Active Learning of Points-To Specifications

OSBERT BASTANI, Stanford University
ALEX AIKEN, Stanford University
PERCY LIANG, Stanford University

Large libraries pose significant challenges to static points-to analysis. A popular solution is to have a human analyst provide *points-to specifications* that summarize relevant behaviors of library code, which can substantially improve precision and scalability, and furthermore handle missing code such as native code. We propose Atlas, a tool that automatically infers points-to specifications. Atlas synthesizes test cases that exercise the library code, and then generates points-to specifications based on observations from these executions. Atlas automatically infers 97% of specifications for the Java Collections API, and discovers 20% more points-to edges than existing, handwritten specifications.

ACM Reference format:
Osbert Bastani, Alex Aiken, and Percy Liang. 2017. Active Learning of Points-To Specifications. 1, 1, Article 1 (January 2017), 36 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large libraries can significantly reduce the effectiveness of static analysis due to (i) native code that cannot be analyzed, (ii) challenging language features such as reflection, and (iii) deep abstractions that reduce precision and scalability. For example, the implementation of the `Vector` class in OpenJDK 1.7 uses multiple levels of indirection and calls the native function `System.arrayCopy`.

A standard workaround is to use *specifications* that summarize the relevant behaviors of library functions (Facebook 2017; Zhu et al. 2013). For a one-time cost of writing specifications for the library, the scalability, precision, and soundness of the static analysis can improve dramatically when analyzing any client code. However, the large number of functions in libraries makes writing specifications for the entire library prohibitively expensive (Bastani et al. 2015b; Zhu et al. 2013), and manually written specifications are often error prone (Heule et al. 2016). Furthermore, every time the library code is updated, the specifications have to be updated as well.

To address these issues, approaches have been proposed for automatically *inferring* specifications for library code, both based on dynamic analysis (Alur et al. 2005; Bastani et al. 2015a; Nimmer and Ernst 2002; Sharma and Aiken 2014; Sharma et al. 2012) and on static analysis (Ammons et al. 2002; Beckman and Nori 2011; Kremenek et al. 2006; Livshits et al. 2009; Ramanathan et al. 2007; Shoham et al. 2008). In particular, tools have been designed to infer properties of missing code, including taint flow properties (Clapp et al. 2015), function models (Heule et al. 2016, 2015), and callback control flow (Jeon et al. 2016). While these approaches are incomplete, and may not infer sound specifications, current static analyses in practice already rely on user-provided specifications (Facebook 2017), and as we will show, such tools can outperform human analysts.

We propose an algorithm based on dynamic analysis that infers library specifications summarizing points-to effects relevant to a flow-insensitive points-to analysis. Two constraints make our problem substantially more challenging than previously studied settings:

2017. XXXX-XXXX/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

```
boolean test() {
  Object in = new Object(); // o_in
  Box box = new Box(); // o_box
  box.set(in);
  Object out = box.get();
  return in == out; }
```

```
class Box { // specification
  Object f;
  void set(Object ob) { f = ob; }
  Object get() { return f; }
  Box clone() {
    Box b = new Box(); // ~o_clone
    b.f = f;
    return b; }}
```

Fig. 1. An example of a program using the Box class in the library (right), and the implementation of the library functions set, get, and clone in the Box class.

- Points-to effects cannot be summarized for a library function in isolation, e.g., in Figure 1, `set`, `get`, and `clone` all refer to the mutual field `f`.
- We may not be able to instrument library code, e.g., native code.

Now, suppose our algorithm proposes a candidate specification, and we want to check whether this candidate is "correct". More precisely, we want to ensure that the candidate is *admissible*, i.e., there is no strictly better specification. The first constraint says that the candidate must simultaneously summarize the points-to effects of `set`, `get`, and `clone`, and the second constraint says that we can only use input-output examples to check the admissibility.

We introduce *path specifications* to describe points-to effects of library code. Each path specification summarizes a single points-to effect of a combination of functions. An example is:

> For two calls `box.set(x)` and `box.get(0)`, the return value of `get` may alias `x`.

Path specifications have two desirable properties:

- We can check if a candidate path specification is admissible using input-output examples.
- A set of individually admissible path specifications is admissible as a whole.

These two properties imply that we can infer path specifications incrementally. In particular, we formulate the problem of inferring path specifications as a *language inference problem* (Oncina and Garcia 1992), and we develop a language inference algorithm tailored to our problem instance. Our algorithm builds the language incrementally in two phases—it first infers a finite language of path specifications that it is certain are admissible (leveraging the two properties described above), and then inductively generalizes this language while trying to retain admissibility.

We implement our algorithm in a tool called Atlas[1], which infers path specifications for functions in Java libraries. In particular, we evaluate Atlas by using it to infer specifications for the Java Collections API, since this API contains many functions that exhibit complex points-to effects. Atlas infers the correct specifications for 97% of these functions. Previously, we had manually written points-to specifications for the Java Collections API—Atlas inferred 10× as many specifications.

We compare our specifications to handwritten specifications on a benchmark of 46 Android apps. Using these inferred specifications increases the precision of our static points-to analysis by 53% compared to analyzing the library code, and increases recall by 20% compared to using handwritten specifications. While the specifications synthesized by Atlas are incomplete, we show that using the inferred specifications achieves 76% recall for nontrivial points-to edges, including 100% recall for almost half the programs in our benchmark. Our contributions are:

- We introduce path specifications, and prove that they are sufficiently expressive to precisely model the library code when using a standard flow-insensitive points-to analysis.

[1] Atlas stands for AcTive Learning of Alias Specifications.

- We formulate the problem of inferring path specifications as a language inference problem, and we design a language inference algorithm tailored to our problem instance.
- We implement our approach in Atlas, and use it to infer a large number of useful specifications for the Java Collections API.

2 OVERVIEW

Our algorithm infers a set of specifications that describe the behaviors of the library functions that are relevant to our static points-to analysis. It requires two inputs:

- **Library interface:** The type signature of each function in the library.
- **Blackbox access:** The ability to execute a library function on a chosen input and obtain the corresponding output.

Because we only have blackbox access to the library code, it is impossible to guarantee that the inferred specifications are both sound and precise. Instead, any inference algorithm must make tradeoffs between these two properties. Our algorithm aims to ensure that the inferred path specifications S are *admissible*, which says that there are no "strictly better" path specifications S', i.e., the precision and recall of S are as good as those of S' and at least one is strictly better.

To this end, our algorithm infers specifications incrementally in two phases. In the first phase, our algorithm only infers specifications it is certain are admissible. In the second phase, it inductively generalizes this set of specifications, using a large number of tests to minimize the chance of inadmissibility. In our experiments, this phase does not infer any inadmissible specifications.

We define precision for path specifications with respect to Andersen's analysis (Andersen 1994), a context- and flow-insensitive points-to analysis, and to context- or object-sensitive extensions of this analysis based on cloning (Whaley and Lam 2004). We show that path specifications can precisely model the library code when using Andersen's analysis; they are also compatible with other points-to analyses, but may be lose precision. For example, the path specifications for the `List` class describe the same points-to effects as the following code:

```
class List {
  Object f;
  void add(Object ob) { f = ob; }
  Object get(int i) { return f; } }
```

Andersen's analysis does not lose any precision by analyzing this code (or path specifications) instead of the true implementation of `List`, but a more precise static analysis may lose precision.

2.1 Path specifications

Our algorithm infers *path specifications* that summarize the effects of library code. A path specification is simply a sequence $s \in \mathcal{V}^*_{\text{path}}$, where $\mathcal{V}^*_{\text{path}}$ are variables in the library interface. For example, a path specification for the library functions `set` and `get` in Figure 1 is

$$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} \rightarrow \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}. \tag{1}$$

Here, $\texttt{this}_m$ and r_m denote the receiver and return value of library function m, respectively. The arrows in the path specification are for clarity; we can equivalently write this path specification as a sequence $\texttt{ob}\ \texttt{this}_{\texttt{set}}\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}$. Its meaning is the following logical formula:

$$(\texttt{this}_{\texttt{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}} \in \overline{G}) \Rightarrow (\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}} \in \overline{G}). \tag{2}$$

Intuitively, the notation $x \xrightarrow{A} y$ is an edge indicating that x and y satisfy relation A (e.g., they are aliased), and the graph $\overline{G}$ is the set of all such edges. Then, this formula says that if the receivers of `set` and `get` are aliased, then the parameter `ob` of `set` may be *transfered* to the return value of `get`.

The transfer relation $x \xrightarrow{\text{Transfer}} y$ essentially encodes that x may be "indirectly assigned" to y. For example, in the code `z = x; y = z;`, `x` is transfered to `y`.

Path specifications have two key benefits. First, we can devise a test case P to check whether a given path specification s is admissible. The premise of s holds for P, but its conclusion only holds for P if the points-to effect specified by s is exhibited by the library code. Upon executing P, if we observe that the conclusion of s holds for P, then we have proven that s is admissible. Second, given a set S of path specifications, if we have determined that every path specification $s \in S$ is admissible, then we guarantee S is admissible as a whole—i.e., path specifications compose.

Continuing our example, the test case for the path specification (1) is the `test` function shown in Figure 1. Essentially, if we ignore the implementation of the `set` and `get` functions, then the premise of (2) holds for this program, but not the conclusion. Upon executing the program, if we see that the conclusion of (2) holds during execution, then we know that the behavior of the library functions specified by the path specification can occur, i.e., it is admissible.

2.2 Phase One: Sampling Positive Examples

Our algorithm initializes the set of inferred specifications to $S \leftarrow \emptyset$, and then repeats:

(1) Propose a *candidate path specification* s.
(2) Synthesize a test case that checks whether s is admissible.
(3) Execute the test cases, and accept s (i.e., $S \leftarrow S \cup \{s\}$) if the test case passes.

By design, the synthesized test case passes only if s is admissible; therefore, at the end of the first phase, S remains admissible. However, the test cases may fail even if s is admissible—we cannot guarantee that no missed corner cases exist in the library code. Thus, test cases are designed heuristically to pass for the majority of admissible candidates.

2.3 Phase Two: Inductive Generalization

We show that path specifications can precisely model any library code with respect to Andersen's analysis (and its context- and object-sensitive extensions). However, the required set of path specifications may be infinitely large. Phase two inductively generalizes the finite set of specifications inferred in phase one to a description of a potentially infinite set of path specifications.

Since a path specification s is a sequence of variables $s \in \mathcal{V}_{\text{path}}^*$ (where $\mathcal{V}_{\text{path}}$ are the variables in the library interface), a set S of path specifications is a formal language over the alphabet $\mathcal{V}_{\text{path}}$. Thus, we can frame the inductive generalization problem as a language inference problem: given (i) the finite set of positive examples from phase one, and (ii) an oracle we can query to determine whether a given path specification s is admissible (though this oracle is *noisy*, i.e., it may return false even if s is admissible), the goal is to infer a (possibly infinite) language $S \subseteq \mathcal{V}_{\text{path}}^*$.

We devise a language inference algorithm based on RPNI (Oncina and Garcia 1992). Our algorithm proposes candidate inductive generalizations of S, and then checks the admissibility of each candidate using a large number of test cases. Unlike phase one, a generalization may be inadmissible even if all the test cases pass; we show empirically that admissibility is maintained.

While our algorithm infers a regular set of path specifications, in general, the set of path specifications required to model the library code may not be regular. We find that regular sets of path specifications suffice to model library code occurring in practice (see Section 9.1).

For example, the path specifications for `set`, `get`, and `clone` functions are

$$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} (\rightarrow \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}})^* \rightarrow \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}. \tag{3}$$

These specifications say that if we call `set`, then call `clone` n times in sequence, and finally call `get` (all with the specified aliasing between receivers and return values), then the parameter `ob` of

$$\text{(assign)}\ \frac{y \leftarrow x}{x \xrightarrow{\text{Assign}} y} \qquad \text{(allocation)}\ \frac{o = (x \leftarrow X())}{o \xrightarrow{\text{New}} o} \qquad \text{(store)}\ \frac{y.f \leftarrow x}{x \xrightarrow{\text{Store}[f]} y} \qquad \text{(load)}\ \frac{y \leftarrow x.f}{x \xrightarrow{\text{Load}[f]} y}$$

$$\text{(call parameter)}\ \frac{y \leftarrow m(x)}{x \xrightarrow{\text{Assign}} p_m} \qquad \text{(call return)}\ \frac{y \leftarrow m(x)}{r_m \xrightarrow{\text{Assign}} y} \qquad \text{(backwards)}\ \frac{x \xrightarrow{\sigma} y}{y \xrightarrow{\overline{\sigma}} x}$$

Fig. 2. Rules for constructing a graph G encoding the relevant semantics of program statements.

$$\begin{aligned}
\text{Transfer} &\to \epsilon \mid \text{Transfer Assign} \mid \text{Transfer Store}[f]\ \text{Alias Load}[f] \\
\overline{\text{Transfer}} &\to \epsilon \mid \overline{\text{Assign}}\ \overline{\text{Transfer}} \mid \overline{\text{Load}[f]}\ \text{Alias}\ \overline{\text{Store}[f]}\ \overline{\text{Transfer}} \\
\text{Alias} &\to \overline{\text{Transfer}}\ \overline{\text{New}}\ \text{New Transfer} \\
\text{FlowsTo} &\to \text{New Transfer}
\end{aligned}$$

Fig. 3. Productions for the context-free grammar C_{pt}. The start symbol of C_{pt} is FlowsTo.

`set` will be transfered to the return value of `get`. Then, phase one may infer

$$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} \to \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}} \to \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}} \to \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}.$$

Then, phase two would inductively generalize this specification to (3).

3 BACKGROUND ON POINTS-TO ANALYSIS

We consider programs with assignments $y \leftarrow x$ (where $x, y \in \mathcal{V}$ are variables), allocations $x \leftarrow X()$ (where $X \in \mathcal{C}$ is a type), stores $y.f \leftarrow x$ and loads $y \leftarrow x.f$ (where $f \in \mathcal{F}$ is a field), and calls to library functions $y \leftarrow m(x)$ (where $m \in \mathcal{M}$ is a library function). For simplicity, we assume that each library function m has a single parameter p_m and a return value r_m.

An *abstract object* $o \in \mathcal{O}$ is an allocation statement $o = (x \leftarrow X())$. A *points-to edge* is a pair $x \hookrightarrow o \in \mathcal{V} \times \mathcal{O}$. A static points-to analysis computes points-to edges $\Pi \subseteq \mathcal{V} \times \mathcal{O}$. Our results are for Andersen's analysis, a flow-insensitive points-to analysis (Andersen 1994), but generalize to object- and context-sensitive extensions based on cloning (Whaley and Lam 2004). We formulate Andersen's analysis as a context-free language reachability problem (Kodumal and Aiken 2004, 2005; Reps 1998; Sridharan and Bodík 2006; Sridharan et al. 2005).

