Data-Driven Inference of Representation Invariants

Anders Miltner
Princeton University
Princeton, NJ, USA
amiltner@cs.princeton.edu

Saswat Padhi
UCLA
Los Angeles, CA, USA
padhi@cs.ucla.edu

Todd Millstein
UCLA
Los Angeles, CA, USA
todd@cs.ucla.edu

David Walker
Princeton University
Princeton, NJ, USA
dpw@cs.princeton.edu

Abstract

A *representation invariant* is a property that holds of all values of abstract type produced by a module. Representation invariants play important roles in software engineering and program verification. In this paper, we develop a counterexample-driven algorithm for inferring a representation invariant that is sufficient to imply a desired specification for a module. The key novelty is a type-directed notion of *visible inductiveness*, which ensures that the algorithm makes progress toward its goal as it alternates between weakening and strengthening candidate invariants. The algorithm is parameterized by an example-based synthesis engine and a verifier, and we prove that it is sound and complete for first-order modules over finite types, assuming that the synthesizer and verifier are as well. We implement these ideas in a tool called Hanoi, which synthesizes representation invariants for recursive data types. Hanoi not only handles invariants for first-order code, but higher-order code as well. In its back end, Hanoi uses an enumerative synthesizer called Myth and an enumerative testing tool as a verifier. Because Hanoi uses testing for verification, it is not sound, though our empirical evaluation shows that it is successful on the benchmarks we investigated.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Theory of computation** → **Invariants**.

Keywords: Type-Directed Synthesis, Logical Relations, Abstract Data Types

ACM Reference Format:
Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3385967

1 Introduction

A *representation invariant* is a property that holds of all values of abstract type produced by a module. For instance, a module that implements a set using a list might maintain a *no duplicates* or *is sorted* invariant over the lists. Module implementers can rely on the invariant for correctness and efficiency and must ensure that it is maintained by each function in the module. Making representation invariants explicit has a number of software engineering benefits: they can be used as documentation, dynamically checked as *contracts* [9, 16], and used for automated testing [3, 6].

Representation invariants also play a key role in modular verification of software components. Consider a module that implements sets; its specification φ might demand that `(lookup (insert s i) i)` return `true` for all sets `s` and items `i`. A standard way to prove such a specification [1] is in two steps: 1) prove that a predicate $\mathcal{I}$ is a representation invariant of the module; and 2) prove that $\mathcal{I}$ is *stronger* than φ, i.e., all module states that satisfy $\mathcal{I}$ also satisfy φ. In other words, modular verification can be reduced to the problem of synthesizing a *sufficient representation invariant.*

In this paper, we develop an approach to automatically infer a sufficient representation invariant given a pure, functional module and a specification. To our knowledge, the only prior work to tackle this problem [15] builds candidate invariants out of a fixed set of atomic predicates and provides no correctness guarantees. We address both of these limitations through a form of *counterexample-guided inductive synthesis* (CEGIS) [28, §5], which consists of an interaction between two black-box components: **(1)** a *synthesizer* that generates a candidate invariant consistent with given sets of positive and negative examples, **(2)** a *verifier* that either proves that a candidate is a sufficient representation invariant or produces a counterexample, which becomes a new example for the synthesizer.

Our approach is inspired by recent work in data-driven inference of inductive invariants in other settings [7, 21, 25, 31]. As in that work, a key challenge is how to handle *inductiveness counterexamples*, pairs of module states $\langle s, s' \rangle$ such that s satisfies the candidate invariant, some module operation transforms state s to state s', but s' does not satisfy the candidate invariant. The problem is that there are two ways to resolve such counterexamples and it is not clear which is correct: treat s as a new negative example or treat s' as a new positive example.

We observe that if s is a *constructible* state of the module, meaning that it is reachable by a sequence of module operations, then s' must be as well. Therefore, any representation invariant will include both states, so we must treat s' as a new positive example. Based on this observation, we define a candidate invariant I to be *visibly inductive* on a module relative to a set S of known constructible states if every module operation produces a state satisfying I when invoked from a state in S. For each candidate invariant, we first iteratively weaken it until it is visibly inductive relative to the current set of known constructible states, in the process adding new states to this set, and only then do we consider other inductiveness violations. Intuitively, this approach eagerly identifies and exploits inductiveness counterexamples for which no "guessing" is required.

We have formalized a general notion of inductiveness as a type-indexed logical relation, of which both our notion of visible inductiveness and the traditional notion of (full) inductiveness are special cases. We have also formalized our overall algorithm using this notion. We have proven the algorithm sound and complete when the module contains first-order code and the implementation of the abstract type is a finite domain, provided the underlying synthesizer and verifier are also sound and complete.

We have implemented our algorithm in OCaml and call the resulting tool Hanoi. To instantiate the synthesis component of the system, we use Myth [20], a type- and example-directed synthesis engine. Myth is capable of synthesizing invariants over recursive data types in many cases, so it is a good fit for tackling proofs about modules that implement recursive data types, which are the focus of our benchmarks. To instantiate the verification component, we use a form of enumerative test generation. Despite the unsoundness of this underlying verifier, our experimental results show that Hanoi still infers sufficient representation invariants in practice. Such *likely* representation invariants can be used by module implementers and verifiers for many purposes.

We have also implemented extensions that allow Hanoi to be used with higher-order code. Here, the main challenge comes in how to extract counterexamples from higher-order arguments. It turns out that our first-order scheme for extracting counterexamples is essentially an application of a first-order contract that guards and logs values passing through the first-order interface. The solution to counterexample-extraction from higher-order code then is to implement higher-order contracts [9] that guard and log values across this higher-order interface.

To evaluate our tools, we constructed a benchmark suite that includes 28 different modules, including a variety of modules over lists and trees, many drawn from Coq libraries and books [1]. We find that Hanoi is able to synthesize 22 of these invariants within 30 minutes.

To summarize, the main contributions of this work are:

- An algorithm for automated synthesis of representation invariants, parameterized by a verifier and synthesizer.
- A formalization of the algorithm and the key notion of visible inductiveness, over a first-order type theory. We prove soundness and completeness in the case of finite domains, if the given verifier and synthesizer are sound and complete.
- An extension of the algorithm capable of extracting counterexamples from higher-order interfaces.
- Implementation, optimization and evaluation of a tool called Hanoi that synthesizes invariants over recursive data types, using an unsound, enumerative testing engine for verification.

2 A Motivating Example

In this section, we give a high-level overview of Hanoi using an example. Consider the interface SET:

```
module type SET = sig
    type t
    val empty : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

The interface declares an abstract type t and a number of functions that operate over t. Figure 1 shows a module ListSet that implements the SET interface, using int list as the concrete type.

We study the problem of verifying that ListSet satisfies some standard properties of sets. An example specification follows.

$$
\begin{aligned}
(\varphi\ s) \triangleq\ & \forall i : \texttt{int}. \\
& \quad \neg(\texttt{lookup empty}\ i) \\
& \wedge (\texttt{lookup (insert}\ s\ i\texttt{)}\ i) \ \wedge\ \neg(\texttt{lookup (delete}\ s\ i\texttt{)}\ i)
\end{aligned}
$$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Figure 1. A module that implements SET using lists.

Note that this specification does *not* hold for arbitrary integer lists. For example, (lookup (delete [1;1] 1) 1) returns true. Nonetheless, the ListSet module is a correct implementation of the SET interface, because the specification holds for all values of the abstract type t that the module can actually construct. Such values are usually called the *representations* of the abstract type t. To emphasize that such values can be constructed by execution of module operations, we say a value v is *constructible* at type τ whenever a client with access to the module can produce v at the type τ.

A standard approach [1] to prove that a module implementation satisfies such a specification is to identify a *sufficient representation invariant*. In our example, such an invariant for ListSet is a predicate $\mathcal{I}_\star$: (int list $\rightarrow$ bool) that is

- sufficient for φ, i.e. $\forall s$: int list. $(\mathcal{I}_\star\ s) \implies (\varphi\ s)$, and
- whenever operations of ListSet module are supplied with argument values of abstract type that satisfy the invariant, they produce values of abstract type that satisfy the invariant, *i.e.*, the module is *inductive* with respect to the invariant.

In other words, $\mathcal{I}_\star$ contains all integer lists that are representations of type t, and is contained in the set of integer lists that satisfy φ. Figure 2 shows this relationship pictorially.

For ListSet, the predicate demanding an integer list has no duplicates is a sufficient representation invariant for φ. Our tool Hanoi automatically generates that invariant:

