on the structure of e. Initially, δ is set to `false`, meaning that bad conditions are only driven by assertions. The parameter i limits that recursive functions are unrolled only i times.

Our `wp` analysis encodes unknown functions into uninterpreted functions. As a result, we can generate constraints over the input/output behaviors of unknown functions for higher-order functions (e.g. δ_5 in Fig. 4.2). Following the definition in Fig. 3.6, the symbolic analysis for the actual function represented by the unknown function is deferred until it becomes known (e.g. δ_3 in Fig. 4.2).

During `wp`, the symbolic conditions collected at the entry and exit point of each function is treated as the pre- and post-bad condition of the function (e.g. δ_{prebad} and δ_{postbad} in Fig. 4.2).

Program Sampling. Our approach instruments the original program at the entry and exit point of a function to collect values for each function parameter and return, together with variables in its lexical scope (for closures). The instrumentation for base type variables is trivial. To sample an unknown function, we adopt two conservative strategies.

1. A side-effect of `wp`'s definition is that it provides hints on how unknown functions are eventually used because the arguments to such functions are already encoded into uninterpreted forms. If the variables that compose the arguments are all in the lexical scope, we call the function with those arguments (e.g. the argument `j` to unknown function `a` inside function `update` in Fig. 4.2 is considered in-scope).

2. The arguments supplied to unknown functions may not be in-scope (e.g. recall that in function `init` in Fig. 4.2 the argument `j` to `a` is supplied in `update` and undefined in `init`). In this case, for a base type argument, we supply integers drawn from $\texttt{min}(\vec{x})$ to $\texttt{max}(\vec{x})$ where $\vec{x}$ are integer parameters from the higher-order function that hosts the unknown function. The goal is to build a refinement type of the unknown function based on its relation (parameterized

```
let app x (f:int→(int→int)→int) g = f x g
let f x k = k x
let check x y = (assert (x = y); x)
let main a b = app (a * b) f (check (a * b))
```

Figure 4.5.: Generating samples for `g` may trigger assertion violations in `check`.

by our hypothesis domain) with variables in $\vec{x}$. The definition of `min` and `max` is in Sec. 4.1. For a function type argument that is not in-scope, we similarly supply ghost functions with return values from the above domain.

For each known function, bad samples (V_B) can be queried from an SMT solver as solutions to its pre- and post-bad conditions generated by `wp`. During the course of sampling good states, the call to an unknown function with arguments according to the second sampling strategy (above) may raise an assertion failure that is associated with an "`err`" return value. We classify the subset of samples involving "`err`" as an additional set of bad samples (V'_B). The rest of the samples from test outcomes constitute good program states (V_G). Intuitively, V_B can constrain the output while V'_B can constrain the input of unknown function in a likely invariant. For example, we may call (`main 0 0`) for the program given in Fig. 4.5 and obtain the sample states for function `app` shown in Fig. 4.6 where the first argument of `f` and `g` are supplied from `x`-1 to `x`+1. Samples in which calls to the unknown function `g` return `err` (because it would trigger an assertion violation in `check`) will be used to strengthen `g`'s pre-condition.

Sample Generalization. Our main idea is to generalize useful invariants from good program states based on the expectation that such invariants (even for unknown functions) should be observable from test runs. By summarizing the properties that hold in all such runs, we can construct likely invariants. In addition, the use of bad program states, which are either solutions of bad-conditions queried from an SMT solver (V_B) or collected from the "`err`" case during sampling of an unknown function

x	$\mathtt{f}_0$	$\mathtt{f}_1$	$\mathtt{f}_r$	$\mathtt{g}_0$	$\mathtt{g}_r$
0	1	g	err	-1	err
0	0	g	0	0	0
0	-1	g	err	1	err
			...		

Figure 4.6.: Sample table for pre-state of app in Fig. 4.5.

(V'_B), enables a demand-driven inference technique. With a set of good (V_G) and bad ($V_B \cup V'_B$) program states, our method exploits a learning algorithm $L(V_G, V_B)$ (resp. $L(V_G, V'_B)$) to produce a likely invariant that separates V_G from V_B (resp. V'_B). We lift these invariants to a refinement type system and check their validity through refinement type checking technique (Sec. 4.4).

4.3 Learning Algorithm

We describe the design and implementation of our learning algorithm $L(V_G, V_B)$ in this section. Suppose we are given a set of good program states V_G and a set of bad program states V_B, where both V_G and V_B contain states which map variables to values. We simplify the sampled states by abstracting away unknown function f: each sampled state σ in V_G and V_B only records the values of its parameters $f_0, \cdots$ and return f_r. We base our analyses on a set of atomic predefined predicates $\Pi = \{\Pi_i\}_{0 \leq i < n}$ from which program invariants are constructed. Recall the hypothesis domain defined in Sec. 4.1. Each atomic predicate Π_i is of the form:

$$c_1 y_1 + \cdots + c_m y_m + d \ \leq \ 0$$

where $\{y_1, \cdots, y_m\}$ are numerical variables from the domains of V_G and V_B, each $c_i \in \mathcal{C}$ $(i = 1, \cdots, m)$ is an integer coefficient and $d \in \mathcal{D}$ is an integer constant. We have restricted $\mathcal{D}$ to a finite set of integer constants and its negations from the program text and $\mathcal{C} = \{-1, 0, 1\}$. Note that further restricting the number of nonzero

c_i to at most 2 enables the learning algorithm to choose predicates from a subset of the octagon domain. In our experience, we have found such a selection to be a feasible approach, attested by our experiments in Sec. 4.6. Thanks to this parameterization, we can draw on predicates from a richer abstract domain without requiring any re-engineering of the learning algorithm.

The problem of inferring an invariant then reduces to a search problem from the chosen predicates. A number of static invariant inference techniques have been proposed for efficient search over the hypothesis space generated by Π [7,35]. Compared to those, our algorithm has the strength of discovering invariants of arbitrary Boolean structure. In our context, given Π, an *abstract state* α over $\sigma \in (V_G \cup V_B)$ is defined as:

$$\alpha(\sigma) \equiv \{\ \langle \Pi_1(\sigma), \cdots, \Pi_n(\sigma) \rangle\ \}$$

We say that $L(V_G, V_B)$ is *consistent* with respect to V_G and V_B, if $\forall \sigma \in V_G\ .\ \alpha(\sigma) \Rightarrow L(V_G, V_B)$, and $\forall \sigma \in V_B\ .\ \alpha(\sigma) \wedge L(V_G, V_B) \Rightarrow$ `false`. Intuitively, we desire L to compute an interpolant or classifier (that is derived from atomic predicates in Π) that separates the good program states from the bad states [36].

However, we would like to discover classifiers from samples with the property that they generalize to yet unseen executions. Therefore, we exploit a simple observation: a general invariant should be simple enough. Specifically, we answer the question by finding the minimal invariant from the samples, in terms of the number of predicates that are used in the likely invariant. This idea has also been explored before in the context of computing simple proofs based on interpolants [27,37].

To this end, we build the following constraint system. Using Π, we transform V_G and V_B that are defined over integers to V_G^b and V_B^b defined over Boolean values. Specifically, $V_G^b = \{\langle(\Pi_1(\sigma), \cdots, \Pi_n(\sigma))\rangle |\ \sigma \in V_G\}$. V_B^b is defined dually. Fig. 4.3b is an example of such conversion from Fig. 4.3a. We associate an integer variable sel_i

Algorithm 1: $L\ (V_G,\ V_B)$

```
1 let (φ1, φ2) = encode (V_G, V_B) in
2 let k := 1 in
3 if sat (φ1 ∧ φ2) ≠ UNSAT) then
4 |   while not (sat (φ1 ∧ φ2 ∧ (Σ_i sel_i = k)) do
5 |   |   k := k + 1
6 |   McCluskey (smt_model (φ1 ∧ φ2 ∧ (Σ_i sel_i = k)))
7 else abort "Invariant not in hypothesis domain"
```

to the i^{th} predicate $\Pi_i (0 \leq i < n)$. If Π_i should be selected for separation in the classifier, sel_i is assigned to 1. Otherwise, it is assigned as 0.

$$\varphi_1 : \bigwedge_{\forall g,b.\ g \in V_G^b, b \in V_B^b} \bigvee_{0 \leq i < n} (g(\Pi_i) \neq b(\Pi_i) \wedge sel_i = 1)$$

$$\varphi_2 : \bigwedge_{0 \leq i < n} 0 \leq sel_i \leq 1$$

$$\varphi_c : min(\Sigma_{0 \leq i < n} sel_i)$$

The first constraint φ_1 specifies the separation of good states from bad states—for each good state g and bad state b, there must exist at least one predicate Π_i labeled by sel_i such that the respective evaluations of Π_i on g and b differs.

The second constraint φ_2 ensures that each x_i must be between 0 and 1. The third constraint φ_c specifies the cost function of the constraint system and minimizing this function is equivalent to minimizing the number of predicates selected for separation, which in turn results in a simple invariant as discussed.

Algorithm 1 computes a solution for likely invariant. It firstly builds φ_1 and φ_2 as stated. Then it iteratively solves the constraint system to find the minimum k that renders the constraint system satisfiable. In our experience, since the number of parameters of a function is not large, and the fact that a few number of samples usually suffice for discovering an invariant, the call to an SMT solver in our algorithm

is very efficient. For example, a solution of the constraint system built over Fig. 4.3b is shown in Fig. 4.3c. By design, our algorithm guarantees that the invariants discovered are the minimum one to separate V_G and V_B and therefore, it is very likely that they will generalize.

When the solution is computed, the likely invariant should be a Boolean combination of the predicates Π_i if sel_i=1 in the solution. We use a Boolean variable $\mathcal{B}_i$ to represent the truth value of predicate Π_i and generate a truth table $\mathcal{T}$ over the $\mathcal{B}_i$ variables for the selected predicates. Formally $\{\mathcal{B} = \mathcal{B}_i |\ sel_i = 1 (0 \leq i < n)\}$. To construct the likely invariant, we firstly generate a table $V^b_{\mathcal{B}}$, which only retains the values corresponding to the selected predicates Π_i ($sel_i = 1$) in V^b_B. Each row of the truth table $\mathcal{T}$ is a configuration (assignment) to the variables in $\mathcal{B}$. If a configuration corresponds to a row in $V^b_{\mathcal{B}}$, its corresponding result in $\mathcal{T}$ is `false`. Otherwise, the result in `true`. Intuitively, $\mathcal{T}$ must reject all the evaluations to $\mathcal{B}$ if they appear in a bad sample in V^b_B and accept all the other possible evaluations to $\mathcal{B}$ (which of course include those in V^b_G). See Fig. 4.3d as an example of the generated truth table from Fig. 4.3c. In line 6 of Algorithm 1, the call to McCluskey applies standard sound logic minimization techniques [34] to $\mathcal{T}$ to compute a compact Boolean structure of the likely invariant.

Lemma 5 *L (V_G, V_B) is consistent.*

Lemma 5 claims that our algorithm will never produce an invariant that misclassifies a good sample or bad sample.

4.4 Verification Procedure

To yield refinement types, we extend standard types with invariants which are automatically synthesized from samples as type refinements. The invariants inferred for a function f are assigned to unknown refinement variables (κ) in the refinement function type of f and verified via the verification procedure defined in Sec. 2.2.

Notably, our approach can properly account for unknown functions whose order is more than one, that is unknown functions which may also takes functional arguments. Recall the sample states generated for function `app` in Fig. 4.6. In the `app` function, the argument `f` is an unknown function whose second argument $\texttt{f}_1$ is also an unknown function as the type in Fig. 4.5 shows. We did not sample the input/output values for function $\texttt{f}_1$ and only recorded its supplier, `g`. We observe that such an unknown function will be eventually supplied with another function. For example, in the body of `app`, `g` will be supplied for $\texttt{f}_1$. This indicates the invariant inferred for `g` is also likely to be invariant for $\texttt{f}_1$ so the type refinements for `g` can flow into that of $\texttt{f}_1$. Formally, consider the refinement function subtyping rule in Fig. 2.3:

$$\frac{\Gamma \vdash P'_x <: P_x \quad \Gamma; x : P'_x \vdash P <: P'}{\Gamma \vdash \{x : P_x \to P\} <: \{x : P'_x \to P'\}}$$

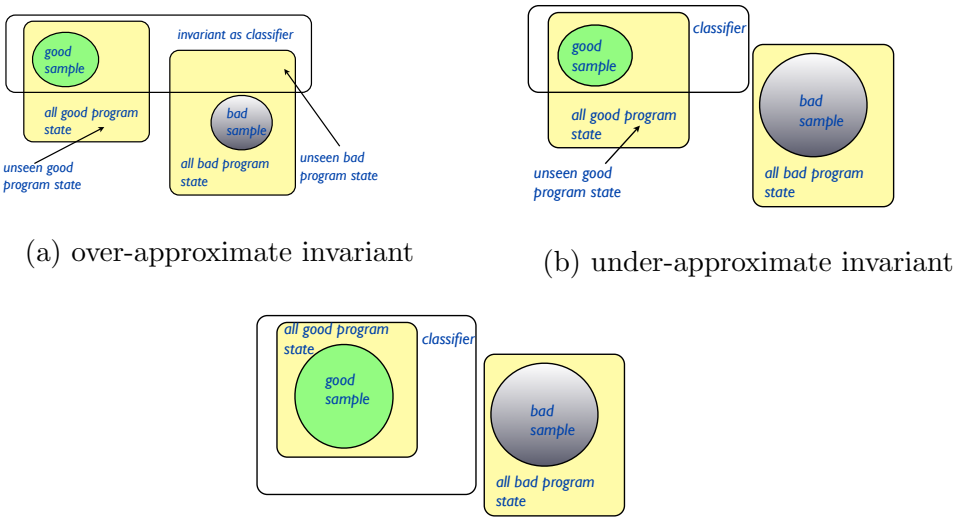
If the type refinement in P_x is synthesized, it can be propagated to that of P'_x.

For example, according to the subtyping rule, `g` must subtype to $\texttt{f}_1$. So $\texttt{f}_1$ can then inherit the type refinements for `g`. We then let our type inference algorithm decide a valid type instantiation, following [7]. In Fig. 4.6, separating the samples that represent good calls to `f` and `g` with the samples that represent bad calls (e.g., calls that raise an `err`), we infer the invariant: $\texttt{f}_0 = \texttt{x}$ and $\texttt{g}_0 = \texttt{x}$. Leveraging the type inference algorithm with the likely type refinement $\nu = \texttt{x}$, we conclude the desired type for `app`:

$$\begin{aligned}\texttt{app} ::(\texttt{x} : \texttt{int} \to \texttt{f} : (\texttt{f}_0 : \{\texttt{int}|\ \nu = \texttt{x}\} \to \\ \texttt{f}_1 : (\{\texttt{int}|\nu = \texttt{x}\} \to \texttt{int}) \to \texttt{int}) \to \\ \texttt{g} : (\texttt{g}_0 : \{\texttt{int}|\nu = \texttt{x}\} \to \texttt{int}) \to \texttt{int})\end{aligned}$$

4.4.1 CEGAR Loop

Algorithms. Our Main algorithm (Algorithm 2) takes as input a higher-order program e with its safety property ψ that is expected to hold at some program point. We first annotate ψ in the source as assertions at that program point and use ran-



(a) over-approximate invariant

(b) under-approximate invariant

(c) desired invariant

Figure 4.7.: CEGAR loop: invariant as classifier.

dom test inputs iv (like [31]) to bootstrap our verification process (line 1). We then instrument the program using the strategy discussed in Sec. 4.2. Function `run` compiles and runs the instrumented code with iv (line 2); concrete program states at the entry and exit of each known function are logged to produce good states V_G. (We omit including additional bad states V'_B caused by calls to unknown functions returning "`err`" in the instrumented code (see Sec. 4.2), for simplicity.) We then enter the main CEGAR loop (line 4-8). With a set of good and bad states for each known function, the function `learn` invokes the L learning algorithm (see Sec. 4.3) to generate likely invariants (line 5) which are subsequently encoded as the function's refinement types for validation (line 6). If the program typechecks, verification is successful. Otherwise, type checking is considered to fail because these invariants are synthesized from an insufficient set of samples. We try generating more samples for the learning algorithm, refining the failed invariants (line 8). Notably, our backward symbolic analysis (`wp`) requires to bound the number of times recursive functions are unrolled. This is achieved by passing the bound parameter i to Refine. Initially i is set to 2 (line 3 of Algorithm 2).