Graph representation. First, our static analysis constructs a labeled graph G representing the program semantics. The vertices of G are $\mathcal{V} \cup \mathcal{O}$. The edge labels are

$$\Sigma_{\text{pt}} = \{\text{Assign}, \text{New}, \text{Store}, \text{Load}, \overline{\text{Assign}}, \overline{\text{New}}, \overline{\text{Load}}, \overline{\text{Store}}\}$$

that encode the semantics of program statements. The rules for constructing G are in Figure 2. For example, the edges extracted for the program `test` in Figure 1 are the solid edges in Figure 4.

Transitive closure. Second, our static analysis computes the *transitive closure* $\overline{G}$ of G according to the context-free grammar C_{pt} in Figure 3. First, a *path* $y \overset{\alpha}{\dashrightarrow} x$ in G is a sequence of edges

$$x \xrightarrow{\sigma_1} v_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_k} y$$

such that $\alpha = \sigma_1 \ldots \sigma_k \in \Sigma_{\text{pt}}^*$. Then, $\overline{G}$ contains (i) the edges $x \xrightarrow{\sigma} y$ in G, and (ii) if there is a path $x \overset{\alpha}{\dashrightarrow} y$ in G such that $A \overset{*}{\Rightarrow} \alpha$ (where A is a nonterminal), the edge $x \xrightarrow{A} y$. The graph $\overline{G}$ can be computed using dynamic programming; see (Melski and Reps 2000).

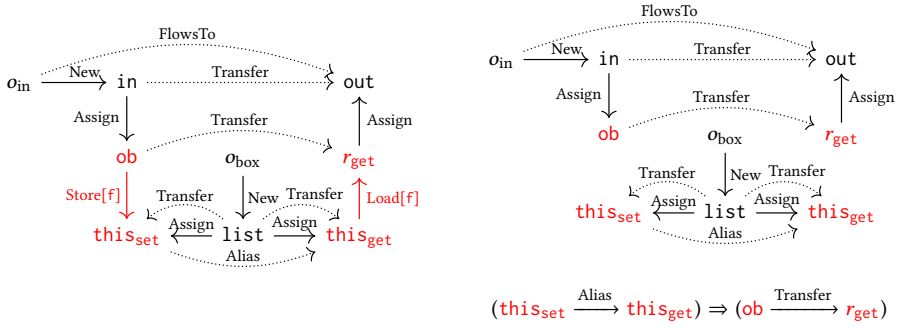


Fig. 4. The solid edges are the graph G extract for the program `test` shown in Figure 1. In addition, the dashed edges are a few of the edges in $\overline{G}$ when computing the transitive closure. We omit backward edges (i.e., with labels $\overline{A}$) for clarity. Vertices and edges corresponding to library code are highlighted in red.

The first production in Figure 3 constructs the *transfer* relation $x \xrightarrow{\text{Transfer}} y$, which says that x may be "indirectly assigned" to y. The second production constructs the "backwards" transfer relation. The third production constructs the *alias* relation $x \xrightarrow{\text{Alias}} y$, which says that x may alias y. The fourth production computes the points-to relation, i.e., $x \hookrightarrow o$ whenever $o \xrightarrow{\text{FlowsTo}} x \in \overline{G}$.

4 PATH SPECIFICATIONS

In this section, we describe our specification language.

4.1 Motivation

Suppose that our static analysis could analyze the library implementation, and that by doing so, the extracted graph G contains additional paths

$$z_1 \overset{\beta_1}{\dashrightarrow} w_1, \ \ldots, \ z_k \overset{\beta_k}{\dashrightarrow} w_k$$

extracted from the library code, where the variables $z_1, w_1, \ldots, z_k, w_k$ are parameters and return values of library functions and $\beta_1, \ldots, \beta_k \in \Sigma_{\text{pt}}^*$. Furthermore, let $A_1, \ldots, A_{k-1}$ be nonterminals that satisfy $A \overset{*}{\Rightarrow} \beta_1 A_1 \ldots \beta_{k-1} A_{k-1} \beta_k$. In this case, while computing the transitive closure $\overline{G}$, if

$$w_1 \xrightarrow{A_1} z_2, \ \ldots, \ w_{k-1} \xrightarrow{A_{k-1}} z_k \in \overline{G},$$

then our static analysis would add edge $z_1 \xrightarrow{A} w_k$ to $\overline{G}$ as well.

However, since we cannot analyze the library implementation, the paths $z_i \overset{\beta_i}{\dashrightarrow} w_i$ are missing from G (and thus from $\overline{G}$), so the static analysis will not add the edge $z_1 \xrightarrow{A} w_k$ to $\overline{G}$. Therefore, we need a specification telling the analysis to add $z_1 \xrightarrow{A} w_k$ to $\overline{G}$ if all the edges $w_i \xrightarrow{A_i} z_{i+1}$ are in $\overline{G}$.

For example, consider the library code in Figure 1. When analyzing the program `test` in the same figure with the library code available, the analysis includes the paths

$$\text{ob} \xrightarrow{\text{Store[f]}} \text{this}_{\text{set}}, \quad \text{this}_{\text{get}} \xrightarrow{\text{Load[f]}} r_{\text{get}}.$$

In this case, we have

$$\text{Transfer} \overset{*}{\Rightarrow} \text{Store[f] Transfer Load[f]},$$

Library Code	Candidate Path Specifications	Generated Test Cases	

`void set(Object ob) { f = ob; }`
`Object get() { return f; }`
	$ob \dashrightarrow this_{set} \rightarrow this_{get} \dashrightarrow r_{get}$	`boolean test() {`
		`  Object in = new Object(); // o_in`
		`  Box box = new Box(); // o_box`
		`  box.set(in);`
		`  Object out = box.get();`
		`  return in == out; }`
			✓

`void set(Object ob) { f = ob; }` `Object get() { return g; }`	$\emptyset$	$\emptyset$	✓

`void set(Object ob) { f = ob; }`
`Object clone() { return f; }`
	$ob \dashrightarrow this_{set} \rightarrow this_{clone} \dashrightarrow r_{clone}$	`boolean test() {`
		`  Object in = new Object(); // o_in`
		`  Box box = new Box(); // o_box`
		`  box.set(in);`
		`  Object out = box.clone();`
		`  return in == out; }`
			✗

`void set(Object ob) { f = ob; }`
`Object get() { return f; }`
`Box clone() {`
`  Box b = new Box(); // ~o_clone`
`  b.f = f;`
`  return b; }}`
	$ob \dashrightarrow this_{set} (\rightarrow this_{clone} \dashrightarrow r_{clone})^{*} \rightarrow this_{get} \dashrightarrow r_{get}$	`boolean test0() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Object out = box0.get();`
		`  return in == out; }`
		`boolean test1() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Object out = box1.get();`
		`  return in == out; }`
		`boolean test2() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Box box2 = box1.clone();`
		`  Object out = box2.get();`
		`  return in == out; }`
		`...`
			✓

`void set(Object ob) { f = ob; }`
`Object get() {`
`  return f;`
`  return g;`
`  return h; }`
`Box clone() {`
`  Box b = new Box(); // ~o_clone`
`  b.g = f;`
`  b.h = g;`
` return b; }}
	$ob \dashrightarrow this_{set} \rightarrow this_{get} \dashrightarrow r_{get}$
	$+\ ob \dashrightarrow this_{set} \rightarrow this_{clone} \dashrightarrow r_{clone} \rightarrow this_{get} \rightarrow r_{get}$
	$+\ ob \dashrightarrow this_{set} \rightarrow this_{clone} \dashrightarrow r_{clone} \rightarrow this_{clone} \dashrightarrow r_{clone} \rightarrow this_{get} \dashrightarrow r_{get}$
		`boolean test0() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Object out = box0.get();`
		`  return in == out; }`
		`boolean test1() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Object out = box1.get();`
		`  return in == out; }`
		`boolean test2() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Box box2 = box1.clone();`
		`  Object out = box2.get();`
		`  return in == out; }`
			✓

Fig. 5. Examples of hypothesized library implementations (left column), an equivalent set of path specifications (middle column), and the synthesized test cases to check the precision of these specifications (right column), with a check mark ✓ (indicating that the tests pass) or a cross mark ✗ (indicating that the tests fail).

so we need a specification encoding the rule

$$(\texttt{this}_{\texttt{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}}) \Rightarrow (\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}}).$$

This rule says that if the static analysis computes $\texttt{this}_{\texttt{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}} \in \overline{G}$, then it also computes $\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}} \in \overline{G}$. For example, this rule is applied in Figure 4 (right) to compute $\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}}$.

Path specifications are a language for expressing such rules. The middle column of Figure 5 shows examples of path specifications. In the first column, we show a hypothetical implementation of the library functions that has the same semantics as the corresponding path specification. In the last column, we show test cases that check the admissibility of the path specifications.

4.2 Syntax and Semantics

Let $\mathcal{V}_{\text{prog}}$ be the set of variables in the program (i.e., excluding variables in the library), and let $\mathcal{V}_{\text{path}} = \bigcup_{m \in \mathcal{M}} \{p_m, r_m\}$ be the set of *visible variables*, i.e., variables in the program or at the library interface. Then, a path specification is a sequence

$$z_1 w_1 z_2 w_2 \ldots z_k w_k \in \mathcal{V}_{\text{path}}^*,$$

where $z_i, w_i \in \mathcal{V}_{m_i}$ for library function $m_i \in \mathcal{M}$. We require that w_i and z_{i+1} are not both return values, and that w_k is a return value. For clarity, we also use the syntax

$$z_1 \dashrightarrow w_1 \rightarrow z_2 \dashrightarrow \ldots \dashrightarrow w_{k-1} \rightarrow z_k \dashrightarrow w_k. \tag{4}$$

Given path specification (4), for each $i \in [k]$, define the nonterminal A_i in the grammar C_{pt} to be

$$A_i = \begin{cases} \text{Transfer} & \text{if } w_i = r_{m_i} \text{ and } z_{i+1} = p_{m_{i+1}} \\ \text{Alias} & \text{if } w_i = p_{m_i} \text{ and } z_{i+1} = p_{m_{i+1}} \\ \overline{\text{Transfer}} & \text{if } w_i = p_{m_i} \text{ and } z_{i+1} = r_{m_{i+1}}. \end{cases}$$

Also, define the nonterminal A by

$$A = \begin{cases} \text{Transfer} & \text{if } z_1 = p_{m_1} \\ \text{Alias} & \text{if } z_1 = r_{m_1}. \end{cases}$$

Then, the path specification corresponds to adding a rule

$$\left(\bigwedge_{i=1}^{k-1} w_i \xrightarrow{A_i} z_{i+1} \in \overline{G} \right) \Rightarrow (z_1 \xrightarrow{A} w_k \in \overline{G})$$

to the static points-to analysis. The corresponding rule also adds the backwards edge $w_k \xrightarrow{\overline{A}} z_1$ to $\overline{G}$, but we omit it for clarity. We refer to the premise of this rule as the premise of the path specification, and the conclusion of this rule as the conclusion of the path specification.

4.3 Admissibility

In this section, we formalize the notion of *admissibility*, which essentially says that a set S of path specifications is Pareto optimal in terms of precision and recall, i.e., there does not exist a set S' of path specifications that is strictly preferable to S.

Let $\overline{G}_*(P)$ denote the true set of relations for a program P (i.e., relations that hold dynamically). Furthermore, given path specifications S, let $\overline{G}(P, S)$ denote the points-to edges computed using S for P, let $\overline{G}_+(P, S) = \overline{G}(P, S) \setminus \overline{G}_*(P)$ be the false positive points-to edges computed, and $\overline{G}_-(P, S) =$

$\overline{G}_*(P) \setminus \overline{G}(P,S)$ be the false negative points-to edges computed. We say S is *sound* if $\overline{G}_-(P,S) = \emptyset$ and *completely precise* if $\overline{G}_+(P,S) = \emptyset$.

Our notions of precision and recall are relative versions of the notions of complete precision and soundness, respectively. More precisely, given sets S and S' of path specifications, we say S has *higher or equal precision* than S' if for all programs P, $\overline{G}_+(P,S) \subseteq \overline{G}_+(P,S')$, i.e., S always produces fewer false positives than S'. Similarly, we say S has *higher or equal recall* than S' if for all programs P, $\overline{G}_-(P,S) \subseteq \overline{G}_-(P,S')$, i.e., S always produces fewer false negatives than S'. If for all programs P, S and S' compute the same relations, i.e., $\overline{G}(P,S) = \overline{G}(P,S')$, then we say S and S' are *equivalent*.

We say a set S of path specifications is *admissible* if, for any other set S' of path specifications, either S' does not have higher or equal precision than S or it does not have higher or equal recall than S. In other words, there is no set of path specifications S' that is strictly better than S.

4.4 Checking Admissibility

Our algorithm needs to generate test cases that check whether a candidate path specification s is admissible. We describe sufficient conditions for a passing test case to prove admissibility, i.e., if the test case passes, then we guarantee that s is admissible. However, the test case may fail even if s is admissible. This property is inevitable since executions are underapproximations; we show empirically that if s is admissible, then the synthesized test case typically passes.

Definition 4.1. Let s be a path specification. We say a program P is a *potential witness* for s if:

- The conclusion ($e \in \overline{G}$) of s does not hold statically for P with empty specifications, i.e., $e \notin \overline{G}(P,\emptyset)$.
- The premise of s holds for P, i.e., $e \in \overline{G}(P,\{s\})$.
- For every set S of path specifications, if $e \in \overline{G}(P,S)$, then $S \cup \{s\}$ is equivalent to S.

We say P is a *witness* for s if furthermore the conclusion of s is a true relation of P, i.e., $e \in \overline{G}_*(P)$.

In other words, s is the most precise path specification that can compute e for P—for any set S of path specifications that can do so, adding s to S does not affect the semantics of S. In Figure 5, the test cases shown in the last column witness the corresponding path specifications.

Intuitively, if program P is a potential witness for path specification s with premise ψ and conclusion $\phi = (e \in \overline{G})$, then s is the only path specification that can be used by the static analysis to compute relation e for P. Therefore, if P witnesses s, then s is guaranteed to be admissible. More precisely, we have the following important result:

Theorem 4.2. For any set S of path specifications, if each $s \in S$ has a witness, then S is admissible.

Proof. Let S' be a set of path specifications. We need to show that (i) if S' has higher recall than S, then S has higher precision than S', and (ii) if S' has recall equal to S, then S has higher or equal precision than S'.

First, we claim that in either case, S' is equivalent to $S' \cup S$. To this end, consider a path specification $s \in S$ with conclusion ($e \in \overline{G}$) and witness P. We claim that if path specifications S' has higher or equal recall than S, then $S' \cup \{s\}$ is equivalent to S'. Since the static analysis is monotone, we have $e \in \overline{G}(P,\{s\}) \subseteq \overline{G}(P,S)$, so since S' has higher or equal recall than S, we have $e \in \overline{G}(P,S')$. By the definition of a witness, $S' \cup \{s\}$ is equivalent to S'. Thus, by induction, $S' \cup S$ is equivalent to S'.