```
let rec I★ : int list -> bool = function
  | [] -> true
  | hd :: tl -> (not (lookup tl hd)) && (I★ tl)
```

2.1 Overview of Hanoi

Given a module, an interface and a specification, Hanoi employs a form of *counterexample-guided inductive synthesis* (CEGIS) [28, §5] to infer a sufficient representation invariant.

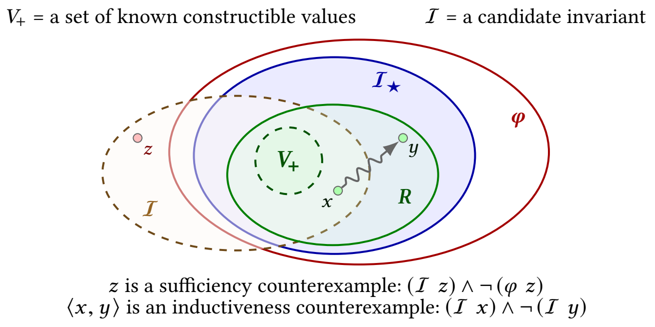
V_+ = a set of known constructible values
$\mathcal{I}$ = a candidate invariant
$\mathcal{I}_\star$
φ
z
V_+
y
x
R
$\mathcal{I}$
z is a sufficiency counterexample: $(\mathcal{I}\ z) \wedge \neg(\varphi\ z)$
$\langle x, y\rangle$ is an inductiveness counterexample: $(\mathcal{I}\ x) \wedge \neg(\mathcal{I}\ y)$

Figure 2. A sufficient representation invariant $\mathcal{I}_\star$ implies the spec and is an overapproximation of the set of representations R of the module's abstract type.

Specifically, we use a generate-and-check approach that iterates between two black-box components: **(1)** a synthesizer **Synth** that generates a candidate invariant, which is a predicate that *separates* a set V_+ of positive and a set V_- of negative examples, and **(2)** a verifier **Verify** that checks if a candidate invariant satisfies the desired properties and otherwise generates a counterexample. CEGIS has been successfully applied to other forms of invariant inference [7, 21, 25, 31].

We illustrate Hanoi and its key challenges via our running example. Initially the V_+ and V_- sets are empty, so suppose that **Synth** generates the candidate invariant fun _ -> true. This invariant is inductive, but not sufficient. Hence **Verify** will provide a counterexample, for instance [1;1], which is an integer list that satisfies the candidate invariant but not the specification φ (see z in Figure 2). As the final invariant must imply φ, [1;1] is added to V_-, which forces **Synth** to choose a stronger candidate invariant in the next round.

The main challenge in using this approach is the need to handle counterexamples to inductiveness. For instance, suppose that at some point during the algorithm we have $V_+ = \{$[], [3]$\}$ and $V_- = \{$[1;1]$\}$, and suppose **Synth** generates the following candidate invariant:

```
let I : int list -> bool = function
  | [] -> true
  | hd :: _ -> hd <> 1
```

This candidate is not inductive over the ListSet module. For instance, [0] satisfies the candidate, but (insert [0] 1) = [1;0] does not. Hence the pair $\langle$[0], [1;0]$\rangle$ constitutes an inductiveness counterexample (see $\langle x, y\rangle$ in Figure 2). Resolving such an inductiveness counterexample requires ensuring that either both x and y satisfy the candidate invariant or that neither does. This leads to two possibilities, and the problem is that it's unclear which one is correct:

- add [0] to V_- so that it will be excluded from the next candidate invariant
- add [1;0] to V_+ so that it will be included in the next candidate invariant

However, observe that if $\langle x, y\rangle$ is an inductiveness counterexample and x is known to be constructible, then y is constructible as well. Since a representation invariant must include all constructible integer lists, there is no choice to make in this case: we must add y to V_+.

Based on this observation, our algorithm maintains the property that all elements of V_+ both satisfy the current candidate invariant and are known to be constructible. To verify inductiveness of a candidate invariant, we first check a property that we call *visible inductiveness*, which informally requires that there are no inductiveness counterexamples $\langle x, y \rangle$ such that $x \in V_+$. If such a counterexample exists, we add y to V_+, ask **Synth** for a new candidate invariant, and re-check visible inductiveness on the updated V_+.

In our running example, the candidate invariant $\mathcal{I}$ shown above is not visibly inductive, so **Verify** would produce a counterexample, for instance ⟨`[]`, `[1]`⟩. Unlike the case for the counterexample ⟨`[0]`, `[1;0]`⟩ shown earlier, by construction the first element of this new pair is in V_+, so we know that we must add `[1]` to V_+. We then re-check visible inductiveness, continuing in this way until the candidate invariant is visibly inductive on V_+.

At that point, we check full inductiveness. Because $\mathcal{I}$ is visibly inductive with respect to V_+, any counterexample to full inductiveness is a pair $\langle x, y \rangle$ where x is not in V_+. In this case, in order to maintain the invariant that V_+ only contains constructible values we resolve the counterexample by adding x to V_-. So in general, the elements of V_- all falsify the current candidate invariant, but they may or may not be constructible. With this new negative example, **Synth** will produce a stronger candidate invariant. We then restart the process all over again, first weakening this new invariant to be visibly inductive and then strengthening it to be inductive.

In §3.4 we show that despite this interplay between weakening and strengthening, Hanoi is sound and complete over finite domains if **Verify** and **Synth** are sound and complete. That is, if a sufficient representation invariant exists then Hanoi will produce one.

The question of how to handle inductiveness counterexamples arises in prior work on data-driven invariant inference. Some of this work also observes that if x is constructible in an inductiveness counterexample $\langle x, y \rangle$, then so is y [25, 31]. However, those approaches only leverage this observation opportunistically, when a counterexample to full inductiveness happens to satisfy it. In contrast, we define the notion of visible inductiveness and use this notion to eagerly weaken a candidate invariant until no such counterexamples exist. We demonstrate empirically in Section 5 that our eager search for visible inductiveness counterexamples provides performance benefits. We also prove a completeness result for our approach in the context of finite domains, which those prior approaches lack. To our knowledge, the only prior CEGIS-based approaches to inductive invariant inference that have a completeness result depend upon special-purpose synthesizers that directly accept inductiveness counterexamples in addition to positive and negative examples [7, 10].

2.2 Handling Binary Functions

Consider the following extension to our `SET` interface, which exposes additional functions for set union and intersection:

```
module type ESET = sig
  include SET
  val union : t -> t -> t
  val inter : t -> t -> t
end
```

Consider an extension of the `ListSet` module that supports these functions (implementation not shown in the interest of space). When verifying inductiveness, an inductiveness counterexample on either `union` or `inter` is now a *triple* $\langle x_1, x_2, y \rangle$. This increases the number of possible ways to resolve the counterexample to four: **(1)** add x_1 to V_-, **(2)** add x_2 to V_-, **(3)** add both x_1 and x_2 to V_-, or **(4)** add y to V_+. More generally, the number of choices grows exponentially in the number of arguments to the function that have type `t`.

Hanoi naturally extends to this setting: By construction, a counterexample to visible inductiveness due to `union` or `inter` will be a triple $\langle x_1, x_2, y \rangle$ where x_1 and x_2 are in V_+, so as before we add y to V_+. On the other hand, a counterexample to inductiveness due to `union` or `inter` will be a triple $\langle x_1, x_2, y \rangle$ where at least one of x_1 and x_2 is *not* in V_+. In this case, we simply add each x_i that is not in V_+ to V_-.

Hanoi handles n-ary specifications in a similar manner. For instance, we may want to prove that a module implementing the `ESET` interface satisfies the following specification:

$$\begin{aligned}(\varphi' \; s_1 \; s_2) \triangleq \; & \forall i : \texttt{int}. \\ & ((\texttt{lookup}\ s_1\ i) \lor (\texttt{lookup}\ s_2\ i) \\ & \quad \implies (\texttt{lookup}\ (\texttt{union}\ s_1\ s_2)\ i)) \\ \land \; & ((\texttt{lookup}\ s_1\ i) \land (\texttt{lookup}\ s_2\ i) \\ & \quad \implies (\texttt{lookup}\ (\texttt{inter}\ s_1\ s_2)\ i))\end{aligned}$$

If a candidate invariant is not strong enough to imply this specification, then a counterexample will consist of a pair $\langle x_1, x_2 \rangle$ where at least one of x_1 and x_2 is not in V_+. In this case, we again add each x_i that is not in V_+ to V_-.

Our algorithm remains sound and complete for finite domains in the presence of these extensions, assuming the verifier and synthesizer are as well.

3 The Inference Algorithm

In this section, we describe our algorithm formally and characterize its key properties.

3.1 Preliminaries

Our programming language is a first-order variant of the simply-typed lambda calculus with functions, pairs, a base type (β) and a single designated abstract type (α). The syntax of 0-order types (σ), 1st-order types (τ), values (v) and expressions e are provided below.

$$\begin{array}{rrcl} \text{(0-types)} & \sigma & ::= & \beta \mid \alpha \mid (\sigma * \sigma) \\ \text{(1-types)} & \tau & ::= & \sigma \mid \sigma \rightarrow \tau \mid (\tau * \tau) \\ \text{(values)} & v & ::= & w \mid \langle v_1, v_2 \rangle \mid (\lambda x : \sigma . \, e) \\ \text{(expressions)} & e & ::= & x \mid v \mid (\pi_i \ e) \mid (e_1 \ e_2) \end{array}$$

We use x for value variables and w for constants of the base type β. The expression $(\pi_i \ e)$ is the i^{th} projection from the pair e. We write $\Gamma \vdash e : \tau$ to indicate that an expression e has type τ in the context Γ, which maps variables to their types. We write $\vdash e : \tau$ when Γ is empty, as will be the case in most of this work. We write $\tau[\alpha \mapsto \tau_c]$ to substitute τ_c for α in τ. Finally, and we use $e \Downarrow v$ to indicate that e evaluates to v. We refer the reader to Pierce [22] for the details.

We assume a module defines a single abstract type (α), which is declared in its interface. A *module interface* ($\mathbf{F} = \exists \alpha \, . \, \tau_m$) is a pair of a name ($\alpha$) for the abstract data type and a signature τ_m that specifies the types for operations over the abstract type. A *module implementation* $\mathbf{M} = \langle \tau_c, v_m \rangle$ is the classic existential package of a concrete type τ_c and a value v_m containing operations over the type τ_c. We say a module $\langle \tau_c, v_m \rangle$ *implements* an interface $\exists \alpha \, . \, \tau_m$ when it is well-typed as per the usual rules for existential introduction [22, §24], i.e. $\vdash v_m : \tau_m[\alpha \mapsto \tau_c]$.

In addition to an interface, we also assume the existence of a target *specification* φ, which captures the desired correctness criteria for a module implementation. These specifications are universal properties of the values of the abstract type; we formalize them as polymorphic functions over the module operations, *i.e.*, $\varphi : \forall \alpha \, . \, (\tau_m \rightarrow \alpha \rightarrow \texttt{bool})$. We saw an example specification for integer sets in §2.

The values of an abstract type α are simply the values that are *constructible* at type α through the module interface. Below we define the notion of a τ-constructible value and then use it to define when a module satisfies a specification.

Definition 3.1 (τ-Constructible Value: $\mathfrak{C}_{\mathbf{M}} [v \, ; \tau]$)**.** A value v is τ-constructible using $\mathbf{M}$, denoted $\mathfrak{C}_{\mathbf{M}} [v \, ; \tau]$, iff there exists a function $f : \forall \alpha \, . \, (\tau_m \rightarrow \tau)$ such that $(f[\tau_c] \ v_m) \Downarrow v$.

Definition 3.2 (Specification Satisfaction: $\mathbf{M} : \mathbf{F} \models \varphi$)**.** A module $\mathbf{M}$ with interface $\mathbf{F}$ is said to satisfy a given specification φ, denoted $\mathbf{M} : \mathbf{F} \models \varphi$, iff every α-constructible value satisfies φ, i.e. $\forall v : \tau_c \, . \, \mathfrak{C}_{\mathbf{M}} [v \, ; \alpha] \implies (\varphi[\tau_c] \ v_m \ v)$.

3.2 Representation Invariants

Loosely speaking, a representation invariant is a property that is preserved by operations over the abstract type of a module. As such, we say that a representation invariant is a *fully inductive* property of a module. The first part of Figure 3 defines a relation that we call *conditional inductiveness*, which is a generalization of both full inductiveness and the notion of visible inductiveness described earlier. Specifically, the relation $v : \tau \blacktriangleright^{P}_{Q} \texttt{Valid}$ may be read as "value v is conditionally inductive at type τ with respect to properties P and Q."

Full inductiveness. When P and Q are the same property I (*i.e.*, $P = Q = I$) , these rules correspond to the standard logical relation over closed values for System F [27], but where there is exactly one free type variable (α) and that type variable is associated with the concrete type τ_c and the unary relation I. Values of the abstract type α are in the relation if they satisfy I (rule I-A). Products satisfy the relation if their components satisfy the relation (rule I-Prod). Functions satisfy the relation if they take arguments in the relation to results in the relation (rule I-Fun).

The following corollary of Reynolds' theory of parametricity [23] says that if I is a representation invariant then all α-constructible values satisfy it.

Corollary 3.3.

$$v_m : \tau_m \blacktriangleright^{\mathcal{I}}_{\mathcal{I}} \texttt{Valid} \implies (\forall v : \tau_c \, . \, \mathfrak{C}_{\mathbf{M}} [v \, ; \alpha] \implies (\mathcal{I} \ v))$$

Therefore, to prove that a module meets a specification it is enough to identify a *sufficient representation invariant.*

Definition 3.4 (Sufficient Predicate: $\mathsf{Suf}^{\varphi}_{\mathbf{M}}[p]$)**.** A predicate $p : (\tau_c \rightarrow \texttt{bool})$ is *sufficient for proving that* $\mathbf{M}$ *satisfies* φ, denoted $\mathsf{Suf}^{\varphi}_{\mathbf{M}}[p]$, iff $\forall v : \tau_c \, . \, (p \ v) \implies (\varphi \, [\tau_c] \ v_m \ v)$.