Algorithm 2: Main $e\ \psi$

Input: e is a program; ψ is its safety property

Output: verification result

```
1 let (e', iv) = (annotate e ψ, randominputs e) in
2 let (V_G, V_B) := ( run (instrument e') iv, ∅) in
3 let i := 2 in
4 while true do
5 |   let φ = learn (V_G, V_B) in
6 |   if verify e' φ then
7 |   |   return "Verified"
8 |   else (V_G, V_B) := Refine (i, e, ψ, φ, V_G, V_B)
```

The Refine algorithm (see Algorithm 3) guides the sample generation to refine a failed likely invariant. The first step of Refine is the invocation of the `wp` procedure over the given higher-order program annotated with the property ψ (line 1 and 2); this step yields pre-and post-bad conditions for each known function sufficient to trigger a failure of some assertion (line 3). A failed invariant may be too over-approximate (failing to incorporate needed sufficient conditions) or too under-approximate (failing to account for important necessary conditions). This is intuitively described in Fig. 4.7a where the classifier (as invariant) only separates the observed good and bad samples but fails to generalize to unseen states.

To account for the case that it is too over-approximate, we firstly try to sample new bad states (line 4). The idea is reflected in Fig. 4.7b. The new bad samples should help the learning algorithm strengthen the invariants it considers. For each known function, we simply conjoin the failed likely pre- and post-invariants with the pre- and post-bad conditions derived earlier from the `wp` procedure. Bad states (V_B) are (SMT) solutions of such conditions (line 5). Note that bad_cond and φ are sets of bad conditions and failed invariants for each known function in the program. Operators like $\wedge$ and $\cup$ in Algorithm 3 are overloaded in the obvious way. If no new

Algorithm 3: Refine (i, e, ψ, φ, V_G, V_B)

Input: (e, ψ) are as in Algorithm 2; φ are failed invariants; i is the number of times a recursive function is unrolled in wp; V_G and V_B are old good and bad samples

Output: good or bad samples (V_G, V_B) that refines φ

```
1  let e' = annotate e ψ in
2  let _ = wp (i, e', false) in
3  let bad_cond = bad conditions of functions from wp call in
4  if sat (bad_cond ∧φ) then
5  |  (V_G, (deduce (bad_cond ∧φ)) ∪ V_B)
6  else
7  |  let test_cond = wp (i, annotate e φ, false) in
8  |  if sat (test_cond) then
9  |  |  let iv = deduce test_cond in
10 |  |  ((run (instrument e') iv) ∪ V_G, V_B)
11 |  else Refine (i + 1, e, ψ, φ, V_G, V_B)
```

bad states can be sampled, we account for the case that failed invariants are too under-approximate (line 6).

Our idea of sampling more good states is reflected in Fig. 4.7c. The new good state should help the learning algorithm weaken the invariants it considers. To this end, we annotate the failed pre- and post-invariant as assertions to the entry and exit of function bodies for the known functions where such invariants are inferred. (Function *annotate* substitutes variables representing unknown function argument and return in a failed invariant with the actual argument and return encoded into uninterpreted form in the corresponding function's pre- and post-bad conditions. For example, $\mathtt{a}_0$ and $\mathtt{a}_\mathtt{r}$ in Fig. 4.3a are replaced with `j` and `a j` in a failed invariant for the `init` function (consider δ_5) in Fig. 4.2.) Note that these invariants only represent an under-approximate set of good states. To direct tests to program states that have not been

seen before, the `wp` procedure executes the negation of these annotated assertions back to the program's `main` entry to yield a symbolic condition (line 7). Function *deduce* generates a new test case for the `main` entry (line 8 and 9) from the (SMT) solutions of the symbolic condition. The new good states from running the generated test inputs are ensured to refine the failed invariant (line 10).

In function Refine, we only consider unrolling recursive function a fixed i times. As stated, if this is not sufficient, we increase the value of i and iterate the refinement strategy (line 11). However, in our experience (see Sec. 4.6), unrolling the definition of a recursive function twice usually suffices based on the observation that the invariant of recursive function can be observed from a shallow execution. Particularly, i is unlikely to be greater than the maximum integer constant used in the `if`-conditions of the program.

Algorithm Output. (a) In the testing phase (Runner), the Main algorithm terminates with test inputs witnessing bugs in function `run` when the tests expose assertion failures in the original program. (b) In the sampling phase (Deducer), since our technique is incomplete in general, if a program has expressions that cannot be encoded into a decidable logic for SMT solving, Refine may be unable to infer necessary new samples because the *sat* function (line 4 and line 8 of Algorithm 3) aborts with undecidable result. (c) In the learning phase (learner), it terminates with "Invariant not in hypothesis domain" in line 7 of Algorithm 1 when no invariant can be found in the search space (which is parameterized by Equation 4.1 in Sec. 4.1). (d) In the verifying phase (verifier), it returns "Verified" in line 5 of Algorithm 2 when specifications are successfully proved.

4.4.2 Soundness and Convergence

Our algorithm is sound since we rely on a sound refinement type system [7] for proving safety properties and a concrete test input for witnessing program bugs.

Desired Invariant. For a program e with some safety property ψ, a *desired invariant* of e should accept all possible (unseen) good states and reject all (unseen) bad states. Recall that our hypothesis domain is the arbitrary Boolean combination of predicates, parameterized by Equation 4.1 in Sec. 4.1. We claim the CEGAR loop in Algorithm 2 *converges*: it could eventually learn a desired invariant φ, provided that one exists expressible as a hypothesis in the hypothesis domain.

Theorem 4.4.1 [*Convergence*] *Algorithm 2 converges to a desired invariant.*

Proof First, assume Refine (line 7 of Algorithm 2) does not take a desired invariant as input; otherwise Algorithm 2 has already converges. We also assume the existence of an efficient SMT solver for checking satisfiability. We claim, by iteratively increasing i, the number of times recursive functions are unrolled in e, Refine can eventually generate a new pair of good/bad samples that refine φ.

To show the reason, we assume such a value of i does not exist. Formally,

$$\forall i.\ \texttt{wp}(i, e, \neg\varphi)\ \Rightarrow \texttt{false} \tag{a}$$

$$\forall i.\ \varphi \wedge \texttt{wp}(i, e, \psi)\ \Rightarrow \texttt{false} \tag{b}$$

We use Σ_G and Σ_B to denote all the possible good states and bad states respectively. 1) By definition, we have

$$\forall \sigma_B \in \Sigma_B.\ \exists i.\ \sigma_B \in \texttt{wp}(i, e, \psi) \tag{c}$$

Combining Equation (b) and Equation (c),

$$\forall \sigma_B \in \Sigma_B.\ \sigma_B \notin \varphi \tag{d}$$

2) Furthermore, by definition,

$$\forall \sigma_G \in \Sigma_G.\ \exists i. \texttt{wp}(i, e, \alpha(\sigma_G)) \not\Rightarrow \texttt{false} \tag{e}$$

Recall that, given the set of atomic predicates Π (from our hypothesis domain), an *abstract state* α is defined as:

$$\alpha(\sigma) \equiv \{\ \langle \Pi_1(\sigma), \cdots, \Pi_n(\sigma) \rangle\ \}$$

It is clear that $\alpha(\sigma_G)$ is a conjunction of the predicates (or their negations) from Π, while φ is a Boolean combination of the predicates (or their negations) from Π, guaranteed by the construction of the learning algorithm L. As a result, for an arbitrary good state $\sigma_G \in \Sigma_G$, the logic relationship between $\alpha(\sigma_G)$ and $\neg\varphi$ is either $\alpha(\sigma_G) \Rightarrow \neg\varphi$ or $\alpha(\sigma_G) \wedge \neg\varphi \Rightarrow \texttt{false}$.

Combing Equation (a) and Equation (e), however, for an arbitrary σ_G, $\alpha(\sigma_G) \Rightarrow \neg\varphi$ is impossible. This is because, if we have $\alpha(\sigma_G) \Rightarrow \neg\varphi$, then we must have $\exists i.\ \texttt{wp}(i, e, \neg\varphi) \not\Rightarrow \texttt{false}$ due to Formula (e), by induction on the structure on e, contradicting with Formula (a).

Hence, we must have,

$$\forall \sigma_G \in \Sigma_G.\ \alpha(\sigma_G) \wedge \neg\varphi \Rightarrow \texttt{false} \tag{f}$$

From Equation (f),

$$\forall \sigma_G \in \Sigma_G.\ \alpha(\sigma_G) \Rightarrow \varphi \tag{g}$$

Combining Equation (d) and Equation (g), φ is a desirable invariant because it accepts all possible good states that are not sampled and rejects all possible bad states, which contradict our assumption that Refine does not take a desired invariant as input.

As a result, we guarantee to find a value of i that drives the sampling for good/bad samples that refine φ. In each CEGAR iteration, by construction, a new sample provides a witness of why a failed invariant should be refuted.

According to Lemma 5, our learning algorithm produces a *consistent* hypothesis that separates all good samples from bad samples. As a result, the CEGAR loop does not repeat hypothesis: a failed invariant, once refuted, cannot be reproduced in later CEGAR iterations. Our technique essentially enumerates the hypothesis space and ensures that all hypotheses will be eventually considered by adding

more samples. Finally, the hypothesis domain is finite since the coefficients and constants of atomic predicates are accordingly bounded (see Sec. 4.3); the CEGAR based sampling-learning-checking loop in Algorithm 2 converges in a finite number of iterations. ■

Intuitively, the above proof assumes Refine (line 8 of Algorithm 2) does not take a desired invariant as input; otherwise Algorithm 2 has already converges. Refine can iteratively increase i, the number of times recursive functions are unrolled in e, to generate a new pair of good/bad samples that refine φ. Otherwise, if such a value of i does not exist, φ already classified all the unseen good/bad samples. Hence, in each CEGAR iteration, by construction, a new sample provides a witness of why a failed invariant should be refuted.

According to Lemma 5, our learning algorithm produces a *consistent* hypothesis that separates all good samples from bad samples. As a result, the CEGAR loop does not repeat failed hypothesis. Our technique essentially enumerates the hypothesis domain. Finally, the hypothesis domain is finite since the coefficients and constants of atomic predicates are accordingly bounded (see Sec. 4.3); the CEGAR based sampling-learning-checking loop in Algorithm 2 converges in a desired invariant in a finite number of iterations.

Inductive Invariant. A desired invariant, however, might not be an inductive invariant. Consider the type checking rule for recursive functions:

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

The refinement type system (Fig. 2.3) we use is an *inductive* invariant checking system. The refinement types encoded from a candidate invariant of $\texttt{fix}\,(\texttt{fun}\ f \to \lambda x.\ e)$ can be type checked in our refinement type system only if it is inductive. However, a program may have many invariants and, for recursive functions, and only *inductive* ones should be used to assist verification.

We can slightly adapt our counterexample-guided refinement algorithm to achieve *relative completeness*, meaning that if an inductive invariant actually exists in the

hypothesis domain the algorithm can eventually find it. If the number of recursive function unrollings i exceeds a certain bound in Algorithm 3, the algorithm can just terminate and uses a set F to save possibly non-inductive invariants. It is OK if the bound is not set to a sufficiently large value, because Algorithm 2 then moves to the next round and we enforce the learning algorithm to return a hypothesis invariant different with any invariants in F. Such a constraint can be easily encoded into our learning algorithm Algorithm 1. Algorithm 2 then essentially enumerates the hypothesis domain and can eventually find an inductive invariant, provided one actually exits expressible.

4.4.3 Algorithm Features

In Algorithm 2, the refinement type system and test system cooperate on invariant inference. The refinement type system benefits from tests because it can extract invariants from test outcomes. Conversely, if previous tests do not expose an error in a buggy program, failed invariants serve as abstractions of sampled good states. By directing tests towards the negation of these abstractions, Algorithm 3 guides test generation towards hitherto unexplored states.

Second, it is well known that intersection types [19] are necessary for verification when an unknown function is used more than once in different contexts [13]. Instead of inferring intersection types directly as in [13], we recover their precision by inferring type refinements (via learning) containing disjunctions (as demonstrated by the example in Fig. 4.2).

4.5 Inductive Data Structures

As stated in Sec. 4.1, we extend our framework to verify inductive data structure programs with specifications that can be encoded into type refinements using measures [8, 38]. For example, a measure `len`, representing list length, is defined

```
let rec len l =
 match l with
  | x :: xs →
    len xs + 1
  | [ ] → 0
```

```
let reverse zs =
 let rec aux xs ys =
  match xs with
   | [ ] → ys
   | x::xs → aux xs (x::ys) in
 let r = aux zs [ ] in
 (assert(len r = len zs); r)
```

Figure 4.8.: Samples of data structures can be classified by measures.

in Fig. 4.8 for lists. We firstly extend the syntax of our language to support inductive data structures.

$$e ::= \cdots \mid \langle v \rangle \mid \mathcal{C}\langle v \rangle \mid \texttt{match}\ v\ \texttt{with}\ |_i\ \mathcal{C}_i\langle x_i \rangle \rightarrow e_i$$
$$M ::= (m, \langle \mathcal{C}_i\langle x_i \rangle \rightarrow \epsilon_i \rangle) \quad \epsilon ::= m \mid c \mid x \mid \epsilon\ \epsilon$$

The first line illustrates the syntax for tuple constructors, data type constructors where $\mathcal{C}$ represent a constructor (e.g. list `cons`), and pattern-matching. M is a map from a measure m to its definition. To ensure decidability, like [8], we restrict measures to be in the class of first order functions over simple expressions (ϵ) so that they are syntactically guaranteed to terminate. The typing rules for the extended syntax are adapted from [8] and are given in Fig. 4.9. In rule T-Constructor, the type refinement for an inductive data structure returned by a constructor is a conjunctions of relation between the measure of the constructed expression and the variables bound by the constructor arguments. The rule T-Match stipulates that the entire expression has type P if and only if P is well-formed in the type environment, and that, for each case expression e_i of the match, e_i must also have type P in the type environment extended with the guard predicate that captures the relation between the measure of the matched expression and the variables bound by the matched pattern.