Note that (ii) follows immediately, since S clearly has higher or equal precision than $S' \cup S$, so S has higher or equal precision than S' as well. To show (i), it remains to show that using S' computes a false positive edge that using S does not. Since S' has higher recall than S, there exists some program P and some edge e such that $e \in \overline{G}(P,S')$ but $e \notin \overline{G}(P,S)$. Let P' be the

program "if False then P"; since our static analysis is flow-insensitive, we have $\overline{G}(P', S) = \overline{G}(P, S)$ and $\overline{G}(P', S') = \overline{G}(P, S')$, so $e \in \overline{G}(P', S')$ and $e \notin \overline{G}(P', S)$. But clearly e is a false positive for P', since P' does not exhibit any points-to edges. Thus, S' is less precise than S, as claimed. □

By Theorem 4.2, we can check whether a candidate path specification s is admissible by synthesizing a test case P that is a potential witness for s. If we execute the test case P and observe that the conclusion of s holds during the execution, then P is a witness for s, so s is admissible (though if P is not a witness of s, s may still be admissible). Finally, if S is the set of path specifications inferred by our algorithm, as long as each $s \in S$ has a witness, then S is admissible as well.

4.5 Equivalence to Library Implementations

It is not obvious that path specifications are sufficiently expressive to precisely model library code. In this section, we show that path specifications are in fact sufficiently expressive to do so in the case of Andersen's analysis (and its cloning-based context- and object-sensitive extensions). More precisely, for any implementation of the library, there exists a (possibly infinite) set of path specifications such that the points-to sets computed using path specifications are both sound and at least as precise as analyzing the library implementation. For convenience, we assume the following:

Assumption 4.3. Let $\mathcal{F}_{\text{lib}}$ be fields accessed by the library and $\mathcal{F}_{\text{prog}}$ be fields accessed by the program, and let the *shared fields* be $\mathcal{F}_{\text{share}} = \mathcal{F}_{\text{lib}} \cap \mathcal{F}_{\text{prog}}$. We assume $\mathcal{F}_{\text{share}} = \emptyset$.

We can remove this assumption by having the static analysis treat accesses to library fields in the program as calls to getter and setter library functions. With this assumption, we have:

Theorem 4.4. Let $\overline{G}(P)$ be the points-to sets computed with the library code. Then, there exists S such that $\overline{G}(P, S)$ is sound and $\overline{G}(P, S) \subseteq \overline{G}(P)$.

We give a proof in Section 7. Note that the set S of path specifications may be infinite. This infinite blowup is unavoidable since we want the ability to test the admissibility of an individual path specification. In particular, the library implementation (e.g., the one shown on the fourth row of Figure 5) may exhibit effects that require infinitely many test cases to check admissibility.

4.6 Regular Sets of Path Specifications

Since the library implementation may correspond to an infinite set of path specifications, we need a mechanism for describing such sets. In particular, since a path specification is a sequence $s \in \mathcal{V}_{\text{path}}^*$, we can think of a set S of path specifications as a formal language $S \subseteq \mathcal{V}_{\text{path}}^*$ over the alphabet $\mathcal{V}_{\text{path}}$. Then, we can express an infinite set of path specifications using standard representations such as regular expressions or context-free grammars.

We make the empirical observation that the set of path specifications corresponding to the library implementation is a regular language. There is no particular reason that this fact should be true, but it holds empirically for all the Java library functions we have examined so far. For example, consider the library implementation shown in the first column of line four of Figure 5. This specification corresonds to the set of path specifications shown as a regular expression in the middle column of the same line (tokens in the regular expression are highlighted in blue for clarity).

One challenge is how to run our static points-to analysis with an infinite set of path specifications; we describe how to do so for the case of regular sets of path specifications in Section 6.

5 SPECIFICATION INFERENCE ALGORITHM

In this section, we describe our algorithm for inferring path specifications. Our system is summarized in Figure 6, which also shows the section where each component is described in detail.

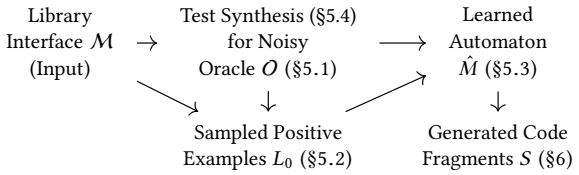


Fig. 6. An overview of our specification inference system. The section describing each component is in parentheses.

5.1 Overview

Let the *target language* $S_* \subseteq \mathcal{V}^*_{\text{path}}$ be the set of all path specifications that have a witness. By Theorem 4.2, S_* is admissible. The goal of our algorithm is to infer a set of path specifications that approximates S_* as closely as possible.

Inputs. Our algorithm is given two inputs:

- **Library interface:** The type signature of each function in the library.
- **Blackbox access:** The ability to execute library functions on a chosen input and obtain the corresponding output.

Using these two inputs, we construct the following two data structures.

Noisy oracle. Given a path specification s, the *noisy oracle* $O : \mathcal{V}^*_{\text{path}} \to \{0, 1\}$ (i) always returns 0 if s is inadmissible, and (ii) ideally returns 1 if s is admissible (but may return 0). This oracle is implemented by synthesizing a potential witness P for s—if the conclusion of the specification holds upon executing P, then P is a witness for s so the oracle returns 1; otherwise, the oracle returns 0. We describe how we synthesize a witness for s in Section 5.4.

Positive examples. Phase one of our algorithm constructs a set of positive examples: it randomly samples candidate path specifications $s \sim \mathcal{V}^*_{\text{path}}$, and then uses O to determine whether each s is admissible. More precisely, given a set $S = \{s \sim \mathcal{V}^*_{\text{path}}\}$ of random samples, it constructs positive examples $S_0 = \{s \in L \mid O(s) = 1\}$. We describe how we sample $s \sim \mathcal{V}^*_{\text{path}}$ in Section 5.2.

Language inference problem. Phase two of our algorithm inductively generalizes S_0 to a regular set of path specifications. We formulate this inductive generalization problem as follows:

Definition 5.1. The *language inference problem* is to, given the noisy oracle O and the positive examples $S_0 \subseteq S_*$, infer a language $\hat{S}$ that approximates S_* as closely as possible.

In Section 5.3, we describe our algorithm for solving this problem. Our algorithm outputs a regular language $\hat{S} = \mathcal{S}(\hat{M})$, where $\hat{M}$ is a finite state automaton. For example, given

$$S_0 = \{\texttt{ob}\ \texttt{this}_{\texttt{set}}\ \texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}}\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}\},$$

our language inference algorithm returns an automaton encoding the regular language

$$\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^*\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}.$$

5.2 Sampling Positive Examples

We sample a path specification $s \in \mathcal{V}^*_{\text{path}}$ by building it one variable at a time, starting from $s = \epsilon$. At each step, we ensure that s satisfies the constraints on path specifications, i.e., (i) z_i and w_i

are parameters or return values of the same library function, (ii) w_i and z_{i+1} are not both return values, and (iii) the last variable w_k is a return value. In particular, given current sequence s, the set $\mathcal{T}(s) \subseteq \mathcal{V}_{\text{path}} \cup \{\emptyset\}$ of choices for the next variable (where $\emptyset$ indicates to terminate and return s) is:

- If $s = z_1 w_1 z_2 \ldots z_i$, then the choices for w_i are $\mathcal{T}(s) = \{p_m, r_m\}$, where $z_i \in \{p_m, r_m\}$.
- If $s = z_1 w_1 z_2 \ldots z_i w_i$, and w_i is a parameter, then the choices for z_{i+1} are $\mathcal{T}(s) = \mathcal{V}_{\text{path}}$.
- If $s = z_1 w_1 z_2 \ldots z_i w_i$, and w_i is a return value, then the choices for z_{i+1} are

$$\mathcal{T}(s) = \{z \in \mathcal{V}_{\text{path}} \mid z \text{ is a parameter}\} \cup \{\emptyset\}.$$

At each step, our algorithm samples $x \sim \mathcal{T}(s)$, and either constructs $s' = sx$ and continues if $x \neq \emptyset$ or returns s if $x = \emptyset$. We consider two sampling strategies.

Random sampling. We uniformly randomly choose $x \sim \mathcal{T}(s)$ at every step.

Monte Carlo tree search. We can exploit the fact that certain choices $x \in \mathcal{T}(s)$ are much more likely to yield an admissible path specification than others. To do so, note that our search space is structured as a tree, where each vertex corresponds to a prefix in $\mathcal{V}_{\text{path}}^*$, the root corresponds to the prefix ϵ, edges are defined by $\mathcal{T}$, and leaves correspond to candidate path specifications.

We can sample $x \sim \mathcal{T}(s)$ using Monte Carlo tree search (MCTS) (Browne et al. 2012), a search algorithm that learns over time which choices are more likely to succeed. In particular, MCTS keeps track of a score $Q(s, x)$ for every visited $s \in \mathcal{V}_{\text{path}}^*$ and every $x \in \mathcal{T}(s)$. Then, the choices are sampled according to the distribution

$$\Pr[x \mid s] = \frac{1}{Z} e^{Q(s,x)} \quad \text{where} \quad Z = \sum_{x' \in \mathcal{T}(s)} e^{Q(s,x')}.$$

Whenever a candidate $s = x_1 \ldots x_k$ is found, the score $Q(x_1 \ldots x_i, x_{i+1})$ (for each $0 \leq i < k$) is increased if s is a positive example (i.e., $O(s) = 1$) and decreased otherwise (i.e., $O(s) = 0$):

$$Q(x_1 \ldots x_i, x_{i+1}) \leftarrow (1 - \alpha) Q(x_1 \ldots x_i, x_{i+1}) + \alpha O(s).$$

We choose the *learning rate* α to be $\alpha = 1/2$.

5.3 Language Inference Algorithm

We modify RPNI (Oncina and Garcia 1992) to leverage access to the noisy oracle. In particular, whereas RPNI takes as input a set of negative examples, we use the oracle to generate them on-the-fly. Our algorithm learns a regular language $\hat{S} = \mathcal{L}(\hat{M})$ represented by the (nondeterministic) finite state automaton (FSA) $\hat{M} = (Q, \mathcal{V}_{\text{path}}, \delta, q_{\text{init}}, Q_{\text{fin}})$, where Q is the set of states, $\delta : Q \times \mathcal{V}_{\text{path}} \to 2^Q$ is the transition function, $q_{\text{init}} \in Q$ is the start state, and $Q_{\text{fin}} \subseteq Q$ are the accept states. If there is a single accept state, we denote it by q_{fin}. We denote transitions $q \in \delta(p, \sigma)$ by $p \xrightarrow{\sigma} q$.

Our algorithm initializes $\hat{M}$ to be the FSA representing the finite language S_0. In particular, it initializes $\hat{M}$ to be the prefix tree acceptor (Oncina and Garcia 1992), which is the FSA where the underlying transition graph is the prefix tree of S_0, the start state is the root of this prefix tree, and the accept states are the leaves of this prefix tree.

Then, our algorithm iteratively considers *merging* pairs of states of $\hat{M}$. More precisely, given two states $p, q \in Q$ (without loss of generality, assume $q \neq q_{\text{init}}$), $\text{Merge}(\hat{M}, p, q)$ is the FSA obtained by (i) replacing transitions

$$(r \xrightarrow{\sigma} q) \mapsto (r \xrightarrow{\sigma} p), \qquad (q \xrightarrow{\sigma} r) \mapsto (p \xrightarrow{\sigma} r),$$

(ii) adding p to Q_{fin} if $q \in Q_{\text{fin}}$, and (iii) removing q from Q.

Our algorithm makes a single pass over all the states Q. We describe how a single step proceeds: let q be the state being processed in the current step, let Q_0 be the states that have been processed so far but not removed from Q, and let $\hat{M}$ be the current FSA. For each $p \in Q_0$, our algorithm checks whether merging q and p overgeneralizes the language, and if not, greedily performs the merge. More precisely, for each $p \in Q_i$, our algorithm constructs

$$M_{\text{diff}} = \mathsf{Merge}(\hat{M}, q_i, p) \setminus \hat{M},$$

which represents the set of strings that are added to $\mathcal{L}(\hat{M})$ if q and p are merged. Then, for each $s \in M_{\text{diff}}$ up to some maximum length N (we take $N = 8$), our algorithm queries $O(s)$. If all queries pass (i.e., $O(s) = 1$), then our algorithm greedily accepts the merge, i.e., $\hat{M} \leftarrow \mathsf{Merge}(\hat{M}, q, p)$ and continues to the next $q \in Q$. Otherwise, it considers merging q with the next $p \in Q_0$. Finally, if q is not merged with any state $p \in Q_0$, then our algorithm does not modify $\hat{M}$. Once it has completed a pass over all states in Q, our algorithm returns $\hat{M}$.

For example, suppose our language learning algorithm is given a single positive example

$$\texttt{ob}\ \texttt{this}_{\texttt{set}}\ \texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}}\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}.$$

Then, our algorithm constructs the finite state automaton

$$\hat{M}_0 = q_{\text{init}} \xrightarrow{\texttt{ob}} q_1 \xrightarrow{\texttt{this}_{\texttt{set}}} q_2 \xrightarrow{\texttt{this}_{\texttt{clone}}} q_3 \xrightarrow{r_{\texttt{clone}}} q_4 \xrightarrow{\texttt{this}_{\texttt{get}}} q_5 \xrightarrow{r_{\texttt{get}}} q_{\text{fin}}.$$

Our algorithm fails to merge q_{init}, q_1, q_2, or q_3 with any previous states. It then tries to merge q_4 with each state $\{q_{\text{init}}, q_1, q_2, q_3\}$; the first two merges fail, but merging q_4 with q_2 produces

$$\hat{M}_1 = q_{\text{init}} \xrightarrow{\texttt{ob}} q_1 \xrightarrow{\texttt{this}_{\texttt{set}}} q_2 \xrightarrow{\texttt{this}_{\texttt{get}}} q_4 \xrightarrow{r_{\texttt{get}}} q_{\text{fin}}, \quad q_2 \xrightarrow{\texttt{this}_{\texttt{clone}}} q_3 \xrightarrow{r_{\texttt{clone}}} q_2$$

Then, the specifications of length at most N in M_{diff} are

$$\begin{aligned}
&\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^0\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}\\
&\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^2\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}\\
&\ldots\\
&\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^N\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}},
\end{aligned}$$

all of which are accepted by our noisy oracle O. Therefore, our algorithm greedily accepts this merge and continues. The remaining merges fail, so our algorithm returns an FSA that equals $\hat{M}_1$.

5.4 Test Case Synthesis

We describe how we synthesize a test case that is a potential witness for a given specification

$$s = (z_1 \dashrightarrow w_1 \rightarrow \ldots \rightarrow z_k \dashrightarrow w_k).$$

We relegate details to Appendix A.