Definition 3.5 (Sufficient Representation Invariant)**.** A predicate $\mathcal{I} : (\tau_c \rightarrow \texttt{bool})$ is called a sufficient representation invariant for a module $\mathbf{M}$ with respect to a specification φ, denoted $\mathbf{M} : \mathbf{F} \models_{\mathcal{I}} \varphi$, iff $\mathsf{Suf}^{\varphi}_{\mathbf{M}}[\mathcal{I}] \ \wedge \ v_m : \tau_m \blacktriangleright^{\mathcal{I}}_{\mathcal{I}} \texttt{Valid}$.

Theorem 3.6. *If a sufficient representation invariant exists, then the module satisfies the specification, i.e.*

$$\begin{gathered} \big(\exists \mathcal{I} : (\tau_c \rightarrow \texttt{bool}) \, . \, \mathbf{M} : \mathbf{F} \models_{\mathcal{I}} \varphi\big) \\ \implies (\forall v : \tau_c \, . \, \mathfrak{C}_{\mathbf{M}} [v \, ; \alpha] \implies (\varphi \, [\tau_c] \ v_m \ v)) \end{gathered}$$

Proof. Follows from Corollary 3.3, Definition 3.4, and Definition 3.5. □

Conditional inductiveness. When P and Q are not the same, conditional inductiveness informally requires that if the client supplies values of abstract type satisfying P then the module will produce values of abstract type satisfying Q. When conditional inductiveness is used to check visible inductiveness in our algorithm, P will be the set V_+ of examples that are known to be α-constructible by the module and Q will be a candidate representation invariant. The most interesting rule when P and Q are different is the I-Fun rule for functions. Specifically, notice the inversion of P and Q in the negative position: If the argument is a value of abstract type, it must satisfy P, not Q. In other words, this element of the formalism codifies the intuition that if the client supplies values that satisfy P then the module will supply values that satisfy Q.

$$w : \beta \blacktriangleright^P_Q \texttt{Valid} \quad (\text{I-B}) \qquad \frac{\vdash v : \tau_c \qquad (Q\ v)}{v : \alpha \blacktriangleright^P_Q \texttt{Valid}} \quad (\text{I-A}) \qquad \frac{v_1 : \tau_1 \blacktriangleright^P_Q \texttt{Valid} \qquad v_2 : \tau_2 \blacktriangleright^P_Q \texttt{Valid}}{\langle v_1, v_2 \rangle : (\tau_1 * \tau_2) \blacktriangleright^P_Q \texttt{Valid}} \quad (\text{I-Prod})$$

$$\frac{\vdash v : (\sigma_1 \to \tau_2)[\alpha \mapsto \tau_c] \qquad \forall v_1 . \forall v_2 . \Big(v_1 : \sigma_1 \blacktriangleright^Q_P \texttt{Valid} \ \wedge\ (v\ v_1) \Downarrow v_2 \implies v_2 : \tau_2 \blacktriangleright^P_Q \texttt{Valid}\Big)}{v : (\sigma_1 \to \tau_2) \blacktriangleright^P_Q \texttt{Valid}} \quad (\text{I-Fun})$$

$$\frac{\vdash v : \tau_c \qquad \neg(Q\ v)}{v : \alpha \blacktriangleright^P_Q \texttt{CEx}\ \langle \{\}, \{v\} \rangle} \quad (\text{I-A-CEx})$$

$$\frac{v_1 : \tau_1 \blacktriangleright^P_Q \texttt{CEx}\ \langle S, V \rangle \qquad \vdash v_2 : \tau_2[\alpha \mapsto \tau_c]}{\langle v_1, v_2 \rangle : (\tau_1 * \tau_2) \blacktriangleright^P_Q \texttt{CEx}\ \langle S, V \rangle} \quad (\text{I-Prod-CEx}_1) \qquad \frac{\vdash v_1 : \tau_1[\alpha \mapsto \tau_c] \qquad v_2 : \tau_2 \blacktriangleright^P_Q \texttt{CEx}\ \langle S, V \rangle}{\langle v_1, v_2 \rangle : (\tau_1 * \tau_2) \blacktriangleright^P_Q \texttt{CEx}\ \langle S, V \rangle} \quad (\text{I-Prod-CEx}_2)$$

$$\frac{\vdash v : (\sigma_1 \to \tau_2)[\alpha \mapsto \tau_c] \qquad v_1 : \sigma_1 \blacktriangleright^Q_P \texttt{Valid} \qquad (v\ v_1) \Downarrow v_2 \qquad v_2 : \tau_2 \blacktriangleright^P_Q \texttt{CEx}\ \langle S, V \rangle}{v : (\sigma_1 \to \tau_2) \blacktriangleright^P_Q \texttt{CEx}\ \big\langle \{\!|\, v_1 \,|\!\}_{\sigma_1} \cup S, V \big\rangle} \quad (\text{I-Fun-CEx})$$

$$\{\!|\, w \,|\!\}_\beta = \{\} \quad (\text{C-Base}) \qquad \{\!|\, v \,|\!\}_\alpha = \{v\} \ \text{ if } \vdash v : \tau_c \quad (\text{C-Abs}) \qquad \{\!|\, \langle v_1, v_2 \rangle \,|\!\}_{(\sigma_1 * \sigma_2)} = \{\!|\, v_1 \,|\!\}_{\sigma_1} \cup \{\!|\, v_2 \,|\!\}_{\sigma_2} \quad (\text{C-Prod})$$

Figure 3. Inference rules for conditional inductiveness.

Counterexamples. Normally logical relations are only used to prove that an invariant is inductive. However, we additionally require counterexamples from failed inductiveness checks, to drive our CEGIS-based invariant inference algorithm. The second section of Figure 3 provides the logic for refuting conditional inductiveness and generating counterexamples. This judgement has the form $v : \tau \blacktriangleright^P_Q \texttt{CEx}\ \langle S, V \rangle$, which can be read as "value v is not conditionally inductive at type τ with respect to properties P and Q, with inductiveness counterexample witnesses S and V." Here the set S contains values that satisfy P, the set V contains values that falsify Q, and intuitively the values in V can be computed using module operations given inputs from S.

As an example, consider the rule for values of abstract type (rule I-A-CEx). Here, a value v of type α is not conditionally inductive if it falsifies Q. The counterexample produced includes v in the set V (and returns the empty S), and hence satisfies the judgemental invariant explained above. As another example, a function is not conditionally inductive (rule I-Fun-CEx) if there is an argument v_1 in the relation that causes the function to produce a result v_2 that is not in the relation. In that case, the function $\{\!|\, v \,|\!\}_\sigma$ is used to collect all values of type α in v_1 to put in the returned set S, since they are the inputs that led to the result v_2.

The completeness of our algorithm for inferring sufficient representation invariants depends critically on these rules for generating counterexamples. In particular, values in S are added to the set V_- of negative examples in order to strengthen a candidate invariant, while values in V are added to the set V_+ of positive examples in order to weaken a candidate invariant. Therefore, the returned set S (V) must be non-empty whenever strengthening (weakening) is required, which we prove as part of our completeness theorem.

Given this theory of counterexamples, one can appreciate why handling higher-order functions is more challenging than first-order functions. Extracting counterexamples from a pair or other data structure requires a walk of the data structure, and such a procedure is trivially complete. However, extracting counterexamples from functional arguments requires execution of those arguments. That said, it is easy to extract counterexamples from functional arguments when the types of those functions do not include the abstract type α—in that case, there are no counterexamples and one could safely return the empty set. Therefore, our theory and formal guarantees extend naturally to modules that contain functions such as maps, folds (other than those that produce values of the abstract type), zips, and iterators, where function argument types refer to the *element* type of a data structure, not the abstract type of the data structure itself. The latter case actually appears surprisingly rare in practice, but it does exist. For instance, the abstract type appears in a higher-order position in a monadic interface. We explain how we lift the first-order restriction in our implementation in §4.

3.3 The Inference Algorithm

The invariant synthesis algorithm is parameterized by a verifier **Verify** and a synthesizer **Synth**. A call **Verify** P returns `Valid` when $P\ v$ is true on all inputs of type τ_c. Otherwise, it returns a counterexample v to the predicate. A call **Synth** $V_+\ V_-$ returns a predicate P that returns true on the positive examples (V_+) and false on the negative ones (V_-). V_+ and V_- should not overlap; if they do then **Synth** will fail.