To support this extension, we also need to extend our `wp` definition in Fig. 4.10. The basic idea is that when a recursive structure is encountered, its measure defi-

$$\textsc{T-Constructor}\quad \frac{}{\Gamma \vdash \mathcal{C}_j\langle v\rangle : \{\nu : ty(\mathcal{C}_j) |\, \wedge_m\, m(\nu) = \epsilon_j(\langle v\rangle)\}}$$

$$\textsc{T-Match}\quad \frac{\Gamma \wedge \bigwedge_m m\ v = \epsilon\ \langle x_i\rangle \vdash e_i : P \quad \Gamma \vdash P}{\Gamma \vdash \texttt{match}\ v\ \texttt{with}\ |_i\ \mathcal{C}_i\langle x_i\rangle \rightarrow e_i : P}$$

Figure 4.9.: Refinement typing rules for inductive data structures.

$$\begin{aligned}
&\texttt{wp}(i, e, \phi) = \ \mathsf{match}\ e\ \mathsf{with}\\
&\quad |\ \mathcal{C}_i\langle v\rangle\ \ \texttt{when}\ \ (m, \langle \mathcal{C}_i\langle x_i\rangle \rightarrow \epsilon_i\rangle) \in M \rightarrow [\epsilon_i\ \langle v\rangle/(m\ \nu)]\phi\\
&\quad |\ \{\texttt{match}\ \ v\ \ \texttt{with}\ \ |_i\ \mathcal{C}_i\langle x_i\rangle \rightarrow e_i\}\,\texttt{when}\,(m, \langle \mathcal{C}_i\langle x_i\rangle \rightarrow \epsilon_i\rangle) \in M \rightarrow\\
&\qquad \bigvee_i \{\exists \langle x_i'\rangle.[\langle x_i'\rangle/\langle x_i\rangle]((m\ v) = \epsilon_i\langle x_i\rangle \wedge (\texttt{wp}(i, e_i, \phi)))\}\\
&\quad |\ \cdots
\end{aligned}$$

Figure 4.10.: `wp` rule for inductive data structures.

nitions are accordingly unrolled: (1) for a structure constructor $\mathcal{C}_i\langle e\rangle$, we derive the appropriate pre-condition by substituting the concrete measure definition $\epsilon_i\langle e\rangle$ for the measure application $m\ \nu$ in the post-condition; this is exemplified in Fig. 4.11 where bad-condition δ_2 is obtained from δ_1 by substituting `len ys` for `len ys` $+ 1$ based on the definition of measure `len`; (2) for a match expression, the pre-condition is derived from a disjunction constructed by recursively calling `wp` over all of its case expressions, which are also extended with the guard predicate capturing the measure relation between e and $\langle x_i\rangle$. All the $\langle x_i\rangle$ need to be existentially quantified and skolemized when fed to an SMT solver to check satisfiability. The bad condition δ_3 in Fig. 4.11 is such an example.

With the extended definition, sampling inductive data structures is fairly straitforward. To collect "good" states, in the instrumentation phase, for each inductive

```
let rec aux xs ys =
δprebad : δ3 ∨ δ4
 match xs with
 δ4 : len xs = 0 ∧ len ys ≠ len zs
 | [ ] → ys
 δ3 : ∃xs′.len xs = 1 + len xs′ ∧ [xs′/xs]δ2
 | x::xs →
  δ2 : len xs = 0 ∧ len ys + 1 ≠ len zs
  let ys = x::ys in
  δ1 : len xs = 0 ∧ len ys ≠ len zs
  aux xs ys in
δpostbad : len ν ≠ len zs
```

Samples:

	len xs	len ys	len zs
G	1	2	3
	2	1	3
B	1	0	2
	1	0	0

Likely invariant:

$$\texttt{len xs} + \texttt{len ys} = \texttt{len zs}$$

Figure 4.11.: Learning a data structure function's precondition from its samples.

data structure serving as a function parameter or return value in some data structure function, we simply call its measure functions and record the measure outputs in the sample state. To collect "bad" states, we invoke an SMT solver on the bad-conditions for each data structure functions to find satisfiability solutions. The solver can generate values for measures because it interprets a measure function in bad-conditions as uninterpreted.

Consider how we might infer a precondition for function `aux` in Fig. 4.8. Note that `aux` is defined inside `reverse` and is a closure which can refer to variable `zs` in its lexical scoping. A good sample (G) presents the values of `len xs`, `len ys` and `len zs`, trivially available from testing. A bad sample (B) captures a bad relation among `len xs`, `len ys` and `len zs` that is sufficient to invalidate the assertion in the `reverse` function, solvable from δ_{prebad} in Fig. 4.11. With these samples, our approach infers the following refinement type for `aux`, which is critical to prove the assertion.

$$\texttt{xs}: \text{'a } \texttt{list} \rightarrow \texttt{ys}: \{\text{'a } \texttt{list} \mid \texttt{len xs} + \texttt{len } \nu = \texttt{len zs}\}$$
$$\rightarrow \{\text{'a } \texttt{list} \mid \texttt{len } \nu = \texttt{len zs}\}$$

If function `aux` is not defined inside of function `reverse` where `zs` is not in the scope of `aux`, our technique infers a different type for `aux`, `xs` : 'a `list` → `ys` : 'a `list` → {'a `list` | `len xs` + `len ys` = `len` ν}.

When there is a need for sampling more good states in the Refinement algorithm (Algorithm 3), generating additional test inputs for data structures from `wp`-condition reduces to Korat [39], a constraint based test generation mechanism. Alternatively, the failed invariants can be considered incorrect specifications. We can directly generate inputs to the program by causing it to violate the specifications following [40, 41]. Notably, the former approach is complete if the underlying SMT solver can always find a model for any satisfiable formula. As an optimization for efficiency, we bootstrap the verification procedure with random testing to generate a random sequence of method calls (e.g. `insert` and `remove` functions) up to a small length s in the Main algorithm (line 1 of Algorithm 2). In our experience in Sec. 4.6, setting s to 300 allows the system to converge for all the container structures we consider without requiring extra good samples; this result supports a large case study [42] showing that test coverage of random testing for container structures is as good as that of systematic testing.

4.6 Experimental Results

We have implemented our approach in a prototype verifier.[1] Our tool is based on OCaml compiler. We use Yices [43] as our SMT solver. To test the utility of our ideas, we consider a suite of around 100 benchmarks from the related work. Our experimental results are collected in a laptop running Intel Core 2 Duo CPU with 4GB memory. Our experiments are set up into three phases. In the first step, we demonstrate the efficiency of our learning based invariant generation algorithm (Sec. 4.3) by comparing it with existing learning based approaches, using non-trivial first-order *loop* programs. In this step, we only compare first-order programs because

[1] `https://www.cs.purdue.edu/homes/zhu103/msolve/`

Loops	N	L	T	CPA	ICE	SC	MC^2
cgr2	2	0.2	0.3s	1.7s	6.9s	2.7s	17.3s
ex23	3	0.3	0.4s	16.7s	17.4s	4.7s	0.1s
sum1	5	0.6	0.8s	1.5s	1.8s	2.6s	29.1s
sum4	2	0.1s	0.1s	3.2s	2.6s	×	×
tcs	2	0.1s	0.1s	1.7s	1.4s	0.5s	×
trex3	2	0.1s	0.3s	×	2.2s	×	×
prog4	3	0.3s	0.5s	1.6s	×	×	0.1s
svd	2	0.5s	1.0s	19.1s	×	5.9s	×

Figure 4.12.: Evaluation using loop programs.

the sampling strategies used in the other learning based approaches do not work in higher-order cases. In the second and third steps, we compare with MoCHI and LiquidTypes, two state-of-the-art verification tools for higher-order programs.

4.6.1 Learning Benchmarks

We collected challenging loop programs found in an invariant learning framework ICE [44]. We list in Fig. 4.12 the programs that took more than 1s to verify in their tool. In the table, N and T are the number of CEGAR iterations and total time of our tool (L is the time in learning). And × means an adequate invariant was not found. We additionally compare our approach to CPA, a static verification tool [45] and three related learning based verification tools that are also based on the idea of inferring invariants as classifiers to good/bad sample program states: ICE [44], SC [46] and MC^2 [47]. Our tool outperforms ICE because it completely abstracts the inference of the Boolean structure of likely invariants while ICE requires to fix a Boolean template prior to learning; it outperforms SC because it guides samples generation via the CEGAR loop; it outperforms MC^2 due to its attempt to find minimal invariants from the samples for generalization.

Program	N	L	T	I	DI	MoCHI
ainit	4	1.9s	2.3s	5	4	5.7s
amax	4	0.6s	0.9s	5	2	2.4s
accpr	3	0.8s	1.1s	7	0	3.9s
fold_fun_list	3	0.2s	0.6s	5	0	3.7s
mapfilter	5	0.7s	1.2s	3	2	18.5s
risers	3	0.1s	0.3s	4	2	2.4s
zip	3	0.1s	0.2s	1	0	2.4s
zipunzip	3	0.1s	0.2s	1	0	1.7s

Figure 4.13.: Evaluation using MoCHI benchmarks.

4.6.2 MoCHI Higher-order Programs

To gauge the effectiveness of our protptype with respect to existing automated higher-order verification tools, we consider benchmarks encoded with complex higher-order control flow, reported from MoCHI [13], including many higher-order list manipulating routines such as `fold`, `forall`, `mem` and `mapfilter`.

We gather the MoCHI results on an Intel Xeon 5570 CPU with 6 GB memory, running an up-to-date MoCHI implementation, a machine notably faster than the environment for our system. A CEGAR loop in MoCHI performs dependent type inference [19,23] on spurious whole program counterexamples from which suitable predicates for refining abstract model are discovered based on interpolations [37]. However, existing limitations of interpolating theorem provers may confound MoCHI. For example, it fails to prove the assertion given in program in Fig. 4.8.

In Fig. 4.13, `N` and `T` are the number of CEGAR iterations and total time of our tool (`L` is the time spent in learning), `I` is the number of discovered type refinements, among which `DI` shows the number of disjunctive type refinements inferred. Column MoCHI shows verification time using MoCHI. Fig. 4.13 only lists results for which MoCHI requires more than 1 second. Our tool also takes less than 1s for the rest

```
let make_list m =
  if m <= 0 then [ ]
  else (m) :: make_list (m-1)

let make_list_list m =
  if m <= 0 then [ ]
  else make_list (m)::
           make_list_list (m-1)

let ne (xs: int list) =
  match xs with
    [ ] -> 1
  | x::xs -> 0

let filter p (xs: 'a list list) =
  match xs with
    [ ] -> [ ]
  | x::xs ->
    if p x >= 1 then filter p xs
    else x::(filter p xs)

let ok (xs: 'a list list) =
  match xs with
    [ ] -> [ ]
  | x::xs -> (assert (length x > 0);
    x :: ok xs)

let main m =
  ok (filter ne (make_list_list m))
```

Figure 4.14.: A case study of `mapfilter`.

of MoCHI benchmarks. Performance improvements range from 2x to 18x. We typically infer smaller and hence more readable types than MoCHI. In the case of `mapfilter`, where the performance differential is greatest, MoCHI spends 6.1s to find a huge dependent intersection type in its CEGAR loop. This results in an additional 10.7s spent on model checking. In contrast, our approach tries to learn a simple classifier from easily-generated samples to permit generalization.

Case Study.

The source code of `mapfilter` is given in Fig. 4.14. The implementation uses `make_list_list` to build a list of list. It then calls the `filter` function which filters out all the empty list using a filter function `p` which essentially binds to `ne`. In the `main` procedure, we use the `ok` function to recursively test a safety property: there should not exist any empty list in the result list of `filter`.

For simplicity, we only apply `wp` to unroll `filter` once. From the application `wp`(`ok` (`filter` $\cdots$), `false`), we obtain

$$\exists \texttt{xs}'.(\texttt{len xs} = 0 \wedge \texttt{false}) \vee (\texttt{len xs} = 1 + \texttt{len xs}' \wedge \texttt{len (hd xs)} > 0)$$

as the pre-bad condition for `ok`. The first disjunction corresponds to the empty list case in `ok` while the second disjunction corresponds to the nonempty list case, which encodes the negation of the assertion. Since only one iteration of `ok` is considered, the formula is an under-approximation of a sound bad-condition. We continue to apply `wp` and push the pre-bad condition of `ok` into the post-bad condition for `filter`:

$$\exists \texttt{xs}'.(\texttt{len } \nu = 0 \wedge \texttt{false}) \vee (\texttt{len } \nu = 1 + \texttt{len xs}' \wedge \texttt{len (hd } \nu) > 0)$$

where the special variable ν denotes the procedure's result list. A pre-bad condition for `filter` is then derived from it as:

$$\exists \texttt{xs}'.(\texttt{len xs} = 0 \wedge \texttt{false}) \vee (\texttt{len xs} = 1 + \texttt{len xs}' \wedge ((\texttt{p (hd xs)} \geq 1 \wedge \texttt{false}) \vee (\texttt{p (hd xs)} < 1 \wedge \texttt{len (hd xs)} \leq 0)))$$

This formula captures the `if-then-else` rule encoded in `wp` (some non-interesting disjunctions and quantifiers are ignored for simplicity). Unknown functions are encoded as uninterpreted.

From `filter`'s pre-bad condition, it is now obvious of how to call the unknown function `p` in the instrumentation phase: we will supply `hd xs` as the arguments; since it is a list, we will sample its *length* in the good states for `filter`. Let us focus on `filter`'s pre-condition inference. After running the program with (`main` 2), a set of good/bad samples is shown in Fig. 4.15 where each bad sample is generated as a model to the pre-bad condition via an SMT solver. Clearly, although `p` is considered uninterpreted, its input/output is still constrained via the term `len` (`hd xs`). Applying our learning algorithm, we can gradually find a likely invariants shown in

class	len p_0	p_r
Pos	2	0
	0	1
Neg	0	0

$$\texttt{len}\ p_0 > 0 \wedge p_r \leq 0 \vee$$
$$\texttt{len}\ p_0 \leq 0 \wedge p_r > 0$$

Figure 4.15.: Learning a specification for `mapfilter`.

the right column of Fig. 4.15. Together with a similar analysis for post-condition inference, we obtain the following refinement type for `filter`:

$$\texttt{p} : \{p_0 : \text{'a list} \rightarrow \{\texttt{int} \mid (\texttt{len}\ p_0 > 0 \wedge \nu \leq 0) \vee (\texttt{len}\ p_0 \leq 0 \wedge \nu > 0)\}\}$$
$$\rightarrow \texttt{xs} : \{\text{'a list list}\} \rightarrow \{\{\text{'a list} \mid \texttt{len}\ \nu > 0\}\texttt{list}\}$$

This type clearly specifies that the unknown function `p` returns an integer that is greater than 0 only if it takes an empty list as input and any nonempty list is filtered away in `filter`'s result. This type is sufficient to prove the assertion in `ok`.

We now show how higher-order model checking [13] (implemented in MoCHI) works for this program. MoCHI develops a CEGAR loop combined with predicate abstraction, which can be considered as a variant of finite state and pushdown model checking [16]. In CEGAR, MoCHI actually performs a dependent type inference [19, 23] on a spurious whole-program counterexample trace from which suitable predicates for refining the abstract model is discovered based on interpolations [37] derived from a theorem prover. For the `mapfilter` program, MoCHI spends 10.2s to find the correct abstraction refinement predicates in its CEGAR. However, we find suitable "interpolants" from a set of program states local to `filter` and do not depend on an interpolating theorem prover.

