Hybrid Top-down and Bottom-up Interprocedural Analysis

Xin Zhang[1] Ravi Mangal[1] Mayur Naik[1] Hongseok Yang[2]

[1] Georgia Institute of Technology [2] Oxford University

Abstract

Interprocedural static analyses are broadly classified into top-down and bottom-up, depending upon how they compute, instantiate, and reuse procedure summaries. Both kinds of analyses are challenging to scale: top-down analyses are hindered by ineffective reuse of summaries whereas bottom-up analyses are hindered by inefficient computation and instantiation of summaries. This paper presents a hybrid approach SWIFT that combines top-down and bottom-up analyses in a manner that gains their benefits without suffering their drawbacks. SWIFT is general in that it is parametrized by the top-down and bottom-up analyses it combines. We show an instantiation of SWIFT on a type-state analysis and evaluate it on a suite of 12 Java programs of size 60-250 KLOC each. SWIFT outperforms both conventional approaches, finishing on all the programs while both of those approaches fail on the larger programs.

Categories and Subject Descriptors D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification

1. Introduction

Interprocedural static analyses are broadly classified into top-down and bottom-up. Top-down analyses start at root procedures of a program and proceed from callers to callees. Bottom-up analyses, on the other hand, begin from leaf procedures and proceed from callees to callers. For reasons of scalability and termination, both kinds of analyses compute and reuse *summaries* of analysis results over procedures, but they do so in fundamentally different ways.

Top-down analyses only analyze procedures under contexts in which they are called in a program. A key drawback of such analyses is that the summaries they compute tend to track details that are specific to individual calling contexts. These analyses thereby fail to sufficiently reuse summaries of a procedure across different calling contexts. This in turn causes a blow-up in the number of summaries and hinders the scalability of the analyses.

Bottom-up analyses analyze procedures under *all* contexts, not just those in which they are called in a program. These analyses have two key strengths: the summaries they compute are highly reusable and they are easier to parallelize. But these analyses also have drawbacks that stem from analyzing procedures in contexts that are unrealizable in a program: they either lose scalability due to the need to reason about too many cases, or they sacrifice precision by eliding needed distinctions between cases. Also, instantiating summaries computed by bottom-up analyses is usually expensive compared to top-down analyses.

PLDI '14,, June 9–11, 2014, Edinburgh, United Kingdom..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594328

It is thus evident that the performance of both top-down and bottom-up analyses depends crucially on how procedure summaries are computed, instantiated, and reused. Top-down analysis summaries are cheap to compute and instantiate but hard to reuse, whereas bottom-up analysis summaries are easy to reuse but expensive to compute and instantiate.

This paper proposes a new hybrid interprocedural analysis approach called SWIFT that synergistically combines top-down and bottom-up analyses. Our approach is based on two observations. First, multiple summaries of the top-down analysis for a procedure can be captured by a single summary of the bottom-up analysis for the procedure. Therefore, applying bottom-up summaries in the top-down analysis can greatly improve summary reuse. Second, although bottom-up analysis reasons about all possible cases over the unknown initial states of a procedure, only few of those cases may be encountered frequently during top-down analysis, or even be reachable from the root procedures of a program. Therefore, making the bottom-up analysis only analyze those cases that are encountered most frequently during top-down analysis can greatly reduce the cost of computing and instantiating bottom-up summaries.

We formalize SWIFT as a generic framework that is parametrized by the top-down and bottom-up analyses, and show how to instantiate it on a type-state analysis for object-oriented programs. This analysis is useful for checking a variety of program safety properties, features both may and must alias reasoning about pointers, and is challenging to scale to large programs. We implemented SWIFT for Java and evaluate it on the type-state analysis using 12 benchmark programs of size 60-250 KLOC each. SWIFT outperforms both conventional approaches in our experiments, achieving speedups upto 59X over the top-down approach and 118X over the bottom-up approach, and finishing successfully on all programs while both conventional approaches fail on the larger programs.

We summarize the main contributions of this paper:

1. We propose SWIFT, a new interprocedural analysis approach that synergistically combines top-down and bottom-up approaches. We formalize SWIFT as a generic framework and illustrate it on a realistic type-state analysis.
2. Central to SWIFT is a new kind of relational analysis that approximates input-output relationships of procedures without information about initial states, but avoids the common problem of generating too many cases by using a *pruning operator* to identify and drop infrequent cases during case splitting.
3. We present empirical results showing the effectiveness of SWIFT over both the top-down and bottom-up approaches for type-state analysis on a suite of real-world Java programs.

2. Overview

We illustrate SWIFT on a type-state analysis of an example program shown in Figure 1. The program creates three file objects and calls procedure `foo` on each of them to open and close the file. Each file `f` can be in state *opened*, *closed*, or *error* at any instant, and starts in state *closed*. A call `f.open()` changes its state to *opened* if it is *closed*, and to *error* otherwise. Similarly, `f.close()` changes its state to *closed* if it is *opened*, and to *error* otherwise.

```
main() {
      v1 = new File(); // h1
pc1: foo(v1);
      v2 = new File(); // h2
pc2: foo(v2);
      v3 = new File(); // h3
pc3: foo(v3);
}

foo(File f) {
    f.open(); f.close();
}
```

Summaries computed for procedure `foo` by a type-state analysis using:	
top-down approach	bottom-up approach
$(h_1, \mathit{closed}, \{f\}, \emptyset) \to \{\, (h_1, \mathit{closed}, \{f\}, \emptyset) \,\}$ [T_1]	$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n})$. if $(f \in \boldsymbol{n})$ then $\{\, (\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n}) \,\}$ [B_1]
$(h_2, \mathit{closed}, \{f\}, \emptyset) \to \{\, (h_2, \mathit{closed}, \{f\}, \emptyset) \,\}$ [T_2]	$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n})$. if $(f \in \boldsymbol{a})$ then $\{\, (\boldsymbol{h}, (\iota_{\mathit{close}} \circ \iota_{\mathit{open}})(\boldsymbol{t}), \boldsymbol{a}, \boldsymbol{n}) \,\}$ [B_2]
$(h_1, \mathit{closed}, \emptyset, \{f\}) \to \{\, (h_1, \mathit{closed}, \emptyset, \{f\}) \,\}$ [T_3]	$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n})$. if $(f \notin \boldsymbol{n} \wedge f \notin \boldsymbol{a} \wedge \mathit{mayalias}(f, \boldsymbol{h}))$ then $\{\, (\boldsymbol{h}, \mathit{error}, \boldsymbol{a}, \boldsymbol{n}) \,\}$ [B_3]
$(h_2, \mathit{closed}, \emptyset, \{f\}) \to \{\, (h_2, \mathit{closed}, \emptyset, \{f\}) \,\}$ [T_4]	$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n})$. if $(f \notin \boldsymbol{n} \wedge f \notin \boldsymbol{a} \wedge \neg\mathit{mayalias}(f, \boldsymbol{h}))$ then $\{\, (\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n}) \,\}$ [B_4]
$(h_3, \mathit{closed}, \{f\}, \emptyset) \to \{\, (h_3, \mathit{closed}, \{f\}, \emptyset) \,\}$ [T_5]	

Figure 1: Example illustrating top-down and bottom-up approaches to interprocedural type-state analysis.

We use a type-state analysis by Fink et al. [8] which computes a set of abstract states at each program point to over-approximate the type-states of all objects. Each abstract state is of the form (h, t, a, n) where h is an allocation site, t is the type-state in which an object created at that site might be in, and a and n are finite sets of program expressions with which that object is aliased (called *must set*) and not aliased (called *must-not set*), respectively.

We next illustrate two common interprocedural approaches: top-down and bottom-up. Informally, we use top-down to mean *global* and *explicit* (or *tabulating*), and bottom-up to mean *compositional* and *symbolic* (or *functional*). Our formal meaning of these terms is given in Sections 3.1 and 3.2.

2.1 The Top-Down Approach

The top-down approach starts from root procedures and summarizes procedures as functions on abstract states. For our example, it starts from `main`, proceeds in the order of program execution, and computes summaries T_1 through T_5 for procedure `foo`, which are shown in Figure 1. Summaries T_1, T_2, and T_5 mean that an incoming abstract state $(h_i, \mathit{closed}, \{f\}, \emptyset)$ is transformed to itself by `foo`, whereas summaries T_3 and T_4 mean the same for an incoming abstract state $(h_i, \mathit{closed}, \emptyset, \{f\})$. Summaries T_1, T_2, and T_5 share the property that the effect of `foo` on the incoming abstract state is a no-op when its argument `f` definitely points to the abstract object h_i (i.e., `f` is in the must set). In this case, the type-state analysis is able to perform a *strong update*, and establishes that the type-state remains *closed* after the calls to `f.open()` and `f.close()`. Summaries T_3 and T_4 share an analogous property that holds when `f` definitely does not point to the abstract object h_i (i.e., `f` is in the must-not set). In this case, the analysis establishes that the type-state remains *closed* despite the calls to `f.open()` and `f.closed()`, since h_i is different from `f`. However, these five top-down summaries are specific to calling contexts, and fail to capture these two general properties. As a result, they are unlikely to be reused heavily. The only summary reused is T_3 at call site pc_3.

