Bottom-up Context-Sensitive Pointer Analysis for Java[⋆]

Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig
UT Austin

Abstract. This paper describes a new bottom-up, subset-based, and context-sensitive pointer analysis for Java. The main novelty of our technique is the constraint-based handling of virtual method calls and instantiation of method summaries. Since our approach generates polymorphic method summaries, it can be context-sensitive without reanalyzing the same method multiple times. We have implemented this algorithm in a tool called Scuba, and we compare it with k-CFA and k-obj algorithms on Java applications from the DaCapo and Ashes benchmarks. Our results show that the new algorithm achieves better or comparable precision to k-CFA and k-obj analyses at only a fraction of the cost.

1 Introduction

Pointer analysis is a key enabling technology underlying many program analysis, software engineering, and compiler optimization tasks. Given a pointer variable p, pointer analysis statically determines the set of all heap objects that p may point to. The result of such an analysis can be used to resolve important program analysis questions, such as whether two pointers can be aliases or whether a heap location may be referenced in a given piece of code.

While existing pointer analysis algorithms differ along many dimensions, a key feature that determines the precision of an algorithm is *context-sensitivity*. In particular, a context-sensitive analysis respects the call/return semantics of procedure calls and does not yield spurious points-to facts that arise from interprocedurally unrealizable paths. Furthermore, a context-sensitive analysis distinguishes heap objects that are allocated at the same program location, but due to different invocations of the same method. While more precise than context-insensitive ones, context-sensitive algorithms are much harder to scale to real programs, and many existing techniques use approximations of full context-sensitivity. For instance, *object-sensitive* analyses [14, 15, 23] only distinguish callsites where the receiver objects are different, and *k-CFA* analyses [22, 12] differentiate contexts by tracking callstrings up to some fixed length k.

Context-sensitivity can be achieved either by performing a *top-down* or *bottom-up* interprocedural analysis. Top-down analyses start at entry methods of a program and analyze callers before callees. In contrast, bottom-up analyses start at leaf methods of the callgraph and analyze callees before callers. Since top-down algorithms analyze every method in a known calling context, they are simpler

⋆ This work is supported in part by the Air Force Research Laboratory under agreement numbers FA8750-14-2-0270 and FA8750-15-2-0096 and in part by NSF Awards #1453386.

```
class X {
   Z f; Z g;
   void bar(Z z) {
      this.f = z;
   }
}
```

```
class Y extends X {
   void bar (Z z) {
      this.g = z;
   }
}
```

```
class A {
   X x; X y;
   void a1() {
      y = new Y();
      x = y;
      Z z = new Z();
      foo(z);
   }
   void a2() {
      x = new X();
      y = x;
      Z z = new Z();
      foo(z);
   }
   void foo(Z a) {
      x.bar(a);
      y.bar(a);
   }
}
```

Fig. 1. Code example to illustrate our approach

to design and implement, but they need to re-analyze the same method multiple times under different contexts. In contrast, bottom-up analyses generate a *polymorphic* method summary that may be used in *any* calling context to get context-sensitive results. While generating a polymorphic points-to summary is trickier than determining points-to information at a particular call site, bottom-up analyses do not need to reanalyze the same method several times[1] and have the potential to scale better. In addition, the results of a bottom-up pointer analysis are *reusable*: For instance, using a bottom-up pointer analysis, we can analyze a library just once and reuse its summary for many different clients.

In this paper, we present a bottom-up context- and field-sensitive pointer analysis algorithm for Java. A key novel feature of our approach is the constraint-based treatment of virtual method calls. Similar to many other approaches, our method starts with an imprecise callgraph and refines the callgraph as points-to facts are discovered. However, we construct the callgraph in a purely bottom-up fashion by predicating points-to facts on the possible dynamic types of the receiver object. As method summaries are propagated up the call chain, these dynamic types are resolved, thereby allowing the refutation of infeasible call targets and spurious points-to facts in a context-sensitive manner.

Another salient feature of our approach is that it can generate polymorphic method summaries without performing expensive case splits on possible aliasing patterns at call sites. In particular, a key challenge in bottom-up pointer analysis is how to generate method summaries that soundly capture the aggregate effect of a call to method m under *any* possible aliasing relation at m's call sites. Most previous techniques deal with this difficulty either by performing case splits on all possible aliasing patterns [6, 3] (which can cause exponential blow-up) or by using unification-based methods [10, 28, 13, 27] (which are imprecise compared to subset-based methods). A main advantage of our technique is that it is as precise as subset-based methods despite modeling the unknown state of the heap in a simple and uniform way. In particular, since our technique does not perform *strong updates* [2] to heap locations, it can soundly account for the callee's side effects by performing a fixed-point computation during summary instantiation.

[1] Bottom-up algorithms only re-analyze methods that belong to SCCs in the callgraph.
[2] A strong update to memory location o kills the existing points-to facts for o, while a *weak update* does not.

