Learning Refinement Types

He Zhu
Purdue University, USA
zhu103@purdue.edu

Aditya V. Nori
Microsoft Research, UK
adityan@microsoft.com

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

Abstract

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. Our approach 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 a refinement type checker. 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.

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

Keywords Refinement Types, Testing, Higher-Order Verification, Learning

1. Introduction

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). In a refinement type system, a *base type* such as `int` is specified into a *refinement base type* written $\{\texttt{int}|\, e\}$ where e (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 $\{x : P_x \rightarrow P\}$, constrains the argument x by the refinement type P_x, and produces a result whose type is specified by P. Refinement type systems such as DML [41] and LIQUIDTYPES [28] have demonstrated their utility in validating useful specifications of higher-order functional programs.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada

http://dx.doi.org/10.1145/2784731.2784766

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

Figure 1. A simple higher-order program

To illustrate, consider the simple program shown in Fig. 1. Intuitive program invariants for `max` and `f` can be expressed in terms of the following refinement types:

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

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 (we show how to find sufficient invariants for this program in Section 2).

On the other hand, random testing, exemplified by systems like QUICKCHECK [6], 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 (CE-

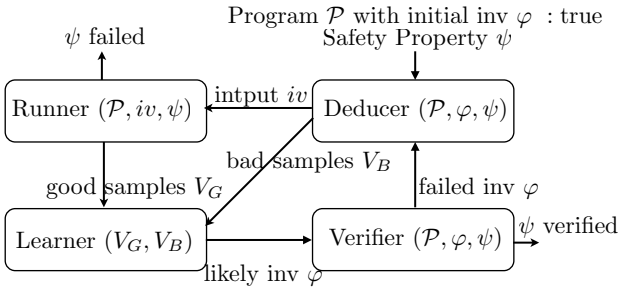


Figure 2. The Main algorithm.

GAR) process [7] 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) 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 [17]), 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 [28]) can verify these programs only if such invariants are manually supplied which can be challenging for programmers.

Contributions. Our paper 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.
- A description of an implementation, along with experimental results over a range of complex higher-order programs, that validates the utility of our ideas.

2. Overview

This section describes the framework of our approach, outlined in Fig. 2. 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 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. 3 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 `update` 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 `update`); 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, $\Omega(\texttt{init})$ includes variables $\texttt{i}, \texttt{n}, \texttt{a}_0, \texttt{a}_\texttt{r}$ where $\texttt{a}_0$ and $\texttt{a}_\texttt{r}$ 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 [28], 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 1:

$$c_1 y_1 + \cdots + c_m y_m + d \ \leq \ 0 \tag{1}$$

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

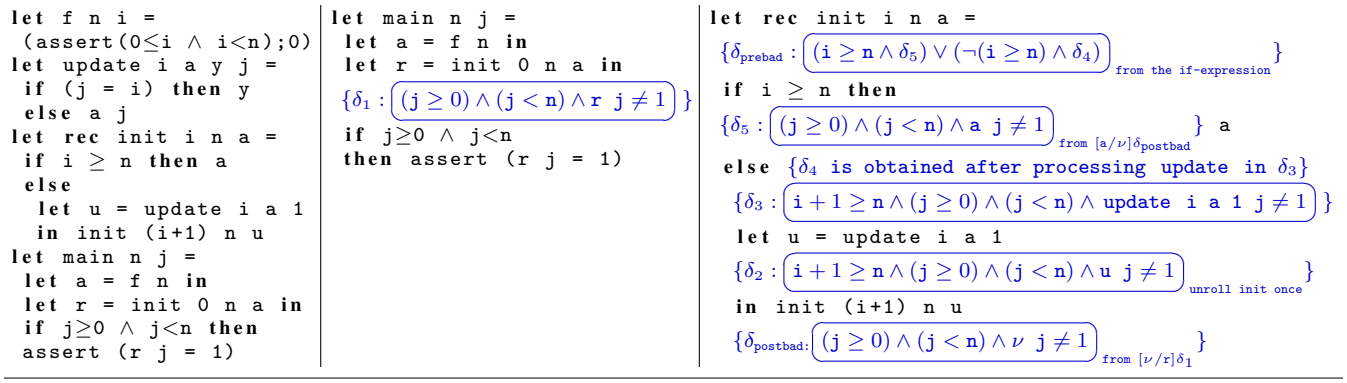


Figure 3. A higher-order program (in the first column) and its bad-conditions (in the latter two columns).