2.2 The Bottom-Up Approach

The bottom-up approach starts from leaf procedures and proceeds up along call chains. Like the top-down approach, it too computes summaries, but they differ in crucial ways. Top-down summaries consider only *reachable* incoming abstract states of procedures whereas bottom-up summaries cover *all* possible incoming abstract states. Also, caller-specific information is not present in bottom-up summaries. For example, the bottom-up approach computes summaries B_1 through B_4 for procedure `foo`, which are shown in Figure 1. These summaries represent four partial functions on abstract states, corresponding to four different cases of incoming abstract states of `foo` based on the kind of aliasing between argument `f` and incoming abstract object $\boldsymbol{h}$. Summary B_1 is applicable when `f` is in the incoming must-not set $\boldsymbol{n}$. In this case, the incoming abstract state is simply returned. Summary B_2 is applicable when `f` is in the incoming must set $\boldsymbol{a}$. In this case, a strong update is performed, returning the incoming abstract state with the type-state updated to $(\iota_{\mathit{close}} \circ \iota_{\mathit{open}})(\boldsymbol{t})$, where

$$\begin{aligned} \iota_{\mathit{open}} &= \lambda \boldsymbol{t}.\ \text{if } (\boldsymbol{t} = \mathit{closed}) \text{ then } \mathit{opened} \text{ else } \mathit{error} \\ \iota_{\mathit{close}} &= \lambda \boldsymbol{t}.\ \text{if } (\boldsymbol{t} = \mathit{opened}) \text{ then } \mathit{closed} \text{ else } \mathit{error} \end{aligned}$$

are type-state transformers associated with `open()` and `close()`, respectively. The remaining two summaries B_3 and B_4 are applicable when `f` is neither in the incoming must nor must-not set; this is possible as the must and must-not sets track only finite numbers of expressions. In this situation, the type-state analysis uses possibly less precise but still safe information from a may-alias analysis, denoted *mayalias* in summaries B_3 and B_4. If the may-alias analysis states that `f` may-alias with $\boldsymbol{h}$, then B_3 applies, and a *weak update* is performed, returning the *error* type-state. Otherwise, B_4 applies and the incoming abstract state is returned similar to B_1.

Procedure `foo` has only two statements yet there are already four cases represented in its bottom-up summaries. Note that this is despite the use of symbolic representations such as type-state transformer $\iota_{\mathit{close}} \circ \iota_{\mathit{open}}$ in B_2 instead of case-splitting on $\boldsymbol{t}$. For real-world programs, the number of cases can grow exponentially with the number of commands, and hinder scalability.

2.3 Our Hybrid Approach: SWIFT

The above drawbacks of the top-down and the bottom-up approaches (namely, lack of generalization in top-down summaries and excessive case splitting in bottom-up summaries) motivate our hybrid approach SWIFT. SWIFT is parametrized by the top-down and the bottom-up versions of the given analysis, plus two thresholds k and θ that control its overall performance. SWIFT triggers the bottom-up analysis on a procedure when the number of incoming abstract states of that procedure computed by the top-down analysis exceeds threshold k. Requiring sufficiently many incoming abstract states not only limits triggering the bottom-up analysis to procedures that are being re-analyzed the most often by the top-down analysis—and thus are likely suffering the most from lack of summary reuse and offer the greatest benefit of generalizing—but also helps the bottom-up analysis to determine the most common cases when it does case splitting, and track only those cases, unlike the original bottom-up analysis which tracks *all* cases. The threshold θ dictates the maximum number of such cases to track and thereby limit excessive case splitting.

We illustrate SWIFT using $k = 2$ and $\theta = 2$ on our example. SWIFT starts the top-down analysis of `main` in the usual order of program execution. Upon reaching program point pc_1 with abstract state $(h_1, \mathit{closed}, \{v_1\}, \emptyset)$, it checks whether any bottom-up summary is available for `foo`. Since there is none, it continues the top-down analysis, analyzes the body of `foo` with initial abstract state

$$(h_1, \mathit{closed}, \{f\}, \emptyset) \qquad [A_1]$$

and computes top-down summary T_1. The number of initial abstract states of `foo` is now one but not above the threshold $(k = 2)$

needed to trigger the bottom-up analysis. So SWIFT continues with the top-down analysis, reaches program point pc_2 with abstract states $(h_2, \mathit{closed}, \{v_2\}, \emptyset)$ and $(h_1, \mathit{closed}, \{v_1\}, \{v_2\})$, proceeds to re-analyze `foo` in the corresponding new initial abstract states

$$(h_2, \mathit{closed}, \{f\}, \emptyset) \qquad [A_2]$$
$$(h_1, \mathit{closed}, \emptyset, \{f\}) \qquad [A_3]$$

and computes top-down summaries T_2 and T_3, respectively. The number of initial abstract states of `foo` is now 3, which has exceeded our threshold $k = 2$. Hence, SWIFT triggers a bottom-up analysis of `foo`. The bottom-up analysis starts analyzing `foo` with

$$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n}).\, \text{if (true) then } \{\ (\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n})\ \}$$

which means the identity function on abstract states, and correctly summarizes that the part of `foo` analyzed so far does not make any state changes. The analysis then proceeds to transform the above function according to the abstract semantics of each command in `foo`, while avoiding case splitting by ignoring certain initial abstract states of `foo`. When it analyzes the first command `f.open()`, it transforms the identity function above to four cases, namely, B_1, B_3, and B_4 shown in Figure 1, plus

$$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n}).\, \text{if } (f \in \boldsymbol{a}) \text{ then } \{\ (\boldsymbol{h}, \iota_{\mathit{open}}(\boldsymbol{t}), \boldsymbol{a}, \boldsymbol{n})\ \} \qquad [B_2']$$

B_2' differs from B_2 in Figure 1 only in skipping the application of ι_{close}, since the second command `f.close()` has not yet been analyzed by the bottom-up analysis. At this point, the bottom-up analysis inspects the three existing abstract states A_1, A_2, and A_3 that have been recorded in the corresponding top-down summaries T_1, T_2, and T_3, respectively, and determines that case B_2' is the most common (applying to A_1 and A_2), case B_1 is the second-most common (applying to A_3), and cases B_3 and B_4 are the least common (applying to none of the existing abstract states). Since $\theta = 2$, it keeps the two most common cases B_2' and B_1, and prunes cases B_3 and B_4. It then proceeds to analyze command `f.close()` similarly to obtain the final bottom-up summaries of `foo` as

$$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n}).\, \text{if } (f \in \boldsymbol{n}) \text{ then } \{\ (\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n})\ \} \qquad [B_1]$$
$$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n}).\, \text{if } (f \in \boldsymbol{a}) \text{ then } \{\ (\boldsymbol{h}, (\iota_{\mathit{close}} \circ \iota_{\mathit{open}})(\boldsymbol{t}), \boldsymbol{a}, \boldsymbol{n})\ \} \quad [B_2]$$

Thus, while four cases (B_1–B_4) existed in the original bottom-up summaries of `foo`, these new summaries represent only two cases, corresponding to the most common incoming abstract states of `foo`.

Once the bottom-up analysis is finished, SWIFT continues from the statement following program point pc_2 in `main`, and reaches the call to `foo` at pc_3 with three abstract states:

$$(h_1, \mathit{closed}, \emptyset, \{f\}) \qquad [A_3]$$
$$(h_2, \mathit{closed}, \emptyset, \{f\}) \qquad [A_4]$$
$$(h_3, \mathit{closed}, \{f\}, \emptyset) \qquad [A_5]$$

Since it had encountered A_3 at pc_2, it reuses top-down summary T_3 and avoids re-analyzing `foo`, similar to a conventional top-down analysis. But more significantly, SWIFT avoids re-analyzing `foo` even for A_4 and A_5: of the bottom-up summaries B_1 and B_2 that it computed for `foo`, B_1 applies to A_4, and B_2 applies to A_5.

In summary, for our example, SWIFT avoids creating top-down summaries T_4 and T_5 that a conventional top-down analysis computes, and it avoids creating bottom-up summaries B_3 and B_4 that a conventional bottom-up analysis computes.

2.4 The Challenge of Pruning

Procedure `foo` has a single control-flow path which causes exactly one of bottom-up summaries B_1–B_4 to apply to any incoming abstract state A. But in general, procedures have multiple control-flow paths, which can cause multiple bottom-up summaries to apply to a given state A. Retaining some of these summaries while pruning others, and reusing only the retained ones upon encountering state A, is unsound. We illustrate this with an alternate definition of `foo`:

```
foo(File f, File g) {
    if (*) then { f.open(); f.close(); } else g.open(); }
```

Its bottom-up analysis yields summaries B_1–B_4 for the true branch, and four similar summaries for the false branch, one of which is

$$\lambda(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{a}, \boldsymbol{n}).\, \text{if } (g \in \boldsymbol{a}) \text{ then } \{\ (\boldsymbol{h}, \iota_{\mathit{open}}(\boldsymbol{t}), \boldsymbol{a}, \boldsymbol{n})\ \} \qquad [B_5]$$

Suppose SWIFT prunes all of them except B_2 (case $f \in \boldsymbol{a}$) and B_5 (case $g \in \boldsymbol{a}$), and then encounters two incoming abstract states:

$$\langle h, \mathit{closed}, \{f, g\}, \emptyset\rangle \qquad [A_1]$$
$$\langle h, \mathit{closed}, \{g\}, \{f\}\rangle \qquad [A_2]$$

Both B_2 and B_5 apply to A_1, yielding result $\{\langle h, \mathit{closed}, \{f, g\}, \emptyset\rangle, \langle h, \mathit{opened}, \{f, g\}, \emptyset\rangle\}$ as expected. In particular, none of the six pruned cases apply to A_1, and therefore they have no effect on the result. On the other hand, B_1 and B_5 apply to A_2. Since B_1 was pruned, applying B_5 alone will yield incorrect result $\{\langle h, \mathit{opened}, \{g\}, \{f\}\rangle\}$, as it omits state $\langle h, \mathit{closed}, \{g\}, \{f\}\rangle$ produced by B_1. SWIFT guarantees correctness; specifically, we prove that the hybrid approach it embodies is equivalent to a conventional top-down approach (see Theorem 3.1). In the above scenario, SWIFT re-analyzes `foo` for A_2, as a conventional top-down approach would; only if both of B_1 and B_5 are retained does it avoid re-analyzing, and reuses both of them. Thus, in either scenario, SWIFT produces the correct result for A_2.