Skeleton construction. Our algorithm first constructs the *skeleton* of the test case. In particular, a witness for s must include a call to each function $m_1, \ldots, m_k$, where the variables $z_i, w_i \in \mathcal{V}_{m_i}$ are parameters or return values of m_i, since the graph G extracted from the test case must by definition contain edges connecting the w_i to z_{i+1}. For each function call $y \leftarrow m(x)$, the argument x and the left-hand side variable y are left as holes ?? to be filled in subsequent steps.

$$\mathsf{ob} \dashrightarrow \mathsf{this}_{\mathsf{set}} \rightarrow \mathsf{this}_{\mathsf{clone}} \rightarrow r_{\mathsf{clone}} \rightarrow \mathsf{this}_{\mathsf{get}} \rightarrow r_{\mathsf{get}}$$

skeleton	`??.add(??);`
	`?? = ??.clone();`
	`?? = ??.get(??);`

fill holes	`list.add(in);`
	`List listClone = list.clone();`
	`Object out = listClone.get(??);`

initialization & scheduling	`Object in = new Object();`
	`List list = new List()`
	`list.add(in);`
	`List listClone = list.clone();`
	`Object out = listClone.get(0);`
	`return in == out;`

Fig. 7. Steps in the test synthesis algorithm (right) for a candidate path specification for Box (left). Code added at each step is highlighted in blue. Scheduling is shown in the same line as initialization—it chooses the final order of the statements.

Fill holes. Second, our algorithm fills the holes in the skeleton corresponding to reference variables. In particular, for each pair of function calls $??_{y,i} \leftarrow m_i(??_{x,i})$ and $??_{y,i+1} \leftarrow m_{i+1}(??_{x,i+1})$, it fills the holes $??_{y,i}$ and $??_{x,i+1}$ depending on the edge $w_i \xrightarrow{A_i} z_{i+1}$:

- **Case** A_i = **Transfer:** In this case, w_i is a return value and z_{i+1} is a parameter. Thus, the algorithm fills $??_{y,i}$ and $??_{x,i+1}$ with the same fresh variable x.
- **Case** A_i = $\overline{\textbf{Transfer}}$**:** This case is analogous to the case A_i = Transfer.
- **Case** A_i = **Alias:** In this case, w_i and z_{i+1} are both parameters. Thus, the algorithm fills $??_{y,i}$ and $??_{x,i+1}$ with the same fresh variable x, and additionally adds to the test case an allocation statement $x \leftarrow X()$.

As we show in Proposition 5.3, the added statements ensure that P is a potential witness for s.

Initialization. Third, our algorithm initializes the remaining reference variables and primitive variables in the test case. In particular, function calls $y \leftarrow m_i(x)$ may have additional parameters that need to be filled, as may constructors $x \leftarrow X()$ added in the previous step. For Proposition 5.3 to hold, we require that remaining reference variables are filled with the value `null`.

However, this approach is likely to synthesize test cases that fail when s is admissible. Therefore, we alternatively use a heuristic where we allocate a fresh variable for each reference variable. For allocating reference variables that are passed as arguments to constructors, we have to be careful to avoid infinite recursion; for example, a constructor `Integer(Integer i);` should be avoided. Our algorithm uses a shortest-path algorithm to generate the smallest possible construct statements; see Appendix A.3 for details. With this approach, we can no longer guarantee that P is a witness, so our oracle may be susceptible to false positives. In our evaluation, we show that using this heuristic substantially improves recall with zero reduction in precision.

Primitive variables can be initialized arbitrarily, but the choice of initialization affects whether P is a witness when s is admissible. We initialize primitive variables using default values (`0` for numeric variables and `true` for boolean variables) that work well in practice.

Scheduling. Fourth, our algorithm determines the ordering of the statements in the test case. There are many possible choices of statement ordering, which affect whether P is a witness when s is admissible. There are *hard constraints* on the ordering (in particular, a variable must be defined before it is used) and *soft constraints* (in particular, statements corresponding to edges $w_i \rightarrow z_{i+1}$ for

smaller i should occur earlier in P). Our scheduling algorithm produces an ordering that satisfies the hard constraints while trying to satisfy as many soft constraints as possible. It does so using a greedy strategy, i.e., it orders the statements sequentially from first to last, choosing at each step the statement that satisfies the hard constraints and the most soft constraints; see Appendix A.4.

Guarantees. First, we establish a general condition for P to be a potential witness:

Proposition 5.2. Let s be a path specification with premise $(e_1 \in \overline{G}) \wedge \ldots \wedge (e_k \in \overline{G})$. A program P is a potential witness of s if the set of edges $\{e_1, \ldots, e_k\}$ in the premise of s exactly equals

$$\left\{ w \xrightarrow{A} z \in \overline{G}(P, \emptyset) \;\middle|\; w, z \in \mathcal{V}_{\text{lib}} \text{ and } A \in \{\text{Transfer}, \overline{\text{Transfer}}, \text{Alias}\} \right\}.$$

Proof. Let P be a potential witness for s, and suppose that the conclusion of s is $(e \in \overline{G})$. Let S be a set of path specifications that computes e for P, i.e., $e \in \overline{G}(P, S)$. We need to show that for any such S, $S \cup \{s\}$ is equivalent to S. Clearly, $S \cup \{s\}$ has higher or equal recall than S, so it suffices to show that it also has higher or equal precision than S. Consider an arbitrary program P'. Then, if s is used during the computation $\overline{G}(P, S \cup \{s\})$, then at that point, the premise of s holds for $\overline{G}$, i.e., $e_1, \ldots, e_k \in \overline{G}$. Since the graph for P is contained in the graph for P', and our static analysis is monotone, we have $e \in \overline{G}(P, S) \subseteq \overline{G}(P', S)$, i.e., e is computed without s. Thus, $\overline{G}(P', S \cup \{s\}) = \overline{G}(P', S)$, so $S \cup \{s\}$ equivalent to S as claimed. □

Then, we have the following guarantee for the test case synthesis algorithm:

Proposition 5.3. The test case P synthesized for path specification s is a potential witness for s.

Proof. (sketch) Let $s = z_1 \rightarrow w_1 \dashrightarrow \ldots \rightarrow z_k \dashrightarrow w_k$. Since the function calls are treated as no-ops by the static analysis (according to the definition of a potential witness), they do not add any edges to the extracted graph G except for assignments to and from parameters and return values. The only other edges in the graph G extracted from P are those corresponding to the allocation statements added to P in the initialization step.

First, we show that the edges in the premise of s are contained in $\overline{G}(P, \emptyset)$. For an edge $w_i \rightarrow z_{i+1}$, there are three possibilities—either $A_i = \text{Transfer}$, $A_i = \overline{\text{Transfer}}$, or $A_i = \text{Alias}$:

- **Case A_i = Transfer:** Then, w_i is a return value and z_{i+1} is a parameter. Then, the test case synthesis algorithm assigns the return value of m_i to the argument of m_{i+1}, i.e., the edges

$$w_i \xrightarrow{\text{Assign}} x \xrightarrow{\text{Assign}} z_{i+1} \in G,$$

  where G is the graph extracted from P. Therefore, we have $(w_i \xrightarrow{\text{Transfer}} z_{i+1}) \in \overline{G}(P, \emptyset)$.
- **Case A_i = $\overline{\text{Transfer}}$:** This case is analogous to the case A = Transfer.
- **Case A_i = Alias:** Then, w_i and z_{i+1} are both parameters. Then, w_i and z_{i+1} are both parameters. Then, the test case synthesis algorithm allocates a new object and passes it as a parameter to each m_i and m_{i+1}, i.e., the edges

$$o \xrightarrow{\text{New}} x \xrightarrow{\text{Assign}} w_i \in G \text{ and } o \xrightarrow{\text{New}} x \xrightarrow{\text{Assign}} z_{i+1} \in G.$$

  Therefore, we have $(w_i \xrightarrow{\text{Alias}} z_{i+1}) \in \overline{G}(P, \emptyset)$.

Second, consider all edges $w \xrightarrow{A_i} z$, where $w, z \in \mathcal{V}_{\text{lib}}$ and $A_i \in \{\text{Transfer}, \overline{\text{Transfer}}, \text{Alias}\}$, that are contained in the premise of s. By inspection, of the edges in G as described above, the only additional edges in $\overline{G}(P, \emptyset)$ of this form are:

$$\text{(initial parameter)}\ \frac{q_{\text{init}} \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\quad z = p_m,\quad w \in \{p_m, r_m\}}{w.f_r \leftarrow z \in m} \qquad \text{(initial return)}\ \frac{q_{\text{init}} \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\quad z = r_m,\quad w \in \{p_m, r_m\}}{t \leftarrow X(),\ z \leftarrow t,\ w.f_r \leftarrow t \in m}$$

$$\text{(final parameter)}\ \frac{p \xrightarrow{z} q \xrightarrow{w} q_{\text{fin}} \in \hat{M},\quad z = p_m,\quad w = r_m}{w \leftarrow z.f_p \in m} \qquad \text{(final return)}\ \frac{p \xrightarrow{z} q \xrightarrow{w} q_{\text{fin}} \in \hat{M},\quad z = r_m,\quad w = r_m}{t \leftarrow X(),\ z.f_p \leftarrow t,\ w \leftarrow t \in m}$$

$$(A_i = \text{Alias})\ \frac{p \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\quad z = p_m,\quad w = p_m}{t \leftarrow z.f_p,\ w.f_r \leftarrow t \in m} \qquad (A_i = \text{Transfer})\ \frac{p \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\quad z = p_m,\quad w = r_m}{w\text{X}(),\ t \leftarrow z.f_p,\ w.f_r \leftarrow t \in m}$$

$$(A_i = \overline{\text{Transfer}})\ \frac{p \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\quad \{z, w\} \subseteq \{p_m, r_m\}}{z \leftarrow X(),\ t \leftarrow w.f_r,\ z.f_p \leftarrow t \in m} \qquad \text{(initial final)}\ \frac{q_{\text{init}} \xrightarrow{z} q \xrightarrow{w} q_{\text{fin}} \in \hat{M},\quad \{z, w\} \subseteq \{p_m, r_m\}}{w \leftarrow z \in m}$$

Fig. 8. Rules for generating code fragment specifications from path specifications defined by a finite state automaton $\hat{M} = (Q, \mathcal{V}_{\text{path}}, \delta, q_{\text{init}}, Q_{\text{fin}})$, where for simplicity we assume $\hat{M}$ has a single accept state q_{fin}.

- The self-loops $z_i \xrightarrow{\text{Transfer}} z_i$ and $w_i \xrightarrow{\text{Transfer}} w_i$ (since there is a production Transfer $\rightarrow \epsilon$ in the points-to grammar C_{pt}).
- The backward edges $z_{i+1} \xrightarrow{\overline{A_i}} w_i$ (when $A_i \in \{\text{Transfer}, \overline{\text{Transfer}}\}$).

If these edges were added to the premise of s for P, then by Proposition 5.2, we could conclude that P is a potential witness of s. However, these edges are in $\overline{G}(P, S)$ for any program P and any specifications S, so we can add them to the premise of s without affecting its semantics. From Definition 4.1, it follows that if P is a witness for s', and s' is equivalent to s, then P is a witness for s as well. Therefore, P is a witness for s as claimed. □

6 STATIC POINTS-TO ANALYSIS WITH REGULAR SETS OF PATH SPECIFICATIONS

In this section, we describe how to run our static points-to analysis in conjunction with a possibly infinite regular set S of path specifications (assumed to be represented as an FSA, i.e., $S = \mathcal{L}(\hat{M})$). In particular, our static analysis converts S to a set $\tilde{S}$ of *code fragment specifications*, which are replacements for the library code that have the same points-to effects as encoded by S.

Given path specifications S, our static analysis constructs *equivalent* code fragment specifications $\tilde{S}$, i.e., $\overline{G}(P, S) = \overline{G}(P, \tilde{S})$. In other words, $\tilde{S}$ has the same semantics as S with respect to our static points-to analysis. One detail in our definition of equivalence is that $\overline{G}(P, \tilde{S})$ may contain additional vertices corresponding to variables and abstract objects in the code fragment specifications; we omit these extra vertices and their relations at the end of the static analysis.

6.1 Converting a Single Path Specification

For intuition, we begin by describing how to convert a single path specification

$$s = (z_1 \dashrightarrow w_1 \rightarrow \ldots \rightarrow z_k \dashrightarrow w_k)$$

into an equivalent set of code fragment specifications, where $A_i = \text{Alias}$ for each i and z_1 is a parameter. Let the code fragment specifications $\tilde{S}$ corresponding to s be:

$$\begin{aligned} m_1 &= \{w_1.f_1 \leftarrow z_1\} \\ m_2 &= \{t_2 \leftarrow z_2.f_1,\quad w_2.f_2 \leftarrow t_2\} \\ &\ldots \\ m_k &= \{w_k \leftarrow z_k.f_{k-1}\}, \end{aligned}$$

where $f_1, \ldots f_{k-1} \in \mathcal{F}$ are fresh fields and $t_2, \ldots, t_{k-1}$ are fresh variables. Then:

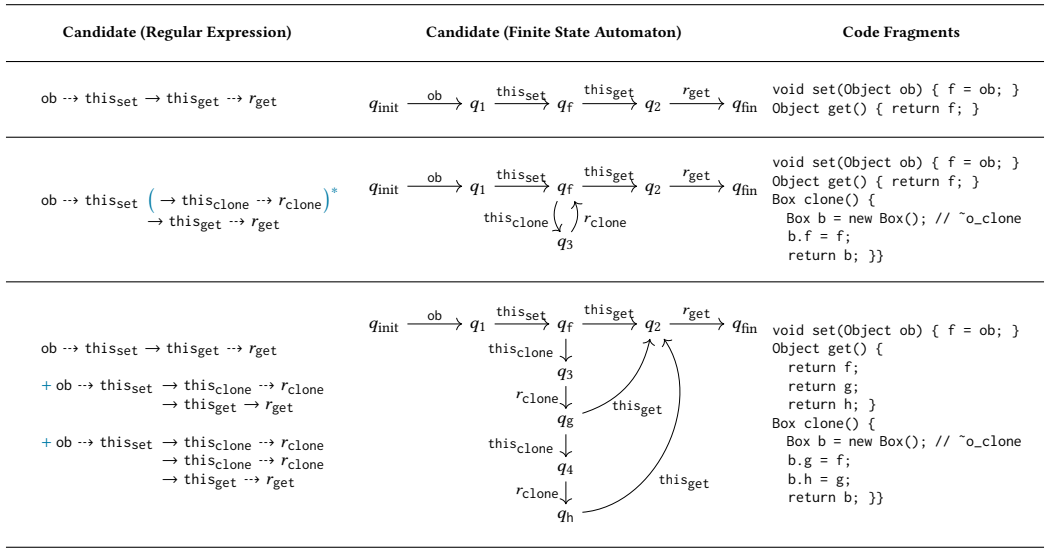


Fig. 9. Examples of candidate code fragment specifications (left column), and the equivalent path specifications as a regular expression (middle column) and as a finite state automaton (right column).

Proposition 6.1. We have $\overline{G}(P, \tilde{S}) = \overline{G}(P, \{s\}) \cup \overline{G}'(P, \tilde{S})$, where $\overline{G}'(P, \tilde{S})$ consists of the edges in $\overline{G}(P, \tilde{S})$ that refer to vertices corresponding to variables and abstract objects in $\tilde{S}$.