Using the abstract refinement predicates from CEGAR, predicate abstraction in MoCHI converts higher-order language semantics into a boolean program capable of being expressed as recursion schemes [11], a recursive tree grammar. The satisifiability of safety properties can be answered by a query from the generated tree. The

Program	LOC	An	LiqTyAn	T	Property
List	62	6	12	2s	Len1
Sieve	15	1	2	1s	Len1
Treelist	24	1	2	1s	Sz
Fifo	46	1	5	2s	Len1
Ralist	102	2	6	2s	Len1, Bal
Avl tree	75	3	9	20s	Bal, Sz, Ht
Bdd	110	5	14	13s	VOrder
Braun tree	39	2	3	1s	Bal,Sz
Set/Map	100	3	10	14s	Bal,Ht
Redblack	150	3	9	27s	Bal,Ht,Clr
Vec	310	15	39	110s	Bal,Len2,Ht

Figure 4.16.: Evaluation using data structure benchmarks.

tool spends 10.7s on model checking the converted program. The primary reason is that, based on the discovered interpolations, a large number of complex dependent intersection types [19] for `filter` are inferred, which challenges the underlying model checker. In contrast, our approach makes an effort to learn a simple classifier from samples to permit generalization. As a result, the time required by proving our verification conditions benefits from the concision of the invariants.

4.6.3 Inductive Functional Data Structure Programs

We further evaluate our approach on some benchmarks that manipulate data structures. `List` is a library that contains standard list routines such as `append`, `length`, `merge`, `sort`, `reverse` and `zip`. `Sieve` implements Eratosthene's sieve procedure. `Treelist` is a data structure that links a number of trees into a list. `Braun tree` is a variant of balanced binary trees. They are described in [3]. `Ralist` is a random-access list library. `Avl tree` and `Redblack` are implementation of two

balanced tree AVL tree and Redblack tree. `Bdd` is a binary decision diagram library. `Vec` is a OCaml extensible array libraray. These benchmarks are used for evaluation in [7]. `Fifo` is a queue structures maintained by two lists, adapted from the SML library [48]. `Set/Map` is the implementation of finite maps taken from the OCaml library [49].

We check the following properties: `Len1`, the various procedures appropriately change the length of lists; `Len2`, the vector access index is nonnegative and properly bounded by the vector length; `Bal`, the trees are recursively balanced (the definition of balance in different tree implementations varies); `Sz` or `Ht`, the functions coordinate to change the number of elements contained and the height of trees; `Clr`, the tree satisfies the redblack color invariant; `VOrder`, the BDD maintains the variable order property.

The results are summarized in Fig. 4.16. `LOC` is the number of lines in the program, `An` is the number of required annotations (for instrumenting data structure specifications), `T` is the total time taken by our system. LiqTyAn is the number of annotations optimized in LiquidTypes system. The number of annotations used in our system is reflected in column `An`. These annotations are simply the `property` in Fig. 4.16. Our experiment shows that we eliminate the burden of annotating a predefined set of likely invariants used to prove these properties, required in LiquidTypes, because we infer such invariants automatically.

For example, in the `Vec` library, an extensible array is represented by a balanced tree with balance factor of at most 2. To prove the correctness of its recursive balancing routine, `recbal ( l , v , r )`, which aims to merge two balanced trees (`l` and `r`) of arbitrarily different heights into a single balanced tree together with a data structure element `v`, our tool infers a complex invariant (equivalent to a 4-DNF formula) describing the result of `recbal`. Without that invariant, the refinement type checker will end up rejecting the correct implementation. In contrast, such a complicated invariant is required to be manually provided in LiquidTypes. Or, at least, the programmer has to provide the shape of the desired invariant (the tool then considers

all likely invariants of the presumed shape). The annotation burden of `recbal` in LiquidTypes is listed as below in which ν refers to the result of `recbal` and `ht` is a measure definition that returns the height a tree structure.

1. $\texttt{Bal}\,(\nu)(A : \texttt{vec}) : \texttt{ht}\ \nu\ \{\leq, \geq\}\ \texttt{ht}\ A\ \{-, +\}\ [1, 2, 3]$
2. $\texttt{Bal}\,(\nu) : \texttt{ht}\ \nu\{\geq, \leq\}(\texttt{ht}\ l >= \texttt{ht}\ r\ ?\ \texttt{ht}\ l : \texttt{ht}\ r)\ \{-, +\}\ [0, 1, 2]$
3. $\texttt{Bal}\,(\nu) : \texttt{ht}\ \nu \geq (\texttt{ht}\ l \leq \texttt{ht}\ r + 2 \wedge \texttt{ht}\ l \geq \texttt{ht}\ r - 2\ ?$
   $(\texttt{ht}\ l \geq \texttt{ht}\ r\ ?\ \texttt{ht}\ l : \texttt{ht}\ r) + 1 : 0)$
4. $\texttt{Bal}\,(\nu) : \texttt{ht}\ \nu \geq (\texttt{ht}\ l \geq \texttt{ht}\ r\ ?\ \texttt{ht}\ l : \texttt{ht}\ r) +$
   $(\texttt{ht}\ l \leq \texttt{ht}\ r + 2 \wedge \texttt{ht}\ l \geq \texttt{ht}\ r - 2\ ?\ 1 : [0, -1])$

The four annotations are already complex because the desired invariant of `recbal` must contain disjunctive clauses. Without suitable expertise, providing such annotations could be challenging. In comparison, our tool automatically generates a Boolean combination of the necessary atomic predicates parameterized from the hypothesis domain (parameterized from Equation 4.1). It learns invariants from sampling the program and closes the gap between the programmer's intuition and inference mechanisms performed by formal verification tools.

Fig. 4.16 does not show the time taken by LiquidTypes because it crucially depends on the relevance of user-provided invariants.

Limitations. There are a few limitations to our current implementation. First, we rely on an incomplete type system Sec. 2.2. In particular, our type system is not as complete as [50] which automatically adds ghost variables into programs to remedy incompleteness in the refinement type system. Second, our tool fails if our hypothesis domain is not sufficiently expressive to compute a classifier for an invariant. As part of future work, we plan to consider ways to gradually increases the expressivity of the hypothesis domain by parameterizing Equation 4.1. Third, this technique does not allow data structure measures to be defined as mappings from datatypes to sets

(e.g. a measure that defines all the elements of a list), preventing us from inferring properties like list-sorting, which requires reasoning about the relation between the head element and all elements in its tail.

4.7 Related Work

There has been much work exploring the incorporation of refinement types into programming languages. DML [3] proposed a sound type-checking system to validate programmer-specified refinement types. LiquidTypes [7] alleviates the burden for annotating full refinement types; it instead blends type inference with predicate abstraction [15], and infers refinement types from *conjunctions* of programmer-annotated Boolean predicates over program variables, following the Houdini approach [35].

There has also been substantial advances in the development of dependent type systems that enable the expression and verification of rich safety and security properties, such as Ynot [51], F* [52], GADTs and type classes [53, 54], albeit without support for invariant inference. The use of directed tests to drive the inference process additionally distinguishes our approach from these efforts.

Higher-order model checkers, such as MoCHI [13], compute predicate abstractions on the fly as a white-box analysis, encoding higher-order programs into recursion schemes [11]. Recent work in higher-order model checking [55] has demonstrated how to scale recursion schemes to several thousand rules. We consider the verification problem from a different angle, applying a black-box analysis to infer likely invariants from sampled states. In a direction opposite to higher-order model checking, HMC [9] translates type constraints from a type derivation tree into a first-order program for verification. However, 1) the size of the constraints might be exponential to that of the original program; 2) the translated program loses the structure of the original, thus making it difficult to provide an actual counterexample for debugging. Popeye [56] suggests how to find invariants from counterexamples on the original higher-order

source, but its expressiveness is limited to conjunctive invariants whose predicates are extracted from the program text.

Refinement types can also be used to direct testing, demonstrated in [40]. A relatively complete approach for counterexample search is proposed in [41] where contracts and code are leveraged to guide program execution in order to synthesize test inputs that satisfy pre-conditions and fail post-conditions. In comparison, our technique can only find first-order test inputs for whole programs. However, existing testing tools can not be used to guarantee full correctness of a general program.

Dynamic analyses can in general improve static analyses. The ACL2 [57] system presents a synergistic integration of testing with interactive theorem proving, which uses random testing to automatically generate counterexamples to refine theorems. We are in part inspired by YOGI [58], which combines testing and first-order model checking. YOGI uses testing to refute spurious counterexamples and find where to refine an imprecise program abstraction. We retrieve likely invariants directly from tests to aid automatic higher-order verification.

There has been much interest in learning program invariants from sampled program states. Daikon [59] uses conjunctive learning to find likely program invariants with respect to user-provided templates with sample states recorded along test runs. A variety of learning algorithms have been leveraged to find *loop* invariants, using both good and bad sample states: some are based on simple equation or template solving [44,60,61]; others are based on off-the-shell machine learning algorithms [36,46,47]. However, none of these efforts attempt to sample and synthesize complex invariants, in the presence of *recursive higher-order* functions.

5 AUTOMATICALLY LEARNING SHAPE SPECIFICATIONS

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 chapter, 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*).

To illustrate the issue, consider the `elements` function shown in Fig. 5.1. 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. 5.2.[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

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

[2] We preserve this convention throughout the chapter.

```
type 'a list =                  // flat:'a tree -> 'a list
 | Nil                          //              -> 'a list
 | Cons 'a *                    let rec flat accu t =
     'a list                     match t with
                                   | Leaf -> accu
type 'a tree =                     | Node (x, l, r) ->
 | Leaf                               flat(x::(flat accu r)) l
 | Node 'a *
     'a tree *                  // elements:'a tree->'a list
     'a tree                    let elements t = flat [ ] t
```

Figure 5.1.: Tree flattening function.

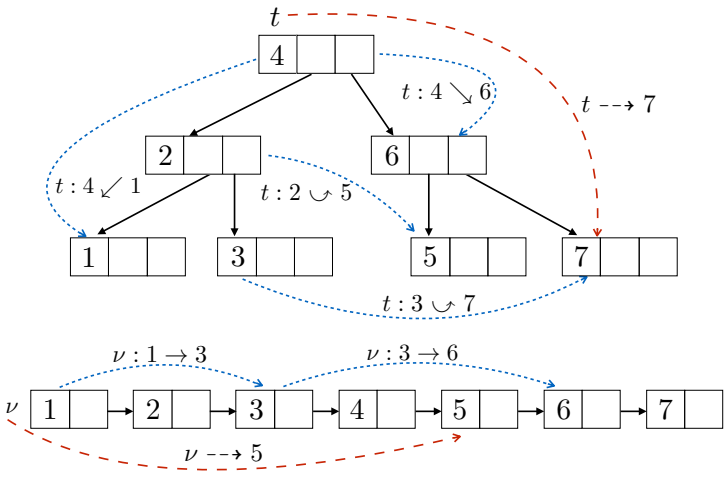


Figure 5.2.: Pictorial example of atomic shape predicates.

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 [62].

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-trivial [63, 64]. 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 [59,61,65] require us to provide the Boolean skeleton of these specifications *a priori*, which we often do not know. Classification-based learning techniques [44,46,47,66,67] 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. 5.1.

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. 5.3.2) 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 chapter provides an overview of our specification language (Sec. 5.1); explains the synthesis mechanism through a detailed example (Sec. 5.2); provides details about type system, verification procedure, as well as soundness and progress results (Sec. 5.3); and describes generalizations of the core technique, presents implementation results, related work and conclusions (Secs. 5.4, 5.5, and 5.6).

5.1 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 [68]. We discuss the basic intuition for how these predicates are extracted for the data types defined in our running example in Fig. 5.1 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. 5.1, 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. 5.3.

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 ei-

$\nu \dashrightarrow u$	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
$t : u \smile v$	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.

Figure 5.3.: Atomic shape predicates for lists and binary trees.

ther 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. 5.3, (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. 5.3, and it is denoted as $t : u \smile v$. 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.

The inductive definitions of the predicates obtained for lists and trees are presented in Fig. 5.4. 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

`list`	$l = \texttt{Nil}$	$l = \texttt{Cons}\,(u', l')$
$l \dashrightarrow u$	`false`	$u = u' \ \vee \ l' \dashrightarrow u$
$l : u \to v$	`false`	$(u = u' \wedge l' \dashrightarrow v) \ \vee \ l' : u \to v$

`tree`	$t = \texttt{Leaf}$	$t = \texttt{Node}\,(u', t_l, t_r)$
$t \dashrightarrow u$	`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`	$(u = u' \wedge t_r \dashrightarrow v) \vee$ $t_l : u \searrow v \vee \ t_r : u \searrow v$
$t : u \cup v$	`false`	$(t_l \dashrightarrow u \ \wedge t_r \dashrightarrow v) \vee$ $t_l : u \cup v \ \vee \ t_r : u \cup v$

Figure 5.4.: Ordering and containment for `list` and `tree`.

given for trees. For example, the predicate $t : u \cup v$ 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. 5.1) with the following two specifications:

$$\begin{aligned} &\big(\forall u,\ \nu \dashrightarrow u \iff t \dashrightarrow u\big) \\ &\big(\forall u\ v,\ \nu : u \to v \iff \begin{pmatrix} t : v \swarrow u \vee \\ t : u \cup v \ \vee \\ t : u \searrow v \end{pmatrix}\big) \end{aligned} \tag{5.1}$$

where the free variables u, v of Fig. 5.3 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. 5.4.2) 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. 5.2:

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

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:

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

where only well-typed predicates are considered (depending on the type of d).
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:

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

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

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

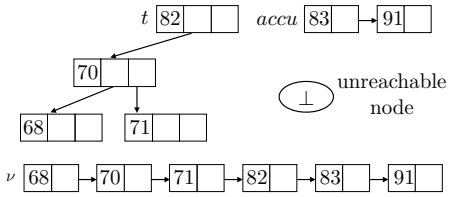
Specification Space. Assume that we denote with $BF(\Omega)$ the smallest set of Boolean formulas containing all the atomic predicates of Ω and closed by standard propositional logic connectives. The *specification space* of a function f, denoted by $Spec(\Omega, f)$, is the set of *input-output specifications*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

5.2 Specification Inference

Fig. 5.7 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. For simplicity, we use a *random testing* approach based on QuickCheck [31], 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



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

Π_I

$\Pi_0 \equiv t : u \swarrow v$ $\Pi_1 \equiv t : u \searrow v$ $\Pi_2 \equiv t : u \circlearrowright v$

$\Pi_3 \equiv t : v \swarrow u$ $\Pi_4 \equiv t : v \searrow u$ $\Pi_5 \equiv t : v \circlearrowright u$

$\Pi_6 \equiv t \dashrightarrow u$ $\Pi_7 \equiv t \dashrightarrow v$

$\Pi_8 \equiv accu : u \to v$ $\Pi_9 \equiv accu : v \to u$

$\Pi_{10} \equiv accu \dashrightarrow u$ $\Pi_{11} \equiv accu \dashrightarrow v$

Π_O

$\Pi_{12} \equiv \nu \dashrightarrow u$ $\Pi_{13} \equiv \nu : u \to v$

(b) Hypothesis domain ($\Omega($ `flat` $)$): $\Pi_I(\Omega($ `flat` $)) = \{\Pi_0, \cdots, \Pi_{11}\}$, and $\Pi_O(\Omega($ `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 Fig. 5.5b.