3. Formalism

This section presents SWIFT as a generic framework. Section 3.1 formulates the top-down analysis. Section 3.2 formulates the bottom-up analysis. Section 3.3 describes conditions relating the two analyses that SWIFT requires. Section 3.4 augments bottom-up analysis with a pruning operator guided by top-down analysis. Section 3.5 extends the formalism to (recursive) procedures.

3.1 Top-Down Analysis

Our formalism targets a language of commands C:

$$C \ ::= \ c \mid C + C \mid C; C \mid C^*$$

It includes primitive commands, non-deterministic choice, sequential composition, and iteration. Section 3.5 adds procedure calls.

A top-down analysis $\mathcal{A} = (\mathbb{S}, \mathsf{trans})$ is specified by:

1. a finite set $\mathbb{S}$ of abstract states, and
2. transfer functions $\mathsf{trans}(c) : \mathbb{S} \to 2^{\mathbb{S}}$ of primitive commands c.

The abstract domain of the analysis is the powerset $\mathbb{D} = 2^{\mathbb{S}}$ with the subset order. The abstract semantics of the analysis is standard:

$$\begin{aligned}
[\![C]\!] &: 2^{\mathbb{S}} \to 2^{\mathbb{S}} \\
[\![c]\!](\Sigma) &= \mathsf{trans}(c)^\dagger(\Sigma) \\
[\![C_1 + C_2]\!](\Sigma) &= [\![C_1]\!](\Sigma) \cup [\![C_2]\!](\Sigma) \\
[\![C_1; C_2]\!](\Sigma) &= [\![C_2]\!]([\![C_1]\!](\Sigma)) \\
[\![C^*]\!](\Sigma) &= \mathsf{lfix}\ (\lambda\Sigma'.\, \Sigma \cup [\![C]\!](\Sigma')).
\end{aligned}$$

where notation $f^\dagger$ denotes lifting of function $f : D_1 \times \ldots \times D_n \to 2^D$ to sets of input arguments, as follows: $f^\dagger(X_1, \ldots, X_n) = \bigcup\{f(x_1, \ldots, x_n) \mid \forall i.\, x_i \in X_i\}$.

Example. We illustrate our formalism using the type-state analysis shown in Figure 2. This analysis computes a set of abstract states (also called abstract objects) at each program point to over-approximate the type-states of all objects. Each abstract state is of the form (h, t, a) and represents that an object allocated at site h may be in type-state t and is pointed to by at least variables in a

Domains:

(method)	$m \in \mathbb{M}$	(allocation site)	$h \in \mathbb{H}$
(variable)	$v \in \mathbb{V}$	(access path set)	$a \in \mathbb{A} = 2^{\mathbb{V}}$

(type state)	$t \in \mathbb{T} = \{\mathit{init}, \mathit{error}, ...\}$
(type-state function)	$[m] \in \mathbb{I} = \mathbb{T} \to \mathbb{T}$
(abstract states)	$\sigma \in \mathbb{S} = \mathbb{H} \times \mathbb{T} \times \mathbb{A}$

Primitive Commands:

$$c \ ::= \ v = \texttt{new}\, h \ \mid \ v = w \ \mid \ v.m()$$

Transfer Functions:

$$\begin{aligned}
\mathsf{trans}(v = \texttt{new}\, h')(h,t,a) &= \{(h, t,\ a \setminus \{v\}), (h', \mathit{init},\ \{v\})\} \\
\mathsf{trans}(v = w)(h,t,a) &= \text{if } (w \in a) \text{ then } \{(h, t,\ a \cup \{v\})\} \\
&\quad\ \text{else } \{(h, t,\ a \setminus \{v\})\} \\
\mathsf{trans}(v.m())(h,t,a) &= \text{if } (v \in a) \text{ then } \{(h, [m](t), a)\} \\
&\quad\ \text{else } \{(h, \mathit{error},\ a)\}
\end{aligned}$$

Figure 2: Top-down type-state analysis.

(the must set). For clarity of exposition, this type-state analysis is simpler than the full version in our experiments, in two respects: first, it omits tracking the must-not set in each abstract state, unlike the type-state analysis in Section 2; second, it restricts the must set to only contain variables, whereas the implementation allows heap access path expressions such as $v.f$ and $w.g.f$.

The transfer function of a primitive command c conservatively updates the type-state and must set of each incoming abstract object (h, t, a). The updated must set includes aliases newly generated by c, and those in a that survive the state change of c. For instance, $\mathsf{trans}(v = w)(h, t, a)$ removes v from a when w is not in a. This is because in that case, any aliases with v can be destroyed by assignment $v = w$. The only command that affects the type-state is a method call. In particular, $\mathsf{trans}(v.m())(h, t, a)$ performs a strong update on the incoming type-state t according to the type-state function $[m]$ associated with m, when v is in a, and transitions it to the *error* type-state otherwise. □

3.2 Bottom-Up Analysis

A bottom-up analysis $\mathcal{B} = (\mathbb{R}, \mathsf{id}^\sharp, \gamma, \mathsf{rtrans}, \mathsf{rcomp})$ is specified by:

1. a ***domain of abstract relations*** $(\mathbb{R}, \mathsf{id}^\sharp, \gamma)$ where $\mathbb{R}$ is a finite set with an element $\mathsf{id}^\sharp$ and γ is a function of type $\mathbb{R} \to 2^{\mathbb{S}\times\mathbb{S}}$ such that $\gamma(\mathsf{id}^\sharp) = \{(\sigma, \sigma) \mid \sigma \in \mathbb{S}\}$;
2. transfer functions $\mathsf{rtrans}(c) : \mathbb{R} \to 2^{\mathbb{R}}$ of primitive commands c; and
3. an operator $\mathsf{rcomp} : \mathbb{R} \times \mathbb{R} \to 2^{\mathbb{R}}$ to compose abstract relations.

Elements r in $\mathbb{R}$ mean relations $\gamma(r)$ over abstract states in $\mathbb{S}$. Hence, they are called abstract relations. We require a relation $\mathsf{id}^\sharp$ in $\mathbb{R}$ that denotes the identity relation on $\mathbb{S}$.[1] We denote the domain of a relation r by $\mathsf{dom}(r) = \{\sigma \mid \exists\sigma' : (\sigma, \sigma') \in \gamma(r)\}$. The input to $\mathsf{rtrans}(c)$ describes past state changes from the entry of the current procedure up to the primitive command c, and the function extends this description with the state change of c. The operator rcomp allows composing two abstract relations, and is used to compute the effects of procedure calls; namely, when a procedure with summary $\{r_1, ..., r_n\}$ is called and an input relation to this call is r, the result of analyzing the call is $\bigcup_{i=1}^{n} \mathsf{rcomp}(r, r_i)$.

Example. Our bottom-up type-state analysis is shown in Figure 3. It contains two types of abstract relations. The first type is (σ, ϕ), and it denotes a constant relation on abstract states that relates any σ' satisfying ϕ to the given σ. The second type is (ι, a_0, a_1, ϕ), and this means a relation that takes an abstract state $\sigma = (h, t, a)$

Domain of Abstract Relations:

(predicate)	$\phi \in \mathbb{Q}$
	$\phi ::= \mathsf{true} \mid \phi \wedge \phi \mid \mathsf{have}(v) \mid \mathsf{notHave}(v)$
(type-state function)	$\iota \in \mathbb{I} = \{\lambda t.t, \lambda t.\mathit{init}, \lambda t.\mathit{error}, ...\}$
(abstract relation)	$r \in \mathbb{R} = (\mathbb{S} \times \mathbb{Q}) \cup (\mathbb{I} \times 2^{\mathbb{V}} \times 2^{\mathbb{V}} \times \mathbb{Q})$

$$\begin{aligned}
\mathsf{id}^\sharp &= (\lambda t.t, \mathbb{V}, \emptyset, \mathsf{true}) \\
\gamma(\sigma, \phi) &= \{(\sigma_0, \sigma) \mid \sigma_0 \models \phi\} \\
\gamma(\iota, a_0, a_1, \phi) &= \{(\sigma_0, \sigma) \mid \sigma_0 \models \phi \wedge \exists h, t, a, a' : (\sigma_0 = (h, t, a) \\
&\qquad \wedge\, \sigma = (h, \iota(t), a') \wedge a' = (a \cap a_0) \cup a_1)\}
\end{aligned}$$

where $\sigma \models \phi$ expresses the satisfaction of ϕ by the abstract state:

$\sigma \models \mathsf{true}$		always
$\sigma \models \phi \wedge \phi'$	$\iff$	$\sigma \models \phi$ and $\sigma \models \phi'$
$\sigma \models \mathsf{have}(v)$	$\iff$	$(h, t, a) = \sigma$ and $v \in a$ for some h, t, a
$\sigma \models \mathsf{notHave}(v)$	$\iff$	$(h, t, a) = \sigma$ and $v \notin a$ for some h, t, a