Proof. (sketch) First, we show that $\overline{G}(P, \{s\}) \subseteq \overline{G}(P, \tilde{S})$. Suppose that the premise of s holds, i.e., $z_i \xrightarrow{A_i} w_{i+1} \in \overline{G}$ for each i. Then, the static analysis computes $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(P, \{s\})$; we need to show that $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(P, \tilde{S})$ as well. Note that we have

$$z_1 \xrightarrow{\text{Store}[f_1]} w_1 \xrightarrow{\text{Alias}} z_2 \xrightarrow{\text{Load}[f_1]} t_2 \in \overline{G}(P, \tilde{S})$$
$$t_2 \xrightarrow{\text{Store}[f_2]} w_2 \xrightarrow{\text{Alias}} z_3 \xrightarrow{\text{Load}[f_2]} t_3 \in \overline{G}(P, \tilde{S})$$
$$\cdots$$
$$t_{k-1} \xrightarrow{\text{Store}[f_{k-1}]} w_{k-1} \xrightarrow{\text{Alias}} z_k \xrightarrow{\text{Load}[f_{k-1}]} w_k \in \overline{G}(P, \tilde{S}).$$

By induction, the static analysis computes $z_1 \xrightarrow{\text{Transfer}} t_i \in \overline{G}(P, \tilde{S})$ for each $i \in [k-1]$. Thus, the static analysis computes $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(P, \tilde{S})$, as claimed.

Next, we show the converse, i.e., that $\overline{G}(P, \tilde{S}) \subseteq \overline{G}(P, S) \cup \overline{G}'(P, \tilde{S})$. First, note that the only production with Store$[f]$ is

$$\text{Transfer} \rightarrow \text{Transfer Store}[f] \text{ Alias Load}[f].$$

Since each f_i is a fresh field, there is only one edge labeled Store$[f_i]$ and only one edge labeled Load$[f_i]$. Thus, this production can only be triggered if (i) $z_i \xrightarrow{\text{Alias}} w_i \in \overline{G}(P, \tilde{S})$, and (ii) for some vertex x, $x \xrightarrow{\text{Transfer}} t_i \in \overline{G}(P, \tilde{S})$. If triggered, the static analysis adds an edge $x \xrightarrow{\text{Transfer}} t_{i+1}$ to $\overline{G}(P, \tilde{S})$. For $i = 1$, the only vertices x satisfying the second condition are $x = z_1$ and $x = t_1$. By

induction, if $w_i \xrightarrow{\text{Alias}} z_{i+1} \in \overline{G}(P, \tilde{S})$ for each i, we have

$$z_1 \xrightarrow{\text{Transfer}} t_i \in \overline{G}(P, \tilde{S})$$
$$t_j \xrightarrow{\text{Transfer}} t_i \in \overline{G}(P, \tilde{S})$$

for each $j \leq i$. None of the t_i are part of an Assign edge except t_1 and t_k; for the latter, the production Transfer $\rightarrow$ Transfer Assign triggers and we get $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(P, \tilde{S})$. This edge is the only one in $\overline{G}(P, \tilde{S})$ that does not refer to vertices extracted from the code fragments, so the claim follows. □

6.2 Converting a Regular Set of Path Specifications

Our construction generalizes straightforwardly to constructing code fragment specifications from $\hat{M}$. For each state $q \in Q$, we introduce a fresh field $f_q \in \mathcal{F}$. Intuitively, transitions into q correspond to stores into f_q, and transitions coming out of q correspond to loads into f_q. In particular, we include statements in m according to the rules in Figure 8.

The following guarantee follows similarly to the proof of Proposition 6.1:

Proposition 6.2. We have $\overline{G}(P, \tilde{S}) = \overline{G}(P, S) \cup \overline{G}'(P, \tilde{S})$, where $\overline{G}'(P, \tilde{S})$ is defined as before.

In Figure 9, we show examples of path specifications (first column), the corresponding FSA (middle column), and the generated code fragment specifications. For example, in the second line, the transitions

$$q_{\text{init}} \xrightarrow{\text{ob}} q_1 \xrightarrow{\texttt{this}_{\texttt{set}}} q_2 \xrightarrow{\texttt{this}_{\texttt{get}}} q_3 \xrightarrow{r_{\texttt{get}}} q_{\text{fin}}$$

generate the specifications for `set` (the first two transitions, with field `f` $= f_{q_2}$) and `get` (the last two transitions), and the self-loop

$$q_2 \xrightarrow{\texttt{this}_{\texttt{clone}}} q_6 \xrightarrow{r_{\texttt{clone}}} q_2$$

generates the specification for `clone`.

7 PROOF OF EQUIVALENCE THEOREM

We prove Theorem 4.4, relegating the proof of technical lemmas to Appendix B.

7.1 Converting the Library Implementation to Path Specifications

First, we describe how to convert the library implementation into a set S of transfer and proxy object specifications. A specification of the form

$$z_1 \dashrightarrow w_1 \rightarrow \ldots \rightarrow z_k \dashrightarrow w_k.$$

is included in S if there exist paths

$$z_1 \overset{\beta_1}{\dashrightarrow} w_1, \quad \ldots, \quad z_k \overset{\beta_k}{\dashrightarrow} w_k$$

such that $A \overset{*}{\Rightarrow} \beta_1 \tilde{\alpha}_1 \ldots \tilde{\alpha}_{k-1} \beta_k$ in C_{pt}, where

$$A = \begin{cases} \text{Transfer} & \text{if } z_1 = p_{m_1} \\ \text{Alias} & \text{if } z_1 = r_{m_1} \end{cases}$$

and

$$\tilde{\alpha}_i = \begin{cases} \text{Assign} & \text{if } w_i = p_{m_i} \text{ and } z_{i+1} = r_{m_{i+1}} \\ \overline{\text{Assign}} & \text{if } w_i = r_{m_i} \text{ and } z_{i+1} = p_{m_{i+1}} \\ \overline{\text{New}}\ \text{New} & \text{if } w_i = p_{m_i} \text{ and } z_{i+1} = p_{m_{i+1}}. \end{cases}$$

Then, we prove that the conclusion of Theorem 4.4 holds for S constructed with this algorithm.

7.2 Proof Overview

Let $\overline{G}$ denote the points-to sets computed by running the static analysis with the library implementation available, and $\overline{G}(S)$ denote the points-to sets computed by running the static analysis with the path specifications S. We have to prove that $\overline{G} = \overline{G}(S)$; the direction $\overline{G}(S) \subseteq \overline{G}$ follows easily, since a path specification s is included in S exactly when the library implementation would imply the same logical formula as the semantics of s.

The challenging direction is to show that S is sound, i.e., $\overline{G} \subseteq \overline{G}(S)$. For simplicity, we focus on points-to edges $o \xrightarrow{\text{FlowsTo}} x$; the alias and transfer relations follow similarly. Suppose that $o \xrightarrow{\text{FlowsTo}} y \in \overline{G}(S)$; then, there must exist a path $o \xrightarrow{\text{New}} x \overset{\alpha}{\dashrightarrow} y$, where Transfer $\overset{*}{\Rightarrow} \alpha$. This path passes into and out of library functions, leading to a decomposition

$$x \overset{\alpha_0}{\dashrightarrow} z_1 \overset{\beta_1}{\dashrightarrow} w_1 \overset{\alpha_1}{\dashrightarrow} \ldots \overset{\beta_k}{\dashrightarrow} w_k \overset{\alpha_k}{\dashrightarrow} y, \tag{5}$$

where $\alpha = \alpha_0\beta_1\alpha_1 \ldots \beta_k\alpha_k$. This decomposition suggests that the following path specification may be applied to derive $x \xrightarrow{\text{Transfer}} y$:

$$z_1 \dashrightarrow w_1 \rightarrow \ldots \rightarrow z_k \dashrightarrow w_k. \tag{6}$$

At a high level, our proof has two parts. First, we prove the case where the segments of α in the program do not contain field accesses, i.e., $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, where

$$\begin{aligned} \Sigma_{\text{free}} &= \{\text{Assign}, \overline{\text{Assign}}, \text{New}, \overline{\text{New}}\} \\ \Sigma_{\text{prog}} &= \{\text{Store}[f], \text{Load}[f], \overline{\text{Store}[f]}, \overline{\text{Load}[f]} \mid f \in \mathcal{F}_{\text{prog}}\} \\ \Sigma_{\text{lib}} &= \{\text{Store}[f], \text{Load}[f], \overline{\text{Store}[f]}, \overline{\text{Load}[f]} \mid f \in \mathcal{F}_{\text{lib}}\}. \end{aligned}$$

Second, we show how "nesting" of fields allows us to reduce the general case to the case $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}} \cup \Sigma_{\text{prog}})^*$. In particular, by Assumption 4.3, the library field accesses and program field accesses do not match one another. As previously discussed, this assumption can be enforced by a purely syntactic program transformation where accesses to library fields in the program are converted into calls to getter and setter functions.

Consider a path of the form (5) such that $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$. We need to show that in this case, we derive the edge $x \xrightarrow{\text{Transfer}} y \in \overline{G}(S)$, where S is constructed as in Section 7.1. Our proof of this claim relies on two results. The first result says that for such a path, the conclusion of (6) holds when each w_i is connected to z_{i+1} by α_i:

Proposition 7.1. For any path of the form (5) such that $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$ we have (i) the case $w_i = r_i$ and $z_{i+1} = r_{i+1}$ cannot happen, and (ii) Transfer $\overset{*}{\Rightarrow} \beta_1\alpha_1\beta_2 \ldots \alpha_{k-1}\beta_k$.

As a consequence of this result, we know that the path specification (6) is contained in S. The second result says that the premise of (6) holds for our case:

Proposition 7.2. For any path of the form (6) such that $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, we have

$$A_i \overset{*}{\Rightarrow} \alpha_i \quad (\forall i \in [k-1])$$
$$A_i \overset{*}{\Rightarrow} \alpha_i \quad (\forall i \in \{0, k\}).$$

Therefore, we can conclude that when running the static analysis using path specifications, we derive the conclusion of the path specification (6), i.e., $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(S)$. In summary, we have the following result:

Theorem 7.3. Theorem 4.4 holds for any $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$.

Proof. Consider an edge $x \xrightarrow{\text{Transfer}} y \in \overline{G}$ derived by the static analysis using the library implementation. We claim that this edge is derived by the static analysis when using path specifications, i.e., $x \xrightarrow{\text{Transfer}} y \in \overline{G}(S)$. By Proposition 7.1, we conclude that (6) is in S. Furthermore, by Proposition 7.2, the premise of (6) holds, so the static analysis derives its conclusion, i.e., $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(S)$. Therefore, we have

$$x \xrightarrow{\text{Transfer}} z_1 \xrightarrow{\text{Transfer}} w_k \xrightarrow{\text{Transfer}} y \in \overline{G}(S),$$

so the static analysis derives $x \xrightarrow{\text{Transfer}} y \in \overline{G}(S)$, as claimed.

Now, we know that any points-to edge $o \xrightarrow{\text{FlowsTo}} y \in \overline{G}$ has the form $o \xrightarrow{\text{New}} x \xrightarrow{\text{Transfer}} y$. Since we have shown that $x \xrightarrow{\text{Transfer}} y \in \overline{G}(S)$, the static analysis also derives $o \xrightarrow{\text{FlowsTo}} y \in \overline{G}(S)$, so the result follows. □

In the remainder of the section, we introduce the technical machinery that enables us to reason about "equivalence" of the semantics of different sequences of statements. Then, we describe how we prove Propositions 7.1 & 7.2. Finally, we reduce Theorem 4.4 to Theorem 7.3.

7.3 Equivalent Semantics

Proving Propositions 7.1 & 7.2 requires reasoning about the *equivalence* of the semantics of sequences of statements in P. For example, to prove Proposition 7.1, we show that each α_i is "equivalent" to $\tilde{\alpha}_i$. Intuitively, for $\tilde{\alpha}_i = \text{Assign}$, we show that the sequence of statements represented by α_i exhibits the same semantics as a single assignment. For example, $y \leftarrow x,\ z \leftarrow y$ has the same points-to effects as $z \leftarrow x$ (assuming y is temporary). We leverage the correspondence established by formulating points-to analysis as context-free language reachability:

$$\text{sequence of statements} \ = \ \text{sequence } \alpha \in \Sigma^*.$$

For example, the first sequence of statements above corresponds to (Assign Assign), and the second to Assign.

Using this correspondence, we can reduce reasoning about sequences of statements with equivalent semantics to studying equivalence classes of strings $\alpha \in \Sigma^*$:

$$\text{equivalent sequences of statements} \quad = \quad \text{equivalence classes } [\alpha] \subseteq \Sigma^* \ .$$

In particular, $\alpha, \beta \in \Sigma^*$ are *equivalent* if

$$\gamma\alpha\delta \in \mathcal{L}(C_{\text{pt}}) \Leftrightarrow \gamma\beta\delta \in \mathcal{L}(C_{\text{pt}}) \quad (\forall \gamma, \delta \in \Sigma^*). \tag{7}$$

In other words, α can be used interchangeably with β in any string without affecting whether the string is contained in $\mathcal{L}(C_{\text{pt}})$. We use $[\alpha] = \{\beta \in \Sigma^* \mid \alpha \sim \beta\}$ to denote the equivalence class of $\alpha \in \Sigma^*$. Then, $[\alpha] = [\beta]$ if for any two paths

$$o \xrightarrow{\gamma} v \xrightarrow{\alpha} w \xrightarrow{\delta} x, \qquad o \xrightarrow{\gamma} v \xrightarrow{\beta} w \xrightarrow{\delta} x,$$

the first results in $x \hookrightarrow o$ if and only if the second does. For example, [Assign Assign] = [Assign].

Then, equivalence is compatible with sequencing:

Lemma 7.4. If $[\alpha] = [\alpha']$ and $[\beta] = [\beta']$, then $[\alpha\beta] = [\alpha'\beta']$.

Proof. By definition, $\gamma\alpha\beta\delta \Leftrightarrow \gamma\alpha'\beta\delta \Leftrightarrow \gamma\alpha'\beta'\delta$. □

In particular, Lemma 7.4 shows that sequencing is well-defined for equivalence classes:

$$[\alpha]\,[\beta] = [\alpha\beta], \tag{8}$$

since different choices $\alpha' \in [\alpha]$ and $\beta' \in [\beta]$ yield the same equivalence class, i.e., $[\alpha\beta] = [\alpha'\beta']$. Abstractly, Σ^* is a semigroup, with sequencing as the semigroup operation; then, Lemma 7.4 shows the equivalence relation is compatible with the semigroup operation, so the quotient $\Sigma/\sim$ is a semigroup with semigroup operation (8).

For convenience, we let ϕ denote an element of the equivalence class of strings such that

$$\text{for all } \gamma, \delta \in \Sigma^*, \quad \gamma\phi\delta \notin \mathcal{L}(C_{\text{pt}}). \tag{9}$$

In other words, $[\phi]$ describes sequences of statements that can never be completed to a valid flows-to path.