Figure 5.5.: Learning shape specifications for the `flat` function in Fig. 5.1.

	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 Fig. 5.5c

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

Figure 5.6.: Predicates selected for separation w.r.t. Π_{13}.

data structure function f into a sample set (which we generally denote with V_f). We assume the existence of generic serialization and deserialization functions, with the obvious recursive structure on the definition of the types. The bookkeeping of inputs and outputs simply records the mappings of variables to values, which in the case of inductive data structures uses their trivial serialization.

Our system analyzes the data type definitions in the program and *automatically generates a set of atomic predicates* (c.f. Sec. 5.1), 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. 5.1), 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. 5.7.

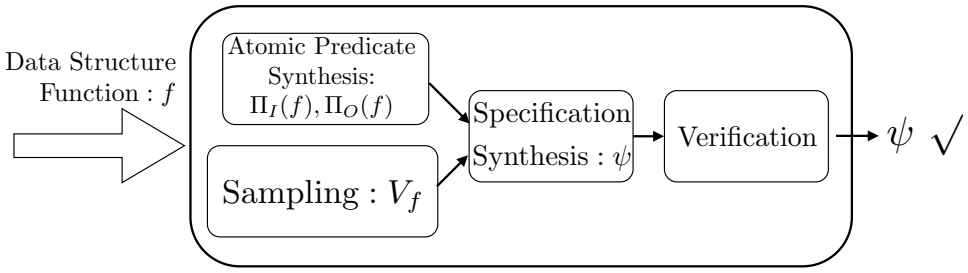


Figure 5.7.: Specification synthesis architecture.

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. 5.3 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. 5.1.

5.2.1 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.

Fig. 5.5a 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 ν (*i.e.* $\nu =$ `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 $\perp$, 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 $\perp$, 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 Fig. 5.5b,[3] we next relate observed samples with these predicates. Fig. 5.5c (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.[4] 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 $\perp$. 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 Fig. 5.5c, containing pairs with the synthetic value $\perp$, and marked in blue, generalize observed data structures,

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

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

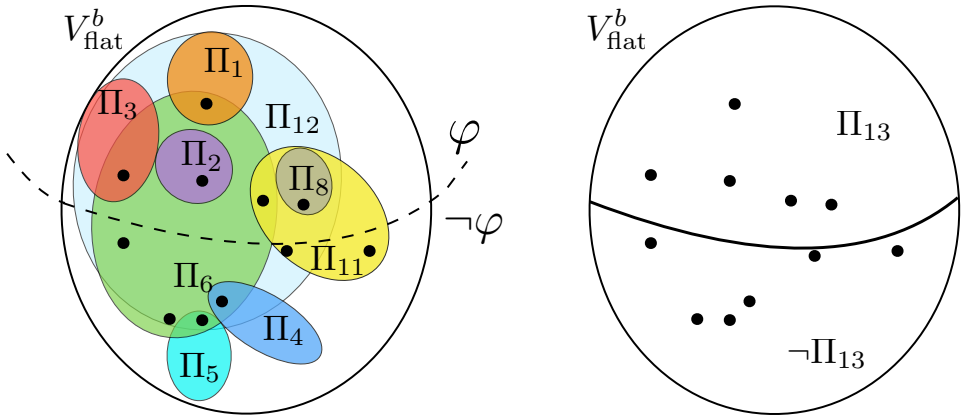


Figure 5.8.: Learning a classifier φ.

relating them to hypothetical elements $\perp$ not accessible by the data structures of `flat`. Thus, the pair $(82, \perp)$ evaluates to true in Π_6 because 82 is reachable in t; all ordering predicates related to t where $u = \perp$ or $v = \perp$ (*i.e.*, $\Pi_0 - \Pi_5$) are false since there is no ordering relation between 82 and a value unreachable from t (see Fig. 5.5a).

5.2.2 Learning Specifications

Fig. 5.8 depicts our specification learning algorithm w.r.t. Π_{13}, in which the full set of observed abstracted samples $V^b_{\texttt{flat}}$ are depicted twice. They represent identical copies of the whole space of abstract samples. Each dot represents a valuation of in Fig. 5.5c. Each set marked with a predicate Π represents the samples that satisfy Π. On the left hand side of the picture, we show the subsets of samples that satisfy each predicate from Π_0 to Π_{12}. For perspicuity, the picture does not present an exact representation of the sets shown in Fig. 5.5c; in particular some predicates not used in the final specification are omitted. 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}.[5]

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 Fig. 5.5c 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 [27, 33].

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. 5.2.3, but we show the final selection informally in Fig. 5.6: 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 Fig. 5.6. The truth table should accept all the samples in `S` from Fig. 5.6 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 Fig. 5.6). We omit this step in our example in Fig. 5.5.

Once this truth table is obtained for the selected predicates, we apply standard logic minimization [34] techniques to infer the Boolean structure of the classifier. The obtained solution is shown in Fig. 5.6, which in turn represents the following candidate specification by unfolding the definitions of the predicates Π_I and Π_O:

$$\left(\forall u\ v,\ \nu : u \to v \iff \begin{pmatrix} t : v \swarrow u \ \vee\ accu : u \to v \\ \vee\ (t \dashrightarrow u \ \wedge\ accu \dashrightarrow v) \\ \vee\ t : u \circlearrowright v \ \vee\ t : u \searrow v \end{pmatrix}\right) \tag{5.2}$$