```
1  Dependencies: A synthesizer Synth and a verifier Verify

3  Globals: An interface $\mathbf{F} = \exists\alpha.\tau_m$, a module $\mathbf{M} = \langle\tau_c, v_m\rangle$
   s.t. $v_m : \tau_m[\alpha \mapsto \tau_c]$, and a spec $\varphi : \forall\alpha.\tau_m \to \alpha \to$ bool

5  (* The specification $\varphi$ interpreted over the concrete type $\tau_c$
6   * and module implementation $v_m$ *)
7  let $\varphi_m$ = $(\varphi[\tau_c]\ v_m)$

9  (* Checks whether a candidate invariant $Q$ is
10  * conditionally inductive with respect to $P$ *)
11 let CondInductive $P$ $Q$ = $R$ where $v_m : \tau_m \blacktriangleright^{P}_{Q} R$

13 (* Checks if the candidate invariant is missing any value
14  * that is constructible from $V_+$ in a single step *)
15 let ClosedPositives $V_+$ $\mathcal{I}$ =
16   match CondInductive $V_+$ $\mathcal{I}$ with
17   | Valid ↦ Valid
18   | CEx $\langle\_, V\rangle$ ↦ CEx $V$

20 (* Checks if the candidate invariant is not inductive,
21  * or includes values that are not constructible *)
22 let NoNegatives $\mathcal{I}$ =
23   match Verify $\mathrm{Suf}^{\varphi}_{\mathbf{M}}[\mathcal{I}]$ with
24   | Valid ↦ begin
25       match CondInductive $\mathcal{I}$ $\mathcal{I}$ with
26       | Valid ↦ Valid
27       | CEx $\langle S, \_\rangle$ ↦ CEx $S$
28     end
29   | CEx $v$ ↦ CEx $\{v\}$

31 (* Returns a sufficient representation invariant *)
32 let rec Hanoi $V_+$ $V_-$ =
33   match Synth $V_+$ $V_-$ with
34   | Failure ↦ failwith "No predicate found"
35   | Success $\mathcal{I}$ ↦ begin
36       match ClosedPositives $V_+$ $\mathcal{I}$ with
37       | CEx $P$ ↦ Hanoi $(V_+ \cup P)$ $\emptyset$
38       | Valid ↦ begin
39           match NoNegatives $\mathcal{I}$ with
40           | CEx $N$ ↦
41             if $N \setminus V_+ = \emptyset$ then
42               failwith "Counterexample $N$";
43             else
44               Hanoi $V_+$ $(V_- \cup (N \setminus V_+))$
45           | Valid ↦ $\mathcal{I}$
46         end
47     end
```

Figure 4. The Hanoi framework.

Figure 4 presents our invariant inference algorithm. To execute the algorithm, a user invokes Hanoi (line 32) with empty sets for V_+ and V_- respectively. Hanoi first generates a candidate invariant $\mathcal{I}$ using **Synth**, given the current V_+ and V_- sets. It then attempts to produce a candidate invariant that is visibly inductive relative to V_+. That is the role of the call to ClosedPositives (line 36). That function simply calls CondInductive, which uses the inference rules in Figure 3. In the implementation these rules are executed through interaction with the verifier **Verify**.

Since everything in $V_+$is known to be constructible, the set V of values that violate the candidate invariant must also be constructible. Therefore, those values are returned from ClosedPositives, and they are added to V_+via a recursive call to Hanoi (line 37). This forces future candidate invariants produced by **Synth** to return `true` on elements in V. Note that each time V_+ is augmented, V_- is reset to the empty set, so the next synthesized invariant will be the constant true function, which is trivially visibly inductive. While we maintain the invariant that the positive examples are constructible and so must be included in the final invariant, negative examples are simply values that violate the current candidate invariant (but may in fact be constructible).

Once the candidate invariant $\mathcal{I}$ is visibly inductive with respect to V_+, Hanoi checks for sufficiency and full inductiveness by calling NoNegatives at line 39. The NoNegatives procedure interacts with **Verify** to check sufficiency and calls CondInductive to check full inductiveness. If either of these checks fail, NoNegatives will return counterexample values that satisfy the current invariant — either a sufficiency violation or the set S from an inductiveness counterexample. Because $\mathcal{I}$ is visibly inductive, $N \setminus V_+$ can only be empty if there is a sufficiency violation. In that case, we have found a constructible violation of the specification φ, so Hanoi terminates and provides this counterexample. If $N \setminus V_+$ is non-empty, Hanoi adds all of these values to V_-, and **Synth** will generate a stronger candidate invariant in the next iteration. If a constructible counterexample is added to V_-, it will eventually be generated by ClosedPositives and moved to V_+. If both checks in NoNegatives succeed, then we have found a sufficient representation invariant and it is returned.

3.4 Soundness and Completeness

We say that **Verify** is *sound* if (**Verify** P) = `Valid` implies $\forall v : \tau_c . (p\ v) \Downarrow$ `true`. Further, **Verify** is said to be complete if **Verify**$(p) = v$ implies $(p\ v) \Downarrow$ `false`. Likewise, we say that **Synth** is *sound* if for all sets V_+ and V_- of τ_c values, (**Synth** V_+ V_-) $= P$ implies $\forall v \in V_+ . (P\ v) \Downarrow$ `true` and $\forall v \in V_- . (P\ v) \Downarrow$ `false`. Further, **Synth** is said to be *complete* if for all sets V_+ and V_- of τ_c values, whenever there exists a predicate $P : \tau_c \to$ bool such that $\forall v^+ \in V_+ . (P\ v^+) \Downarrow$ `true` and $\forall v^- \in V_- . (P\ v^-) \Downarrow$ `false`, **Synth** always returns *some* predicate P'.

Definition 3.7 (Soundness)**.** An inference system for representation invariants is said to be sound iff whenever the system generates a predicate $\mathcal{I}$, it is indeed a sufficient representation invariant, i.e. $\mathbf{M} : \mathbf{F} \models_{\mathcal{I}} \varphi$.

Definition 3.8 (Completeness)**.** An inference system for representation invariants is said to be complete iff whenever there exists a sufficient representation invariant $\mathcal{I}$ such that $\mathbf{M} : \mathbf{F} \models_{\mathcal{I}_\star} \varphi$, the system always generates (terminates with) some predicate $\mathcal{I} : (\tau_c \rightarrow \texttt{bool})$.

Theorem 3.9. *If* **Verify** *is sound, then* Hanoi $\emptyset$ $\emptyset$ *is sound.*

Theorem 3.10. *If* **Verify** *and* **Synth** *are both sound and complete, and* τ_c *is a finite domain, then* Hanoi $\emptyset$ $\emptyset$ *is complete.*

The proofs can be found in the appendix of the full version of this paper [17]. The soundness of Hanoi is straightforward and follows from the fact that an invariant is only returned if it is both sufficient and inductive. The completeness argument for finite domains is much more involved. As mentioned earlier, it depends on several properties of the rules for generating counterexamples in Figure 3. Further, we must prove that the Hanoi algorithm always terminates. Notice that the size of the set V_+ monotonically increases during the algorithm. While V_- is reset to empty on some recursive calls, this is only done when V_+ is augmented. Hence the following is a rank function that is bounded from below and decreases lexicographically with each recursive Hanoi call, where $|\tau_c|$ denotes the number of values of type τ_c:

$$\mathcal{R}(V_+, V_-) \triangleq \left\langle |\tau_c| - |V_+|, |\tau_c| - |V_-| \right\rangle$$

4 Implementation

This section describes a variety of additional aspects of our ~5 KLOC OCaml implementation of Hanoi.

4.1 The Programming Language

We have implemented a pure, simply-typed, call-by-value functional language with recursive data types. Numbers are implemented as a recursive data type, where a number is either 0 or the successor of a number. Each program includes a prelude that may contain data type declarations and functions over those data types. A program also contains a single module declaring an abstract type together with operations over that abstract type. Finally, a program includes a universally quantified specification that defines the intended behavior of the module in terms of its operations.

4.2 Tackling Higher-Order Functions

While the theory presented in the previous section only supports first-order terms, our implementation allows modules to include arbitrary higher-order functions. As mentioned earlier, the key extension required is the ability to extract counterexample values of the abstract type from functions. Here we discuss how our implementation does that.

First, consider a natural extension to the SET interface from §2 to include a map function.