7.4 Proofs of Propositions 7.1 & 7.2

Now, we describe how to prove that under the conditions of Proposition 7.1, $[\alpha_i] = [\tilde{\alpha}_i]$, which suffices to prove the proposition. We focus on the case $\tilde{\alpha}_i$ = Assign; the other cases are similar. We need the following technical lemma (we give a proof in Appendix B.1):

Lemma 7.5. For any $\alpha \in \Sigma^*_{\text{free}}$, we have

$$[\text{Assign}]\,[\alpha]\,[\text{Assign}] \in \{[\text{Assign}],\ [\phi]\}.$$

With this lemma, since $\tilde{\alpha}_i$ = Assign, $w_{m_i} = r_{m_i}$ and $z_{m_{i+1}} = p_{m_i}$, so the path $w_{m_i} \xrightarrow{\alpha_i} z_{m_{i+1}}$ has form

$$w_{m_i} = r_{m_i} \xrightarrow{\text{Assign}} y_i \xrightarrow{\alpha'_i} x_{i+1} \xrightarrow{\text{Assign}} p_{m_{i+1}} = z_{m_{i+1}},$$

where $\alpha_i = \text{Assign}\ \alpha'_i\ \text{Assign}$. By Lemma 7.5,

$$[\alpha_i] = [\text{Assign}]\,[\alpha'_i]\,[\text{Assign}] \in \{[\text{Assign}],\ [\phi]\}.$$

Since (New α) $\in \mathcal{L}(C_{\text{pt}})$, we cannot have $[\alpha_i] = [\phi]$, so

$$[\alpha_i] = [\text{Assign}] = [\tilde{\alpha}_i],$$

as claimed. We have also proven the claim in Proposition 7.2 that $A_i \overset{*}{\Rightarrow} \alpha_i$ (with A_i = Transfer) also follows. The other claims in Propositions 7.1 & 7.2 follow similarly. □

7.5 Reduction of Theorem 4.4 to Theorem 7.3

To handle field accesses, we use the fact that pairs of terminals $(\text{Store}[f], \text{Load}[f])$ and $(\overline{\text{Load}[f]}, \overline{\text{Store}[f]})$ in strings $\alpha \in \mathcal{L}(C_{\text{pt}})$ are matching. Therefore, can identify an inner-most nested pair (σ, τ) such that the string β between σ and τ contains no field accesses, i.e., $\beta \in \Sigma_{\text{free}}$. Furthermore, by Assumption 4.3, library field accesses and program field accesses do not match one another. In particular, the set of matching program field accesses is

$$\Delta_{\text{prog}} = \bigcup_{f \in \mathcal{F}_{\text{prog}}} \{(\text{Store}[f], \text{Load}[f]),\ (\overline{\text{Load}[f]}, \overline{\text{Store}[f]})\}.$$

Lemma 7.6. For any $\alpha \in \mathcal{L}(C_{\text{pt}})$, either $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, or there exists a pair of terminals $(\sigma, \tau) \in \Delta_{\text{prog}}$ such that $\alpha = \gamma\sigma\beta\tau\delta$, where $\gamma, \delta \in \Sigma^*$ and $\beta \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$.

The next step is to characterize $[\sigma\beta\tau]$:

Lemma 7.7. For any $(\sigma, \tau) \in \Delta_{\text{prog}}$ and $\beta \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$,

$$[\sigma]\ [\beta]\ [\tau] \in \{[\text{Assign}],\ [\phi]\}.$$

Finally, β must be an aliasing relation:

Lemma 7.8. For any $\beta \in \Sigma^*$,

$$[\text{Store}[f]]\ [\beta]\ [\text{Load}[f]] = [\text{Assign}] \Rightarrow [\beta] = [\overline{\text{New}}\ \text{New}]$$
$$[\overline{\text{Load}[f]}]\ [\beta]\ [\overline{\text{Store}[f]}] = [\text{Assign}] \Rightarrow [\beta] = [\overline{\text{New}}\ \text{New}].$$

Now, if $\alpha \in \Sigma^*_{\text{free}}$, we are done. Otherwise, putting the three lemmas together, we perform the following procedure:

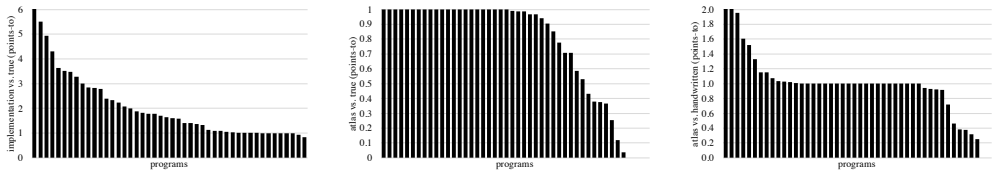
(1) By Lemma 7.6, we can write $\alpha = \gamma\sigma\beta\tau\delta$, where $(\sigma, \tau) \in \Delta_{\text{prog}}$ and $\beta \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, such that
$$y \overset{\gamma}{\dashrightarrow} v \overset{\sigma}{\longrightarrow} w \overset{\beta}{\dashrightarrow} t \overset{\tau}{\longrightarrow} u \overset{\delta}{\dashrightarrow} x.$$
(2) By Lemma 7.7, $[\sigma]\ [\beta]\ [\tau] = [\text{Assign}]$.
(3) By Lemma 7.8, $[\beta] = [\overline{\text{New}}\ \text{New}]$.
(4) By Theorem 4.4, we have $w \xrightarrow{\text{Alias}} t \in \overline{G}(\tilde{S})$; therefore, $v \overset{\text{Transfer}}{\dashrightarrow} u \in \overline{G}(\tilde{S})$ as well.
(5) Recursively apply the procedure to $\alpha' = \gamma\ \text{Assign}\ \delta$.

This procedure must terminate, since α has finitely many pairs of store and load statements. Theorem 4.4 follows. □

8 IMPLEMENTATION

We have implemented our specification inference algorithm as Atlas. We use Atlas to infer specifications for our static analysis tool (a variant of Chord (Naik et al. 2006) modified to use Soot (Vallée-Rai et al. 1999) as a backend) for Java and Android programs, which runs a 1-object-sensitive points-to analysis. Our tool omits analyzing the Android framework and the Java standard library, and instead analyzes user-provided code fragment specifications. Over two years, we have handwritten several hundred code fragment specifications, including many written specifically for our benchmark of programs.

In our evaluation, we focus on specifications for the Java 1.7 Collections API, in particular, for 31 classes that implement the `Collection` and `Map` interfaces. We focus on the Java Collections API since the functions it contains exhibit a variety of interesting points-to effects, which makes them a challenging target for inferring specifications. Indeed, the most complex points-to specifications



(a) (b) (c)

Fig. 10. The ratio of nontrivial program points-to edges discovered using (a) ground truth specifications versus the Collections API implementation, (b) Atlas versus ground truth specifications, and (c) Atlas versus existing specifications. The ratios are sorted from highest to lowest for the 46 benchmark programs with nontrivial points-to edges. In (a) and (c), some values exceeded the graph scale.

we have written by hand for the entire Android framework are all for functions in this API. We can easily use our approach to infer specifications for the entire Android framework, but limit ourselves to these APIs to focus our evaluation—in particular, we provide ground truth specifications for a large fraction of the Java Collections API.

In total, there are four sets of code fragments for the Java Collections API that our tool can use:

- Specifications inferred by Atlas for the Colletions API.
- The 58 existing, handwritten specifications for the Collections API added to our system over the past two years (many written specifically for our benchmark).
- Ground truth specifications we wrote by hand for the 12 classes in the Collections API that are most frequently used by our benchmark—98.5% of calls to the Collections API target a function in one of these 12 classes.
- The class files comprising the actual implementation of the Collections API (developed by Oracle).

9 EVALUATION

First, we evaluate the precision and recall of Atlas compared to both ground truth specifications and existing specifications. Second, we compare the points-to edges computed by our static points-to analysis using different code fragment specifications on a benchmark of 78 Android programs.

9.1 Specification Inference

We sampled a total of four million candidate path specifications (two million using each random sampling and MCTS), and run Atlas using the positive examples.

Positive examples: random sampling vs. MCTS.. We sampled two million candidate path specifications using each sampling algorithm. Random sampling found 3,124 positive examples, whereas MCTS found 10,153. We aggregated all examples for a total of 11,613 positive examples.

Object initialization: null vs. instantiation. Each of the 11,613 positive examples passed the test case constructed using instantiation, but only 7,721 passed when using null initialization, i.e., instantiation finds 50% more specifications.

Inferred specifications. We inferred code fragment specifications for 733 functions; 591 included a non-proxy-object specification and 330 included a proxy-object specification.

Precision and recall. We examine the top 50 most frequently called functions in our benchmark (in total, accounting for 95% of the function calls). We count a specification as admissible if it is

identical to the specification we would have written. For specifications with multiple statements, we count each statement fractionally. The recall of our algorithm was 97% (i.e., we inferred the admissible specification for 97% of the 50 functions) and the precision was 100% (i.e., each specification was as precise as the true specification).

Handwritten specifications. We inferred 92% of the 58 handwritten specifications; in addition, we infer an order of magnitude more new specifications (733 versus 58).

Discussion. Each of the 5 false negatives we examined was due to a false negative in our test case synthesis. For example, the function `subList(int,int)` in the `List` class requires a call of the form `subList(0, 1)` to retrieve the first object in the list. Similarly, the function `set(int,Object)` in the `List` class requires an object to already be in the list or it raises an index out-of-bounds exception.

9.2 Points-To Analysis

We evaluate different code fragment specifications using our 1-object-sensitive points-to analysis; this analysis is particularly sensitive to missing code fragment specifications, since many points-to edges depend on multiple specifications, and will not be computed if any of these specifications are missing. Furthermore, oftentimes a given program makes significant use of a small number of specifications, so a single missing specification can have a large effect. We show that nevertheless, Atlas can recover a large fraction of points-to edges. Overall, we make the following comparisons:

- We demonstrate the effectiveness of using specifications by showing that using ground truth specifications significantly decreases false positives compared to analyzing the actual implementation.
- We evaluate Atlas by comparing the specifications it infers to our ground truth specifications.
- We show that Atlas improves upon our existing, handwritten specifications, even though many of these specifications were written specifically for our benchmark.

To compare two sets of code fragments S and S', we replace the handwritten specifications for the Collections API with each S and S' and run our points-to analysis. We use $\Pi(S) \subseteq \mathcal{V} \times \mathcal{O}$ to denote the points-to edges computed using code fragments S, restricting to points-to edges in the program, i.e., $x \hookrightarrow o \in \Pi(S)$ only if x and o are in the program. Additionally, many points-to edges can be discovered without specifications—we disregard such *trivial* points-to edges $\Pi(\emptyset)$, where $\emptyset$ denotes the code fragments where each function is a no-op. Then, we report the ratio of number of nontrivial points-to edges discovered using S to the number discovered using S', i.e.,

$$R(S, S') = \frac{|\Pi(S) \setminus \Pi(\emptyset)|}{|\Pi(S') \setminus \Pi(\emptyset)|}.$$

We omit the 46 programs for which there are no nontrivial points-to edges, i.e., $\Pi(S) = \Pi(S') = \Pi(\emptyset)$. Finally, we focus on points-to edges since results for alias edges $x \xrightarrow{\text{Alias}} y$ (where both x and y are in the program) are very similar.

Benefit of specifications. To show the benefit of using specifications, we study the ratio $R(S_{\text{impl-12}}, S_{\text{true-12}})$ of analyzing the library implementation S_{impl} to analyzing the ground truth specifications S_{true}, both restricted to the 12 most frequently used classes. This ratio measures the number of false positives due to analyzing the implementation instead of using ground truth specifications, since every points-to edge computed using the implementation but not the ground truth specifications is a false positive. Figure 10 (a) shows this ratio $R(S_{\text{impl-12}}, S_{\text{true-12}})$. For a third of programs, the false positive rate is more than 100% (i.e., when $R \geq 2$), and for four programs,

the false positive rate was more than 300% (i.e., $R \geq 4$). The average false positive rate was 115.2%, and the median was 62.1%. Furthermore, for two programs, there were actually false negatives (i.e., $R < 1$) due to unanalyzable calls to native code.

Finally, running time decreased by an average of 7.5%, and by 12.4% when restricted to analyses that ran for more than five minutes, even though we only analyzed 12 classes in the library implementation. In our experience, our points-to analysis is substantially more scalable than analyzing the Java standard library implementation.

Comparison to ground truth specifications. To show the quality of the specifications inferred by Atlas, we study the ratio $R(S_{\text{atlas-12}}, S_{\text{true-12}})$ of using specifications inferred by Atlas to using ground truth specifications, both restricted to the 12 most frequently used classes. We found that using Atlas does not compute a single false positive points-to edge compared to using ground truth specifications, i.e., the precision of Atlas is 100%. Then, the ratio $R(S_{\text{atlas-12}}, S_{\text{true}-12})$ measures the number of false negative points-to edges when using Atlas compared to ground truth. Figure 10 (b) shows $R(S_{\text{atlas-12}}, S_{\text{true-12}})$. This number is almost one for more than half the programs, i.e., for almost half the programs, there are no false negatives. The median recall is 99.0%, and the average recall is 75.8%.

Improving upon handwritten specifications. To show how Atlas can improve upon our existing, handwritten specifications, we study the ratio $R(S_{\text{atlas}}, S_{\text{hand}})$ of using specifications inferred by Atlas to using the 58 existing specifications (on all 31 classes in the Collections API). This ratio compares the recall of Atlas to that of our existing specifications—a higher ratio says that Atlas has better recall, and a lower ratio says that our existing specifications have better recall. Figure 10 (c) shows $R(S_{\text{atlas}}, S_{\text{hand}})$. Atlas finds a number of new points-to edges compared to the existing, handwritten specifications, despite the fact that many of the existing specifications were written specifically for this benchmark. On average, Atlas discovers 20.1% new points-to edges. The median is 100%, i.e., Atlas find the same number of points-to edges as our existing system.

Discussion. Our results show how the specifications inferred by Atlas substantially improve recall compared to handwritten specifications. Oftentimes code is simply unavailable for analysis, e.g., due to native code, dynamically loaded code, or significant use of reflection. For such code, specifications are the only practical solution for precise and scalable static analysis. However, specifications are expensive and error prone to write—writing ground truth specifications for just 12 classes took one student more than a week of time, and bugs were discovered in the specifications during the course of our evaluation.

Atlas is an automatic approach to generating specifications, and produces higher quality specifications compared to writing them by hand. Production systems often already require handwritten specifications to handle missing or hard-to-analyze code (Facebook 2017), but typically only provide specifications for the most frequently used functions. Tools like Atlas that infer specifications for missing code are crucial for improving the usability of static analysis.

10 RELATED WORK

Inferring specifications for missing code. Techniques have been proposed for mining specifications for missing code from executions, e.g., taint specifications (i.e., whether taint flows from the argument to the return value) (Clapp et al. 2015), functional specifications of library functions (Heule et al. 2015), specifications for x86 instructions (Heule et al. 2016), and specifications for callback control flow (Jeon et al. 2016). In contrast, points-to specifications are more complex properties that span multiple functions, and our goal is to infer summaries of the points-to effects of the implementation rather than the actual operations performed.