[5] 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.

```
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 5.9.: Binary search tree insertion function.

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 Fig. 5.5b, 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:

$$\big(\forall u,\ \nu \dashrightarrow u \iff (t \dashrightarrow u \vee accu \dashrightarrow u)\big) \tag{5.3}$$

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. 5.3), which can validate specifications of this kind.

Precision. The structure of Fig. 5.5c 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. 5.9), whose hypothesis domain is shown in Fig. 5.10. 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 Fig. 5.11.

$$\Pi_I \begin{cases} \Pi_0 \equiv t : u \swarrow v \quad \Pi_1 \equiv t : u \searrow v \quad \Pi_2 \equiv t : u \circlearrowleft v \\ \Pi_3 \equiv t : v \swarrow u \quad \Pi_4 \equiv t : v \searrow u \quad \Pi_5 \equiv t : v \circlearrowleft u \\ \Pi_6 \equiv t \dashrightarrow u \quad \Pi_7 \equiv t \dashrightarrow v \\ \Pi_8 \equiv u = x \quad \Pi_9 \equiv v = x \end{cases}$$

$$\Pi_O \qquad \Pi_{10} \equiv \nu : u \swarrow v \qquad \cdots$$

Figure 5.10.: Hypothesis domain for the `insert` function.

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 Fig. 5.10. If we consider the first `S` sample in Fig. 5.11, we see that it is exactly the same as the first sample in `U` (except for the value of Π_{10}), meaning that no classifier can be generated from Fig. 5.11 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 Fig. 5.11 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 "$\Longleftrightarrow$" 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,

	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

Figure 5.11.: Partition $V^b_{\texttt{insert}}$ evaluated from predicates in Fig. 5.10 using Π_{10}.

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`:

$$\big(\forall u\ v,\ t : u \swarrow v \ \Rightarrow\ \nu : u \swarrow v\big) \wedge \big(\forall u\ v,\ \nu : u \swarrow v \ \Rightarrow \begin{pmatrix} (t \dashrightarrow u \wedge v = x) \vee \\ t : u \swarrow v \end{pmatrix}\big)$$

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.

5.2.3 Formalization of Learning System

We now formalize the learning algorithm discussed in Sec. 5.2.2. Given a function f and the hypothesis domain Ω (Sec. 5.1), 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 chapter, 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 [15] 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. 5.2.1) 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 4 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_f^b). 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 5 described below) to synthesize a candidate specification for f, which efficiently searches over the hypothesis domain of

Algorithm 4: Synthesize (f)

let $V_f = \texttt{test}(f)$ **in**;
let $V_f^b = \alpha(V_f, \Omega(f))$ **in**;
let ξ = Learn (V_f^b, $\Pi_I(\Omega(f))$, $\Pi_O(\Omega(f))$) **in** $(\forall\ u\ v,\ \xi)$

f, based on the valuation V_f^b. The resulting specification is returned after universally quantifying the free variables u and v.

Algorithm 5 takes as input a set of abstract samples (Boolean vectors) V^b, each of which is an assignment to 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_S^b which satisfy Π and the *unsat* samples V_U^b which do not. Each abstract sample $\sigma^b \in V_S^b \cup V_U^b$ is a Boolean vector over the predicates $\Pi_C \equiv \Pi_I \cup \Pi_O \backslash \{\Pi\}$.

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

$$\big(\forall \sigma^b \in V_S^b,\ \varphi(\sigma^b)\big)\ \ \&\ \ \big(\forall \sigma^b \in V_U^b,\ \neg\varphi(\sigma^b)\big)$$

In other words, the result of $L(V_S^b, V_U^b, \Pi_C)$ should be an interpolant [36] separating the *sat* samples (V_S^b) from the *unsat* samples (V_U^b). If this classification algorithm succeeds, $\Pi \iff L(V_S^b, V_U^b, \Pi_C)$ 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_S^b and V_U^b must be separable because there could be coinciding samples in V_S^b and V_U^b. To address this possibility, we first remove coinciding samples from V_U^b and infer $\Pi \Rightarrow L(V_S^b, V_U^b \backslash V_S^b, \Pi_C)$, and similarly remove them from V_S^b, resulting in the specification $L(V_S^b \backslash V_U^b, V_U^b, \Pi_C) \Rightarrow \Pi$. Algorithm 5 does not list the cases when $V_U^b \backslash V_S^b$ or $V_S^b \backslash V_U^b$ are empty. In such cases, it is impossible for

Algorithm 5: Learn (V^b, Π_I, Π_O)

$$\bigwedge_{\Pi \in \Pi_O} \Big(\textbf{let } (V_S^b, V_U^b) = \texttt{partition}(\Pi, V^b) \textbf{ in}$$
$$\textbf{let } \Pi_C = \Pi_I \cup \Pi_O \backslash \{\Pi\} \textbf{ in}$$
$$\textbf{if } V_S^b = \emptyset \textbf{ then } \neg\Pi$$
$$\textbf{else if } V_U^b = \emptyset \textbf{ then } \Pi$$
$$\textbf{else } \; (\Pi \Rightarrow L(V_S^b, (V_U^b \backslash V_S^b), \Pi_C)) \wedge$$
$$(L((V_S^b \backslash V_U^b), V_U^b, \Pi_C) \Rightarrow \Pi)\Big)$$

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 $L(V_S^b, V_U^b, \Pi_C)$ reduces to the well-studied problem of inferring a classifier separating some samples V_S^b from the other samples V_U^b using predicates form Π_C [46,67]. 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 Fig. 5.6. A number of off-the-shelf solvers can be used to solve this constraint optimization problem [69, 70]. We employ the simple classifier described in Sec. 4.3 to implement L.

5.3 Verification

To formally verify program specifications, we encode them into refinement types (`RType` in Fig. 2.1) 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 chapter, 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. 5.1.

Recall that a *refinement function type*, written $\{x : P_x \to P\}$, constrains the argument x by the refinement type P_x, and produces a result whose type is specified by P. We use the `specType` function Sec. 2.1 to encode a function specification into a refinement function type. For example, the specification (5.3) is encoded as the following type:

$$\texttt{flat} : \mathit{accu} : \texttt{'a list} \to t : \texttt{'a tree} \to \left\{\nu : \texttt{'a list} \mid \left(\forall u,\ \nu \dashrightarrow u \iff (t \dashrightarrow u \ \vee\ \mathit{accu} \dashrightarrow u)\right)\right\}$$

5.3.1 Refinement Type System

An excerpt of our refinement type system is given in Fig. 5.12. The type system is an extension of Liquid Types [7, 8]. 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. The full type system provides general rules for arbitrary inductive data types and is presented in Sec. 5.4.1.

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,

List Match

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

List Constructor (Cons)

$$\frac{\|\Gamma\| \Vdash x : \texttt{'a} \qquad \|\Gamma\| \Vdash xs : \texttt{'a list}}{\Gamma \vdash \texttt{Cons}(x, xs) : \left\{\ \nu : \texttt{'a list} \mid \begin{array}{l} \forall u\ v,\ \nu : u \to v \iff ((u = x \wedge xs \dashrightarrow v) \vee xs : u \to v) \\ \wedge \forall u,\ \nu \dashrightarrow u \iff (u = x \vee xs \dashrightarrow u) \end{array}\right\}}$$

Function

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

Subtype DType

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

Figure 5.12.: Refinement typing rules for shape specifications (list excerpt).

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 Fig. 5.4. In summary, the guard predicates unfold the inductive definitions introduced in Fig. 5.4. This strategy of unfolding inductive definitions when explicitly deconstructing (with pattern matching) data structures is reminiscent of the ones used in [8, 71, 72]. The typing rule for List Constructor follows the same idea as the rule for List Match in the type of the consequent.

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 [7, 8]:

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

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:

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

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 [73].

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 5.3.1 *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 [74]) based on first-order axiomatizations of transitive closures

found in [17, 18] to bound the shape of list or tree data structures in BSR models to ensure completeness.

5.3.2 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

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

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 5.3.2 under the assumption that the underlying solver is complete (c.f. Sec. 5.3.1).

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 [75], 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 [68, 76], ensures that *we can construct a finite number of test cases to guarantee convergence in the presumed specification space.*

Theorem 5.3.2 *Given a function f with a hypothesis domain Ω, and assuming that an inductive invariant of f exists in Spec(Ω, f), if $\Gamma_f \not\vdash$ `fix(fun` $f \rightarrow \lambda x.\ e$) : `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(Ω, f).*

Proof We begin the proof by firstly introducing several notations that will be used throughout the proof. Without loss of generality, we assume f is a recursive function.

Let Σ be the entire sample space of f (each sample collects an input-output behavior of f). We define the set of samples reachable up to the nth recursive call of f as $Reach(\varphi_{init}, n)$ where φ_{init} is the set of all the possible inputs to f (the most general precondition of f). Intuitively, the precondition φ_{init} represents all the valid input values of f. For example, if the type of a parameter of f is a tree, then φ_{init} defines that the corresponding input must be a tree instead of other data structures.

We assume ψ is the candidate invariant produced by our specification inference algorithm (Algorithm 4).

Let us define the predicate $\Psi(n)$, which is `true` if and only if the candidate invariant ψ can be invalidated in less than $n+1$ recursive calls to f, meaning the recursive call to f is unfolded at most n times:

$$\Psi(n) = \exists \sigma \in Reach(\varphi_{init}, n).\ \neg\psi(\sigma)$$

This predicate is `true` if there exists a test input (satisfying φ_{init} and we do not care the specific values of the input), which unfolds f's recursion n times and then produce a sample σ that renders ψ unsatisfiable. Essentially, the satisifiability of $\Psi(n)$ explains why ψ is not an invariant of f.

We now split the proof into two parts:

(A). To prove the theorem, when $\Gamma_f \not\vdash \texttt{fix}(\texttt{fun}\ f \rightarrow \lambda x.\ e) : \texttt{specType}(\Gamma_f, f, \psi)$ there must exist an input to f that invalidate ψ, we assume there exists no such test input to f that can produce at least one unseen program sample σ, for which $\psi(\sigma)$ does *not* hold. In turn, this means $\forall n.\ \Psi(n)$ is unsatisfiable. According to the definition of $\Psi(n)$, we have

$$\forall n. Reach(\varphi_{init}, n) \Rightarrow \psi$$

Recall that φ_{init} is the most general precondition of f. Because n can be lifted to infinity, ψ is obviously an invariant of f, possibly not an inductive invariant though.

We keep our proof simple by assuming $\Omega_O(f) = \{\Pi_O\}$, that is, we assume that there is only one predicate in $\Omega_O(f)$. The proof can be trivially extended to consider the case in which there are multiple predicates in $\Omega_O(f)$, due to the construction of Algorithm 5.

We also assume that $Spec(\Omega, f)$ is expressive enough to represent inductive invariants of f. Particularly, we assume that the strongest inductive invariant in $Spec(\Omega, f)$ is ψ_{ind}. Note that we are unaware of the contend of ψ_{ind}. We only know its existence. We can assume that ψ_{ind} is given in the following form, capturing the input-output relations of f,

$$\psi_{ind} = \forall \boldsymbol{u}\ \boldsymbol{v},\ \psi_{ind1} \wedge \psi_{ind2}$$
$$\psi_{ind1} = (\varphi_{ind1} \Rightarrow \Pi_O)$$
$$\psi_{ind2} = (\Pi_O \Rightarrow \varphi_{ind2})$$

Recall that ψ is the candidate invariant produced by our learning algorithm. Based on the construction of Algorithm 5, we can observe that ψ is of the form

$$\psi = \forall \boldsymbol{u}\ \boldsymbol{v},\ \psi_1 \wedge \psi_2$$
$$\psi_1 = (\varphi_1 \Rightarrow \Pi_O)$$
$$\psi_2 = (\Pi_O \Rightarrow \varphi_2)$$

where φ_1 and φ_2 are classifiers learned by the L algorithm (Sec. 4.3) invoked by Algorithm 5 (which range over $\Omega_I(f)$ in this case).

We denote by $\Omega(\varphi_{ind_i})$ the set of atomic predicates used to construct φ_{ind_i} where $i = 1, 2$. Each atomic predicate in $\Omega(\psi_{ind_i})$ belongs to the hypothesis domain of Algorithm 5. We also use $\Omega(\varphi_i)$ to denote the set of atomic predicates appearing in φ_i where $i = 1, 2$. Of course, these predicates belong to the hypothesis domain of our learning algorithm, following our assumptions.

Without loss of generality, in what follows we assume $\Omega(\varphi_i) \subseteq \Omega(\varphi_{ind_i})$ where $i = 1, 2$. This is valid because if there indeed exists an atomic predicate Π that belongs to $\Omega(\varphi_i) \backslash \Omega(\varphi_{ind_i})$ where $i = 1, 2$, we can add the predicate Π into φ_{ind_i} by rewriting φ_{ind_i} as $\varphi_{ind_i} \wedge (\Pi \vee \neg\Pi)$. Therefore we only need to consider the case in which $\Omega(\varphi_i) \subseteq \Omega(\varphi_{ind_i})$ holds where $i = 1, 2$.

We further claim that $\Omega(\varphi_i) \subset \Omega(\varphi_{ind_i})$ is not possible. Recall that our learning algorithm finds the minimum classifier φ_i in terms of the number of selected atomic predicates from $\Omega(f)$ to classify all the abstract Boolean samples S (which are derived from the predicate-abstraction function α defined in Sec. 5.2.3 applied to input-output samples that satisfy Π_O) and all the abstract Boolean samples U (which are derived from the α function applied to input-output samples that do not satisfy Π_O) where $i = 1, 2$. (Algorithm 5 carefully deals with the case in which there exist identical samples in S and U.) Besides, ψ_{ind} is assumed to be the strongest inductive invariant of f in $Spec(\Omega, f)$. If the number of predicates in $\Omega(\varphi_i)$ is less than that in $\Omega(\varphi_{ind_i})$, the samples used to build the classifier φ_i are not sufficient in the sense that, after removing coinciding samples, either S is less than the set of (Boolean) satisfiable assignments to the predicates in φ_{ind_i} (which are restricted to the predicates in $\Omega(\varphi_i)$ only) or U is less than the set of (Boolean) unsatisfiable assignments to the predicates in φ_{ind_i} (which are also restricted to the predicates in $\Omega(\varphi_i)$ only) where $i = 1, 2$. Otherwise, it is impossible that $\Omega(\varphi_{ind_i})$ involves more predicates than $\Omega(\varphi_i)$. If the S samples are not sufficient and consider $i = 2$, it is not possible that the candidate invariant $\forall \boldsymbol{u}\ \boldsymbol{v},\ \psi_2$ is a true invariant because Sec. 4.3 *rejects* any Boolean assignment

to the selected atomic predicates that are *not observed*; if the S samples are sufficient but the U samples are not sufficient and consider $i = 1$, for similar reasons, it is not possible that the candidate invariant $\forall \boldsymbol{u}\ \boldsymbol{v},\ \psi_1$ is a true invariant. As a result, the fact that either S or U is not sufficient therefore immediately contradicts our previous assumption that ψ is an invariant of f.

Hence, we only need to consider the case $\Omega(\varphi_i) = \Omega(\varphi_{ind_i})$ where $i = 1, 2$. Finally, we are able to give our 2-stage proofs.

Proof goal 1. *To prove* that $\psi_2 \Rightarrow \psi_{ind_2}$, it suffices to *show* that $\varphi_2 \Rightarrow \varphi_{ind_2}$.

Recall that φ_2 is constructed from a truth table $\mathcal{T}$ over the predicates in $\Omega(\varphi_2)$ or $\Omega(\varphi_{ind_2})$ using logic minimization based on our learning algorithm (Sec. 4.3). By abuse of notation, in the following, we choose the same variable $\mathcal{T}$ to represent the truth table over the predicates in $\Omega(\varphi_2)$ and the logic formula that it can be reduced to.

By our assumption that ψ_{ind} is an inductive invariant, we conclude that so is $\forall \boldsymbol{u}\ \boldsymbol{v},\ \psi_{ind_2}$. Recall that in Sec. 4.3, $\mathcal{T}$ is encoded using the α function from samples using the same atomic predicates that compose φ_{ind_2} (coinciding abstract samples between S and U removed from U only). We must have

$$\mathcal{T} \Rightarrow \varphi_{ind_2}$$

Otherwise, ψ_{ind} cannot be an invariant for f because it fails on input-output samples of f. Based on the fact that the logic minimization algorithm that we use [34] to get φ_2 from $\mathcal{T}$ ensures that φ_2 is logically equivalent to $\mathcal{T}$. We therefore conclude $\varphi_2 \Rightarrow \varphi_{ind_2}$, from which it is obvious that

$$\psi_2 \Rightarrow \psi_{ind_2}$$

Proof goal 2. *To prove* that $\psi_1 \Rightarrow \psi_{ind_1}$, it suffices to *show* that $\varphi_{ind_1} \Rightarrow \varphi_1$.

Recall that φ_1 is constructed from a truth table $\mathcal{T}$ over the predicates in $\Omega(\varphi_1)$ or $\Omega(\varphi_{ind_1})$, using logic minimization based on our learning algorithm (Sec. 4.3).

Again, by abuse of notation, in the following, we choose the same variable $\mathcal{T}$ to represent the truth table over the predicates in $\Omega(\varphi_1)$ and the logic formula that it can be reduced to.

Let us firstly assume that there exist some test input iv and two values $(u,\ v)$ such that $(iv, u, v) \in \varphi_{ind_1}$ (i.e., (iv, u, v) evaluates φ_{ind_1} to `true`) but $(iv, u, v) \notin \mathcal{T}$. Because we assumed that ψ_{ind} (and $\forall u\ v,\ \psi_{ind_1}$) is an inductive invariant of f, $(iv, f(iv), u, v)$ is a valid sample that *satisfies* Π_O. Then, if $(iv, u, v) \notin \mathcal{T}$ we have that $\mathcal{T}$ was not obtained from a sufficient test suite for inferring an inductive invariant of f. Particularly, $\forall u\ v,\ \psi_2$ (here ψ_2 is $\Pi_O \Rightarrow \varphi_2$ where φ_2 is also computed from the samples) cannot be a true invariant because a Boolean sample in $\alpha((iv, f(iv)), \Omega(f))$ which satisfies Π_O is not included in the Boolean samples to learn φ_2.

We conclude by contradiction that there must exist no test input iv (and u, v) such that $(iv, u, v) \in \varphi_{ind_1}$ but $(iv, u, v) \notin \mathcal{T}$. Recall that in Sec. 4.3, $\mathcal{T}$ is encoded using the α function from samples using the same atomic predicates that compose φ_{ind_1}. We are therefore able to claim that

$$\mathcal{T} \Leftarrow \varphi_{ind_1}$$

The logic minimization algorithm [34] guarantees that $\mathcal{T}$ is logically equivalent to φ_1. We can therefore conclude that $\varphi_{ind_1} \Rightarrow \varphi_1$, from which it is obvious that

$$\psi_1 \Rightarrow \psi_{ind_1}$$

Now we combine the above 2-stage proofs and obtain,

$$(\psi_1 \Rightarrow \psi_{ind_1}) \wedge$$
$$(\psi_2 \Rightarrow \psi_{ind_2})$$

As a result,

$$\psi \Rightarrow \psi_{ind}$$

(The above conclusions essentially claim that our learning algorithm *rejects* any Boolean assignment to the atomic predicates in ψ that are *not observed or inconsistent with the samples.*)

Because we assumed that ψ_{ind} is the strongest inductive invariant in $Spec(\Omega, f)$ and ψ is at least an invariant for the f function learnt by Algorithm 4, ψ is then an inductive invariant as well. However, this contradicts our assumption that $\Gamma_f \nvdash \texttt{fix}\,(\texttt{fun}\ f \rightarrow \lambda x.\ e) : \texttt{specType}(\Gamma_f, f, \psi)$ (recall that we assume the underlying solver is complete). So there must exist a value of n, under which $\Psi(n)$ holds, i.e., there exists a test input to f, which can unfold the f's recursion n times and then produce at least one unseen program sample σ, for which $\psi(\sigma)$ does *not* hold (we can find such a value of n by the validation procedure from [76] based on bounded model checking or the relatively complete test generation approach in [75]).

(B). We now prove that, if $\Gamma_f \vdash \texttt{fix}\,(\texttt{fun}\ f \rightarrow \lambda x.\ e) : \texttt{specType}(\Gamma_f, f, \psi)$ holds, then ψ is the strongest inductive invariant for f in $Spec(\Omega, f)$.

In this case, because ψ which is inferred by Algorithm 4 is already an invariant, we can then reuse our proof above (Proof part A) to conclude that, if ψ_{ind} is the strongest inductive invariant in $Spec(\Omega, f)$,

$$\psi \Rightarrow \psi_{ind}$$

Trivially, ψ must be the strongest inductive invariant in $Spec(\Omega, f)$.

Our proof completes by incorporating proof part A and proof part B. ■

The key idea of the above proof 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. 5.1. 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 (5.2), 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. 5.12. 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 `U(nsat)` samples of $\nu : u \to v$.

5.4 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.

5.4.1 Arbitrary User-defined Inductive Data Structures

The language in Fig. 2.1 supports arbitrary user-defined inductive data types D at the type level. And we use $\mathcal{C}$ to represent data type constructors. To simplify the presentation, Fig. 2.1 only considers polymorphic inductive data type definitions, and requires all type variables (`'a`) to appear before all the data types in constructor expressions.

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

We first present the general definitions for ordering and containment predicates. For a data structure $\mathcal{C}_h\langle\vec{x},\vec{d}\rangle$, its *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. 5.1.

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),

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

Figure 5.13.: Definitions of shape predicates for list and tree.

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. 5.1 as shown in Fig. 5.13.

Refinement Type System. The type refinements of each constructor are returned by the *ty* function. Specifically, the type refinements of a constructor $\mathcal{C}_i$ are conjunctive aggregations of its ordering and containment relation definitions. To help us pre-

cisely define the ${\psi_\tau}^{\texttt{Contain}}$ and ${\psi_\tau}^{\texttt{Ord}}$ relations, as illustrated in the bottom of Fig. 2.3, we assume the presence of a globally-defined finite map (`Ord`) that maps data types τ to all possible tuples of its constructors and their arguments indices, i.e. $(\mathcal{C}, n, m)$ in which $\mathcal{C}$ is a data type constructor of τ, and n, m are the indices (place) of the arguments to the constructor $\mathcal{C}$ (starting from 1). Obviously, we require n, m to be no less than 1 and no greater than the total size of the arguments of $\mathcal{C}$. For example, (`Node`, 2, 3) is a triple in `Ord(tree)` using the `Node` constructor, and the left and right subtrees indices.

The definitions ${\psi_\tau}^{\texttt{Contain}}$ and ${\psi_\tau}^{\texttt{Ord}}$ essentially unfold the inductive definitions of ordering and containment predicates. The unfolding is in line with the general definitions discussed in Sec. 5.4.1. Notably, through the call to `Ord`, the definitions exhaust all the possible ordering relations from the constructors of the data type of v.

For instance, considering the list case with constructors `Nil` and `Cons`. We have,

$$ty(\texttt{Cons}) = x : \texttt{'a} \rightarrow y : \texttt{'a list} \rightarrow \{\nu : \texttt{'a list} \mid \psi_c\}$$
$$ty(\texttt{Nil}) = \{\nu : \texttt{'a list} \mid \psi_n\}$$

where the type refinement ψ_n for the `Nil` constructor is

$$\psi_n = (\forall u\ v,\ \nu : u \rightarrow v \iff \texttt{false}$$
$$\wedge\ \forall u,\ \nu \dashrightarrow u \iff \texttt{false})$$

and the type refinement ψ_c for the `Cons` constructor (recall that x and y are bound by `Cons`) is

$$\psi_c = (\forall u,\ \nu \dashrightarrow u \iff (u = x \vee y \dashrightarrow u)$$
$$\wedge\ \forall u\ v,\ \nu : u \rightarrow v \iff$$
$$((u = x \wedge y \dashrightarrow v) \vee y : u \rightarrow v))$$

Note that the ordering and containment predicates are encoded as *uninterpreted* relations in the refinement logic. For any arbitrary values, their ordering and containment relations with respect to a data structure are guaranteed, by the construction

T-Match

$$\frac{\begin{array}{c}\Gamma \vdash v : P_v \qquad \tau \equiv \|P_v\| \equiv \mu t\ \Sigma_i\ \mathcal{C}_i\langle {}'\vec{\mathtt{a}}, \vec{D}_i\rangle \\ \forall i.\ \Gamma \vdash \mathcal{C}_i\ :\ \vec{x_i} : {}'\vec{\mathtt{a}}\ \rightarrow\ \vec{d_i} : \vec{D_i}\ \rightarrow\ \{\nu : \tau \mid \psi_\tau{}^{\mathtt{Contain}}(\nu,\ \mathcal{C}_i\langle \vec{x_i}, \vec{d_i}\rangle)\ \wedge\ \psi_\tau{}^{\mathtt{Ord}}(\nu,\ \mathcal{C}_i\langle \vec{x_i}, \vec{d_i}\rangle)\} \\ \Gamma \vdash P \qquad \forall i.\ \Gamma_i\ =\ \vec{x_i} : {}'\vec{\mathtt{a}};\ \vec{d_i} : \vec{D_i};\ [v/\nu](\psi_\tau{}^{\mathtt{Contain}}(\nu,\ \mathcal{C}_i\langle \vec{x_i}, \vec{d_i}\rangle)\ \wedge\ \psi_\tau{}^{\mathtt{Ord}}(\nu,\ \mathcal{C}_i\langle \vec{x_i}, \vec{d_i}\rangle)) \\ \forall i.\ \Gamma,\ \Gamma_i\ \vdash e_i : P\end{array}}{\Gamma \vdash \mathtt{match}\ v\ \mathtt{with}\ \big|_i\ \mathcal{C}_i\langle \vec{x_i}, \vec{d_i}\rangle \rightarrow e_i : P}$$

T-Constructor

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{C}_h\ :\ \vec{x} : {}'\vec{\mathtt{a}}\ \rightarrow\ \vec{d} : \vec{D}\ \rightarrow\ \{\nu : \tau \mid \psi_\tau{}^{\mathtt{Contain}}(\nu,\ \mathcal{C}_h\langle \vec{x}, \vec{d}\rangle)\ \wedge\ \psi_\tau{}^{\mathtt{Ord}}(\nu,\ \mathcal{C}_h\langle \vec{x}, \vec{d}\rangle)\} \\ \|\Gamma\| \Vdash \vec{x} : {}'\vec{\mathtt{a}} \qquad \|\Gamma\| \Vdash \vec{d} : \vec{D}\end{array}}{\Gamma \vdash \mathcal{C}_h\langle \vec{x}, \vec{d}\rangle : \left\{\ \nu : \tau\ \middle|\ \psi_\tau{}^{\mathtt{Contain}}(\nu,\ \mathcal{C}_h\langle \vec{x}, \vec{d}\rangle)\ \wedge\ \psi_\tau{}^{\mathtt{Ord}}(\nu,\ \mathcal{C}_h\langle \vec{x}, \vec{d}\rangle)\right\}}$$

$$\psi_\tau{}^{\mathtt{Ord}}(\nu,\ \mathcal{C}_i\langle \vec{x_i}, \vec{d_i}\rangle) = \bigwedge_{\mathcal{C}, n, m \in \mathtt{Ord}(\tau)} \forall u\ v,\ \nu : u@n \xrightarrow{\mathcal{C}} v@m \iff$$

$$\left(\left(\bigvee_{q=1}^{l_i} d_{iq} : u@n \xrightarrow{\mathcal{C}} v@m\right)\ \vee\ \left\{\begin{array}{cl} x_{in} = u \wedge x_{im} = v & \text{if } \mathcal{C}_i = \mathcal{C} \\ & \wedge\ n \leq k_i \\ & \wedge\ m \leq k_i \\ x_{in} = u \wedge d_{im-k_i} \dashrightarrow v & \text{if } \mathcal{C}_i = \mathcal{C} \\ & \wedge\ n \leq k_i \\ d_{in-k_i} \dashrightarrow u\ \wedge d_{im-k_i} \dashrightarrow v & \text{if } \mathcal{C}_i = \mathcal{C} \\ \mathtt{false} & \text{otherwise}\end{array}\right\}\right)$$

where $k_i = |\vec{x_i}|$ and $l_i = |\vec{d_i}|$

$$\psi_\tau{}^{\mathtt{Contain}}(\nu,\ \mathcal{C}_i\langle \vec{x_i}, \vec{d_i}\rangle) = \left(\ \forall u,\ \nu \dashrightarrow u \iff \bigvee_{p=1}^{k_i} x_{ip} = u\ \vee\ \bigvee_{q=1}^{l_i} d_{iq} \dashrightarrow u\ \right)$$

where $k_i = |\vec{x_i}|$ and $l_i = |\vec{d_i}|$

Figure 5.14.: Refinement typing rules for shape specifications.

in Sec. 5.4.1, decidable and therefore we can soundly use them as uninterpreted relations in the refinement logic. Also notice that we can define *multiple* ordering and containment relations for a data type and we can simply conjoin the relational terms for each relation when refining each data type constructor.

The T-MATCH rule stipulates that the entire expression has type P if and only if P is well-formed in the type environment, and that, for each case expression e_i of the match, e_i must also have type P in the type environment extended with the guard predicate that captures the relation between the ordering and containment definition of the matched expression and the variables bound by the matched pattern.

We can reduce the T-MATCH rule to the following form, tailored for the list case.

T-LIST-MATCH

$$\frac{\Gamma \vdash v : P_v \qquad \tau \equiv \|P_v\| \equiv \texttt{'a list} \qquad \Gamma \vdash \texttt{Nil} \ : \ \{\nu : \tau \mid \psi_n\} \qquad \Gamma \vdash \texttt{Cons} \ : \ x : \texttt{'a} \rightarrow y : \texttt{'a list} \rightarrow \{\nu : \tau \mid \psi_c\} \qquad \Gamma \vdash P \qquad \Gamma_c \ = \ x : \texttt{'a}; \ y : \texttt{'a list}; \ [v/\nu]\psi_c \qquad \Gamma_n \ = \ [v/\nu]\psi_n \qquad \Gamma; \Gamma_c \ \vdash e_1 : P \qquad \Gamma; \Gamma_n \ \vdash e_2 : P}{\Gamma \vdash \texttt{match} \ v \ \texttt{with} \ \texttt{Cons}\,(x, y) \ \rightarrow \ e_1 \mid \texttt{Nil} \rightarrow e_2 : P}$$

It is apparent that the above rule can be further simplified to the LIST MATCH rule defined in Fig. 5.12. Observe that the T-LIST-MATCH rule type checks each branch of the `match` expression under an environment that records the corresponding branch condition. Additionally, the type environment for the `Cons` branch is also extended with the types of matched pattern variables (x and y). The branch condition for the `Cons` case is obtained by substituting the test value (v) for the bound variable (ν) in the type refinement of `Cons`. Intuitively, the branch condition of `Cons` captures the fact that the value v was obtained by applying the constructor `Cons`; therefore, it should satisfy the invariant of `Cons`. The `Nil` case is similar.

The typing rule for T-CONSTRUCTOR follows the same idea as the rule for T-MATCH in the type of the consequent. It describes how inductively constructed data structures should be type checked. Notice that for each possible ordering and containment definition of the corresponding inductive data type, a predicate specifying

the relation between the constructed expression and the variables bound by the constructor arguments is offered.

5.4.2 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:

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

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:

$$\begin{aligned}\Pi_{\texttt{shape}}(f) =& \{d \dashrightarrow u, d \dashrightarrow v, d : u \rightarrow v, \\ & d : u \swarrow v, d : u \searrow v, d : u \curvearrowright v \mid d \in \theta_D(f)\}\end{aligned}$$

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 5 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. 5.2.3 based on the predicates from $\Pi_{\texttt{data}}(f) \cup \Pi_{\texttt{shape}}(f)$.

To discharge the candidate specifications produced by this domain, following [64, 77], 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.

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

$$\Pi_{\texttt{shape}} \begin{cases} \Pi_4 \equiv t : u \swarrow v & \Pi_5 \equiv t : u \searrow v & \Pi_6 \equiv t : u \circlearrowleft v \\ \Pi_7 \equiv \nu : u \swarrow v & \Pi_8 \equiv \nu : u \searrow v & \Pi_9 \equiv \nu : u \circlearrowleft v \\ \multicolumn{3}{c}{\Pi_{10} \equiv t \dashrightarrow u \quad \Pi_{11} \equiv \nu \dashrightarrow u} \end{cases}$$

Figure 5.15.: Hypothesis domain for synthesizing shape and data specifications.

Consider the binary search tree `insert` function in Fig. 5.9. Fig. 5.15 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 $\texttt{BST}(t)$. 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`:

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

5.4.3 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 Vec library implementation (Sec. 5.5).