Transfer Functions:

$$\begin{aligned}
&\mathsf{rtrans}(c)(\sigma, \phi) = \{(\sigma', \phi) \mid \sigma' \in \mathsf{trans}(c)(\sigma)\} \\
&\mathsf{rtrans}(v = \texttt{new}\, h)(\iota, a_0, a_1, \phi) = \\
&\qquad\qquad \{(\iota, a_0 \setminus \{v\}, a_1 \setminus \{v\}, \phi),\ ((h, \mathit{init}, \{v\}), \phi)\} \\
&\mathsf{rtrans}(v = w)(\iota, a_0, a_1, \phi) = \\
&\qquad\qquad \text{if } (w \in a_1) \text{ then } \{(\iota, a_0, a_1 \cup \{v\}, \phi)\} \\
&\qquad\qquad \text{else if } (w \notin a_0) \text{ then } \{(\iota, a_0 \setminus \{v\}, a_1 \setminus \{v\}, \phi)\} \\
&\qquad\qquad \text{else } \{(\iota, a_0, a_1 \cup \{v\}, \phi \wedge \mathsf{have}(w)), \\
&\qquad\qquad\qquad (\iota, a_0 \setminus \{v\}, a_1 \setminus \{v\}, \phi \wedge \mathsf{notHave}(w))\} \\
&\mathsf{rtrans}(v.m())(\iota, a_0, a_1, \phi) = \\
&\qquad\qquad \text{if } (v \in a_1) \text{ then } \{([m] \circ \iota, a_0, a_1, \phi)\} \\
&\qquad\qquad \text{else if } (v \notin a_0) \text{ then } \{(\lambda t.\mathit{error}, a_0, a_1, \phi)\} \\
&\qquad\qquad \text{else } \{([m] \circ \iota, a_0, a_1, \phi \wedge \mathsf{have}(v)), \\
&\qquad\qquad\qquad (\lambda t.\mathit{error}, a_0, a_1, \phi \wedge \mathsf{notHave}(v))\}
\end{aligned}$$

Relation Composition:

$$\begin{aligned}
&\mathsf{rcomp}(r, r') = \\
&\quad \text{let } (\_, \phi) = r \text{ and } (\_, \phi') = r' \text{ in} \\
&\quad \text{if } (\mathsf{wp}(r, \phi') \Leftrightarrow \mathsf{false}) \text{ then } \emptyset \text{ else } \{(r; r', \phi \wedge \mathsf{wp}(r, \phi'))\}
\end{aligned}$$

Here the routines wp and $r; r'$ are defined as follows:

$$\begin{aligned}
&\mathsf{wp}(r, \mathsf{true}) = \mathsf{true}, \qquad \mathsf{wp}(r, \phi \wedge \phi') = \mathsf{wp}(r, \phi) \wedge \mathsf{wp}(r, \phi'), \\
&\mathsf{wp}((\sigma, \phi), \mathsf{have}(v)) = \text{if } (\sigma \models \mathsf{have}(v)) \text{ then } \mathsf{true} \text{ else } \mathsf{false} \\
&\mathsf{wp}((\sigma, \phi), \mathsf{notHave}(v)) = \text{if } (\sigma \models \mathsf{notHave}(v)) \text{ then } \mathsf{true} \text{ else } \mathsf{false} \\
&\mathsf{wp}((\iota, a_0, a_1, \phi), \mathsf{have}(v)) = \\
&\quad \text{if } (v \in a_1) \text{ then } \mathsf{true} \text{ else } (\text{if } (v \notin a_0) \text{ then } \mathsf{have}(v) \text{ else } \mathsf{false}) \\
&\mathsf{wp}((\iota, a_0, a_1, \phi), \mathsf{notHave}(v)) = \\
&\quad \text{if } (v \in a_1) \text{ then } \mathsf{false} \text{ else } (\text{if } (v \notin a_0) \text{ then } \mathsf{true} \text{ else } \mathsf{notHave}(v)) \\
&r; (\sigma', \_) = \sigma' \\
&((h, t, a), \_); (\iota', a'_0, a'_1, \_) = (h, \iota'(t), a \cap a'_0 \cup a'_1) \\
&(\iota, a_0, a_1, \_); (\iota', a'_0, a'_1, \_) = (\iota' \circ \iota, a_0 \cap a'_0, a_1 \cap a'_0 \cup a'_1)
\end{aligned}$$

Figure 3: Bottom-up type-state analysis.

satisfying ϕ and updates its type-state to $\iota(t)$ and its must set to $(a \cap a_0) \cup a_1$. For instance, $(\iota_{\mathit{close}} \circ \iota_{\mathit{open}}, \mathbb{V}, \emptyset, \mathsf{have}(f))$ denotes the relation that is defined only when the input must set contains f, and that produces as output the same must set but updates type-state t to $t' = (\iota_{\mathit{close}} \circ \iota_{\mathit{open}})(t)$ (e.g., if $t = \mathit{closed}$ then $t' = \mathit{closed}$, and if $t = \mathit{opened}$ then $t' = \mathit{error}$). This is precisely the bottom-up summary B_2 of procedure `foo` in the example in Section 2.

We next describe transfer functions rtrans of primitive commands. When the input abstract relation is (σ, ϕ), $\mathsf{rtrans}(c)$ simply updates the σ part using trans from the top-down type-state analysis. When the input is (ι, a_0, a_1, ϕ), $\mathsf{rtrans}(c)$ does not have such a uniform behavior—it changes each component of the input according to $\mathsf{trans}(c)$. We explain this with $\mathsf{rtrans}(v = w)$. Recall that $\mathsf{trans}(v = w)(h, t, a)$ splits cases based on whether the must set a contains w or not. To implement this case split, $\mathsf{rtrans}(v = w)$ views the input (ι, a_0, a_1, ϕ) as a transformer on abstract states, and

[1] Our bottom-up analysis uses $\mathsf{id}^\sharp$ as the initial abstract relation when analyzing procedures. See Section 3.5.

[C1] For all commands c, relations $r \in \mathbb{R}$, and states $\sigma, \sigma' \in \mathbb{S}$:

$$(\exists r_0 : r_0 \in \mathsf{rtrans}(c)(r) \wedge (\sigma, \sigma') \in \gamma(r_0)) \iff (\exists \sigma_0 : (\sigma, \sigma_0) \in \gamma(r) \wedge \sigma' \in \mathsf{trans}(c)(\sigma_0))$$

[C2] For all $r_1, r_2 \in \mathbb{R}$ and $\sigma, \sigma' \in \mathbb{S}$:

$$(\sigma, \sigma') \in \gamma^\dagger(\mathsf{rcomp}(r_1, r_2)) \iff \exists \sigma_0 : (\sigma, \sigma_0) \in \gamma(r_1) \wedge (\sigma_0, \sigma') \in \gamma(r_2)$$

[C3] For all $r \in \mathbb{R}$ and $\sigma \in \mathbb{S}$ and $\Sigma \in 2^{\mathbb{S}}$:

$$\sigma \in \mathsf{wp}(r, \Sigma) \iff (\forall \sigma' : (\sigma, \sigma') \in \gamma(r) \Rightarrow \sigma' \in \Sigma)$$

Figure 4: Conditions required by our SWIFT framework.

checks whether this transformer results in an abstract state with w in its must set. The three cases in the definition of $\mathsf{rtrans}(v = w)$ correspond to the three answers to this question: always, never and sometimes. In the first two cases, $\mathsf{rtrans}(v = w)$ updates its input so that v is included (first case) or excluded (second case) from the must set. In the third case, it generates two abstract relations that cover both possibilities of including and excluding v.

The remaining part is the composition operator $\mathsf{rcomp}(r, r')$. One difficulty for composing relations r and r' symbolically is that the precondition ϕ' of r' is not a property of initial states of the composed relation, but that of intermediate states. Hence, we need to compute the weakest precondition of ϕ' with respect to the first relation r. Our rcomp operator calls the routine $\mathsf{wp}(r, \phi')$ to do this computation, and constructs a new precondition by conjoining the precondition ϕ of the first relation r with the result of this call. Computing the other state-transformation part of the result of $\mathsf{rcomp}(r, r')$ is relatively easier, and follows from the semantics of $\gamma(r)$ and $\gamma(r')$. It is described by the other routine $r; r'$. $\square$

3.3 Conditions of SWIFT Framework

SWIFT allows combining a top-down analysis $\mathcal{A}$ and a bottom-up analysis $\mathcal{B}$ as specified in the preceding subsections. Since it is a generic framework, however, SWIFT lets users decide the relative degrees of these two analyses in the resulting hybrid analysis, by means of thresholds k and θ. SWIFT has to guarantee the correctness and equivalence of the resulting analyses for all choices of these thresholds. For this purpose, SWIFT requires three conditions **C1–C3** shown in Figure 4 on the two analyses that it combines.

Condition **C1** requires trans and rtrans—the transfer functions of the top-down and bottom-up analyses for primitive commands—to be equally precise. Our top-down and bottom-up type-state analyses satisfy this condition. In Section 5, we discuss ways to avoid manually specifying both trans and rtrans, by automatically synthesizing one from the other while satisfying this condition.

Condition **C2** requires the operator $\mathsf{rcomp} : \mathbb{R} \times \mathbb{R} \to 2^{\mathbb{R}}$ to accurately model the composition of relations $\gamma(r_1)$ and $\gamma(r_2)$. Note that $\gamma(r_i)$ is a relation over abstract states, not concrete states, which makes it easier to discharge the condition. Besides, the bottom-up analysis might define rcomp in a manner that already satisfies this condition. The rcomp operator for our type-state analysis in Figure 3 illustrates both of these aspects.

Condition **C3** requires an operator $\mathsf{wp} : \mathbb{R} \times 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ to compute the weakest precondition of an abstract relation over a given set of abstract states. SWIFT uses this operator to adjust bottom-up summaries of called procedures. Those summaries involve preconditions on incoming abstract states to the callee, and need to be recast as preconditions to the caller. The wp operator satisfies this need. Designing wp is relatively simple, because it computes a weakest precondition on abstract relations in $\mathbb{R}$, not on relations over concrete states. Besides, this operator might already be defined as part of the bottom-up analysis. Both of these observations hold for the wp operator in Figure 3 for our type-state analysis.