```
module type HOSET = sig
   include SET
   val map : (int -> int) -> t -> t
end
```

Notice that while map is a higher-order function, the type of the higher-order argument does not involve any occurrences of the abstract type t. The same is true of iter, zip, and many other variants. Consequently, if, during invariant inference, (map f v) fails an inductiveness check on some candidate representation invariant, the counterexample values that represent the "reason" for this failure will never come from f. More generally, when the abstract type does not appear in a higher type τ, the value with type τ cannot contain counterexamples. Our implementation therefore simply ignores such higher-order values when extracting counterexamples, just as it ignores ordinary base types such `int`.

Now consider a further extension that includes a fold.

```
module type FSET = sig
   include HOSET
   val fold : (int -> t -> t) -> t -> t -> t
end
```

Here, `fold` contains a function argument with a type including `t`. The fold might be implemented as follows.

```
let rec fold f a s =
   match s with
   | [] -> a
   | hd :: tl -> f hd (fold f a tl)
```

Given a call `fold f s1 s2` and a result `s'` that does not satisfy the current candidate invariant, how do we extract the counterexamples from the functional argument? The solution arises from reflecting back on the intuitive definition of conditional inductiveness: "if clients supply values in P then the module implementation should supply values in Q." In the higher-order case, there are simply more ways for client and implementation to interact across the module boundary. Specifically,the implementation supplies a value to the client when it calls a function argument, and the client supplies a value to the module when it returns from such a function. Fortunately, a mechanism already exists for tracking such boundary crossings in the general case: the higher-order contracts of Findler and Felleisen [9].

Therefore, our implementation extracts counterexamples through higher-order contract checking. The first-order case is straightforward. For example, when the type is `t -> t`, we generate a contract P `->` Q to check that arguments satisfy P and results satisfy Q, and we log situations where P is satisfied by an argument but Q is violated by the result. This is a direct implementation of the rule I-Fun-CEx in Figure 3.

For a type such as `(int -> t -> t) -> t -> t -> t`, we simply extend the idea, giving rise to the following contract.

$$\texttt{(any\_int -> } Q \texttt{ -> } P\texttt{) -> } P \texttt{ -> } P \texttt{ -> } Q$$

As per usual, all negative positions must satisfy P and the positive ones Q. Then contract checking is used to identify runs that satisfy all of the P checks but fail a Q check. In that case, if S is the set of values that satisfy P and v is the value that violates Q, then the extracted inductiveness counterexample is $\langle S, \{v\}\rangle$.

With this extension, Hanoi is trivially sound, for the same reason that the first-order algorithm is sound (the algorithm checks for soundness just before termination). We conjecture that Hanoi with this extension is also complete for finite domains but have not proven it. However, in the next section, we demonstrate empirically that our implementation can infer representation invariants in the presence of higher-order functions.

4.3 Verifier and Synthesizer

To implement **Verify**, we use a size-bounded enumerative tester, which is unsound but effective in practice. To validate a predicate with a single quantifier, we test the predicate on data structures, from smallest to largest, until either 3000 data structures have been processed, or the data structure has over 30 AST nodes, whichever comes first. To validate predicates with two or more quantifiers, we instantiate each quantifier with the smallest 3000 data structures with under 15 AST nodes. We further limit the total number of data structures processed to 30000. These restrictions limit the total amount of time spent in the verifier at the cost of soundness guarantees.

To implement **Synth**, we use Myth [20], adapting it slightly in two ways. First, we modified it to return a set of candidate invariants, instead of just one. Doing so permits the caching of synthesis results described earlier. Second, we had to manage Myth's requirement for *trace completeness*. Trace completeness requires that whenever we provide an input-output example $\langle x, y\rangle$ for a recursive data type, we also provide input-output examples for each subvalue of x. We generate input-output pairs for Myth by pairing each element of V_+ with true and each element of V_- with false. To handle trace completeness, we first identify all subvalues of the values in V_+ and V_-. For each such subvalue that does not already appear in V_+ or V_- we simply add it to V_-, which has the effect of mapping it to false. However, it could be that these values are actually constructible; if they are, future visible inductiveness checks will find this inconsistency, and move the value to V_+. However, this solution sometimes does make our synthesis task more difficult, as these additional values in V_- can force candidate invariants to be stronger than necessary. In such cases the synthesizer can spend more time searching for a complex invariant, when a simpler (though weaker) one would suffice.

4.4 Optimizations

To accelerate invariant inference, we have implemented two key optimizations: *synthesis result caching* and *counterexample list caching*. Synthesis result caching reduces the number of synthesis calls, and counterexample list caching reduces the number of verification and synthesis calls. Since the bulk of the system run time is spent in one or both kinds of calls, reducing them can have a substantial impact on performance.

V_+	V_-		Invariant		CEx
$\{v_1\}$	$\{\}$	$\xrightarrow{\text{synth}}$	$\lambda x.\mathtt{true}$	$\xrightarrow{\text{verify}}$	Neg v_2
$\{v_1\}$	$\{v_2\}$	$\xrightarrow{\text{synth}}$	$\lambda x.e_1$	$\xrightarrow{\text{verify}}$	Neg v_3
$\{v_1\}$	$\{v_2,v_3\}$	$\xrightarrow{\text{synth}}$	$\lambda x.e_2$	$\xrightarrow{\text{verify}}$	Neg v_4
$\{v_1\}$	$\{v_2,v_3,v_4\}$	$\xrightarrow{\text{synth}}$	$\lambda x.e_3$	$\xrightarrow{\text{verify}}$	Pos v_5

(a)

V_+	V_-		Invariant		CEx
$\{v_1,v_5\}$	$\{\}$	$\xrightarrow{\text{synth}}$	$\lambda x.\mathtt{true}$	$\xrightarrow{\text{verify}}$	Neg v_2
$\{v_1,v_5\}$	$\{v_2\}$	$\xrightarrow{\text{synth}}$	$\lambda x.e_1$	$\xrightarrow{\text{verify}}$	Neg v_3
$\{v_1,v_5\}$	$\{v_2,v_3\}$				

(b)

Figure 5. Partial traces of synthesis and verification results from a run of Hanoi without counterexample list caching enabled.

V^+	V^-		Invariant		CEx	
$\{v_1\}$	$\{\}$	$\xrightarrow{\text{synth}}$	$\lambda x.\mathtt{true}$	$\xrightarrow{\text{verify}}$	Neg v_2	$\mathtt{true}[v_5/x] \rightarrow^* \mathtt{true}$
$\{v_1\}$	$\{v_2\}$	$\xrightarrow{\text{synth}}$	$\lambda x.e_1$	$\xrightarrow{\text{verify}}$	Neg v_3	$e_1[v_5/x] \rightarrow^* \mathtt{true}$
$\{v_1\}$	$\{v_2,v_3\}$	$\xrightarrow{\text{synth}}$	$\lambda x.e_2$	$\xrightarrow{\text{verify}}$	Neg v_4	$e_2[v_5/x] \rightarrow^* \mathtt{false}$
$\{v_1\}$	$\{v_2,v_3,v_4\}$	$\xrightarrow{\text{synth}}$	$\lambda x.e_3$	$\xrightarrow{\text{verify}}$	Pos v_5	$e_3[v_5/x] \rightarrow^* \mathtt{false}$

Figure 6. Results of running a positive counterexample on the trace shown in Figure 5(a).

Synthesis Result Caching. When synthesizing, Myth often finds multiple possible solutions for a given set of input/output examples. Instead of throwing the unchosen solutions away, we store them for future synthesis calls. When given a set of input/output examples, before making a call to Myth, we check if any of the previously synthesized invariants satisfy the input/output example set. If one does, that invariant is used instead of a freshly synthesized one.

Counterexample List Caching. Consider the trace of Hanoi shown in Figure 5(a). In this example, Hanoi was just called with v_1 as the only positive example, and with no negative examples. With no negative examples, the synthesizer proposes $\lambda x.\mathtt{true}$ as a candidate invariant and then verification subsequently provides the negative counterexample, v_2, which then becomes the only negative example in the next attempt at synthesis. This loop of proposing new invariants, and adding their negative counterexamples to the negative example set continues until $\lambda x.e_3$ is proposed, which provides the positive counterexample, v_5.

Next, according to the unoptimized algorithm, one should begin a run with $\{v_1, v_5\}$ as positive examples and no negative examples—see Figure 5(b) for a partial trace of this subsequent execution. Suppose that v_5 satisfies the first two synthesized invariants from the original run. In that case, those invariants will simply be synthesized again as the first two candidates of this new run, as shown in the figure. To avoid this recomputation, we cache these traces of synthesis and verification calls. When we receive a new positive example, we run it on the synthesized invariants in the trace, as shown in Figure 6. Because $\lambda x.$`true` and $\lambda x.e_1$ both return `true` on v_5, we can skip the entirety of the trace shown in Figure 5(b) and begin by synthesizing from the testbed $V_+ = \{v_1, v_5\}$ and $V_- = \{v_2, v_3\}$.

5 Experimental Results

We aim to answer the following research questions:

1. Can we infer sufficient representation invariants in practice?
2. What are the primary performance factors?
3. What effect on performance do our optimizations have?
4. How does our algorithm compare with prior work?

5.1 Benchmark Suite

We evaluate Hanoi on a total of 28 verification problems, most of which require reasoning over list or tree structures. We categorize them into the following four groups.

- VFA (5): Four modules from Verified Functional Algorithms (VFA) [1] that have interfaces and specifications over those interfaces, including tree- and list-based implementations of lookup tables and priority queues. We also experimented with a second version of priority queues that excludes the `merge` function.
- VFAExt (3): Three VFA modules with additional function(s) and corresponding specifications from the Coq [30] standard library.
- Coq (14): Five tree- and list-based implementations of data structures from the Coq [30] standard library. One additional problem for each of the five by introducing additional binary functions. Four more problems by extending interfaces with higher-order functions.
- Other (6): Six additional benchmarks of our own creation requiring reasoning over lists, natural numbers, monads and other basic data structures.

5.2 Experimental Setup

All experiments were performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 RAM running macOS Mojave. We ran each benchmark 10 times with a timeout of 30 minutes and report the average time. If any of the 10 runs time out then we consider the benchmark as a whole to have timed out.

5.3 Inferred Invariants

Figure 7 presents our results. Overall, Hanoi terminated with an invariant on 22 out of 28 benchmarks within the timeout bound. The second column shows the sizes of the inferred invariants, in terms of their abstract syntax trees.

Though our verifier is unsound, there was no effect on the reliability of the system on our benchmark suite: 22 of the 22 inferred invariants are correct. Further, some of them are quite sophisticated. For example, we synthesize a heap invariant, which requires that the the elements of each node's subtrees is smaller than that node's label. We synthesize invariants over lists including "max element first," "no duplicates," and "ordered." If we allow the system to exceed the 30 min threshold, the system will infer a binary search tree invariant as well.