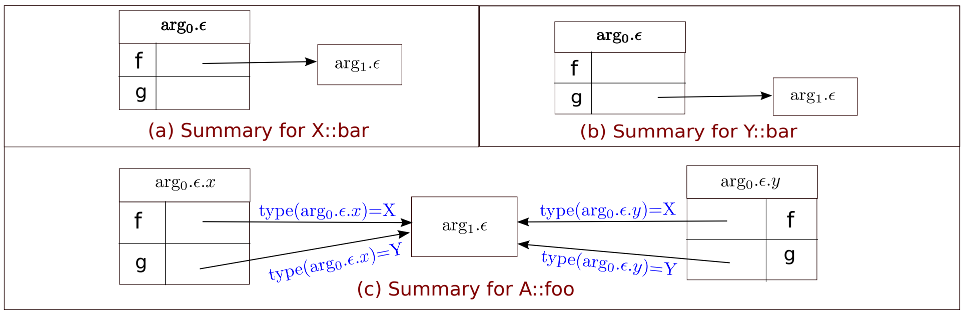

Fig. 2. Method summaries computed by our algorithm

We have implemented our algorithm in a tool called SCUBA , and we compare its scalability and precision with top-down pointer analysis algorithms implemented in CHORD [16]. Our experimental results on programs from the DaCapo and Ashes benchmark suites indicate that SCUBA achieves better or comparable precision to k-CFA and k-object-sensitive algorithm at a fraction of the cost.

To summarize, this paper makes the following contributions:

– We present a bottom-up, subset-based, and context-sensitive pointer analysis for Java. A key novelty of our approach is the handling of virtual method calls using constraint-based techniques.
– We describe a new method for summarizing and instantiating points-to facts. Unlike previous techniques, our approach does not case-split on aliasing patterns and guarantees soundness by performing fixed-point computation during summary instantiation.
– We describe an implementation of our algorithm and compare it with k-CFA and k-obj algorithms on the DaCapo and Ashes benchmarks.

2 Example

This section illustrates our approach on an example that showcases virtual method calls and the need for context-sensitivity. Consider the code shown in Figure 1, which defines classes `X`, `Y`, and `A`. Here, `Y` is a subclass of `X` and overrides `X`'s `bar` method. Class `A` has two instance variables `x` and `y` of type `X`. For concreteness, suppose we want to know whether `x.f` and `y.g` can be aliases at the end of `a1` and `a2`.

Our algorithm starts by analyzing `X::bar` and `Y::bar`, which are leaf procedures in the callgraph. The summaries for `X::bar` and `Y::bar` are shown in Figure 2 (a) and (b). We depict both method summaries as well as local points-to facts in the form of a graph, where nodes correspond to abstract heap objects and directed edges denote may-point-to relations. Our method summaries only include points-to edges that may be *added* due to an invocation of the summarized method. In particular, method summaries do not include points-to relations that already exist on method entry, and since our analysis does not apply strong updates, no points-to edges can be removed as a result of analyzing a method. Hence, the summary for a method `m` can be thought of as a bag of (symbolic) points-to edges that are introduced due to an invocation of `m`.

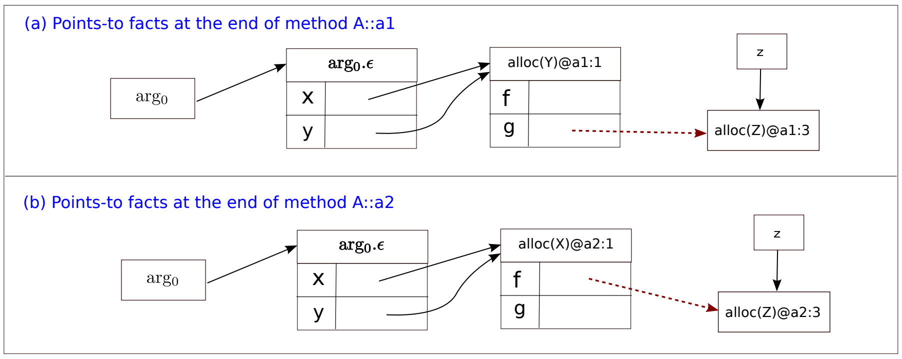
(a) Points-to facts at the end of method A::a1
arg0
arg0.ε
x
y
alloc(Y)@a1:1
f
g
z
alloc(Z)@a1:3
(b) Points-to facts at the end of method A::a2
arg0
arg0.ε
x
y
alloc(X)@a2:1
f
g
z
alloc(Z)@a2:3

Fig. 3. Dashed lines indicate points-to edges added after the `foo` call.

Now, consider the summary for `X::bar` shown in Figure 2(a). We use $\texttt{arg}_0$ to denote the `this` pointer, and $\texttt{arg}_1$, $\texttt{arg}_2$ to denote the first and second formal parameters. For parameter i, we use $\texttt{arg}_i.\epsilon$ to denote the heap object pointed to by $\texttt{arg}_i$ on method entry. Hence, according to the summary in Figure 2(a), `X::bar` adds a points-to edge via field `f` from the object pointed to by the `this` pointer to the object pointed to by `X::bar`'s first parameter. Note that, at a call site of `X::bar`, parameter `z` may have multiple points-to targets, hence, the single edge in `X::bar`'s summary may introduce multiple points-to edges at a call site. The summary for `Y::bar`, which is shown in Figure 2(b), is very similar.

Now consider method `A::foo`, whose summary is shown in Figure 2(c). Since `x` has type `X`, the call `x.bar(a)` could either invoke `X::bar` or `Y::bar`. Hence, to analyze the method call, we instantiate the summaries of both `X::bar` and `Y::bar`, but the points-to edges induced by instantiating method `T::bar` are qualified by a constraint that stipulates that the dynamic type of `x` is `T`.