$d \in \mathcal{D}$. 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. 2 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 (`wp` generation defined in Sec. 4), 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. 3, 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 [23]. 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 $\texttt{a}_0$ and $\texttt{a}_\texttt{r}$ 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 Table 1(a). 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 `update`, which can drive sampling for `update`.

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 `update`, which happens within a series of recursive calls to `init`? Note that the argument to the application of `a` takes place in `update` 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 Table 1(a), 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 `(a 0) = 1` and `(a` $\texttt{a}_0$`) = 0` for `0 <` $\texttt{a}_0$ `< n`.

Learner. A classifier that admits all good samples and prohibits all bad ones is considered a likely invariant. We rely on predicate abstraction [11] to build these classifiers. For illustration, consider a subset of the atomic predicates obtained from Equition 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 [1]. Similar arguments have been demonstrated in machine learning and static verification techniques [13].

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 Table 1(b) and we show the selection informally in Table 1(c) (Π_3 and Π_6 constitute a sufficient classifier). To compute a likely invariant, we generate its

Table 1. Classifying good (G) and bad (B) samples to construct an invariant (precondition) for `init`.

(a) samples

	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

(b) relate samples to predicates

	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

(c) select predicates

	Π_3	Π_6
	1	1
G	0	0
B	1	0

(d) truth table

	Π_3	Π_6
G	1	1
	0	0
	0	1
B	1	0

truth table Table 1(d). The table rejects all (Boolean) bad samples in Table 1(c) and accepts all the other possible samples, including the good samples in Table 1(c). Note that we generalize good states. The truth table accepts more good states than sampled. We apply standard logic minimization techniques [20] 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:

$$\neg(\mathtt{a}_0 < \mathtt{i}) \vee \mathtt{a}_r = 1$$

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: $\psi_0 \equiv \mathtt{a}_0 \geq 0 \wedge \mathtt{a}_0 < n$, 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`:

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

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 [28], 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 Table 1(a). 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 Table 1(a). 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. To bootstrap, we may run the program with arguments `1,2` and `3`, and infer the following types:

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

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` ($\nu > \mathtt{x} \wedge \nu \geq y$) 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:

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

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

```
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. A simple data structure example.

than $\texttt{i} + \texttt{len xs}$. 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.

3. Language

Syntax. 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. 5, with support for refinement types defined in Sec. 1. We cover recursive data structures in Sec. 7.

Typically, x and y are bound to variables; f is bound to function symbol. A refinement expression is either a refinement variable (κ) that represents an initially unknown type refinement or a concrete boolean expression (e). Instantiation of the refinement variables to concrete predicates takes place through the type refinement algorithm described in Sec. 6.

Note that the `let rec` binding (in our examples) is syntactic sugar for the `fix` operator: $\texttt{let rec}\, f\ \tilde{x} = e\ \texttt{in}\ e'$ is converted from $\texttt{let}\, f = \texttt{fix}\,(\texttt{fun}\, f \rightarrow \lambda\tilde{x}.e)\,\texttt{in}\, e'$. Here, $\tilde{x}$ abbreviates a (possibly empty) sequence of arguments $\{x_0, \cdots, x_n\}$. The length of $\tilde{x}$ is called the *arity* of f.

Our language is A-normalized. For example, in function applications $f\ \tilde{y}$, we ensure every function and its arguments are associated with a program variable. When the length of $\tilde{y}$ is smaller than the arity of f, $f\ \tilde{y}$ is a partial application. For any expression of the form $\texttt{let}\, f = \lambda\tilde{x}.e\,\texttt{in}\, e'$, 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'; similarly in $\lambda x.\ e'$ if x is used as a function in e', x is an unknown function in e'). The statement "$\texttt{assert}\ p$" is standard. Programs with assertion failure would immediately terminates.

Semantics. We reuse the refinement type system defined in [28]; the type checking rules are given in [44].

$$B \in \texttt{Base} ::= \texttt{int} \mid \texttt{bool}$$
$$P \in \texttt{RefinementType} ::= \{B \mid \kappa\} \mid \{B \mid e\} \mid \{x : P \rightarrow P\}$$
$$x, y, \nu, f \in \texttt{Var} \quad c \in \texttt{Constant} ::= \texttt{0}, \ldots, \texttt{true}, \texttt{false}$$
$$v \in \texttt{Value} ::= c \mid x \mid y \mid \texttt{fix}\,(\texttt{fun}\, f \rightarrow \lambda\tilde{x}.\ e) \mid \lambda\tilde{x}.e$$
$$\texttt{op} \in \texttt{Operator} ::= \{+, -, \geq, \leq, \neg, \cdots\}$$
$$e ::= v \mid \texttt{op}\ (v_0, \cdots, v_n) \mid \texttt{assert}\ v \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid f\ \tilde{y}$$

Figure 5. Syntax

4. 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 Table 1(a)), 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 implement a backward symbolic analysis, `wp`, analogous to weakest precondition generation; the analysis simply pushes up the negation of assertions backwards, substituting terms for values in a bad condition δ based on the structure of the term e. 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. To ensure termination, recursive functions are unrolled a fixed number of times, defined by the parameter i. The definition of `wp` is given as follows:

$$\begin{array}{l}
\texttt{wp}(i, e, \delta) = \\
\quad \mathsf{let}\ \delta = \ \mathsf{match}\ e\ \mathsf{with} \\
\qquad \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow \\
\qquad\quad ((v \wedge \texttt{wp}(i, e_1,\ \delta)) \vee (\neg v \wedge \texttt{wp}(i, e_2,\ \delta))) \\
\qquad \mid \texttt{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 \rightarrow \texttt{wp}(i, e_1, [\nu/x]\texttt{wp}(i, e_2, \delta))) \\
\qquad \mid v \Rightarrow [e/\nu]\delta \\
\qquad \mid \ \texttt{op}\,(v_0, \cdots, v_n) \Rightarrow [e/\nu]\delta \\
\qquad \mid \texttt{assert}\ p \rightarrow \neg p \vee \delta \\
\qquad \mid f\ \tilde{y} \rightarrow (\mathsf{match}\ f\ \mathsf{with} \\
\qquad\quad \mid \mathsf{unknown\ fun\ or\ partial\ application} \rightarrow [(f\ \tilde{y})/\nu]\delta \\
\qquad\quad \mid \mathsf{known\ fun}\ (\mathsf{when}\ \texttt{let}\ f = \lambda\tilde{x}.\ e) \rightarrow [\tilde{y}/\tilde{x}]\texttt{wp}(i, e, \delta) \\
\qquad\quad \mid \mathsf{known\ fun}\ (\mathsf{when}\ \texttt{let}\ f = \ \texttt{fix}\,(\texttt{fun}\, f \rightarrow \lambda\tilde{x}.e)) \rightarrow \\
\qquad\qquad \mathsf{if}\ i > 0\ \mathsf{then}\ [\tilde{y}/\tilde{x}]\texttt{wp}(i - 1, e, \delta)\ \mathsf{else}\ \texttt{false}\,) \\
\quad \mathsf{in} \\
\qquad \mathsf{if\ exists}\ f\ \tilde{y}\ \mathsf{in}\ \delta\ \mathsf{and}\ f\ \mathsf{is\ a\ known\ fun} \\
\qquad \mathsf{then}\ \texttt{wp}(i, f\ \tilde{y}, [\nu/(f\ \tilde{y})]\delta)\ \mathsf{else}\ \delta
\end{array}$$

Our `wp` function is standard, extended to deal with unknown function calls. The concept of known function and unknown function is defined in Sec. 3. Our idea is to encode unknown functions into uninterpreted functions, reflected in the $f\ \tilde{y}$ case for an application expression when f is an unknown function with a list of argument $\tilde{y}$ or $f\ \tilde{y}$ is a partial application. 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. 3). The symbolic analysis for the actual function represented by the unknown function is deferred until it becomes known (e.g. δ_3 in Fig. 3), 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 $f\ \tilde{y} \in \delta$ where $\tilde{y}$ is a list of arguments, we perform $\texttt{wp}(i, f\ \tilde{y}, [\nu/(f\ \tilde{y})]\delta)$.

In the $f\ \tilde{y}$ 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

```
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 6. Generating samples for `g`, bound to parameter `k` in `f`, may trigger assertion violations in `check`.