In seven of the cases above, we run into a limitation of the Myth synthesizer rather than our algorithm: Myth cannot synthesize functions that require recursive "helper" functions. To bypass this restriction, we added a `true_maximum` function (that finds the maximum element of a tree) to our tree-heap benchmark and a `min_max_tree` function (that finds the minimum and maximum elements of trees) to our bst and red-black-tree benchmarks. We added a * next to the names of benchmarks that we altered by providing a helper function in this way (see Figure 7).

5.4 Primary Performance Factors

When benchmarks complete within the 30 minute bound, most of the time is spent in verification. Indeed, for all but two of the terminating benchmarks, the total time spent synthesizing is under two seconds.

Three factors affect verification times significantly: (1) the strength of the specification, (2) the complexity of the underlying data structure, and (3) the presence of higher order functions. First, it takes our verifier longer to validate a true fact than to find a counterexample to a false one (validation requires enumeration of all tests; in contrast, the moment a counterexample is found, the enumeration is short-circuited). Many candidate invariants imply weak specifications but are not inductive. Hence weak specifications, ironically, are quite costly, because many candidate invariants wind up being sufficient (incurring a significant verification expense each time), only to be thrown away later when it turns out they are not inductive. Second, relatively simple data structures, like natural numbers and lists with numbers as their elements, take less time to verify than more complex data structures, like trees, tries, and lists with more complex elements. Third, like other complicated data types, the use of higher-order functions increases verification time. There are many ways to build a function, so enumeratively verifying a higher-order function requires searching through many possible functions.

However, the story is different for the complex benchmarks that do not complete within 30 minutes. When we ran Hanoi on our bst set benchmark, it completed in 78.4 minutes. Unlike the prior benchmarks, the majority of the time (65%) was spent in synthesis. Moreover, 30% of the total time was spent on the synthesis call that generated the final invariant. This indicates that Hanoi is currently not gated

Name	**Size**	**Time** (s)	**TVT** (s)	**TVC**	**MVT** (s)	**TST** (s)	**TSC**	**MST** (s)
/coq/bst-::-set*	t/o	t/o	t/o	t/o	97	t/o	t/o	t/o
/coq/bst-::-set+binfuncs	15	42.0	41.8	11	3.80	0.2	3	0.07
/coq/bst-::-set+hofs*	t/o	t/o	t/o	t/o	97	t/o	t/o	t/o
/coq/rbtree-::-set*	t/o	t/o	t/o	t/o	103	t/o	t/o	t/o
/coq/rbtree-::-set+binfuncs	t/o	t/o	t/o	t/o	103	t/o	t/o	t/o
/coq/rbtree-::-set+hofs*	t/o	t/o	t/o	t/o	103	t/o	t/o	t/o
/coq/maxfirst-list-::-heap	35	6.2	5.8	16	0.36	0.3	5	0.06
/coq/maxfirst-list-::-heap+binfuncs	35	7.4	7.0	16	0.44	0.3	5	0.06
/coq/sorted-list-::-set	49	22.9	22.0	21	1.05	0.7	7	0.10
/coq/sorted-list-::-set+binfuncs	49	17.3	16.0	21	0.76	1.2	8	0.15
/coq/sorted-list-::-set+hofs	49	101.3	99.6	21	4.74	1.6	8	0.20
/coq/unique-list-::-set	35	13.2	12.9	20	0.65	0.3	6	0.05
/coq/unique-list-::-set+binfuncs	15	15.7	15.6	8	1.95	0.1	2	0.05
/coq/unique-list-::-set+hofs	17	81.7	81.4	8	10.18	0.2	2	0.10
/other/cache	29	1.3	1.0	16	0.06	0.2	5	0.04
/other/listlike-tree	53	9.0	8.8	14	0.63	0.1	4	0.03
/other/nat-nat-option-::-range	23	1.6	1.5	12	0.12	0.1	4	0.03
/other/rational	28	8.6	7.8	8	0.97	0.7	2	0.35
/other/sized-list	45	15.4	15.0	20	0.75	0.3	6	0.05
/other/stutter-list	49	6.9	5.6	20	0.28	1.1	7	0.16
/vfa-extended/assoc-list-::-table	4	2.6	2.5	3	0.83	0.0	0	undef
/vfa-extended/bst-::-table	t/o	t/o	t/o	t/o	103	t/o	t/o	t/o
/vfa-extended/trie-::-table	4	15.5	15.4	3	5.13	0.0	0	undef
/vfa/assoc-list-::-table	4	1.9	1.9	3	0.63	0.0	0	undef
/vfa/bst-::-table	4	12.9	12.8	3	4.27	0.0	0	undef
/vfa/tree-::-priqueue*	47	65.7	53.6	54	0.99	9.6	15	0.64
/vfa/tree-::-priqueue+binfuncs*	47	79.4	64.5	54	1.19	12.4	15	0.83
/vfa/trie-::-table	4	17.7	17.6	3	5.87	0.0	0	undef

Figure 7. Information from running Hanoi on our benchmark suite. **Name** is the name of the benchmark. **Size** is the size of the inferred invariant. **Time** is the time to run the benchmark from start to end. **TVT** is the total time spent verifying. **TVC** is the total number of verification calls. **MVT** is the average time for a single verification call. **TST** is the total time spent synthesizing. **TSC** is the total number of synthesis calls. **MST** is the average time for a single synthesis call. Benchmarks marked with a * were provided an additional function to enable synthesis by Myth.

by the verifier, but by the synthesizer. Indeed, the implementation of bst set that includes binary functions like union and intersection is actually much faster than that without union and intersection, terminating within our 30 minute timeout. Adding these functions makes the verification harder, but Myth can use them to generate simpler invariants. Adding helper functions that permit simpler invariants also makes our implementation of a bst table verifiable in under 30 minutes.

Due to these limitations, we believe that a smarter synthesizer would be able to find more invariants. To this end, we built a prototype synthesizer that can generate more complex types of functions. This synthesizer has similarities to Myth as it is type-and-example directed and enumerative. However, where Myth can only synthesize simple recursive functions, this alternate synthesizer can synthesize *folds*, letting our synthesizer generate functions that require accumulators. Our synthesizer performs comparably to Myth, synthesizing invariants for the 20 benchmarks Myth that can solve an average of 11% slower. However, our synthesizer is also able to find the invariant for a binary heap (/vfa/tree-::-priqueue) without requiring helper functions or functions defined in the module in 185.4 seconds (55.8 seconds for /vfa/tree-::-priqueue+binfuncs), while Myth fails.

5.5 Comparisons

Figure 8 summarizes the results of running of Hanoi, Hanoi without optimizations, and our implementations of prior approaches adapted to our setting.

Impact of Optimizations. The modes Hanoi$_{\text{-SRC}}$ and Hanoi$_{\text{-CLC}}$ tested the impacts of our optimizations described in §4.4. Hanoi$_{\text{-SRC}}$ runs the benchmarks with synthesis result caching turned off. Hanoi$_{\text{-CLC}}$ runs the benchmarks with counterexample list caching turned off.

Removing synthesis result caching does not have a large impact on the majority of benchmarks as the majority of our benchmarks spend relatively little time in synthesis. However, more complex benchmarks are able to enjoy the benefits of this optimization.

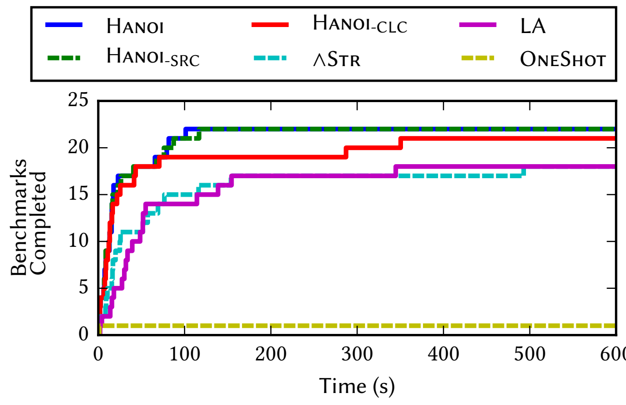

Figure 8. Number of benchmarks that terminate in a given time. Hanoi is the full Hanoi tool. Hanoi-SRC is Hanoi with synthesis result caching turned off. Hanoi-CLC is Hanoi with counterexample list caching turned off. ∧Str is Hanoi using a conjunctive strengthening algorithm similar to that of LoopInvGen. LA is Hanoi using a counterexample generation strategy similar to that of LinearArbitrary. OneShot synthesizes based on the results of running the 30 smallest values on the specification. All the tests that terminated within the 30 minute timeout terminated within the first 10.

Counterexample list caching has significant impact on complex benchmarks as they have more synthesis and verification calls. The synthesizer requires more input/output examples to synthesize the correct invariant on complex benchmarks, so saving time reconstructing the negative examples via counterexample list caching has great impact.

Comparison to **∧Str.** The ∧Str mode simulates the LoopInvGen algorithm [21], a related data-driven system for inferring loop invariants. When running ∧Str, if a candidate invariant $\mathcal{I}_1$ is sufficient to prove the specification, but is not inductive, the algorithm attempts to synthesize a new predicate $\mathcal{I}_2$ such that $v_m : \tau_m \blacktriangleright^{\mathcal{I}_1 \wedge \mathcal{I}_2}_{\mathcal{I}_1}$ `Valid`. In that case, $\mathcal{I}_1 \wedge \mathcal{I}_2$ is considered the new candidate invariant. This process continues until either the conjoined invariants are inductive, or they are overly strong so a new positive counterexample is found, at which point the whole process restarts. Hanoi outperforms ∧Str on all the benchmarks and solves 3 more benchmarks within 30 minutes. The main downside of ∧Str is that it can only add new positive examples in order to weaken the candidate invariant after it has obviously overstrengthened. Hanoi, in contrast, uses visible inductiveness checks to eagerly weaken in a directed manner.

Comparison to **LA..** LA mode simulates the LinearArbitrary algorithm [31], which is used in a data-driven CHC solver. There are two differences from Hanoi. First, LA tries to satisfy individual inductiveness constraints, generated for each function in the module, one at a time rather than all at once. Second, rather than eagerly searching for visible inductiveness violations, only full inductiveness counterexamples are obtained. However, if a full inductiveness counterexample happens to also be a visible inductiveness counterexample then it is treated accordingly. Hanoi outperforms LA on all the benchmarks and is able to solve 4 more benchmarks within 30 minutes. While Hanoi checks eagerly for positive counterexamples, LA finds them nondeterministically. Without performing the guided search through visible inductiveness checks, the algorithm sometimes gets "stuck" in holes of negative counterexamples. While the algorithm does seem to emerge from these holes eventually, it takes time.

Comparison to **OneShot.** The OneShot mode uses "one shot learning" rather than an interative CEGIS algorithm. The OneShot algorithm runs the specification over the smallest 30 elements of the concrete implementation type, tagging each element as either positive or negative. Doing so generates sets V_+ and V_-, which may be supplied to the synthesizer. Whatever invariant synthesized is returned as the result. (This algorithm only works when the specification quantifies over a single element of the abstract type, which is true for all but 7 of our benchmarks.) Running the OneShot algorithm fails on all but one of our benchmarks, `coq/unique-list-set`, and does so for a variety of reasons. On some benchmarks, Myth times out, indicating that the given synthesis problem was too hard, and Myth needed to be provided fewer examples to find the right invariant. On some benchmarks, Myth returns the wrong invariant, indicating that the synthesis problem was underspecified, and too few examples were given. Merely choosing some fixed number of examples to build the invariant with is insufficient, that fixed number is too high for some benchmarks, and too low for others.

6 Related Work

Inferring Representation Invariants. To our knowledge, the only prior work that attempts to automatically infer representation invariants for data structures is the Deryaft system by Malik et al. [15], which targets Java classes. There are three key differences between systems. First, Deryaft requires a fixed set of predicates (e.g., sortedness) as an input; the invariants generated are conjunctions of these predicates. In contrast, Hanoi can learn new predicates from a general grammar of programs. Second, the conjunction of predicates that Deryaft produces consists of those predicates that hold on a fixed set of examples (generated from test executions). There is no guarantee the final invariant is inductive. In contrast, Hanoi employs a CEGIS-based approach to refining the candidate invariant, terminating only when a sufficient representation invariant has been identified. Third, the Deryaft algorithm comes with no theoretical completeness guarantee.