As an example, consider the potential target `X::bar` of the call `x.bar(a)`. Here, the only points-to edge in the summary for `X::bar` is from the `f` field of $\texttt{arg}_0.\epsilon$ to $\texttt{arg}_1.\epsilon$. Since $\texttt{arg}_0$ of `X::bar` corresponds to `this.x` at the call site, $\texttt{arg}_0.\epsilon$ instantiates to $\texttt{arg}_0.\epsilon.x$, which denotes the memory location *pointed to* by `this.x`. On the other hand, $\texttt{arg}_1$ in `X::bar` corresponds to parameter `z` at the call site, hence $\texttt{arg}_1.\epsilon$ translates to a location with the same name, i.e., $\texttt{arg}_1.\epsilon$. Therefore, as shown in Figure 2(c), instantiating the summary of `X::bar` for this call site induces an edge from $\texttt{arg}_0.\epsilon.x$ to $\texttt{arg}_1.\epsilon$, but this edge is qualified by the constraint $\text{type}(\text{arg}_0.\epsilon.x) = \text{X}$.

Continuing with the method invocation `y.bar(a)` in the second line of `A::foo`, we again instantiate the summaries of `X::bar` and `Y::bar`, since the type of `y` is `X`. However, this time, the location named $\texttt{arg}_1.\epsilon$ used in the summaries of `X::bar` and `Y::bar` correspond to the location pointed to by `this.y`, which is denoted by $\texttt{arg}_0.\epsilon.y$ in method `foo`. Hence, as shown in Figure 2(c), the summary for `foo` includes points-to edges from the `f` and `g` fields of $\texttt{arg}_0.\epsilon.y$ to $\texttt{arg}_1.\epsilon$, again qualified by the appropriate type constraints. Observe that our approach is context-sensitive because two different invocations of `bar` induce different points-to relations. Also, even though `this.x` and `this.y` may alias at a call site of `foo`, we represent their points-to targets on method entry using two *separate* locations called $\texttt{arg}_0.\epsilon.x$ and $\texttt{arg}_0.\epsilon.y$. As we explain shortly, this approach is sound as long as we do not perform strong updates.

Now consider method `a1` defined in `A`. The *solid* black edges in Figure 3(a) denote points-to facts that hold right before the call to `foo`. In particular, the location named $\texttt{arg}_0.\epsilon$ denotes the object pointed to by the `this` pointer, and both the `x` and `y` fields of $\texttt{arg}_0.\epsilon$ point to a location called alloc(Y)@a1 : 1, which corresponds to the memory allocated at the first line of method `a1`. Throughout the paper, we use the notation alloc(T)@Ctx to denote heap objects of type `T` that are allocated in context Ctx.

We now turn to the method invocation `foo(z)` in `a1`. Here, $\texttt{arg}_1.\epsilon$ in `foo`'s summary corresponds to the location pointed to by `z` in `a1`, which is alloc(Z)@a1 : 3. On the other hand, the locations named $\texttt{arg}_0.\epsilon.x$ and $\texttt{arg}_0.\epsilon.y$ in `foo` represent the locations pointed to by `this.x` and `this.y` in `a1` respectively. Following the chain of points-to edges in Figure 3(a), we see that $\texttt{arg}_0.\epsilon.x$ and $\texttt{arg}_0.\epsilon.y$ both correspond to the location alloc(Y)@a1 : 1. Since this allocation is tagged with type `Y`, the constraints $\text{type}(\text{arg}_0.\epsilon.x) = \text{X}$ and $\text{type}(\text{arg}_0.\epsilon.y) = \text{X}$ evaluate to false, while $\text{type}(\text{arg}_0.\epsilon.x) = \text{Y}$ and $\text{type}(\text{arg}_0.\epsilon.y) = \text{Y}$ evaluate to true. Hence, instantiating `foo`'s summary induces a single points-to edge from the `g` field of alloc(Y)@a1 : 1 to alloc(Z)@a1 : 3, which is shown with the dotted edge in Figure 3(a). A similar chain of reasoning allows us to obtain the points-to facts shown in Figure 3(b) for method `a2`.

As we can see from Figure 3, the analysis determines that `x.f` and `y.g` are not aliases in either `a1` or `a2`. Observe that an analysis that is either context-insensitive or based on an imprecise callgraph would conclude otherwise. Also, even though there are two different calls to `foo` and four different calls to `bar`, observe that our algorithm analyzes each method only once.

3 Conceptual Foundations

Before describing our analysis in detail, we first describe a conceptual framework that lays the foundations of our algorithm. We describe the main ideas using a may-points-to graph, which we refer to as an *abstract heap*:

Definition 1 (Abstract heap) *An abstract heap H is a graph (N, E) where N is a set of nodes corresponding to abstract memory locations, and E is a set of directed edges between nodes labeled with field names or ϵ. An edge (o_1, o_2, f) indicates that the f field of o_1* may *point to o_2.*

Here, an abstract memory location represents either the stack location of a variable or a set of heap objects. The edge label ϵ is used to model points-to relations from stack locations to heap objects. The root nodes of an abstract heap denote locations of variables, and we write $root(H)$ to indicate the set of root nodes of H. Given two abstract heaps H_1 and H_2, $H_1 \cup H_2$ represents the abstract heap containing nodes and edges from both H_1 and H_2. Given a heap H and edges E, we write $H \backslash E$ to denote the heap that contains all nodes and edges in H except the set of edges E.