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.

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. 3).

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. 3 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. 3 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 $\texttt{min}(\tilde{x})$ to $\texttt{max}(\tilde{x})$ where $\tilde{x}$ 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 by our hypothesis domain) with variables in $\tilde{x}$. The definition of `min` and `max` is in Sec. 2. 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. 6 and obtain the sample states for function `app` shown in Fig. 7 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.

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

Figure 7. Sample table for pre-state of `app` in Fig. 6

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 (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. 6).

5. 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 $\Pi = \{\Pi_i\}_{0 \le i < n}$ from which program invariants are constructed. Recall the hypothesis domain defined in Sec. 2. Each atomic predicate Π_i is of the form:

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

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. 8. 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 [9, 28]. 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 $\forall \sigma \in V_B \ .\ \alpha(\sigma) \wedge L(V_G, V_B) \Rightarrow \texttt{false}$. 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 [32].

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 [13, 21].

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^b_G and V^b_B defined over Boolean values. Specifically, $V^b_G = \{\langle(\Pi_1(\sigma), \cdots, \Pi_n(\sigma))\rangle |\ \sigma \in V_G\}$. V^b_B is defined dually. Table 1(b) is an example of such conversion from Table 1(a). We associate an integer variable sel_i to the i^{th} predicate $\Pi_i (0 \le i < n)$.