In summary, the conditions required by SWIFT are not onerous, and may even already hold for the analyses to be combined.

3.4 Pruning and Coincidence

We now proceed to augment our bottom-up analysis with a pruning operator and show that it coincides with the top-down analysis. We first define an operation $\mathsf{excl} : 2^{\mathbb{R}} \times 2^{\mathbb{S}} \to 2^{\mathbb{R}}$ as follows:

$$\mathsf{excl}(R, \Sigma) = \{r \in R \mid \mathsf{dom}(r) \not\subseteq \Sigma\},$$

which removes abstract relations r that become void if we ignore abstract states in Σ from the domain of r.

We then define a ***pruning operator*** as a function $f : 2^{\mathbb{R}} \times 2^{\mathbb{S}} \to 2^{\mathbb{R}} \times 2^{\mathbb{S}}$ such that for all $R, R' \subseteq \mathbb{R}$ and $\Sigma, \Sigma' \subseteq \mathbb{S}$,

$$f(R, \Sigma) = (R', \Sigma') \Rightarrow (\Sigma \subseteq \Sigma' \wedge R' = \mathsf{excl}(R, \Sigma')).$$

The purpose of a pruning operator f is to filter out some abstract relations from its input R. When making this filtering decision, the operator also takes Σ, which contains abstract states that the bottom-up analysis has already decided to ignore. Given such an R and Σ, the operator increases the set of ignored states to Σ', and removes all abstract relations r that do not relate any abstract states outside of Σ' (i.e., $\mathsf{dom}(r) \subseteq \Sigma'$).

SWIFT automatically constructs a pruning operator, denoted prune, by ranking abstract relations and choosing the top θ relations, where θ is a parameter to SWIFT. The ranking is based on the frequencies of incoming abstract states of the current procedure, which are encountered during top-down analysis performed by SWIFT. Formally, prune is built in four steps described next.

First, we assume a multi-set M of incoming abstract states to a given command, which the top-down analysis has previously encountered while analyzing that command in a bigger context.

Second, we define a function $\mathsf{rank} : \mathbb{R} \to \mathbb{N}$ that ranks abstract relations based on M as follows:

$$\mathsf{rank}(r) = \sum_{\sigma \in \mathsf{dom}(r)} \# \text{ of copies of } \sigma \text{ in } M$$

Third, we define a function $\mathsf{best}_\theta : 2^{\mathbb{R}} \to 2^{\mathbb{R}}$ that chooses the top $\theta \geq 1$ abstract relations by their rank values as follows:

$$\mathsf{best}_\theta(R) = \text{set of the top } \theta \text{ elements in } R \text{ according to rank}$$

Finally, we define the pruning operator prune as follows:

$$\begin{aligned}\mathsf{prune}(R, \Sigma) = \ & \text{let } R_0 = \mathsf{best}_\theta(R) \text{ in} \\ & \text{let } \Sigma' = \Sigma \cup \bigcup\{\mathsf{dom}(r) \mid r \in R \setminus R_0\} \text{ in} \\ & (\mathsf{excl}(R_0, \Sigma'), \Sigma').\end{aligned}$$

The operator first chooses the top θ abstract relations from R, and forms a new set R_0 with these chosen relations. The next step is to increase Σ. The operator goes through every unchosen relation in R, computes its domain, and adds abstract states in the domain to Σ. The reason for performing this step is to find an appropriate restriction on the domains of R and R_0 such that they have the same meaning under this restriction, although R_0 contains only selected few of R. The result of this iteration, Σ', is such a restriction:

$$\gamma^\dagger(R) \cap ((\mathbb{S} \setminus \Sigma') \times \mathbb{S}) = \gamma^\dagger(R_0) \cap ((\mathbb{S} \setminus \Sigma') \times \mathbb{S}).$$

Note that once we decide to ignore abstract states in Σ' from the domain of an abstract relation, some abstract relations in R_0 become redundant, because they do not relate any abstract states outside of Σ'. In the last step, the operator removes such redundant elements from R_0 using the operator $\mathsf{excl}(-, \Sigma')$. The result of this further reduction and the set Σ' become the output of the operator.

Example. We illustrate the prune operator on our type-state analysis using the example in Section 2. Suppose SWIFT has analyzed procedure `foo` thrice using the top-down type-state analysis in incoming abstract states A_1, A_2, and A_3, and suppose the bottom-up analysis it triggered has just finished analyzing command `f.open()` in the body of `foo`. At this point, SWIFT has data:

$$M = \{(h_1, closed, \{f\}), (h_2, closed, \{f\}), (h_1, closed, \emptyset)\}, \Sigma = \emptyset,$$
$$R = \{(\iota_{open}, \mathbb{V}, \emptyset, \mathsf{have}(f)),\ (\lambda t.error, \mathbb{V}, \emptyset, \mathsf{notHave}(f))\}.$$

If θ is 1, the pruning operator will retain $(\iota_{open}, \mathbb{V}, \emptyset, \mathsf{have}(f))$ from R, since two abstract states in M satisfy $\mathsf{have}(f)$ (namely, the first two listed in M) whereas only one satisfies $\mathsf{notHave}(f)$. The result of the operator in this case will become:

$$\Sigma' = \{(h, t, a) \mid f \notin a\}, \quad R' = \{(\iota_{open}, \mathbb{V}, \emptyset, \mathsf{have}(f))\}. \qquad \square$$

SWIFT automatically augments the given bottom-up analysis with the prune operator to yield an analysis whose abstract domain is

$$\mathbb{D}_r = \{(R, \Sigma) \in 2^{\mathbb{R}} \times 2^{\mathbb{S}} \mid \forall r \in R.\, \mathsf{dom}(r) \not\subseteq \Sigma\}$$
$$(R, \Sigma) \sqsubseteq (R', \Sigma') \iff \Sigma \subseteq \Sigma' \wedge \mathsf{excl}(R, \Sigma') \subseteq R'$$

An element (R, Σ) means a set of relations R on abstract states together with the set of ignored input abstract states Σ. We require that every $r \in R$ carries non-empty information when the input abstract states are restricted to $\mathbb{S} \setminus \Sigma$ (i.e., $\mathsf{dom}(r) \not\subseteq \Sigma$). Our order $\sqsubseteq$ says that increasing the ignored set Σ or increasing the set of relations R makes (R, Σ) bigger. It is a partial order with the following join operation $\sqcup$ where $\mathsf{clean}(R, \Sigma) = (\mathsf{excl}(R, \Sigma), \Sigma)$:

$$\bigsqcup_{i \in I} (R_i, \Sigma_i) = \mathsf{clean}(\bigcup_{i \in I} R_i, \bigcup_{i \in I} \Sigma_i)$$

The semantics of the bottom-up analysis with pruning is as follows:

$$\begin{aligned}
\llbracket C \rrbracket^r &: \mathbb{D}_r \to \mathbb{D}_r \\
\llbracket c \rrbracket^r(R, \Sigma) &= (\mathsf{prune} \circ \mathsf{clean})(\mathsf{rtrans}(c)^\dagger(R), \Sigma) \\
\llbracket C_1 + C_2 \rrbracket^r(R, \Sigma) &= \mathsf{prune}(\llbracket C_1 \rrbracket^r(R, \Sigma) \sqcup \llbracket C_2 \rrbracket^r(R, \Sigma)) \\
\llbracket C_1; C_2 \rrbracket^r(R, \Sigma) &= \llbracket C_2 \rrbracket^r(\llbracket C_1 \rrbracket^r(R, \Sigma)) \\
\llbracket C^* \rrbracket^r(R, \Sigma) &= \mathsf{fix}^{(R, \Sigma)}\ F \\
&\text{(where } F(R', \Sigma') = \mathsf{prune}((R', \Sigma') \sqcup \llbracket C \rrbracket^r(R', \Sigma'))).
\end{aligned}$$

$\mathsf{fix}^{(R, \Sigma)}\ F$ gives the stable element of the increasing sequence:[2]

$$(R, \Sigma),\ F(R, \Sigma),\ F^2(R, \Sigma),\ F^3(R, \Sigma),\ F^4(R, \Sigma),\ \ldots$$

This sequence is increasing because $(R', \Sigma') \sqsubseteq F(R', \Sigma')$ for every (R', Σ'), and the sequence has a stable element because the domain $\mathbb{D}_r$ has finite height. Notice that whenever a new abstract relation r can be generated in the above semantics, prune is applied to filter r out unless it is considered one of common cases. The degree of filtering done by prune is the main knob of SWIFT for balancing performance against other factors such as the generality of computed abstract relations.

We now present our main theorem, which says that bottom-up analysis with pruning computes the same result as top-down analysis, provided the set of incoming abstract states does not include any abstract state that the bottom-up analysis ignores.

THEOREM 3.1 (Coincidence). *For all Σ, Σ', R', if*

$$(\llbracket C \rrbracket^r(\{\mathsf{id}^\sharp\}, \emptyset) = (R', \Sigma')) \wedge (\Sigma \cap \Sigma' = \emptyset),$$

we have that for every σ' in $\mathbb{S}$,

$$\sigma' \in \llbracket C \rrbracket(\Sigma) \iff \exists \sigma \in \Sigma : (\sigma, \sigma') \in \gamma^\dagger(R').$$

3.5 Extension for Procedures

To handle procedures that are potentially mutually recursive, we extend the formalism we have so far, as follows.

First, we include procedure calls $f()$ in our language of commands, where $f \in \mathsf{PName}$, a set of procedure names. We also define a program Γ as a map from procedure names to commands.

[2] A stable element of a sequence $\{x_i\}_{i \geq 0}$ is x_n such that $x_n = x_{n+1}$.