Solving constrained Horn clauses. Recently, several tools have been developed to infer predicates that satisfy a given set of *constrained Horn clauses* (CHCs). Inferring representation invariants can be seen as a special case of CHC solving, since all of our inductiveness constraints are Horn clauses (e.g., $(\mathcal{I}_\star\ s_1) \wedge (\mathcal{I}_\star\ s_2) \implies (\mathcal{I}_\star\ (\texttt{union}\ s_1\ s_2))$). CHCs can include multiple unknown predicates in their inference problem, whereas there is only one in ours.

However, existing CHC solvers do not support inference of recursive predicates, which is necessary to handle representation invariants over recursive data types. Several solvers support only arithmetic constraints [7, 8, 31], while others support arrays or bit vectors [5, 11, 13] as well. To our knowledge, Eldarica [11] is the only CHC solver that supports algebraic data types. However, Eldarica only computes recursion-free solutions [11, Sec III C] and therefore cannot express the *sortedness* or *no-duplicates* properties, for instance.

Though they only handle arithmetic constraints, two of these solvers employ a data-driven technique that is similar to our approach, iterating between a synthesizer and a verifier [7, 31]. Like our work, the approach of Zhu et al. [31] leverages the observation that handling inductiveness counterexamples $\langle x, y\rangle$ is easy when we know that x is constructible. However, their approach simply checks if a counterexample to full inductiveness happens to have this property, while we exhaustively iterate through these counterexamples until a candidate invariant is *visibly inductive*. Intuitively, our approach minimizes the number of inductiveness counterexamples that must be treated heuristically. We show that this difference results in a significant performance improvement, and we have proven a completeness result for finite domains, while the approach of Zhu et al. [31] lacks a completeness result. The approach of Ezudheen et al. [7] does have a completeness result, and it applies to the infinite domain of integers. However, they achieve this guarantee through the use of a specialized synthesizer designed to handle inductiveness counterexamples directly, while our approach can use any off-the-shelf synthesizer. There is also no obvious analogue to our analysis of higher-order programs in this context.

Inferring inductive invariants. There have been many techniques developed to infer individual inductive invariants for program verification, for example an inductive invariant for a loop or for a system transition relation. As discussed in §2.2, module functions may consume or produce multiple arguments or results of the abstract type, which results in a more general class of inductiveness counterexamples, whereas loops and transition relations, when viewed as functions, consume and produce exactly one "state" (the analogue of an abstract value in our setting).

Hanoi is similar in structure to several data-driven invariant synthesis engines [2, 7, 8, 10, 14, 18, 19, 21, 25, 26]. We experimentally compared our algorithm to our implementations of the most closely related ones, adapted to the context of representation invariant synthesis (§5.5). Broadly, the technical distinctions are similar to those described above for data-driven CHC solvers. In particular: (1) our development of visible inductiveness is novel; (2) aside from one tool [10] that depends upon a special-purpose synthesizer to handles inductiveness counterexamples directly, none are proven complete; (3) they cannot process higher-order programs; and (4) to our knowledge, none infer recursive invariants.

The `ic3` algorithm for SAT-based model checking employs a notion of *relative inductiveness* [4], which is closely related to our notions of conditional and visible inductiveness. Formally, relative inductiveness is the special case of our conditional inductiveness relation $v : \tau \blacktriangleright^{P}_{Q} \texttt{Valid}$ where P has the form $Q' \wedge Q$ and $\tau = \alpha \rightarrow \alpha$. The `ic3` algorithm uses relative inductiveness to incrementally produce an inductive invariant, by iteratively identifying a state s that leads to a property violation and then conjoining an inductive strengthening of $\neg s$ to the candidate invariant. Our notion of visible inductiveness is also used to incrementally produce an inductive invariant, but it works in the opposite direction: we iteratively identify constructible values of the abstract type and use them to weaken the candidate invariant. This approach is a natural fit for our data-driven setting.

Automatic data structure verification. The Leon framework [29] can automatically verify correctness of data structure implementations, but to do so, a user must manually define an *abstraction function*, which plays a similar role to a representation invariant. Namely, the abstraction function is a partial function mapping an element of the concrete type to an element of the abstract type.

There are many techniques for proving properties of heap-based data structures, including shape analysis [24] and liquid types [12]. These techniques can prove and/or infer sophisticated invariants, often of imperative code. However, they are designed to tackle a different problem and do not infer the inductive representation invariants that are needed to prove modules correct.

7 Conclusion

We present a novel algorithm for synthesis of representation invariants. Our key insight is that it is possible to drive progress of the algorithm towards its goal not by eagerly searching for fully inductive invariants, but rather by searching first for *visibly inductive* invariants. We have proven that our algorthm is sound and complete, given a sound and complete verifier and synthesizer, for a first-order type theory with finite types. We also explain how to extend the algorithm to modules containing higher-order functions, which involves using contracts to validate and collect objects that cross the module boundary. We evaluate our algorithm on 28 benchmarks and find that we are able to synthesize 22 of the invariants within 30 minutes (and most of those in under a minute). Our algorithm is defined independently of the black-box verifier and synthesizers; as research in verifier and synthesizer technologies improve, so too will the capabilities of our overall system. While the tool is currently fully automated, we view this as a step towards an interactive approach to helping users of proof assistants produce correct representation invariants.

Acknowledgments

Thanks to our shepherd James Bornholt and the other reviewers for their helpful comments. This work is supported in part by NSF grant CCF-1837129 and a PhD fellowship from Microsoft Research.

References

[1] Andrew Appel. 2018. Software Foundations Volume 3: Verified Functional Algorithms. https://softwarefoundations.cis.upenn.edu/vfa-current/index.html

[2] Haniel Barbosa, Andrew Reynolds, Daniel Larraz, and Cesare Tinelli. 2019. Extending Enumerative Function Synthesis via SMT-Driven Classification. In *2019 Formal Methods in Computer Aided Design, FMCAD*. IEEE, 212–220. https://doi.org/10.23919/FMCAD.2019.8894267

[3] C. Boyapati, S. Khurshid, and D. Marinov. 2002. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*. ACM, Roma, Italy, 123–133.

[4] Aaron R. Bradley. [n.d.]. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science)*. Springer, 70–87.

[5] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS (Lecture Notes in Computer Science)*, Vol. 10805. Springer, 365–384. https://doi.org/10.1007/978-3-319-89960-2_20

[6] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs.. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00) (ACM Sigplan Notices)*, Vol. 35.9. ACM Press, N.Y., 268–279.

[7] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE Learning for Synthesizing Invariants and Contracts. *PACMPL* 2, OOPSLA (2018), 131:1–131:25. https://doi.org/10.1145/3276501

[8] Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. 2017. Sampling Invariants from Frequency Distributions. In *2017 Formal Methods in Computer Aided Design, FMCAD*. IEEE, 100–107. https://doi.org/10.23919/FMCAD.2017.8102247

[9] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, NY, 48–59.

[10] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM, 499–512. https://doi.org/10.1145/2837614.2837664

[11] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *2018 Formal Methods in Computer Aided Design, FMCAD*. IEEE, 1–7. https://doi.org/10.23919/FMCAD.2018.8603013

[12] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. 2009. Type-based Data Structure Verification. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 304–315. https://doi.org/10.1145/1542476.1542510

[13] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-Based Model Checking for Recursive Programs. *Formal Methods in System Design* 48, 3 (2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4

[14] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: Using Dynamic Analysis to Infer Program Invariants in Separation Logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 788–801. https://doi.org/10.1145/3314221.3314634

[15] Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid. 2007. Generating Representation Invariants of Structurally Complex Data. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS (Lecture Notes in Computer Science)*, Vol. 4424. Springer, 34–49. https://doi.org/10.1007/978-3-540-71209-1_5

[16] Bertrand Meyer. 1997. Design by Contract: Making Object-Oriented Programs that Work. In *TOOLS (25)*. IEEE Computer Society, 360. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5604

[17] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. arXiv:cs.PL/2003.12106

[18] Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. 2018. Invariant Synthesis for Incomplete Verification Engines. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS (Lecture Notes in Computer Science)*, Vol. 10805. Springer, 232–250. https://doi.org/10.1007/978-3-319-89960-2_13

[19] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-Guided Approach to Finding Numerical Invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*. ACM, 605–615. https://doi.org/10.1145/3106237.3106281

[20] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, POPL*. ACM, 619–630. https://doi.org/10.1145/2737924.2738007

[21] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 42–56. https://doi.org/10.1145/2908080.2908099

[22] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press. https://www.cis.upenn.edu/~bcpierce/tapl/

[23] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. 513–523.

[24] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1999. Parametric Shape Analysis via 3-valued Logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/292540.292552

[25] Rahul Sharma and Alex Aiken. 2016. From Invariant Checking to Invariant Inference using Randomized Search. *Formal Methods in System Design* 48, 3 (2016), 235–256. https://doi.org/10.1007/s10703-016-0248-5

[26] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In

Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS. 7762–7773. http://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification

[27] Lau Skorstengaard. 2015. An Introduction to Logical Relations. https://www.cs.uoregon.edu/research/summerschool/summer16/notes/AhmedLR.pdf Notes based on lectures by Amal Ahmed at the Oregon Programming Languages Summer School.

[28] Armando Solar-Lezama. 2013. Program Sketching. *STTT* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[29] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability Modulo Recursive Programs. In *Static Analysis - 18th International Symposium, SAS (Lecture Notes in Computer Science)*, Vol. 6887. Springer, 298–315. https://doi.org/10.1007/978-3-642-23702-7_23

[30] The Coq Development Team. 2019. *The Coq Proof Assistant, version 8.10.0*. https://doi.org/10.5281/zenodo.3476303

[31] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 707–721. https://doi.org/10.1145/3192366.3192416