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"
```

If Π_i should be selected for separation in the classifier, sel_i is assigned to 1. Otherwise, it is assigned as 0.

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

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

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

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 Table 1(b) is shown in Table 1(c). 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 $\{\mathcal{B} = \mathcal{B}_i |\ sel_i = 1 (0 \le i < n)\}$. To construct the likely invariant, we firstly generate a table $V_{\mathcal{B}}^b$, 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 $V_{\mathcal{B}}^b$, 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_G^b). See Table 1(d) as an example of the generated truth table from Table 1(c). In line 6 of Algorithm 1, the call to McCluskey applies standard sound logic minimization techniques [20] to $\mathcal{T}$ to compute a compact Boolean structure of the likely invariant.

Lemma 1. *$L\ (V_G, V_B)$ is consistent.*

Lemma 1 claims that our algorithm will never produce an invariant that misclassifies a good sample or bad sample.

6. 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. Other unknown refinement variables, associated with local expressions inside function definitions, are still undefined.

To solve this problem, we have implemented an algorithm that extracts path-sensitive verification conditions from refinement typing rules, which extends the inference algorithm in [28]. Therefore it does not need to explicitly infer the refinement types for local expressions. It also can verify programmer-supplied program assertions using synthesized likely invariants. We present the full algorithm in [44].

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. 7. In the `app` function, the argument `f` is an unknown function whose second argument $\mathtt{f}_1$ is also an unknown function as the type in Fig. 6 shows. We did not sample the input/output values for function $\mathtt{f}_1$ 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 $\mathtt{f}_1$. This indicates the invariant inferred for `g` is also likely to be invariant for $\mathtt{f}_1$ so the type refinements for `g` can flow into that of $\mathtt{f}_1$. Formally, consider the refinement function subtyping rule in [28]:

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

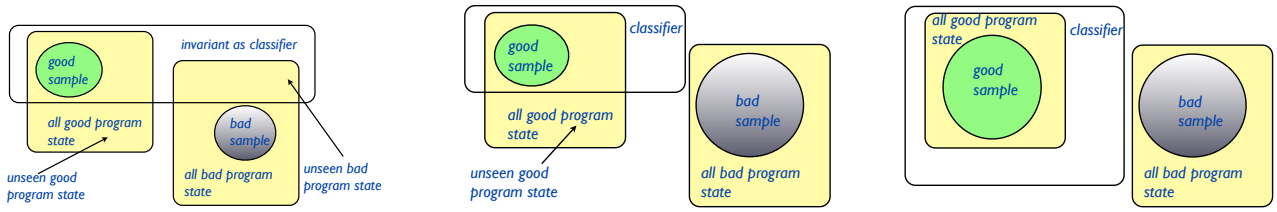
If the type refinement in P_x is synthesized, it can be propagated to that of P_x'.

For example, according to the subtyping rule, `g` must subtype to $\mathtt{f}_1$. $\mathtt{f}_1$ can then inherit the type refinements for `g`. We then let our type inference algorithm decide a valid type instantiation, following [28]. In Fig. 7, 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: $\mathtt{f}_0 = \mathtt{x}$ and $\mathtt{g}_0 = \mathtt{x}$. Leveraging the type inference algorithm with the likely type refinement $\nu = \mathtt{x}$, we conclude the desired type for `app`:

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

6.1 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 random test inputs iv (like [6]) to bootstrap our verification process (line 1). We then instrument the program using the strategy discussed in Sec. 4. 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), 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. 5) 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



(a) over-approximate invariant (b) under-approximate invariant (c) desired invariant

Figure 8. CEGAR loop: Invariant as classifier.

Algorithm 2: Main $e\ \psi$

Input: e is a program; ψ is its safety property
Output: verification result

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

Algorithm 3: Refine $(i, e, \psi, \varphi, V_G, V_B)$

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* $e\ \psi$ **in**
2 **let** _ = `wp` $(i, e', \texttt{false})$ **in**
3 **let** bad_cond = bad conditions of functions from `wp` call **in**
4 **if** *sat* (bad_cond $\wedge\varphi$) **then**
5 | $(V_G, (\texttt{deduce}\ (\text{bad\_cond} \wedge\varphi)) \cup V_B)$
6 **else**
7 | **let** test_cond = `wp` $(i$, *annotate* $e\ \varphi$, `false`$)$ **in**
8 | **if** *sat* (test_cond) **then**
9 | | let iv = `deduce` test_cond in
10 | | ((`run` (*instrument* e') $iv) \cup V_G, V_B)$
11 | **else** Refine $(i + 1, e, \psi, \varphi, V_G, V_B)$

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).

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. 8(a) 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. 8(b). 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 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. 8(c). 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, $\texttt{a}_0$ and $\texttt{a}_\texttt{r}$ in Table 1(a) are replaced with `j` and `a j` in a failed invariant for the `init` function (consider δ_5) in Fig. 3.) 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. 8), 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 1 in Sec. 2). (d) In the verifying phase (verifier), it returns "Verified" in line 5 of Algorithm 2 when specifications are successfully proved.

6.2 Soundness and Convergence

Our algorithm is sound since we rely on a sound refinement type system [28] for proving safety properties (proved in [44]) or a test input for witnessing bugs.

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 (according to [44], the desired invariant found in our system is an *inductive invariant*, which hence can be encoded into the refinement type system in [28] for verification).

Recall that our hypothesis domain is the arbitrary Boolean combination of predicates, parameterized by Equation 1 in Sec. 2. We claim the CEGAR loop in Algorithm 2 *converges*: it could eventually learn the desired invariant φ, provided one exists expressible as a hypothesis in the hypothesis domain.

Theorem 1. *[Convergence] Algorithm 2 converges.*[1]

To derive a proof, assume 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 1, 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. 5); the CEGAR based sampling-learning-checking loop in Algorithm 2 converges in a finite number of iterations.

6.3 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 [37] are necessary for verification when an unknown function is used more than once in different contexts [17]. Instead of inferring intersection types directly as in [17], we recover their precision by inferring type refinements (via learning) containing disjunctions (as demonstrated by the example in Fig. 3).

7. Recursive Data Structures

As stated in Sec. 2, we extend our framework to verify data structure programs with specifications that can be encoded into type refinements using measures [15, 40]. For example, a measure `len`, representing list length, is defined in Fig. 9 for lists. We firstly extend the syntax of our language to support recursive data structures.

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

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 [15], we restrict measures to be in the class of first order functions over simple expressions (ϵ) so that they

[1] All proofs can be found in [44].