Second, we allow the prune operator to be parametrized by procedure names, so that it can be specialized to each procedure. For instance, the operator can use a different pruning strategy for each procedure that depends on the incoming abstract states of that procedure. After this change, prune has the following type:

$$\mathsf{prune} : \mathsf{PName} \to (2^{\mathbb{R}} \times 2^{\mathbb{S}} \to 2^{\mathbb{R}} \times 2^{\mathbb{S}})$$

Given the above two changes, the abstract semantics of the bottom-up analysis augmented with pruning is defined for a program Γ as a map from procedure names to their bottom-up summaries, and it is computed by an iterative fixpoint computation:

$$\begin{aligned}
\llbracket \Gamma \rrbracket^r &: \mathsf{PName} \to 2^{\mathbb{R}} \times 2^{\mathbb{S}} \\
\llbracket \Gamma \rrbracket^r &= \mathsf{fix}^{\eta_0}\ (\lambda \eta.\, \lambda f.\, \eta(f) \sqcup \llbracket \Gamma(f) \rrbracket^r_{f,\eta}(\{\mathsf{id}^\sharp\}, \emptyset))
\end{aligned}$$

The fix^{η_0} operator iterates its argument function from $\eta_0 = \lambda f.\, (\emptyset, \emptyset)$ until it obtains a stable point. It is the same operator as in the intraprocedural setting in Section 3.4. The abstract semantics repeatedly updates a map η of procedure summaries until it reaches a fixpoint. The update is done by analyzing the body $\Gamma(f)$ of each procedure f separately, while using the given η to handle procedure calls in the procedure body. The analysis of $\Gamma(f)$ is formally specified by its abstract semantics $\llbracket \Gamma(f) \rrbracket^r_{f,\eta}$, which is analogous to that in Section 3.4 (namely, we replace each occurrence of $\llbracket . \rrbracket^r$ by $\llbracket . \rrbracket^r_{f,\eta}$, and each occurrence of prune by $\mathsf{prune}(f)$), augmented with a case for procedure calls $g()$ defined as follows:

$$\begin{aligned}
\llbracket C \rrbracket^r_{f,\eta} &: 2^{\mathbb{R}} \times 2^{\mathbb{S}} \to 2^{\mathbb{R}} \times 2^{\mathbb{S}} \\
\llbracket g() \rrbracket^r_{f,\eta}(R, \Sigma) &= \mathsf{let}\ (R', \Sigma') = \eta(g)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ R'' = \mathsf{rcomp}^\dagger(R, R')\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \Sigma'' = \mathbb{S} \setminus \bigcap \{\mathsf{wp}(r, \mathbb{S} \setminus \Sigma') \mid r \in R\}\ \mathsf{in} \\
&\quad\ (\mathsf{prune}(f) \circ \mathsf{clean})(R'', \Sigma' \cup \Sigma'')
\end{aligned}$$

In the case of a procedure call $g()$, the analysis first looks up a summary (R', Σ') of g from the given summary map η. Then, it incorporates the effects of this procedure call by composing R and R', and propagates backward the set Σ' of abstract states to be pruned at the call site, all the way to the entry of the current procedure. This propagation uses the wp operator defined in Section 3.3.

4. Algorithm

The overall algorithm of SWIFT is presented as Algorithm 1. It takes an initial abstract state to the procedure $\mathsf{main} \in \mathsf{PName}$, and a program. For convenience, we assume that the program is specified by both a control-flow graph G and a map Γ from procedure names to commands. Given these inputs, the algorithm iteratively updates three data structures:

$$\mathsf{td} : \mathsf{PC} \to 2^{\mathbb{S} \times \mathbb{S}},\ \ \mathsf{workset} : 2^{\mathsf{PC} \times \mathbb{S} \times \mathbb{S}},\ \ \mathsf{bu} : \mathsf{PName} \rightharpoonup 2^{\mathbb{R}} \times 2^{\mathbb{S}}.$$

Here PC is a set of program points (i.e., vertices in the control flow graph G), $\mathbb{S}$ a set of abstract states and $\mathbb{R}$ a set of abstract relations. The map td and the set workset are data structures of the top-down analysis. The former records the result of the top-down analysis. It maps a program point pc to a set of pairs (σ, σ'), which represent state changes from the entry of the procedure containing pc up to the program point pc. The first component σ of a pair is an incoming abstract state of the procedure containing pc, and the second σ' is an abstract state that arises at pc when σ is the abstract state at the procedure entry. The top-down analysis works by generating new abstract states from pairs $(\sigma, \sigma') \in \mathsf{td}(pc)$ and propagating them to successors of pc. The set workset keeps a set of newly generated abstract states that should be handled by the top-down analysis subsequently. The remaining bu is a table of procedure summaries computed by the bottom-up analysis. Note that bu is a partial function. If $\mathsf{bu}(f)$ is undefined, it means that the procedure f is not yet analyzed by the bottom-up analysis.

Algorithm 1 The SWIFT algorithm.

```
1: INPUTS: Initial abstract state $\sigma_I$ and program $(G, \Gamma)$
2: OUTPUTS: Analysis results td and bu
3: var workset, $R_0, \Sigma_0, \Sigma, f, \sigma$
4: td $= \lambda pc.\emptyset$, bu $= \lambda f.$undef, workset $= \{(\text{entry}_{\text{main}}, \sigma_I, \sigma_I)\}$
5: while (workset $\neq \emptyset$) do
6:     pop $w$ from workset
7:     if (command at $w$ is not a procedure call) then
8:         (td, workset) = run_td($G, w$, td, workset)
9:     else
10:        let $f$ be the procedure invoked at $w$
11:        let $\sigma$ be the current abstract state in $w$
12:        if ($\exists R_0, \Sigma_0 :$ bu$(f) = (R_0, \Sigma_0) \wedge \sigma \notin \Sigma_0$) then
13:            $\Sigma = \{\sigma' \mid (\sigma, \sigma') \in \gamma^\dagger(R_0)\}$
14:            (td, workset) = update_td($\Sigma, w$, td, workset)
15:        else
16:            (td, workset) = run_td($G, w$, td, workset)
17:            if (# of input abstract states to $f$ in td > threshold $k$
               and bu is undefined for $f$) then
18:                bu = run_bu($\Gamma, \theta, f$, bu)
19:            end if
20:        end if
21:    end if
22: end while
```

The algorithm repeatedly pops a $w \in$ workset and processes it until workset becomes empty. Handling w normally means the update of workset and td according to the top-down analysis. We express this normal case in the algorithm by

$$(\mathsf{td}, \mathsf{workset}) := \mathsf{run\_td}(G, w, \mathsf{td}, \mathsf{workset})$$

where run_td is a standard tabulation-based computation [14] that we omit here. The exception to this normal case occurs when the command at the program point of w is a call to a procedure f. In this case, the algorithm checks whether the bottom-up analysis has a summary for f that can be applied to the current abstract state σ of w (i.e., the third component of w). If the check passes, it uses bu(f) and computes the result Σ of analyzing f with σ, and updates (td, workset) with Σ according to the top-down analysis (line 14). If the check fails, the algorithm resorts to the top-down analysis, and updates (td, workset). However, unlike the non-procedure call case, the handling of w does not stop here. Instead, it examines the possibility of running the bottom-up analysis on the body of f. If the number of incoming abstract states of f in the top-down analysis exceeds the threshold k, and the bottom-up analysis is not yet run on f and so bu(f) is undefined, then the algorithm runs the bottom-up analysis for all procedures F reachable from f via call chains, computes summaries for procedures in F, and updates bu with these summaries. All these steps are expressed by a single instruction bu := run_bu$(\Gamma, \theta, f, \mathsf{bu})$ in the algorithm (line 18), which gets expanded to the following pseudo code:

```
run_bu($\Gamma, \theta, f$, bu) =
    let $F$ = {procedures reachable from $f$} in
    let bu' = $[\![\Gamma|_F]\!]^r$ with $\theta$ used during pruning in
    $\lambda g.$ if ($g \in F$) then bu'$(g)$ else bu$(g)$
```

where $[\![\Gamma|_F]\!]^r$ denotes the run of the bottom-up analysis of Section 3.5 on procedures in F.

We conclude by discussing two scenarios where the pruning operator could have difficulty in identifying common cases during the execution of run_bu. Both scenarios could arise in theory but occurred rarely in our experiments. Assume a setting where the number of incoming abstract states to a procedure f has exceeded the threshold k, and a procedure g is reachable from f. In the first scenario, g has not been analyzed in the top-down analysis, although f was analyzed multiple times. Our pruning operator lacks data about g from the top-down analysis and so cannot steer the bottom-up analysis of g towards its common cases. Our implementation handles this issue by postponing executing run_bu$(\Gamma, \theta, f, \mathsf{bu})$ until there is at least one incoming abstract state of g. The second scenario is that g has been analyzed multiple times in the top-down analysis but most of these analyses do not originate from f. In this scenario, our pruning operator uses the dominating incoming abstract states of g over the whole program, even though these may not be the dominating ones for g's calling context from f.

5. Discussion

Most existing interprocedural analyses either use the top-down approach or the bottom-up approach. This section discusses obligations that analysis developers using either of these approaches must satisfy in order to apply SWIFT for improving their scalability.