3.1 Normalization of Abstract Heaps

Given an abstract heap H, we define a *normalization operation* $\mathbb{N}(H)$, which yields a *normalized heap* H^* and a mapping ζ from nodes in H to nodes in H^*.

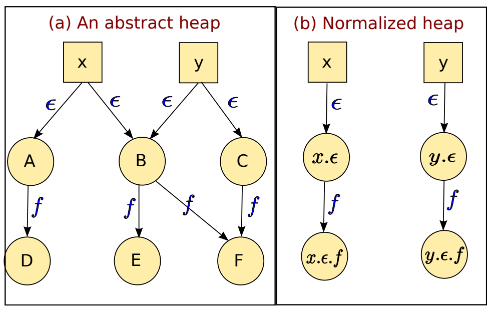

Fig. 4. An abstract heap and its normal form

Definition 2 (Normal form) *Given heap* $H = (N, E)$*,* $\mathbb{N}(H)$ *yields* normalized heap $H^* = (N^*, E^*)$ *and mapping* $\zeta : N \to 2^{N^*}$ *such that:*

1. *If* $x \in \text{root}(H)$*, then* $x \in N^*$ *and* $\zeta(x) = \{x\}$*.*
2. *If* $(o, o', f) \in E$ *and* $o^* \in \zeta(o)$*, then* $o^*.f \in N^*$*,* $o^*.f \in \zeta(o')$*, and* $(o^*, o^*.f, f) \in E^*$*.*

We use the notation $\mathbb{N}(H) = H^*$ to indicate that H^* is in normal form, and we write $Map(H, H^*) = \zeta$ to indicate that ζ maps nodes of H to nodes in H^*.

Example 1. Consider the abstract heap H shown in Figure 4(a) and its normal form H^* in Figure 4(b). Here, $Map(H, H^*)$ yields the following mapping ζ:

$$\begin{array}{lll} \zeta(x) = \{x\} & \zeta(y) = \{y\} & \zeta(A) = \{x.\epsilon\} \\ \zeta(B) = \{x.\epsilon, y.\epsilon\} & \zeta(C) = \{y.\epsilon\} & \zeta(D) = \{x.\epsilon.f\} \\ \zeta(E) = \{x.\epsilon.f, y.\epsilon.f\} & & \zeta(F) = \{x.\epsilon.f, y.\epsilon.f\} \end{array}$$

We use heap normal forms to model the heap on entry to a method. In particular, H^* corresponds to a "generic" heap representing the unknown points-to targets of object o's f field as $o.f$. While the mapping ζ from abstract heap H to its normal form H^* differs for each call site, the normalized heap for a method m is the same irrespective of calling context. Observe that no pair of abstract memory locations alias each other in a normalized heap, and every location has exactly one points-to target for a given field.

Given an abstract heap $H = (N, E)$ and its normal form $H^* = (N^*, E^*)$ such that $Map(H, H^*) = \zeta$, we define ζ^{-1} to be a mapping from N^* to 2^N such that $n \in \zeta^{-1}(n^*)$ iff $n^* \in \zeta(n)$. In general, there is a many-to-many relationship between the nodes of an abstract heap and its corresponding normal form.

Example 2. Consider the heap from Ex 1. We have:

$$\begin{array}{lll} \zeta^{-1}(x) = \{x\} & \zeta^{-1}(y) = \{y\} & \zeta^{-1}(x.\epsilon) = \{A, B\} \\ \zeta^{-1}(y.\epsilon) = \{B, C\} & \zeta^{-1}(x.\epsilon.f) = \{D, E, F\} & \zeta^{-1}(y.\epsilon.f) = \{E, F\} \end{array}$$

The mapping ζ^{-1} is important in summary-based analysis because it allows us to instantiate a method summary to a particular abstract heap at a call site. We use the notation $\overline{\zeta^{-1}}$ to denote the extension of ζ^{-1} that maps any element that is not in the domain of ζ^{-1} to itself. Given heap $H^* = (N^*, E^*)$, we also write $\zeta^{-1}(H^*)$ to denote a heap $H = (N, E)$ where $(o_1, o_2, f) \in E$ iff there exists an edge $(o_1^*, o_2^*, f) \in E^*$ such that $o_1 \in \zeta^{-1}(o_1^*)$ and $o_2 \in \zeta^{-1}(o_2^*)$

Definition 3 (Default edge) *We say an edge* (n, n', f) *is a* default edge *of an abstract heap if* $n' = n.f$*.*