```
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 9. Samples of data structures can be classified by measures.

$$\begin{array}{l}\texttt{wp}(e, \phi) = \ \mathsf{case}\ e\ \mathsf{of} \\ \quad \mid \mathcal{C}_i\langle e \rangle\ \texttt{when}\ (m, \langle \mathcal{C}_i\langle x_i \rangle \rightarrow \epsilon_i \rangle) \in M \rightarrow [\epsilon_i\ \langle e \rangle / (m\ \nu)]\phi \\ \quad \mid \{\texttt{match}\ e\ \texttt{with}\ |_i\, C_i\langle x_i \rangle \rightarrow e_i\}\ \texttt{when}\ (m, \langle \mathcal{C}_i\langle x_i \rangle \rightarrow \epsilon_i \rangle) \in M \rightarrow \\ \qquad \bigvee_i \{\exists \langle x_i' \rangle . [\langle x_i' \rangle / \langle x_i \rangle]((m\ e) = \epsilon_i \langle x_i \rangle \wedge (\texttt{wp}(e_i, \phi)))\}\end{array}$$

Figure 10. `wp` rule for recursive data type

are syntactically guaranteed to terminate. The typing rules for the extended syntax are adapted from [15] and are available as part of the supplementary material. To support this extension, we also need to extend our `wp` definition in Fig. 10.

The basic idea is that when a recursive structure is encountered, its measure definitions 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. 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. 11 is such an example.

With the extended definition, sampling recursive data structures is fairly strait-forward. To collect "good" states, in the instrumentation phase, for each recursive 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. 9. Note that `aux` is defined inside `reverse` and is a closure which can refer to variable `zs` in its lexical scoping. A good sample presents the values of `len(xs)`, `len(ys)` and `len(zs)`, trivially available from testing. A bad sample 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. 11. With these samples, our approach infers the following refinement type for `aux`, which is critical to prove the assertion.

$$\begin{array}{c}\texttt{xs:list} \rightarrow \texttt{ys:}\ \{\texttt{list} \mid \texttt{len xs}\ + \texttt{len}\ \nu = \texttt{len zs}\} \\ \rightarrow \{\texttt{list} \mid \texttt{len}\ \nu = \texttt{len zs}\}\end{array}$$

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`, $\texttt{xs:list} \rightarrow \texttt{ys:list} \rightarrow \{\texttt{list} \mid \texttt{len xs}\ + \texttt{len ys} = \texttt{len}\ \nu\}$.