5.1 From Bottom-Up Analysis to SWIFT

Suppose a bottom-up analysis exists as specified in Section 3.2. To employ SWIFT, an analysis designer must supply a top-down analysis as specified in Section 3.1, and ensure that it satisfies condition **C1** in Section 3.3 that relates the transfer functions for primitive commands trans and rtrans by the two analyses. (We discuss the remaining two conditions **C2** and **C3** momentarily.) Such a top-down analysis can be synthesized automatically from the bottom-up analysis:

$$\mathsf{trans}(c)(\sigma) = \{\sigma' \mid (\sigma, \sigma') \in \gamma(\mathsf{rtrans}(c)(\mathsf{id}^\sharp))\}$$

The only obligations for using SWIFT on an existing bottom-up analysis, then, are defining operators rcomp and wp of the bottom-up analysis in a manner that satisfies conditions **C2** and **C3**, respectively. As we discussed in Section 3.3, discharging these conditions is not onerous, since they are stated over abstract semantics, not the concrete one. Also, the bottom-up analysis may already define these operators, as in the case of our type-state analysis.

5.2 From Top-Down Analysis to SWIFT

Suppose a top-down analysis exists as specified in Section 3.1. To make use of SWIFT, an analysis designer must supply a bottom-up analysis as specified in Section 3.2, and ensure that it satisfies conditions **C1–C3**. Unlike the opposite direction above, there is no general recipe to synthesize the bottom-up analysis automatically from the top-down analysis. Intuitively, the naïve synthesis approach, which defines

$$\mathsf{rtrans}(c)(r) = \{(\sigma_1, \sigma_3) | \exists \sigma_2 : (\sigma_1, \sigma_2) \in r \wedge \sigma_3 \in \mathsf{trans}(c)(\sigma_2)\}$$

does not generalize input-output relationships any more than the top-down analysis. Nevertheless, we have identified a general class of analyses for which we *can* automatically synthesize. We call these *kill/gen analyses*. The intuition is that transfer functions of primitive commands for kill/gen analyses have a special simple form: they transform the input by using the meet and join with fixed abstract states, which are selected based on a transformer-specific case analysis on the input. These kill/gen analyses include bitvector dataflow analyses as well as certain alias analyses (e.g., connection analysis [9]). Our type-state analysis is not an instance of kill/gen analysis, due to the transfer function of $v.m()$, but its handling of the must sets re-uses our kill/gen recipe for synthesizing bottom-up analysis from top-down analysis.

6. Empirical Evaluation

This section empirically evaluates SWIFT. Section 6.1 describes our experiment setup. Section 6.2 compares SWIFT to conventional

	description	# classes		# methods		bytecode (KB)		KLOC	
		app	total	app	total	app	total	app	total
jpat-p	protein analysis tools	5	176	13	766	1	39	1.5	78
elevator	discrete event simulator	5	188	24	899	2.3	52	0.6	88
toba-s	java bytecode to C compiler	25	158	149	745	32	56	6	69
javasrc-p	java source code to HTML translator	49	135	461	789	43	60	13	66
hedc	web crawler from ETH	44	353	230	2,134	16	140	6	153
antlr	A parser/translator generator	111	350	1,150	2,370	128	186	29	131
luindex	document indexing and search tool	206	619	1,390	3,732	102	235	39	190
lusearch	text indexing and search tool	219	640	1,399	3,923	94	250	40	198
kawa-c	scheme to java bytecode compiler	151	529	1,049	3,412	62	174	21	186
avrora	microcontroller simulator/analyzer	1,158	1,544	4,234	6,247	223	325	64	193
rhino-a	JavaScript interpreter	66	330	686	2,288	64	162	26	153
sablecc-j	parser generator	294	876	1,742	5,143	75	276	40	257

Table 1: Benchmark characteristics. All the reported numbers are computed using a 0-CFA call-graph analysis.

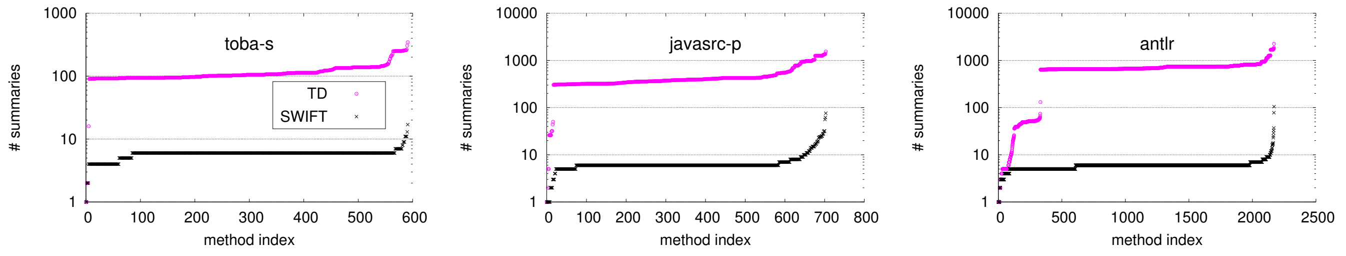

Figure 5: Number of top-down summaries computed for each method by TD and SWIFT for three different benchmarks. The X-axis represents indices of methods sorted by the number of summaries. The Y-axis, which uses log scale, represents the number of summaries.

top-down and bottom-up approaches, called TD and BU, respectively. Section 6.3 shows the effect of varying thresholds k and θ.

6.1 Experimental Setup

We implemented SWIFT for building hybrid interprocedural analyses for Java bytecode using the Chord program analysis platform. The top-down part of the framework is based on the tabulation algorithm [14] while the bottom-up part is based on the relational analysis with pruning described in Section 3.

For concreteness, we built an interprocedural type-state analysis using SWIFT. Unlike the type-state analysis from Section 3, it allows tracking access path expressions formed using variables and fields (upto two), such as $v.f$ and $v.f.g$. By tracking more forms of access path expressions and handling field updates more precisely, the type-state analysis implemented is more precise but also more difficult to scale than the simplified one in Section 3.

We obtained the baseline TD and BU type-state analyses by switching off the bottom-up part and the top-down part, respectively, in SWIFT. Throughout this section, a top-down summary means a pair of input-output abstract states (σ, σ') computed for a method by TD or the top-down part of SWIFT. A bottom-up summary is a pair (r, ϕ) computed for a method either by BU or the bottom-up part of SWIFT, such that r is an abstract relation and ϕ the set of input abstract states to which it applies.

We compared SWIFT with TD and BU on 12 real-world Java programs shown in Table 1. The programs and type-state properties are from the Ashes Suite and the DaCapo Suite. All experiments were done using Oracle HotSpot JVM 1.6.0 on Linux machines with 3.0GHz processors and 16GB memory per JVM process.

6.2 Performance of SWIFT vs. Baseline Approaches

Table 2 shows the running time and the total number of top-down and bottom-up summaries computed by SWIFT and the baseline approaches on each benchmark. For SWIFT, we set the threshold on the number of top-down summaries to trigger bottom-up analysis to five (i.e., $k = 5$) and limited to keeping a single case in the pruned bottom-up analysis (i.e., $\theta = 1$), which we found optimal overall.

SWIFT successfully finished on all benchmarks and outperformed both TD and BU significantly on most benchmarks, while TD and BU only finished on a subset of the benchmarks.

SWIFT ***vs.*** TD. SWIFT significantly boosted the performance of the type-state analysis over TD, achieving speedups of 4X–59X for most benchmarks. Moreover, for the largest three benchmarks, SWIFT only took under seven minutes each, whereas TD ran out of memory. Besides running time, the table also shows the total number of top-down summaries computed by the two approaches. SWIFT avoided computing over 95% of the summaries computed by TD for most benchmarks.

Figure 5 shows the number of top-down summaries computed for each method by the two approaches. The Y-axis represents the number of summaries using log scale and the X-axis lists the methods sorted by the number of summaries. A point on the X-axis does not necessarily denote the same method for both approaches since they may differ in the sorted order. The graphs show that SWIFT greatly reduces the numbers of top-down summaries, keeping them close to the threshold ($k = 5$) for most methods. It shows that the pruned bottom-up analysis successfully finds the dominating case. We inspected the top-down summaries of some methods and found that, for most of the inspected methods, the identity function with a certain precondition was the dominating case.

SWIFT ***vs.*** BU. BU only finished on our two smallest benchmarks. Compared with TD, BU is much harder to scale due to the exponential case splitting and expensive summary instantiation, especially for an analysis like type-state analysis, which tracks nontrivial aliasing information. Even on these two benchmarks, SWIFT achieved speedups of 9X–118X over BU. Besides running time, Table 2 also shows the total number of bottom-up summaries computed in both approaches. SWIFT avoided computing 87% to 96% of the bottom-up summaries that BU computed.

6.3 Effect of Varying SWIFT Thresholds

Effect of Varying k**.** Table 3 shows the total running time and the number of top-down summaries generated for one of our largest benchmarks `avrora`, using seven different values of k. The trend

	running time (m=min., s=sec.)					# summaries (k=thousands)					
	TD	BU	SWIFT	speedup over TD	speedup over BU	top-down			bottom-up		
						TD	SWIFT	drop	BU	SWIFT	drop
jpat-p	0.91s	15.62s	1.79s	0.5X	9X	6.5k	1.7k	74%	2.3k	0.3k	87%
elevator	1.59s	6m35s	3.36s	0.5X	118X	8.4k	2.9k	66%	12k	0.5k	96%
toba-s	20.4s	***timeout***	5s	4X	-	68.5k	3.5k	95%	-	0.6k	-
javasrc-p	4m44s	***timeout***	12s	24X	-	319k	5k	98%	-	0.7k	-
hedc	22m57s	***timeout***	41s	33X	-	891k	11k	99%	-	1.8k	-
antlr	35m28s	***timeout***	36s	59X	-	1,357k	13k	99%	-	2k	-
luindex	43m26s	***timeout***	1m53s	23X	-	2,260k	20k	99%	-	3k	-
lusearch	31m39s	***timeout***	1m52s	17X	-	1,922k	21k	99%	-	3.5k	-
kawa-c	23m52s	***timeout***	1m6s	22X	-	1,661k	19k	99%	-	3k	-
avrora	***timeout***	***timeout***	6m35s	-	-	-	91k	-	-	5.4k	-
rhino-a	***timeout***	***timeout***	6m39s	-	-	-	16k	-	-	2k	-
sablecc-j	***timeout***	***timeout***	4m25s	-	-	-	26k	-	-	4.8k	-

Table 2: Running time and total number of summaries computed by SWIFT and the baseline approaches TD and BU. Experiments that timed out either ran out of memory or failed to terminate in 24 hours.

is similar for the other benchmarks. According to this table, $k = 50$ is optimal in terms of the running time while $k = 10$ is optimal in the number of top-down summaries generated.