Given a heap H, we write $default(H)$ to denote the set of default edges in H.

```
SumAnalyze($H, S, A$):
    input: abstract heap $H$, code $S$, intraprocedural analysis $A$
    output: abstract heap $H'$

    (1)  let $H^* = \text{NormalForm}(H)$
    (2)  let $H'^* = \text{Analyze}(H^*, S, A)$
    (3)  let $\Delta = H'^* \backslash \text{default}(H'^*)$
    (4)  let $\zeta_0 = \text{Map}(H, H^*)$
    (5)  let $H' = H$;   let $\zeta = \zeta_0$
    (6)  do {
    (7)          $\zeta_0 = \zeta$
    (8)          $H' = \overline{\zeta^{-1}}(\Delta) \cup H'$
    (9)          $\zeta = \text{Map}(H', H^*)$
    (10) }
    (11) while($\zeta \neq \zeta_0$)
    (12) return $H'$
```

Fig. 5. Basic structure of summary-based analysis

3.2 Summary-Based Pointer Analysis

We now explain the basic idea underlying our summary-based analysis, assuming a family of pointer analyses that are sound and weakly-updating. Given code snippet S, an abstract heap H, and pointer analysis A, we write $H' = \mathit{Analyze}(H, S, A)$ to indicate that, if statement S is executed in an environment that satisfies abstract heap H, then analyzing code S using pointer analysis A yields a heap H' which conservatively models the concrete heap after S.

The basic structure of our summary-based pointer analysis is shown in Figure 5. The algorithm SumAnalyze takes as input an abstract heap H, a code snippet S, and a weakly-updating pointer analysis A, and works as follows. Line (1) constructs the normalized heap H^* representing the unknown state of the heap before executing S, and line (2) analyzes S without making any assumptions about points-to facts that hold before S. Line (3) generates a polymorphic points-to summary Δ which characterizes side effects of code S. Lines (4)-(11) instantiate the summary Δ by performing a fixed-point computation. Finally, H' at line (12) models the state of the heap after S when S is executed in an environment satisfying H.

Before discussing details, let us first understand in what way this algorithm is "summary-based". Since H^* can be constructed in a context-independent manner, we can analyze S in isolation and compute its side effects without knowing the points-to facts that hold before S. Hence, lines (1)-(3) in Figure 5 correspond to *summary generation.* On the other hand, lines (4)-(11) perform *summary instantiation* by computing the context-specific mapping ζ_0 and by adding all points-to edges that represent S's side effects.

The most involved part of the above algorithm is the fixed-point computation at lines (4)-(11). Intuitively, the algorithm maps each edge in the summary to a set of edges at the callsite by using the mapping $\overline{\zeta^{-1}}$ and adds these edges to the initial abstract heap H. However, as new edges are added to H, the

$$
\begin{array}{lll}
\text{Program } P & := & C^+ \\
\text{ClassDecl } C & := & \text{class } T_1 \text{ [extends } T_2]? \ \{F^*; M^*\} \\
\text{FieldDecl } F & := & T \text{ fld\_name;} \\
\text{MethodDecl } M & := & m(T_0\ v_0,\ \ldots,\ T_k\ v_k) = \{V^*; I;\} \\
\text{VarDecl } V & := & T \text{ var\_name;} \\
\text{Instruction } I & := & v_1 = v_2 \mid v_1 = v_2.f \mid v_1.f = v_2 \mid v = \text{new}^\rho\ T \\
 & & \mid \text{if}(*)\ I_1 \text{ else } I_2 \mid I_1; I_2 \mid m^\rho @T(v_1, \ldots, v_n) \mid v_0.m^\rho(v_1, \ldots, v_n)
\end{array}
$$

Fig. 6. Core language used for our formalization

mapping ζ must to be recomputed since locations used in the summary may map to new additional locations after the summary has been applied. Hence, H' is recomputed until the mapping ζ from H' to H^* stabilizes. It is easy to see that SumAnalyze is sound because (i) the underlying pointer analysis does not apply strong updates to memory locations, and (ii) the summary is applied to a fixed-point. In particular, observe that H' overapproximates H and $\{H'\}S\{H'\}$ is a valid Hoare triple [8].

The reader may wonder why it is necessary to re-apply the summary until the mapping ζ reaches a fixed point in Figure 5. This is necessary because our summary Δ encodes all possible side effects of code snippet S, but not the *order* in which they happen. Hence, while the fixed point computation is required for soundness, an immediate corollary is that the procedure SumAnalyze can be less precise than *Analyze* if the underlying pointer analysis A is flow-sensitive.

4 Formalization of Algorithm

While the previous section describes core ideas of the analysis, it omits many important details. In this section, we describe our full algorithm using the core object-oriented language of Figure 6. Here, a program consists of one or more class declarations C, which defines a class T_1 with optional superclass T_2. Instructions include assignments, loads, stores, heap allocations (marked with a unique program point ρ), non-deterministic conditionals, sequences, static method calls $m^\rho @T(...)$ (also marked with program point ρ), and virtual method calls $v_0.m^\rho(\ldots)$. We assume that the first argument of a method is always the `this` pointer, and if a class T inherits a method m from its superclass T', then T' also contains a definition of m with the same implementation.