Static approaches can infer specifications by interacting with a human (Albarghouthi et al. 2016; Bastani et al. 2015b; Zhu et al. 2013). These approaches have soundness guarantees, but rely on human effort. Finally, (Ali and Lhoták 2013) infers callgraph specifications for library code, but these may be imprecise.

Inferring program properties. There has been work on inferring specifications encoding desired properties of programs (rather than encoding behaviors of missing code). There has been work inferring program invariants such as loop invariantes from executions (Nimmer and Ernst 2002), including approaches using inductive inference (Sharma and Aiken 2014; Sharma et al. 2012, 2013). Approaches have also been developed for inferring other program properties, both using dynamic analysis (Bastani et al. 2015a, 2017; Beckman and Nori 2011; Kremenek et al. 2006; Livshits et al. 2009; Ramanathan et al. 2007; Shoham et al. 2008) and using static analysis (Ammons et al. 2002; Yang et al. 2006).

Static points-to analysis. There is a large literature on static points-to analysis (Andersen 1994; Fähndrich et al. 1998; Milanova et al. 2002; Shivers 1991; Wilson and Lam 1995), including formulations based on set-constraints and context-free language reachability (Kodumal and Aiken 2004, 2005; Reps 1998; Sridharan et al. 2005). Recent work has focused on improving context-sensitivity (Liang and Naik 2011; Smaragdakis et al. 2014; Sridharan and Bodík 2006; Whaley and Lam 2004; Zhang et al. 2014). Using specifications in conjunction with these analyses can improve precision, scalability, and even soundness.

One alternative is to use demand driven static analyses to avoid analyzing the entire library code (Sridharan et al. 2005); however, these approaches are not designed to work with missing code, and furthermore do not provide much benefit for demanding clients that require analyzing a substantial fraction of the library code.

11 CONCLUSION

Specifications summarizing the points-to effects of library code can be used to increase precision, recall, and scalability of running a static points-to analysis on any client code. By automatically inferring such specifications, Atlas fully automatically automatically achieves all of these benefits without the typical time-consuming and error-prone process of writing specifications. We believe that Atlas is an important step towards improving the usability of static analysis in practice.

REFERENCES

Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 789–801.

Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*. Springer, 378–400.

Rajeev Alur, Pavol Černỳ, Parthasarathy Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices* 40, 1 (2005), 98–109.

Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. University of Cophenhagen.

Osbert Bastani, Saswat Anand, and Alex Aiken. 2015a. Interactively verifying absence of explicit information flows in Android apps. *ACM SIGPLAN Notices* 50, 10 (2015), 299–315.

Osbert Bastani, Saswat Anand, and Alex Aiken. 2015b. Specification inference using context-free language reachability. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 553–566.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 95–110.

Nels E Beckman and Aditya V Nori. 2011. Probabilistic, modular and scalable inference of typestate specifications. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 211–221.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43.

Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: mining explicit information flow specifications from concrete executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 129–140.

Facebook. 2017. Adding models. (2017). http://fbinfer.com/docs/adding-models.html

Manuel Fähndrich, Jeffrey S Foster, Zhendong Su, and Alexander Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices* 33, 5 (1998), 85–96.

Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified synthesis: automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 237–250.

Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 710–720.

Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S Foster, and Armando Solar-Lezama. 2016. Synthesizing framework models for symbolic execution. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 156–167.

John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. *ACM Sigplan Notices* 39, 6 (2004), 207–218.

John Kodumal and Alexander Aiken. 2005. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, Vol. 5. Springer, 218–234.

Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 161–176.

Percy Liang and Mayur Naik. 2011. Scaling abstraction refinement via pruning. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 590–601.

Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. *ACM Sigplan Notices* 44, 6 (2009), 75–86.

David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science* 248, 1 (2000), 29–98.

Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ACM SIGSOFT Software Engineering Notes*, Vol. 27. ACM, 1–11.

Mayur Naik, Alex Aiken, and John Whaley. 2006. *Effective static race detection for Java*. Vol. 41. ACM.

Jeremy W Nimmer and Michael D Ernst. 2002. Automatic generation of program specifications. *ACM SIGSOFT Software Engineering Notes* 27, 4 (2002), 229–239.

Jose Oncina and Pedro Garcia. 1992. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition* 5, 99-108 (1992), 15–20.

Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 123–134.

Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* 40, 11 (1998), 701–726.

Rahul Sharma and Alex Aiken. 2014. From invariant checking to invariant inference using randomized search. In *International Conference on Computer Aided Verification*. Springer, 88–105.

Rahul Sharma, Aditya V Nori, and Alex Aiken. 2012. Interpolants as classifiers. In *International Conference on Computer Aided Verification*. Springer, 71–87.

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 391–406.

Olin Shivers. 1991. *Control-flow analysis of higher-order languages*. Ph.D. Dissertation. Citeseer.

Sharon Shoham, Eran Yahav, Stephen J Fink, and Marco Pistoia. 2008. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering* 34, 5 (2008), 651–666.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 485–495.

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 387–400.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 59–76.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 13.

John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 131–144.

Robert P Wilson and Monica S Lam. 1995. *Efficient context-sensitive pointer analysis for C programs*. Vol. 30. ACM.

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*. ACM, 282–291.

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. *ACM SIGPLAN Notices* 49, 6 (2014), 239–248.

Haiyan Zhu, Thomas Dillig, and Isil Dillig. 2013. Automated inference of library specifications for source-sink property verification. In *Asian Symposium on Programming Languages and Systems*. Springer, 290–306.

A TEST CASE SYNTHESIS ALGORITHM

In this section, we describe our algorithm for synthesizing a test case to check correctness of a candidate path specification. For example, in Figure 11, the synthesized test case contains exactly the external edges in the candidate's premise:

$$\texttt{this}_{\texttt{add}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{clone}}, \quad r_{\texttt{clone}} \xrightarrow{\text{Transfer}} \texttt{this}_{\texttt{get}}.$$

Upon executing this test case, the candidate's conclusion

$$\texttt{in} \xrightarrow{\text{Transfer}} \texttt{out}$$

holds dynamically. Therefore, this test case witnesses the correctness of the given candidate.

Our algorithm first constructs a *skeleton* containing a call to each function in the specification. Then, it (i) fills in *holes* with variable names, (ii) initializes variables, and (iii) orders (or *schedules*) statements. The last step also adds a statement returning whether the candidate's conclusion holds.

There are certain constraints on the choices that ensure that the synthesized test case is a valid witness. Even with these constraints, a number of additional choices remain. Each choice produces a valid test case, but some of these test cases may not pass even if the candidate specification is correct. We describe the choices made by our algorithm, which empirically finds almost all correct candidate specifications.

A.1 Skeleton Construction

To witness correctness of the candidate path specification, the synthesized test case must exhibit *exactly* the external edges in its premise. In particular, the test case must include a call to each function in the candidate. Our algorithm constructs a *skeleton* consisting of these calls, for example, the skeleton on the second step of Figure 11. A symbol ??, called a *hole*, is included for each parameter and return value of each function call, and must be filled in with a variable name.

A.2 Filling Holes

The external edges in the candidate specification impose constraints on the arguments that should be used in each function call. In particular, the synthesized test case must exhibit every behavior encoded by the external edges in the candidate specification:

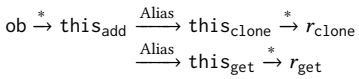
- **Alias:** For an aliasing edge $p_{m_i} \xrightarrow{\text{Alias}} p_{m_{i+1}}$, the algorithm has to ensure that the arguments p_{m_i} (passed to m_i) and $p_{m_{i+1}}$ (passed to m_{i+1}) are aliased.
- **Transfer:** For a transfer edge $r_{m_i} \xrightarrow{\text{Transfer}} p_{m_{i+1}}$, the algorithm has to use the return value of m_i as the argument passed to m_{i+1} (and similarly for backwards transfer edges $p_{m_i} \xrightarrow{\overline{\text{Transfer}}} r_{m_{i+1}}$).

For example, the holes in the skeleton in Figure 11 are filled so that the following premises are satisfied:

$$\texttt{this}_{\texttt{add}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{clone}}, \quad r_{\texttt{clone}} \xrightarrow{\text{Transfer}} \texttt{this}_{\texttt{get}}.$$

One issue is that internal edges may be self-loops, in which case more than two parameters may need to be aliased. For example, consider the following candidate:

$$\begin{aligned} \texttt{ob} \xrightarrow{*} \texttt{this}_{\texttt{add}} &\xrightarrow{\text{Alias}} \texttt{this}_{\texttt{clone}} \xrightarrow{*} \texttt{this}_{\texttt{id}} \\ &\xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}} \xrightarrow{*} r_{\texttt{get}}. \end{aligned} \tag{10}$$



skeleton	`??.add(??);`
	`?? = ??.clone();`
	`?? = ??.get(??);`

fill holes	`list.add(in);`
	`List listClone = list.clone();`
	`Object out = listClone.get(??);`

initialization & scheduling	`Object in = new Object();`
	`List list = new List()`
	`list.add(in);`
	`List listClone = list.clone();`
	`Object out = listClone.get(0);`
	`return in == out;`

Fig. 11. Steps in the test synthesis algorithm (right) for a candidate path specification for `List` (left). Code added at each step is highlighted in blue. Scheduling is shown in the same line as initialization—it chooses the final order of the statements. This figure is a duplicate of Figure 7, and is shown here for clarity.

For the test case for this candidate, the three calls to `add`, `clone`, and `get` must all share the same receiver:

```
list.add(in);
List listClone = list.clone();
Object out = list.get(??);
```

Our algorithm partitions the holes into subsets that must be aliased—since aliasing is a transitive relation, every hole in a subset has to be aliased with every other hole in that subset. To do so, the algorithm constructs an undirected graph where the vertices are the holes, and an edge $(h, h') \in E$ connects two holes h and h' in the following cases:

- There is an external edge $w_{m_i} \xrightarrow{T} z_{m_{i+1}}$ in the candidate specification, where h is the hole corresponding to w_{m_i} and h' is the hole corresponding to $z_{m_{i+1}}$.
- There is an internal edge $p_{m_i} \xrightarrow{*} p_{m_i}$ in the candidate specification, where h is the hole corresponding to the p_{m_i} on the left-hand side and h' is the hole corresponding to the p_{m_i} on the right-hand side.

Then, our algorithm computes the connected components in this graph. For each connected component, the algorithm chooses a fresh variable name, and each hole in that connected component is filled with this variable name.

For example, for the candidate in Figure 11, our algorithm computes the following partitions:

$$\{\texttt{ob}\},\ \{\texttt{this}_{\texttt{add}}, \texttt{this}_{\texttt{clone}}\},\ \{r_{\texttt{clone}}, \texttt{this}_{\texttt{get}}\},\ \{r_{\texttt{get}}\},$$

and fills the corresponding holes with the variables names

$$\texttt{in},\ \texttt{list},\ \texttt{listClone},\ \texttt{out},$$

respectively. Similarly, for (10), we compute partitions

$$\{\texttt{ob}\},\ \{\texttt{this}_{\texttt{add}}, \texttt{this}_{\texttt{clone}}, \texttt{this}_{\texttt{get}}\},\ \{r_{\texttt{clone}}\},\ \{r_{\texttt{get}}\}.$$

The variable names are the same as those chosen in Figure 11.

A.3 Variable Initialization

We describe primitive variables and reference variables separately. For the case of initializing reference variables, we describe two different strategies:

- **Null:** Whenever possible, initialize to `null`.
- **Instantiation:** Whenever possible, use constructor calls.

The first strategy ensures that the test case does not exhibit additional transfer and alias edges beyond those in the candidate specification. The second strategy may produce a test case that does not witness correctness, since it may include spurious edges not in the premise of the candidate. However, certain functions require that some of their arguments are not null; for example, the `put` function in the `Hashtable` class. We show empirically that the second variant identifies a number of candidates missed by the first, and that these additional specifications are in fact correct.

Primitive initialization. We initialize all primitive variables with `0` (except characters, which are initialized as `'a'`). In our experience, the only important choice of primitive value is the `index` parameter passed to functions such as `get`, which retrieve data from collections. Choosing the `index = 0` retrieves the single object the test case previously added to the collection. Testing more primitive values is possible but has so far been unnecessary.

Reference initialization using null. Reference variables for which aliasing relations hold must be instantiated (unless they have already been initialized as the return value of a function call). Any other reference variable is initialized to `null`. For example, in Figure 11, the variables `list` and `out` must be instantiated, but `cloneList` has already been initialized as the return value of `clone`. In general, the test case we synthesize calls the constructor with the fewest number of arguments; primitive arguments are initialized as before, and reference arguments are initialized using `null`.

Reference initialization using instantiation. In this approach, we have to synthesize constructor calls when empty constructors are unavailable. For example, if the only constructor for the `List` class was `List(Object val)`, then we would have to initialize an object of type `Object` as well:

```
List list = new List(new Object());
```

We encode the problem of synthesizing a valid constructor call as a directed hypergraph reachability problem. A *directed hypergraph* is a pair $G = (V, E)$, where V is a set of vertices, and edges $e \in E$ have the form $e = (h, B)$, where $h \in V$ is the *head* of the edge, and $B \subseteq V$ is its *body*. For our purposes, B is a list rather than a set, and may contain a single vertex multiple times.

We construct a hypergraph $G = (V, E)$ where vertices correspond to classes, and edges to constructors:

- **Vertices:** A vertex $v \in V$ is a library class.
- **Edges:** An edge $e = (h, B) \in E$ is a constructor, where h is the class of the constructed object and B is the list of classes of the constructor parameters.

For convenience, we also include primitive types as vertices in G, along with an edge representing the "empty constructor", which returns the initialization value described above.

Now, a *path* T in the hypergraph $G = (V, E)$ is a finite tree with root $v_T \in V$ (called the *root* of the path), such that for each vertex $v \in T$, v and its (ordered) children $[v_1, ..., v_k]$ are an edge $e_{T,v} = (v, [v_1, ..., v_k]) \in E$. Note that for each leaf v of T, there must necessarily be an edge $(v, []) \in E$, since v has no children. Also, we say a vertex $v \in V$ is *reachable* is there exists a path with root v.

In our setting, a path in our hypergraph G corresponds to a call to a constructor—for each vertex $\texttt{X} \in T$ with children `X1`, ..., `Xk`, we recursively define the constructor

$$C_T(\texttt{X}) = \texttt{new X}(C_T(\texttt{X1}),\ \ldots,\ C_T(\texttt{Xk})).$$

Therefore, devising a constructor call to instantiate an object of type `X` amounts to computing a path in G with root `X`. Paths to every reachable vertex can be efficiently computed using a standard

dynamic programming algorithm. Furthermore, we can add a weight w_e to each edge in $e \in E$. Then, the *shortest* path (i.e., the path minimizing the total weight $\sum_{v \in T} e_{T,v}$) can similarly be efficiently computed. We choose all weights $w_e = 1$ for each $e \in E$.