If we fix the running time of bottom-up analysis, the performance of SWIFT can be approximated by the number of times the top-down analysis is performed on each method, which in turn can be captured by the number of top-down summaries generated. On one hand, setting k too high generates too many top-down summaries before the bottom-up analysis is triggered on the method. As Table 3 shows, the numbers of top-down summaries generated for each method grow dramatically from $k = 10$ to $k = 500$. On the other hand, setting k too low triggers the bottom-up analysis too early, resulting in failure to predict the dominating case in the top-down analysis. For example, using $k = 2$ and $k = 5$ generates more top-down summaries than using $k = 10$.

If we consider the running time of bottom-up analysis, there is a tradeoff between the time spent on running bottom-up analysis and the time saved by avoiding repeatedly running top-down analysis on the same method. Generally, bottom-up analysis is much more expensive than the top-down analysis, even under our setting using pruning. The run using $k = 10$ thus consumes more time than that using $k = 50$ by running bottom-up analysis too frequently. However, the run using $k = 50$ is faster than $k = 100$, as the time it saves on the top-down analysis compensates for the time spent on running the bottom-up analysis. The run using $k = 2$ not only invokes the bottom-up analysis too often but also the top-down analysis, due to its failure to predict the dominating case. It is thus even slower than the run using $k = 200$.

Effect of Varying θ. Table 4 shows the effect of varying θ, the maximum number of cases to be kept by the pruned bottom-up analysis. Using $\theta = 2$ reduced the number of top-down summaries computed by SWIFT, as expected, but on most benchmarks overall performance was worse since the overhead incurred in the bottom-up analysis offset the savings of applying bottom-up summaries in the top-down analysis. The only exception was the run on one of our largest benchmarks `avrora`, which received a slight boost in running time and a large drop in top-down summaries generated when increasing θ from 1 to 2. As shown in Table 3, using $k = 5$ triggers the bottom-up analysis too early, resulting in failure to predict the dominating case in the top-down analysis when just one case is tracked. Using $\theta = 2$ gives the bottom-up analysis additional budget and reduces the risk of pruning away the dominating case.

7. Related Work

Sharir and Pnueli [19] present the *call-strings* and *functional* approaches to interprocedural analysis. The call-strings approach is a particular kind of top-down approach in which procedure contexts are distinguished using call strings. They present two variants of

k	running time (m=min., s=sec.)	# summaries (in thousands)
2	28m4s	372
5	6m35s	91
10	4m5s	68
50	2m37s	280
100	4m9s	543
200	22m7s	1,150
500	48m49s	2,663

Table 3: Running time and total number of top-down summaries computed by SWIFT using different k on `avrora` with $\theta = 1$.

	running time (m=min., s=sec.)		# summaries (in thousands)	
	$\theta = 1$	$\theta = 2$	$\theta = 1$	$\theta = 2$
toba-s	5s	6s	3.5	3.5
javasrc-p	12s	20s	5	4.6
hedc	41s	1m36s	11	10
antlr	36s	1m18s	13	13
luindex	1m53s	3m48s	20	20
lusearch	1m52s	3m48s	21	20.5
kawa-c	1m6s	2m10s	19	18
avrora	6m35s	6m20s	91	39
rhino-a	6m39s	12m28s	16	16
sablecc-j	4m25s	13m28s	26	22

Table 4: Running time and total number of top-down summaries computed by SWIFT using different θ with $k = 5$.

the functional approach: one that uses a symbolic representation of summaries, which may be viewed as a bottom-up approach, and another that uses an explicit representation, which may be viewed as a top-down approach where procedure contexts are abstract states as opposed to call strings. They provide an iterative fixpoint algorithm for the functional approach variant that uses an explicit representation, but it may use exponential space. Reps et al. [14] identify a representation for the class of IFDS dataflow analysis problems that takes quadratic space and can be computed in cubic time. Sagiv et al. [17] generalize the IFDS problem into IDE problem, which takes quadratic space and can be computed in cubic time, if there exits efficient representations for the transfer functions.

Our work differs from the above works as follows. First, even when there is a compact representation of procedure summaries with polynomial guarantee, our approach aims at empirically outperforming this guarantee. Second, we do not require a compact representation of transfer functions or procedure summaries. For instance, a summary of our bottom-up type-state analysis is a set of tuples, whose size can be exponential in the number of program variables. The above works do not attempt to avoid such blowup, whereas we provide a recipe for it, via the pruning operator and the interaction between the top-down and bottom-up versions.

Jeannet et al. [12] propose an analysis to generate relational summaries for shape properties in three-valued logic. Yorsh et al. [22] present a logic to relate reachable heap patterns between procedure inputs and outputs. These works involve relational analyses, but they do not focus on controlling the amount of case splitting in a bottom-up relational analysis, as we do with a pruning operator. For instance, Jeannet et al.'s relational shape analysis works top-down, and computes summaries for given incoming abstract states, unlike our bottom-up summaries that can be applied to unseen states. The issue of case splitting does not arise in their setting.

Bottom-up analyses have been studied theoretically in [6, 11], but are less popular in practice than top-down analyses, because of the challenges in designing and implementing them efficiently. Notable exceptions are analyses about pointer and heap reasoning, such as those for may-alias information [10, 13, 18, 20], must-alias information [4, 7] and shape properties [3, 10], where the analyses typically use symbolic abstract domains.

In contrast, top-down analysis [5, 14, 19] is much better understood, and generic implementations are available in analysis frameworks such as Chord, Soot, and Wala. Various approaches have been proposed to scale top-down analyses. Rinetzky et al. [15, 16] improve summary reuse in heap reasoning by separating the part of the heap that can be locally changed by the procedure from the rest of the heap. Yorsh et al. [23] generalize top-down summaries by replacing explicit summaries with symbolic representations, thereby increasing reuse without losing precision. Ball et al. [2] generalize highly reusable summaries by encoding the transfer functions using BDDs. Our work provides a new technique to scale top-down analysis by tightly integrating it with a bottom-up counterpart.

Others have hinted at the existence of dominating cases in bottom-up analyses. The shape analysis of Calcagno et al. [3] assumes that a dereferenced heap cell at a program point pc of a procedure f is usually not aliased with any of the previously accessed cells in the same procedure, unless f creates such an aliasing explicitly before reaching pc. The analysis, then, focuses on initial states to procedures that are considered common based on this assumption, and thereby avoids excessive case splitting. Bottom-up alias analyses [21] likewise assume non-aliasing between procedure arguments. These assumptions are not robust since they conjecture a property of common initial states to procedures without actually seeing any of them. Our hybrid analysis suggests a way to overcome this issue by collecting samples from the top-down analysis and identifying common cases based on these samples.

An orthogonal approach for scaling interprocedural analysis is parallelization. Bottom-up analyses are easy to parallelize—independent procedures can be analyzed in parallel—and recent work also addresses parallelizing top-down analyses [1]. A possible way to parallelize our hybrid approach is to modify it such that whenever a bottom-up summary is to be computed, it spawns a new thread to do this bottom-up analysis, and itself continues the top-down analysis. Developing this idea further and exploring other parallelizing strategies (such as [1]) is future work.

8. Conclusion

We proposed a new approach to scale interprocedural analysis by synergistically combining the top-down and bottom-up approaches. We formalized our approach in a generic framework and showed its effectiveness on a realistic type-state analysis. Our approach holds promise in contrast or complementation with existing techniques to scale interprocedural analysis, including those that use domain knowledge to increase summary reuse in top-down approaches, and those that use sophisticated symbolic techniques to efficiently compute and instantiate summaries in bottom-up approaches.

Acknowledgements We thank the referees for useful feedback. This work was supported by DARPA under agreement #FA8750-12-2-0020, NSF award #1253867, and EPSRC. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.

References

[1] A. Albarghouthi, R. Kumar, A. Nori, and S. Rajamani. Parallelizing top-down interprocedural analyses. In *PLDI*, 2012.

[2] T. Ball and S. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE*, 2001.

[3] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), 2011.

[4] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *POPL*, 1999.

[5] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *Formal Descriptions of Programming Concepts*. North-Holland, 1978.

[6] P. Cousot and R. Cousot. Modular static program analysis. In *CC*, 2002.

[7] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, 2011.

[8] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2), 2008.

[9] R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *IJPP*, 24(6), 1996.

[10] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori. Bottom-up shape analysis using lisf. *ACM TOPLAS*, 33(5), 2011.

[11] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, 2007.

[12] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. *ACM TOPLAS*, 32(2), 2010.

[13] R. Madhavan, G. Ramalingam, and K. Vaswani. Modular heap analysis for higher-order programs. In *SAS*, 2012.

[14] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[15] N. Rinetzky, J. Bauer, T. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.

[16] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, 2005.

[17] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2), 1996.

[18] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, 2005.

[19] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, 1981.

[20] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, 1999.

[21] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM TOPLAS*, 29(3), 2007.

[22] G. Yorsh, E. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *FOSSACS*, 2006.

[23] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, 2008.