4.1 Abstract Domains

Figure 7 shows the abstract domains that we need for describing our algorithm. Since our analysis is bottom-up, we differentiate between two kinds of heap objects o: *Access paths* of the form $a_i.\eta$ represent (caller-allocated) unknown heap objects reachable through the i'th argument, whereas objects named alloc$(T)@\boldsymbol{\rho}$ represent heap objects of type T that are allocated either in the currently analyzed method or in a transitive callee. Specifically, $a_i.f_1...f_n$ denotes the unknown locations reachable on method entry through a series of field accesses $f_1...f_n$ from the i'th argument. We use the notation $a_i.(f_1...f_n)^\star$ to denote all

(Field selector)	$\eta : f \mid \eta.f \mid \eta^\star$
(Heap obj)	$o : a_i.\eta \mid \mathsf{alloc}(T)@\boldsymbol{\rho}$
(Abstract loc)	$\pi : o \mid a_i \mid v_i@\boldsymbol{\rho}$
(Pts set)	$\theta : o \to \phi$
(Abstract heap)	$\Gamma : (\pi \times f) \to \theta$
(Summaries)	$\Upsilon : (T \times M) \to \Gamma$

Fig. 7. Abstract domains used in our analysis

unknown locations reachable from a_i through any combination of field selectors $f_1, ..., f_n$. For instance, $a_0.(f.g)^\star$ represents the infinite set of access paths $a_0.f$, $a_0.g$, $a_0.f.f$, $a_0.g.f$ and so on. As we will see in Section 4.3, access paths in our analysis correspond to node labels of the normalized heap from Section 3.

Abstract memory locations π are either heap objects o or stack locations. In particular, a_i denotes the stack location of the i'th argument, and $v_i@\boldsymbol{\rho}$ denotes the location of local variable v_i under *context* $\boldsymbol{\rho}$. We represent calling contexts using a sequence of program points $\rho_1, \ldots, \rho_n$, where each ρ_i corresponds to some call or allocation site. For instance, a memory location named $\mathsf{alloc}(T)@\rho_1\rho_2$ corresponds to a heap object allocated at program point ρ_2 of some method m which is invoked at call site ρ_1. Similarly, $v@\rho_1$ denotes the local variable v of some method m when m is invoked at callsite ρ_1. Since our analysis builds contexts in a bottom-up way, local variables declared in the currently analyzed method m do not have any context information; hence, we abbreviate the locations of locals in the current method as v_i.

Definition 4 *(***Argument-derived location***) We say location π is* derived from an argument*, written* $\arg(\pi)$*, if it is either (i) a_i representing the location of the i'th argument or (ii) a heap object represented with an access path $a_i.\eta$.*

An *abstract heap Γ* maps each field f of location π to a points-to set. A (guarded) points-to set θ is a set of pairs (o, ϕ) where o is a heap object and ϕ is a constraint. As discussed in Section 2, we use constraints to predicate points-to facts on dynamic types of receivers. Constraints ϕ belong to the theory of equality with uninterpreted functions, defined according to the following grammar:

$$\begin{array}{lll}
\text{Function } f & := & \mathsf{pts} \mid \mathsf{alloc} \mid \varsigma_i \\
\text{Term } t & := & c \mid v \mid f(\boldsymbol{t}) \\
\text{Formula } \phi & := & \top \mid \bot \mid \mathsf{type}(t) = T \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2
\end{array}$$

Here, terms include constants c, variables v, and function applications $f(\boldsymbol{v})$ where f is either the binary function $\mathsf{pts}, \mathsf{alloc}$, or an n-ary function drawn from $\varsigma_1, \ldots, \varsigma_k$. Formulas are composed of $\top$ (true), $\bot$ (false), and conjunctions and disjunctions of equality constraints of the form $\mathsf{type}(t) = T$, where T is a type constant. In addition to the usual function and equality axioms, the alloc and type functions obey the additional axiom $\forall x.\ \mathsf{type}(\mathsf{alloc}(T, \boldsymbol{\rho})) = T$, which states that the type of an allocation of type T is T.

Since we will convert heap objects to terms, we define an operation called *lift*(π), abbreviated $\overline{\pi}$, as follows:

$$\overline{a_i} = a_i \quad \overline{\mathsf{alloc}(T)@\boldsymbol{\rho}} = \mathsf{alloc}(T, \boldsymbol{\rho}) \quad \overline{\pi.f} = \mathsf{pts}(\overline{\pi}, f) \quad \overline{\pi.(\boldsymbol{f})^\star} = \varsigma_i(\overline{\pi}, \boldsymbol{f}) \ (\varsigma_i \text{ fresh})$$

We also assume an operation $\mathit{lift}^{-1}(t)$ which is the inverse of $\mathit{lift}(\pi)$. Given a term t, $\mathit{lift}^{-1}(t)$ yields an abstract memory location representation of that term.

Example 3. The constraint $\mathrm{type}(\overline{a.\epsilon.f}) = A \wedge \mathrm{type}(\overline{a.\epsilon.f}) = B$ is unsatisfiable since the dynamic type of $a.\epsilon.f$ cannot simultaneously be A and B. But the constraint $\mathrm{type}(\overline{a.\epsilon.g^\star}) = A \wedge \mathrm{type}(\overline{a.\epsilon.g^\star}) = B$, which translates to $\mathrm{type}(\varsigma_1(\mathrm{pts}(a, \epsilon), g)) = A \wedge \mathrm{type}(\varsigma_2(\mathrm{pts}(a, \epsilon), g)) = B$, is satisfiable because two distinct occurrences of $a.\epsilon.g^\star$ may correspond to different objects (e.g., $a.\epsilon.g$ and $a.\epsilon.g.g$).