```
let rec bal l v r =
  let hl = ht l in
  let hr = ht r in
  if hl > hr + 2 then
    ... /* call bal on subtrees of l and r */
  else if hr > hl + 2 then
    ... /* call bal on subtrees of r and l */
  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 5 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:

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

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

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

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 $\texttt{bal}$ is guaranteed to return a balanced tree.

5.5 Experiments

DOrder is an implementation of our learning procedure and type-based verification technique.[6] We use the Z3 SMT solver [78] 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 5.3.2 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 [31]. 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 Fig. 5.16) 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 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.

[6] 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	54	8	1s	1s	$\texttt{O}_{\texttt{rd}}$
Lazy Queue	28	91	14	4s	4s	$\texttt{O}_{\texttt{rd}}$
List Lib	133	306	54	7s	10s	$\texttt{O}_{\texttt{rd}}$
List Set	51	96	50	11s	17s	$\texttt{O}_{\texttt{rd}}$, Set
Quicksort	19	49	25	1s	5s	$\texttt{O}_{\texttt{rd}}$, Sorted
Mergesort	30	32	11	1s	5s	$\texttt{O}_{\texttt{rd}}$, Sorted
Insertionsort	12	22	8	1s	1s	$\texttt{O}_{\texttt{rd}}$, Sorted
Selectionsort	22	32	11	1s	2s	$\texttt{O}_{\texttt{rd}}$, Sorted
Heap_1	85	139	48	37s	133s	$\texttt{O}_{\texttt{rd}}$, Min, Heap
Heap_2	77	70	24	5s	28s	$\texttt{O}_{\texttt{rd}}$, Min, Heap
Heapsort	37	81	28	9s	29s	$\texttt{O}_{\texttt{rd}}$, Sorted, Heap
Leftist Heap	43	106	32	12s	18s	$\texttt{O}_{\texttt{rd}}$, Min, Heap
Skew Heap	32	71	25	16s	22s	$\texttt{O}_{\texttt{rd}}$, Min, Heap
Splay Heap	58	98	44	9s	38s	$\texttt{O}_{\texttt{rd}}$, Min, BST
Pairing Heap	42	49	21	1s	7s	$\texttt{O}_{\texttt{rd}}$, Min, Heap
Binomial Heap	70	107	34	5s	26s	$\texttt{O}_{\texttt{rd}}$, Min, Heap
Treap	107	95	17	20s	39s	$\texttt{O}_{\texttt{rd}}$, BST
AVL Tree	176	127	39	27s	56s	$\texttt{O}_{\texttt{rd}}$, BST
Splay Tree	127	110	56	45s	170s	$\texttt{O}_{\texttt{rd}}$, BST
Braun Tree	75	111	42	19s	53s	$\texttt{O}_{\texttt{rd}}$, BST
Redblack Tree	228	260	81	53s	177s	$\texttt{O}_{\texttt{rd}}$, BST
OCaml Set	313	457	73	56s	134s	$\texttt{O}_{\texttt{rd}}$, BST, Min, Set
Proposition	58	94	8	2s	5s	$\texttt{O}_{\texttt{rd}}$
Randaccesslist	73	142	19	4s	7s	$\texttt{O}_{\texttt{rd}}$

Figure 5.16.: Experimental results on inferring shape specifications.

Results. In Fig. 5.16, 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

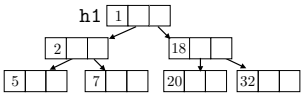
of verified ordering specifications in terms of either input-output or shape-data relations. T is the total time taken (learning and verification), while LT is the time spent solely on learning (including the time spent in sampling). Inferred Spec summarizes the learnt and verified specifications by DOrder.

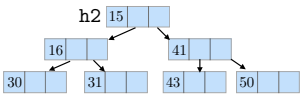
On the benchmarks, DOrder inferred the following specifications: (a) $\texttt{O}_{\texttt{rd}}$: specifications expressed using ordering and containment predicates. For instance, the specification for a balanced tree insertion function ensures that *the output tree preserves the in-order of the input 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 $\texttt{O}_{\texttt{rd}}$ 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 Fig. 5.16 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. Our technical report [79] provides detailed case studies for several of these benchmarks with more complex specifications discovered.

```
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)
```





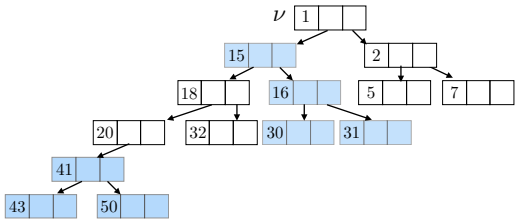


Figure 5.17.: Skew heap with input-output samples of `merge`.

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. 5.17.

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. 5.17. This operation merges two input skew heaps `h1` and `h2` into one output skew heap ν.

DOrder inferred the following functional specifications for the `merge` function from the samples in Fig. 5.17, 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. 5.17, 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 ($\nu : u \smile v$), 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):

$$\texttt{merge} : h1 : \texttt{'a tree} \rightarrow h2 : \texttt{'a tree} \rightarrow \Big\{ \nu : \texttt{'a tree} \mid$$
$$\big(\forall u,\ \nu \dashrightarrow u \iff (h1 \dashrightarrow u \vee h2 \dashrightarrow u)\big) \wedge$$
$$\big(\forall u\ v,\ \nu : u \swarrow v \Rightarrow \begin{pmatrix} h1 : u \swarrow v \ \vee\ h1 : u \searrow v \vee \\ h2 : u \swarrow v \ \vee\ h2 : u \searrow v \vee \\ (h1 \dashrightarrow u \ \wedge\ h2 \dashrightarrow v) \vee \\ (h2 \dashrightarrow u \ \wedge\ h1 \dashrightarrow v) \end{pmatrix}\big) \ \wedge$$
$$\big(\forall u\ v,\ \nu : u \searrow v \Rightarrow \begin{pmatrix} h1 : u \swarrow v \ \vee\ h1 : u \searrow v \vee \\ h2 : u \swarrow v \ \vee\ h2 : u \searrow v \end{pmatrix}\big) \wedge$$
$$\big(\forall u\ v,\ \nu : u \smile v \Rightarrow \begin{pmatrix} h1 : u \smile v \ \vee\ h1 : v \smile u \vee \\ h2 : u \smile v \ \vee\ h2 : v \smile u \vee \\ (h1 \dashrightarrow u \ \wedge\ h2 \dashrightarrow v) \vee \\ (h2 \dashrightarrow u \ \wedge\ h1 \dashrightarrow v) \end{pmatrix}\big) \Big\}$$

The specification reflects 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 [8] and compare the specifications discovered by DOrder with those inferred by [8]. The benchmark suites include realistic data structure implementations such as Bdd, a binary decision diagram library, and Vec, a dynamic functional array library.

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 Tree	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

Figure 5.18.: Experimental results on inferring numeric specifications.

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. The results are collected in Fig. 5.18, whose column interpretation is identical to Fig. 5.16. 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.

DOrder inferred and verified a number of measure based specifications in these programs, reflected in column I in Fig. 5.18, 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. The previous work [67] can also verify these benchmarks but requires user-provided assertions and a complex symbolic execution algorithm to drive so-called bad program states.