For example, suppose that the `List` class has a single constructor `List(Object val)`. Then, our algorithm constructs a hypergraph with two vertices and two edges:

$$V = \{\texttt{Object}, \texttt{List}\}$$
$$E = \{(\texttt{Object}, []), (\texttt{List}, [\texttt{Object}])\}.$$

Then, the path corresponding to `List` is the tree $T = \frac{\texttt{List}}{\texttt{Object}}$, which corresponds to the constructor call

```
new List(new Object())
```

used to instantiate variables of type `List`.

As with initializing primitive variables, multiple choices of constructor calls could be used, but selecting a single constructor suffices has been sufficient so far.

A.4 Statement Scheduling

Note that the test case now contains both function call statements as well as variable initialization statements added in the previous step. All the added variable initialization statements can be executed first, so it suffices to schedule the function call statements.

There are two kinds of constraints on scheduling function calls. First, edges in the candidate specification of the form

$$r_{m_i} \xrightarrow{\text{Transfer}} p_{m_{i+1}}$$

impose *hard constraints* on the schedule, since m_i must be called before m_{i+1} so its return value can be transfered to $p_{m_{i+1}}$ (edges of the form $p_{m_i} \xrightarrow{\overline{\text{Transfer}}} r_{m_{i+1}}$ impose hard constraints as well). For example, in Figure 11, the edge $r_{\texttt{clone}} \xrightarrow{\text{Transfer}} \texttt{this}_{\texttt{get}}$ imposes the hard constraint that the call to `clone` must be scheduled before the call to `get`. Then, any of the following orderings is permitted:

$$[\texttt{add}, \texttt{clone}, \texttt{get}],\ [\texttt{clone}, \texttt{add}, \texttt{get}],\ [\texttt{clone}, \texttt{get}, \texttt{add}].$$

We use *soft constraints* to choose among schedules satisfying the hard constraints. Empirically, we observe that the order of the functions in the specification is typically the same as the order in which they must be called for the conclusion to be exhibited dynamically. More precisely, function m_i should be called before function m_j whenever $i < j$. In our example, the soft constraint says that `add` should be scheduled before both `clone` and `get`.

Our algorithm iteratively constructs a schedule $[i_1, ..., i_k]$ of the function calls $F = \{m_1, ..., m_k\}$. At iteration t, it selects the tth function call m_{i_t} from the remaining calls $F_t \subseteq F$. It does so greedily, by identifying the choices $G_t \subseteq F_t$ that satisfy the hard constraints, and then selecting $m_{i_t} \in G_t$ to be optimal according to the soft constraints. These conditions uniquely specify m_{i_t}, since our soft constraints are a total ordering.

Our algorithm keeps track of the remaining statements F_t as a directed acyclic graph (DAG), which includes an edge $m_i \to m_j$ for each hard constraint that m_i should be scheduled before m_j. Then, G_t is the set of roots of F_t. Furthermore, our algorithm maintains G_t as a priority queue, where the priority of m_i is i (the highest priority element in G_t is the element with the smallest index i).

We initialize $F_1 = F$; then, G_1 is the subset of vertices in F_1 without a parent. Updates are computed as follows:

(1) The highest priority function call m_{i_t} in G_t is removed from both G_t and from F_t.
(2) For each child m_i of m_{i_t} in F_t, we determine if m_i is now a root of F_t (i.e., none of its parents are in F_t).
(3) For every child m_i that is now a root of F_t, we add m_i to G_t with priority i.

In Figure 11, F_1 has three vertices `add` (priority 1), `clone` (priority 2), and `get` (priority 3), and a single edge `clone` $\rightarrow$ `get`, and G_1 includes `add` and `clone`. Therefore, the selected schedule is [`add`, `clone`, `get`].

B PROOF OF TECHNICAL LEMMAS

We prove the technical lemmas used in Appendix 7.

B.1 Proof of Lemma 7.5

We first show the following lemma, which completely characterizes the subgroupoid of elements $\Sigma^*_{\text{free}} \subseteq \Sigma^*$:

Lemma B.1. We have

$$\begin{aligned}
[\text{Assign}]\ [\text{Assign}] &= [\text{Assign}] \\
[\text{Assign}]\ [\overline{\text{Assign}}] &= [\phi] \\
[\overline{\text{Assign}}]\ [\text{Assign}] &= [\phi] \\
[\overline{\text{Assign}}]\ [\overline{\text{Assign}}] &= [\overline{\text{Assign}}] \\
[\text{Assign}]\ [\overline{\text{New}}\ \text{New}] &= [\phi] \\
[\overline{\text{New}}\ \text{New}]\ [\text{Assign}] &= [\overline{\text{New}}\ \text{New}] \\
[\overline{\text{Assign}}]\ [\overline{\text{New}}\ \text{New}] &= [\overline{\text{Assign}}] \\
[\overline{\text{New}}\ \text{New}]\ [\overline{\text{Assign}}] &= [\phi] \\
[\overline{\text{New}}\ \text{New}]\ [\overline{\text{New}}\ \text{New}] &= [\phi].
\end{aligned}$$

Proof. We show the first relation; the others follow similarly. First, we show that if γ Assign $\delta \in \mathcal{L}(C_{\text{pt}})$, then γ Assign Assign$\delta \in \mathcal{L}(C_{\text{pt}})$. There must exist a derivation

$$\begin{aligned}
\text{FlowsTo} &\Rightarrow \ldots \Rightarrow \text{Transfer } u_\delta \\
&\Rightarrow \text{Transfer Assign } u_\delta \\
&\Rightarrow \ldots \\
&\Rightarrow \gamma \text{ Assign } \delta
\end{aligned}$$

since the only production in C_{pt} containing the terminal symbol Assign is Transfer $\rightarrow$ Transfer Assign. Therefore, the following derivation also exists:

$$\begin{aligned}
\text{FlowsTo} &\Rightarrow \ldots \Rightarrow \text{Transfer } \delta \\
&\Rightarrow \text{Transfer Assign } u_\delta \\
&\Rightarrow \text{Transfer Assign Assign } u_\delta \\
&\Rightarrow \ldots \\
&\Rightarrow \gamma \text{ Assign Assign } \delta,
\end{aligned}$$

i.e., γ Assign Assign $\delta \in \mathcal{L}(C_{\text{pt}})$. By a similar argument, it follows that if γ Assign Assign $\delta \in \mathcal{L}(C_{\text{pt}})$, then γ Assign $\delta \in \mathcal{L}(C_{\text{pt}})$, so [Assign] [Assign] = [Assign]. □

It follows directly that if $\alpha \in \Sigma^*_{\text{free}}$, then

$$[\alpha] \in \{[\phi],\ [\epsilon],\ [\text{Assign}],\ [\overline{\text{Assign}}],\ [\overline{\text{New}}\ \text{New}]\}.$$

In particular, for $\alpha' \in \Sigma^*_{\text{free}}$, $[\text{Assign}]\ [\alpha'] \in \{[\text{Assign}],\ [\phi]\}$, so the lemma follows by taking $\alpha' = \alpha$ Assign. □

B.2 Proof of Lemma 7.6

If we replace the terminal symbols $\sigma \in \Sigma_{\text{free}}$ with ϵ in C_{pt}, then C_{pt} is a parentheses matching grammar where each "open parentheses" $\text{Store}[f]$ (resp., $\overline{\text{Load}[f]}$) must be matched with a corresponding "closed parentheses" $\text{Load}[f]$ (resp., $\overline{\text{Store}[f]}$). Also, by Assumption 4.3, $\Sigma_{\text{lib}} \cap \Sigma_{\text{prog}} = \emptyset$.

Now, we prove by induction on the length of α. The base case $\alpha = \epsilon$ is clear. If $\alpha \in \Sigma^*$ does not contain a pair of matched parentheses $(\text{Store}[f], \text{Load}[f]) \in \Sigma^2_{\text{prog}}$, then $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, so we are done. Otherwise, for any such pair of matched parentheses, we can express $\alpha = \gamma\sigma\alpha'\tau\delta$. By induction, the lemma holds for α', so we can write $\alpha = \gamma'\sigma'\beta'\tau'\delta'$ as in the lemma. Therefore, we have

$$\alpha = (\gamma\sigma\gamma')\sigma'\beta'(\tau'\delta'\tau\delta),$$

so the claim follows. □

B.3 Proof of Lemma 7.7

We show the case $(\sigma, \tau) = (\text{Store}[f], \text{Load}[f])$, where $f \in \mathcal{F}_{\text{prog}}$; the case $(\sigma, \tau) = (\overline{\text{Load}[f]}, \overline{\text{Store}[f]})$ is similar. First, suppose that $\gamma\sigma\beta\tau\delta \in \mathcal{L}(C_{\text{pt}})$. Then, there must exist a derivation of form

$$\begin{aligned}\text{FlowsTo} \Rightarrow \ldots &\Rightarrow u_\gamma\ \overline{\text{Transfer}}\ u_\delta \\ &\Rightarrow u_\gamma\ \text{Transfer}\ \sigma\ \text{Alias}\ \tau\ u_\delta \\ &\Rightarrow \ldots \\ &\Rightarrow \gamma\sigma\beta\tau\delta,\end{aligned}$$

so the following derivation exists:

$$\begin{aligned}\text{FlowsTo} \Rightarrow \ldots \Rightarrow u_\gamma\ \overline{\text{Transfer}}\ u_\delta &\\ &\Rightarrow u_\gamma\ \text{Transfer Assign}\ u_\delta \\ &\Rightarrow \ldots \\ &\Rightarrow \gamma\ \text{Assign}\ \delta.\end{aligned}$$

The converse follows similarly, so the claim follows. □

B.4 Proof of Lemma 7.8

We show two preliminary lemmas.

Lemma B.2. We have

$$\begin{aligned}[\text{Store}[f]]\ [\overline{\text{New}}\ \text{New}]\ [\text{Load}[f]] &= [\text{Assign}] \\ [\overline{\text{Load}[f]}]\ [\overline{\text{New}}\ \text{New}]\ [\text{Store}[f]] &= [\overline{\text{Assign}}].\end{aligned}$$

Proof. Suppose that γ Store$[f]$ $\overline{\text{New}}$ New Load$[f]$ $\delta \in \mathcal{L}(C_{\text{pt}})$. Then, we must have derivation

$$\begin{aligned}\text{FlowsTo} &\Rightarrow \ldots \Rightarrow u_\gamma \text{ Transfer } u_\delta \\ &\Rightarrow u_\gamma \text{ Store}[f] \text{ Alias Load}[f] \; u_\delta \\ &\Rightarrow \ldots \\ &\Rightarrow \gamma \text{ Store}[f] \; \alpha \text{ Load}[f] \; \delta,\end{aligned}$$

so we also have derivation

$$\begin{aligned}\text{FlowsTo} &\Rightarrow \ldots \Rightarrow u_\gamma \text{ Transfer } u_\delta \\ &\Rightarrow u_\gamma \text{ Assign } u_\delta \\ &\Rightarrow \ldots \\ &\Rightarrow \gamma \text{ Assign Load}[f] \; \delta.\end{aligned}$$

Thus, γ Assign $\delta \in \mathcal{L}(C_{\text{pt}})$. The converse follows similarly, as does the second claim. □

Lemma B.3. For any $\beta \in \Sigma^* \setminus \{\epsilon\}$, we have

$$\begin{aligned}[\beta] = [\text{Assign}] &\Leftrightarrow \beta \in \mathcal{L}(C_{\text{pt}}, \text{Transfer}) \\ [\beta] = [\overline{\text{Assign}}] &\Leftrightarrow \beta \in \mathcal{L}(C_{\text{pt}}, \overline{\text{Transfer}}) \\ [\beta] = [\overline{\text{New}} \text{ New}] &\Leftrightarrow \beta \in \mathcal{L}(C_{\text{pt}}, \text{Alias}).\end{aligned}$$

Proof. We first show the forward implication. If $[\beta] = [\text{Assign}]$, then New Assign $\in \mathcal{L}(C_{\text{pt}})$, so New $\beta \in \mathcal{L}(C_{\text{pt}})$. Therefore, there must exist a derivation

$$\text{FlowsTo} \Rightarrow \text{New Transfer} \Rightarrow \ldots \Rightarrow \text{New } \beta,$$

so $\beta \in \mathcal{L}(C_{\text{pt}}, \text{Transfer})$. The other two cases follow similarly. Now, we show the backward implication. Suppose that $\beta \in \mathcal{L}(C_{\text{pt}}, \text{Transfer})$. We prove by structural induction on the derivation of β from Transfer. Since $\beta \neq \epsilon$, β cannot have been produced by Transfer $\Rightarrow \epsilon$. If β is produced by Transfer $\rightarrow$ Transfer Assign, then $\beta = \beta'$Assign, where $\beta' \in \mathcal{L}(C_{\text{pt}}, \text{Transfer})$. By induction, $[\beta'] = [\text{Assign}]$, so

$$[\beta] = [\beta'] \; [\text{Assign}] = [\text{Assign}] \; [\text{Assign}] = [\text{Assign}],$$

where the last step follows from Lemma B.1. Next, if β is produced using the production

$$\text{Transfer} \rightarrow \text{Transfer Store}[f] \text{ Alias Load}[f],$$

then $\beta = \beta'$ Store$[f]$ β'' Load$[f]$, where $\beta' \in \mathcal{L}(C_{\text{pt}}, \text{Transfer})$ and $\beta'' \in \mathcal{L}(C_{\text{pt}}, \text{Alias})$. By induction, $[\beta'] = [\text{Assign}]$ and $[\beta''] = [\overline{\text{New}} \text{ New}]$, so

$$\begin{aligned}\beta &= [\beta'] \; [\text{Store}[f]] \; [\beta''] \; [\text{Load}[f]] \\ &= [\text{Assign}] \; [\text{Store}[f]] \; [\overline{\text{New}} \text{ New}] \; [\text{Load}[f]] \\ &= [\text{Assign}],\end{aligned}$$

where the last step follows from Lemma B.2 and Lemma B.1. The remaining cases follow similarly. □

Now, suppose that $[\text{Store}[f]] \; [\beta] \; [\text{Load}[f]] = [\text{Assign}]$. Since

$$\text{New Store}[f] \; \overline{\text{New}} \text{ New Load}[f] \in \mathcal{L}(C_{\text{pt}}),$$

we have

$$\text{New Store}[f]\ \beta\ \text{Load}[f] \in \mathcal{L}(C_{\text{pt}}),$$

so the following derivation must exist:

$$\begin{aligned}\text{FlowsTo} &\Rightarrow \text{New Store}[f]\ \text{Alias Load}[f] \\ &\Rightarrow \ldots \\ &\Rightarrow \text{New Store}[f]\ \beta\ \text{Load}[f],\end{aligned}$$

i.e., $\beta \in \mathcal{L}(C_{\text{pt}}, \text{Alias})$. By Lemma B.3, we have $[\beta] = [\overline{\text{New}}\ \text{New}]$. The second case follows similarly. □