We now define a function *has_type*(θ, T) which generates a constraint that evaluates to true if *some* element in points-to set θ can have dynamic type T:

Definition 5 has_type(θ, T) *Given a points-to set θ, the function* has_type(θ, T) *yields the following constraint:*

$$\bigvee_{(\pi_i, \phi_i) \in \theta} (((\mathrm{type})(\overline{\pi_i}) = T) \wedge \phi_i)$$

Now, going back to Figure 7, an environment Υ maps each method M in class T to its corresponding summary, which is an abstract heap Γ summarizing M's side effects. Applying the summary at a call site allows us to determine points-to relations of the caller without having to reanalyze the callee. In addition, our method summaries include points-to information for locals in the summarized method. In particular, this design choice allows us to determine points-to sets for *all* program variables without employing a separate top-down pass.

4.2 Operations on Abstract Domains

In this section, we describe some operations on abstract domains that simplify the description of our algorithm. Since our algorithm constructs the initial heap on method entry in a demand-driven way, we first define *default targets* for argument-derived locations:

Definition 6 (Default target) *Given an argument-derived location π and a field f, the* default target *of the f field of π, written* $\mathrm{def}(\pi, f)$*, is given as follows:*

$$\mathrm{def}(\pi, f) = \begin{cases} \pi & \text{if } \pi = \pi'.(\boldsymbol{f})^\star \text{ and } f \in \boldsymbol{f} \quad (1) \\ \pi'.(f.\boldsymbol{g})^\star & \text{if } \pi = \pi'.f.\boldsymbol{g} \quad (2) \\ \pi.f & \text{otherwise} \quad (3) \end{cases}$$

In other words, if field f is *not* part of a recursive field cycle (line 3), then the default target for field f of an argument derived location π is $\pi.f$, just like the normal form heaps from Section 3. However, if f is part of a recursive field cycle, then our analysis collapses this cycle into a single abstract memory location (lines 1-2). For example, $\mathrm{def}(a.\mathrm{next}, \mathrm{next}) = a.\mathrm{next}^\star$ (line 2), and $\mathrm{def}(a.\mathrm{next}^\star, \mathrm{next}) = a.\mathrm{next}^\star$ (line 1). The summarization of recursive field cycles into access paths of the form $a.\boldsymbol{f}^\star$ is needed to ensure termination of the fixed-point computation performed by our algorithm.

Next, we define a *field lookup* operation on abstract heaps Γ:

Definition 7 (Field look-up) *Given heap Γ, field f, and location π, the* field lookup *operation $\Gamma[\pi, f]$ retrieves the points-to target for π's f field:*

$$\Gamma[\pi, f] = \begin{cases} \Gamma(\pi, f) \cup \{(\text{def}(\pi, f), \top)\} & \text{if } \arg(\pi) \\ \Gamma(\pi, f) & \text{otherwise} \end{cases}$$

Since our algorithm does not explicitly add default edges to the abstract heap, $\Gamma[\pi, f]$ always yields $\mathit{def}(\pi, f)$ as part of the points-to set of $\pi.f$ if π is an argument derived location. Now, since our analysis performs weak updates, we need to merge two points-to sets using the following join operator:

Definition 8 (Join $\sqcup$ of points-to sets θ_1, θ_2)

$$(\theta_1 \sqcup \theta_2)(o) = \begin{cases} \theta_1(o) \vee \theta_2(o) & \text{if } o \in \mathit{dom}(\theta_1) \cap \mathit{dom}(\theta_2) \\ \theta_1(o) & \text{if } o \in \mathit{dom}(\theta_1) \text{ and } o \notin \mathit{dom}(\theta_2) \\ \theta_2(o) & \text{if } o \in \mathit{dom}(\theta_2) \text{ and } o \notin \mathit{dom}(\theta_1) \end{cases}$$

Observe that if an object o is in both points to sets θ_1 and θ_2, we take the disjunction of the constraints associated with o. We also extend this join operator to abstract heaps in the expected way. That is, for a location π and field f, $(\Gamma_1 \sqcup \Gamma_2)(\pi, f)$ yields $\Gamma_1(\pi, f) \sqcup \Gamma_2(\pi, f)$. In our analysis, we sometimes need to predicate points-to information on constraints. For this purpose, an operation $\theta \downarrow \phi$ conjoins ϕ with every constraint in θ:

Definition 9 (Projection of θ on ϕ) $\theta \downarrow \phi = \{(\pi_i, \phi_i \wedge \phi) \mid (\pi_i, \phi_i) \in \theta\}$

Finally, we extend the field lookup operation on points-to sets as follows:

Definition 10 (Field lookup for pts-to set) $\Gamma[\theta, f] = \bigsqcup_{(\pi_i, \phi_i) \in \theta} \Gamma[\pi_i, f] \downarrow \phi_i$

That is, $\Gamma[\theta, f]$ includes the points-to target of every element in θ under the appropriate constraints.

4.3 Intraprocedural Analysis

Figure 8 describes the intraprocedural analysis using judgements of the form $\Upsilon, \Gamma \vdash I : \Gamma'$. which indicates that, if statement I is executed in an environment that satisfies summary environment Υ and abstract heap Γ, we obtain a new heap Γ'. Since the analysis is (partially) flow-sensitive, we distinguish between heaps Γ, Γ' before and after executing I.