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.

5.6 Related Work

Learning Based Invariant Inference. Compared to earlier sampling-based approaches [80–83] 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 [59] and [61,65,84–86] 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 [44, 46, 47, 66, 67, 87] (in order to rule out program states not seen in normal executions) or non-commutativity conditions [88] 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. Moreover, 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. Fig. 5.11); 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 5.3.2). 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.

Relational Data Structure Verification. Our technique is closely related to [64, 77], 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 [63]. 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 [89–94]. These non-learning based techniques differ substantially from ours, owing to the nature of pointer manipulations in their imperative program model.

Static Analysis. 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 [3, 8, 17, 18, 71, 72, 95–104]. Our approach complements these tools with an inference procedure that can learn specifications for fully automatic data structure verification.

Following the Houdini approach [35], the LiquidTypes system [8, 62, 105] blends type inference for data structures with predicate abstraction, and infers refinement types from conjunctions of programmer-annotated predicates. To infer more ex-

pressive invariants, [106] 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 [44,107–109]. They either require the programmer to provide non-trivial post-conditions [44, 108, 109] or lack a notion of *progress* (c.f. Sec. 5.3.2) [107]. Unlike other static synthesis techniques that perform shape analyses on the source code [110–115], DOrder discovers shape specifications entirely from tests.

6 CONCLUSIONS AND FUTURE WORK

In Chapter 3, we present a compositional inter-procedural verification technique for functional programs. We use refinement type checking rules to generate refinement type templates for local expressions inside a procedure. Refinement subtyping rules are then used to generate verification conditions. From an unprovable verification condition, we can construct a counterexample path to infer refinement types for procedure arguments and results, and to propagate inferred specifications between procedures and call-sites where they are applied. Thus, our technique effectively leverages a variety of strategies used in the verification of first-order imperative programs within a higher-order setting.

In future work, we plan to incorporate more first-order verification techniques into our framework.

Chapter 4 presentes a new CEGAR based framework that integrates testing with a refinement type system to automatically infer and verify specifications of higher-order functional programs using a lightweight learning algorithm as an effective intermediary. Our experiments demonstrate that this integration is efficient.

In future work, we plan to integrate our idea into more expressive type systems. The work of [116] shows that a refinement type system can verify the type safety of higher-order **dynamic languages** like Javascript. However, it does not give an inference algorithm. It would be particularly useful to adapt the learning based inference techniques shown here to the type system for dynamic languages, relieving the annotation burden for non-trivial specifications and proof terms.

Chapter 5 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 tests, DOrder can synthesize sophisticated and high quality shape specifications for versatile data structure manipulating functions with reasonable cost.

For future work, we would like to extend DOrder for specification synthesis for **software defined networking programs (SDN)**. In DOrder, we investigate reachability (*i.e.,* ordering) relations between data structure elements. It is possibly to infer reachability relation between network hosts with DOrder. We can synthesize specification like "a network host A can receive a packet from a network host B only if A sent some packets to B previously." Specifications of suck kind are the key to prove functional correctness of SDN programs.

We also propose to build a refinement session type system for complex **distributed systems**, which should precisely prescribe the communication behaviors between concurrent message-passing processes. The type system should also soundly verify session protocols when spawning new processes is allowed. Accurately blaming undesired actions in these systems via the refinement type system is an important challenge.

It is also important to generalize our technique for checking **security properties and source-sink specifications**. An instantiation of DOrder to synthesize information-flow properties from concrete executions would possibly scale to complex iOS, Android and Java software systems.

Automatically synthesized specifications are unable to prove user-supplied properties if programs are indeed buggy. It is possible to identify a few locations closely

related to verification failures using Max-SMT solvers [78]. We can leverage program synthesis techniques for possible **bug-fixes** in these locations and exploit synthesized specifications in the context of these locations to help the synthesizers. Our strategy is able to combine the specification synthesis technique proposed here with state-of-the-art program synthesis techniques.

We are very interested in applying more advanced machine learning techniques to improve software systems' robustness. Software source code is usually accompanied with software documents. While we showed that high-quality specifications can be discovered from code, it is possible to learn language models from documents, using natural language processing (NLP) or information retrieval (IR) techniques. We can then measure the consistency between specification models (from code) and language models (from documents), in order to verify whether software code is compliant with software documents.

REFERENCES

REFERENCES

[1] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *The Royal Society Discussion Meeting on Mathematical Logic and Programming Languages*, 1985.

[2] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

[3] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[4] Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1982.

[5] O. Shivers. Control-Flow Analysis in Scheme. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[6] Knowles, Kenneth and Flanagan, Cormac. Type Reconstruction for General Refinement Types. In *European Symposium on Programming*, 2007.

[7] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[8] Ming Kawaguci, Patrick Rondon, and Ranjit Jhala. Type-based Data Structure Verification. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[9] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. HMC: Verifying Functional Programs Using Abstract Interpreters. In *International Conference on Computer Aided Verification*, 2011.

[10] Naoki Kobayashi. Model-Checking Higher-order Functions. In *Principles and Practice of Declarative Programming*, 2009.

[11] Naoki Kobayashi. Types and Higher-order Recursion Schemes for Verification of Higher-order Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[12] Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. Higher-order Multi-Parameter Tree Transducers and Recursion Schemes for Program Verification. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[13] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate Abstraction and CEGAR for Higher-order Model Checking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[14] C.-H. Luke Ong and Steven James Ramsay. Verifying Higher-order Functional Programs with Pattern-matching Algebraic Data Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[15] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *International Conference on Computer Aided Verification*, 1997.

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003.

[17] Shuvendu Lahiri and Shaz Qadeer. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[18] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic with Trees and Data. In *International Conference on Computer Aided Verification*, 2014.

[19] Tachio Terauchi. Dependent Types from Counterexamples. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[20] Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis*. Prentice-Hall, 1981.

[21] Suresh Jagannathan and Stephen Weeks. A Unified Treatment of Flow Analysis in Higher-order Languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[22] K. McMillan. Interpolation and SAT-based Model Checking. In *International Conference on Computer Aided Verification*, 2003.

[23] Hiroshi Unno and Naoki Kobayashi. Dependent Type Inference with Interpolants. In *Principles and Practice of Declarative Programming*, 2010.

[24] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In *International Conference on Computer Aided Verification*, 2008.

[25] Bitv Library. `http://www.lri.fr/~filliatr/software.en.html`.

[26] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. In *Formal Methods in Computer-Aided Design*, 2010.

[27] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from Proofs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[28] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, 2006.

[29] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[30] Domagoj Babic and Alan J. Hu. Structural Abstraction of Software Verification Conditions. In *International Conference on Computer Aided Verification*, 2007.

[31] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming*, 2000.

[32] Charles Gregory Nelson. Techniques for Program Verification. Technical report, XEROX Research Center, 1981.

[33] Aws Albarghouthi and Kenneth L McMillan. Beautiful Interpolants. In *International Conference on Computer Aided Verification*, 2013.

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

[35] Cormac Flanagan and K. Rustan M. Leino. Houdini, An Annotation Assistant for ESC/Java. In *Formal Methods Europe*, 2001.

[36] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants As Classifiers. In *International Conference on Computer Aided Verification*, 2012.

[37] K. L. McMillan. An Interpolating Theorem Prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[38] Niki Vazou, Patrick M Rondon, and Ranjit Jhala. Abstract Refinement Types. In *European Symposium on Programming*, 2013.

[39] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates. In *ACM International Symposium on Software Testing and Analysis*, 2002.

[40] Eric L. Seidel, Niki Vazou, and Ranjit Jhala. Type Targeted Testing. In *European Symposium on Programming*, 2015.

[41] Phuc C. Nguyen and David Van Horn. Relatively Complete Counterexamples for Higher-order Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[42] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. Testing Container Classes: Random or Systematic? In *International Conference on Fundamental Approaches to Software Engineering*, 2011.

[43] B Dutertre and L De Moura. Yices SMT Solver. `http://yices.csl.sri.com/`.

[44] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. ICE: A Robust Framework for Learning Invariants. In *International Conference on Computer Aided Verification*, 2014.

[45] Dirk Beyer and M Erkan Keremoglu. CPACHECKER: A Tool for Configurable Software Verification. In *International Conference on Computer Aided Verification*, 2011.

[46] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya Nori. Verification as Learning Geometric Concepts. In *International Static Analysis Symposium*, 2013.

[47] Rahul Sharma and Alex Aiken. From Invariant Checking to Invariant Inference Using Randomized Search. In *International Conference on Computer Aided Verification*, 2014.

[48] SML Library. `http://www.smlnj.org/doc/smlnj-lib/`.

[49] OCAML Library. `http://caml.inria.fr/pub/docs/`.

[50] Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. Automating Relatively Complete Verification of Higher-order Functional Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

[51] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent Types for Imperative Programs. In *International Conference on Functional Programming*, 2008.

[52] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[53] Conor McBride. Faking It Simulating Dependent Types in Haskell. *Journal of Functional Programming*, 12(5):375–392, 2002.

[54] Sam Lindley and Conor McBride. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In *ACM SIGPLAN Haskell Symposium*, 2013.

[55] Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. A Type-directed Abstraction Refinement Approach to Higher-order Model Checking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[56] He Zhu and Suresh Jagannathan. Compositional and Lightweight Dependent Type Inference for ML. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2013.

[57] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating Testing and Interactive Theorem Proving. In *International Workshop on the ACL2 Theorem Prover and Its Applications*, 2011.

[58] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: A New Algorithm for Property Checking. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2006.

[59] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[60] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *International Conference on Software Engineering*, 2012.

[61] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A Data Driven Approach for Algebraic Loop Invariants. In *European Symposium on Programming*, 2013.

[62] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *International Conference on Functional Programming*, 2014.

[63] Gowtham Kaki and Suresh Jagannathan. A Relational Framework for Higher-order Shape Analysis. In *International Conference on Functional Programming*, 2014.

[64] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. Modular Reasoning About Heap Paths via Effectively Propositional Formulas. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[65] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using Dynamic Analysis to Generate Disjunctive Invariants. In *International Conference on Software Engineering*, 2014.

[66] Pranav Garg, P Madhusudan, Daniel Neider, and Dan Roth. Learning Invariants Using Decision Trees and Implication Counterexamples. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[67] He Zhu, Aditya Nori, and Suresh Jagannathan. Learning Refinement Types. In *International Conference on Functional Programming*, 2015.

[68] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, 2015.

[69] Mirko Stojadinović and Filip Marić. meSAT: Multiple Encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014.

[70] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. Solving Constraint Satisfaction Problems with SAT Modulo Theories. *Constraints*, 17(3):273–303, 2012.

[71] Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive Proofs for Inductive Tree Data-structures. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[72] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. Natural Proofs for Structure, Data, and Separation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[73] Ruzica Piskac, Leonardo Moura, and Nikolaj Bjørner. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. *Journal of Automated Reasoning*, 44:401–424, 2010.

[74] Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. On Local Reasoning in Verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[75] Phuc C. Nguyen and David Van Horn. Relatively Complete Counterexamples for Higher-order Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

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

[77] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. Effectively-Propositional Reasoning About Reachability in Linked Data Structures. In *International Conference on Computer Aided Verification*, 2013.

[78] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[79] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically Learning Shape Specifications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[80] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic Implementation of the Best Transformer. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2004.

[81] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically Computing Most-precise Abstract Operations for Shape Analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2004.

[82] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, Abstraction, Theorem Proving: Better Together! In *ACM International Symposium on Software Testing and Analysis*, 2006.

[83] A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and All That: Automating Abstract Interpretation. *Theoretical Computer Science*, 311:15–32, 2015.

[84] He Zhu, Aditya Nori, and Suresh Jagannathan. Dependent Array Type Inference from Tests. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2015.

[85] Patrice Godefroid and Ankur Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[86] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From Tests to Proofs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[87] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction Refinement via Inductive Learning. In *International Conference on Computer Aided Verification*, 2005.

[88] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. Learning Commutativity Specifications. In *International Conference on Computer Aided Verification*, 2015.

[89] Bor-Yuh Evan Chang and Xavier Rival. Relational Inductive Shape Analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[90] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A Relational Approach to Interprocedural Shape Analysis. *ACM Transactions on Programming Languages and Systems*, 32:5:1–5:52, 2010.

[91] Roman Manevich, E. Yahav, G. Ramalingam, and Mooly Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-linked Lists. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2005.

[92] Tal Lev-Ami and Shmuel Sagiv. TVLA: A System for Implementing Static Analyses. In *International Static Analysis Symposium*, 2000.

[93] 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 *International Conference on Automated Deduction*, 2005.

[94] Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A Logic of Reachable Patterns in Linked Data-structures. In *International Conference on Foundations of Software Science and Computation Structures*, 2006.

[95] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software Verification with VeriFast: Industrial Case Studies. *Science of Computer Programming*, 82:77–97, 2014.

[96] Adam Chlipala. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[97] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*, 2009.

[98] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 2010.

[99] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[100] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[101] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability Modulo Recursive Programs. In *International Static Analysis Symposium*, 2011.

[102] Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper - Complete Heap Verification with Mixed Specifications. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[103] Karen Zee, Viktor Kuncak, and Martin Rinard. Full Functional Verification of Linked Data Structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[104] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2007.

[105] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded Refinement Types. In *International Conference on Functional Programming*, 2015.

[106] Saurabh Srivastava and Sumit Gulwani. Program Verification Using Templates over Predicate Abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[107] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Learning Universally Quantified Invariants of Linear Data Structures. In *International Conference on Computer Aided Verification*, 2013.

[108] Shachar Itzhaky, Nikolaj Bjørner, Thomas W. Reps, Mooly Sagiv, and Aditya V. Thakur. Property-Directed Shape Analysis. In *International Conference on Computer Aided Verification*, 2014.

[109] Aleksandr Karbyshev, Nikolaj Bjorner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-Directed Inference of Universal Invariants or Proving Their Absence. In *International Conference on Computer Aided Verification*, 2015.

[110] Bolei Guo, Neil Vachharajani, and David I. August. Shape Analysis with Inductive Recursion Synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[111] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-abduction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[112] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape Analysis via Second-Order Bi-Abduction. In *International Conference on Computer Aided Verification*, 2014.

[113] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable Shape Analysis for Systems Code. In *International Conference on Computer Aided Verification*, 2008.

[114] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAYER: Memory Safety for Systems-level Code. In *International Conference on Computer Aided Verification*, 2011.

[115] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[116] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

VITA

VITA

He Zhu was born and brought up in Beijing, China. He obtained his Ph.D. from the Department of Computer Science at Purdue University. At Purdue, he worked in the Programming Languages Group, advised by Prof. Suresh Jagannathan. During his Ph.D. studies, he interned at Microsoft Research, Cambridge, UK (July 2015 - October 2015).

He's research interests include static analysis, automated reasoning, and machine learning applied to software verification. His work in program specification synthesis provides higher assurance of software correctness with lower human effort through the design and implementation of novel data-driven learning techniques. His work was published in top-tier programming language research conferences including CAV, ICFP and PLDI. He also received a Ross Fellowship and Maurice H. Halstead Memorial Award for outstanding research in software engineering at Purdue University.