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 [3], a constraint based test generation mechanism. Alternatively, the failed

```
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:

	l_{xs}	l_{ys}	l_{zs}
G	1	2	3
	2	1	3
B	1	0	2
	1	0	0

Likely invariant:

$$l_{xs} + l_{ys} = l_{zs}$$

Figure 11. Classifying good (G) and bad (B) samples to construct an invariant (precondition) for `aux`. l_{xs} abbreviates `len xs`, etc.

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 12. Evaluation using loop programs: `N` and `T` are the number of CEGAR iterations and total time of our tool (`L` is the time in learning). × means an adequate invariant was not found.

invariants can be considered incorrect specifications. We can directly generate inputs to the program by causing it to violate the specifications following [24, 29]. 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`) up to a small length s in the Main algorithm (line 1 of Algorithm 2). In our experience in Sec. 8, 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 [31] showing that test coverage of random testing for container structures is as good as that of systematic testing.

8. Experimental Results

We have implemented our approach in a prototype verifier.[2] Our tool is based on OCaml compiler. We use Yices [42] 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. 5) 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 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.

8.1 Learning Benchmarks

We collected challenging loop programs found in an invariant learning framework ICE [10]. We list in Fig. 12 the programs that took more than 1s to verify in their tool. We additionally compare our approach to `CPA`, a static verification tool [2] 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 [10], `SC` [34] and `MC`^{2} [30]. 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.

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

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 13. Evaluation using MoCHI benchmarks: `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.

8.2 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 [17], 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 [37, 38] on spurious whole program counterexamples from which suitable predicates for refining abstract model are discovered based on interpolations [21]. However, existing limitations of interpolating theorem provers may confound MoCHI. For example, it fails to prove the assertion given in program in Fig. 9.

Fig. 13 only lists results for which MoCHI requires more than 1 second. Our tool also takes less than 1s for the rest 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.

8.3 Recursive 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. `Brauntree` is a variant of balanced binary trees. They are described in [41]. `Ralist` is a random-access list library. `Avltree` and `Redblack` are implementation of two balanced tree AVL-tree and Redblack tree. `Bdd` is a binary decision diagram library. `Vec` is a OCaml ex-

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 14. Evaluation using data structure benchmarks: 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.

tensible array libraray. These benchmarks are used for evaluation in [28]. Fifo is a queue structures maintained by two lists, adapted from the SML library [35]. Set/Map is the implementation of finite maps taken from the OCaml library [26].

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. 14. The number of annotations used in our system is reflected in column An. These annotations are simply the property in Fig. 14. 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, $\texttt{recbal}(\texttt{l},\texttt{r})$, which aims to merge two balanced trees (l and r) of arbitrarily different heights into a single balanced tree, 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 v refers to the result of recbal and ht is a measure definition that returns the height a tree structure.

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

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 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. 14 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 [28]. In particular, our type system is not as complete as [39] 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 1. Third, we do not currently 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. We leave such extensions for future work.

9. Related Work and Conclusions

There has been much work exploring the incorporation of refinement types into programming languages. DML [41] proposed a sound type-checking system to validate programmer-specified refinement types. LIQUIDTYPES [28] alleviates the burden for annotating full refinement types; it instead blends type inference with predicate abstraction [11], and infers refinement types from *conjunctions* of programmer-annotated Boolean predicates over program variables, following the Houdini approach [9].

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 [22], F* [36], GADTs and type classes [18, 19], 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 [17], compute predicate abstractions on the fly as a white-box analysis, encoding higher-order programs into recursion schemes [16]. Recent work in higher-order model checking [27] 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 [14] 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 [43] 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 [29]. A relatively complete approach for counterexample search is proposed in [24] 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 [4] 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 [12], which combines testing and first-order model checking. YOGI uses testing to refute spurious counterex-

amples 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 [8] 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 [10, 25, 33]; others are based on off-the-shell machine learning algorithms [30, 32, 34]. However, none of these efforts attempt to sample and synthesize complex invariants, in the presence of *recursive higher-order* functions.

Conclusion. We have presented 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 [5] 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.

Acknowledgement

We thank Gustavo Petri, Gowtham Kaki, and the anonymous reviewers for their comments and suggestions in improving the paper. This work was supported in part by the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

References

[1] A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *CAV*, 2013.

[2] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, 2011.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ISSTA*, 2002.

[4] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. In *ACL2*, 2011.

[5] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012.

[6] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, 2000.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[9] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME*, 2001.

[10] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust learning framework for learning invariants. In *CAV*, 2014.

[11] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *CAV*, 1997.

[12] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *FSE*, 2006.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.

[14] R. Jhala, R. Majumdar, and A. Rybalchenko. Hmc: Verifying functional programs using abstract interpreters. In *CAV*, 2011.

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

[16] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, 2009.

[17] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and cegar for higher-order model checking. In *PLDI*, 2011.

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

[19] C. McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002.

[20] E. J. McCluskey. Minimization of boolean functions. *Bell system technical Journal*, 35(6):1417–1444, 1956.

[21] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[22] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.

[23] C. G. Nelson. Techniques for program verification. Technical report, XEROX Research Center, 1981.

[24] P. C. Nguyen and D. V. Horn. Relatively complete counterexamples for higher-order programs. In *PLDI*, 2015.

[25] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, 2012.

[26] OCAML Library. `http://caml.inria.fr/pub/docs/`.

[27] S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. In *POPL*, 2014.

[28] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *PLDI*, 2008.

[29] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In *ESOP*, 2015.

[30] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, 2014.

[31] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *FASE*, 2011.

[32] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*, 2012.

[33] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, 2013.

[34] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS*, 2013.

[35] SML Library. `http://www.smlnj.org/doc/smlnj-lib/`.

[36] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, 2013.

[37] T. Terauchi. Dependent types from counterexamples. In *POPL*, 2010.

[38] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP*, 2009.

[39] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, 2013.

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

[41] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.

[42] Yices SMT solver. `http://yices.csl.sri.com/`.

[43] H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ml. In *VMCAI*, 2013.

[44] H. Zhu, A. V. Nori, and S. Jagannathan. Learning refinement types. Technical report, Purdue Univsersity, 2015. `https://www.cs.purdue.edu/homes/zhu103/msolve/tech.pdf`.