Rule (1) in Figure 8 describes the analysis of assignments. Although our analysis only performs weak updates to heap objects, it does apply strong updates to variables. Hence, rule (1) updates the points-to set for (v_1, ϵ) to be $\Gamma[v_2, \epsilon]$, where the lookup operation is defined in Section 4.2. Rule (2) for memory allocations $v = \text{new}^\rho\ T$ introduces a new abstract location named $\text{alloc}(T)@\rho$ and assigns v_1 to this singleton.

Rule (3) concerns loads of the form $v_1 = v_2.f$. Here, we first look up the points-to set θ of v_2 and then use $\Gamma[\theta, f]$ to retrieve the targets of memory locations in θ. Finally, since our analysis applies strong updates to variables, we override v_1's existing targets and change its points-to set to $\Gamma[\theta, f]$.

$$(1)\ \frac{\varGamma' = \varGamma[(v_1,\epsilon) \leftarrow \varGamma[v_2,\epsilon]]}{\varUpsilon, \varGamma \vdash v_1 = v_2 : \varGamma'} \qquad (2)\ \frac{\varGamma' = \varGamma[v \leftarrow \{(\text{alloc}(T)@\rho, \top)\}]}{\varUpsilon, \varGamma \vdash v = \text{new}^{\rho}\ T : \varGamma'}$$

$$(3)\ \frac{\begin{array}{c}\theta = \varGamma[v_2,\epsilon] \\ \varGamma' = \varGamma[(v_1,\epsilon) \leftarrow \varGamma[\theta, f]]\end{array}}{\varUpsilon, \varGamma \vdash v_1 = v_2.f : \varGamma'}$$

$$(4)\ \frac{\begin{array}{c}\theta_1 = \varGamma[v_1,\epsilon] \quad \theta_2 = \varGamma[v_2,\epsilon] \\ \varGamma' = \varGamma[(o_i, f) \leftarrow (\varGamma(o_i, f) \sqcup (\theta_2 \downarrow \phi_i)) \mid (o_i, \phi_i) \in \theta_1]\end{array}}{\varUpsilon, \varGamma \vdash v_1.f = v_2 : \varGamma'}$$

$$(5)\ \frac{\begin{array}{c}\varUpsilon, \varGamma \vdash I_1 : \varGamma_1 \\ \varUpsilon, \varGamma \vdash I_2 : \varGamma_2\end{array}}{\varUpsilon, \varGamma \vdash \text{if}(*)\ I_1 \text{ else } I_2 : \varGamma_1 \sqcup \varGamma_2} \qquad (6)\ \frac{\begin{array}{c}\varUpsilon, \varGamma \vdash I_1 : \varGamma_1 \\ \varUpsilon, \varGamma_1 \vdash I_2 : \varGamma_2\end{array}}{\varUpsilon, \varGamma \vdash I_1; I_2 : \varGamma_2}$$

Fig. 8. Rules for intraprocedural analysis

$$\frac{}{\mathcal{M}, \varGamma, \rho \vdash \mathit{inst\_loc}(a_i) : \{\mathcal{M}(a_i), \top\}} \quad \frac{\mathcal{M}, \varGamma, \rho \vdash \mathit{inst\_loc}(\pi) : \theta}{\mathcal{M}, \varGamma, \rho \vdash \mathit{inst\_loc}(\pi.f) : \varGamma[\theta, f]}$$

$$\frac{\begin{array}{c}\mathcal{M}, \varGamma, \rho \vdash \mathit{inst\_loc}(\pi) : \theta_0 \\ \theta_i = \bigsqcup_{1 \le j \le n} \varGamma[\theta_{i-1}, f_j]\end{array}}{\mathcal{M}, \varGamma, \rho \vdash \mathit{inst\_loc}(\pi.(f_1 ... f_n)^{\star} : \bigsqcup_{i \ge 0} \theta_i)} \quad \frac{\boldsymbol{\rho}_{\text{new}} = \text{new\_ctx}(\rho, \boldsymbol{\rho})}{\mathcal{M}, \varGamma, \rho \vdash \mathit{inst\_loc}(v@\boldsymbol{\rho}) : \{(v@\boldsymbol{\rho}_{\text{new}}, \top)\}}$$

$$\frac{\boldsymbol{\rho}_{\text{new}} = \text{new\_ctx}(\rho, \boldsymbol{\rho})}{\mathcal{M}, \varGamma, \rho \vdash \mathit{inst\_loc}(\text{alloc}(T)@\boldsymbol{\rho}) : \{(\text{alloc}(T)@\boldsymbol{\rho}_{\text{new}}, \top)\}}$$

Fig. 9. Rules for instantiating memory locations

Rule (4) analyzes stores $v_1.f = v_2$. First, we look up the points-to sets θ_1 and θ_2 of v_1 and v_2. Now, the store operation will update every location o_i such that $(o_i, \phi_i) \in \theta_1$. However, since we apply only weak updates to heap objects, we preserve the existing points-to targets $\varGamma(o_i, f)$ for each o_i. Furthermore, since v_1 points to o_i under constraint ϕ_i, o_i points to elements in θ_2 only when ϕ_i holds. Hence, the new points-to set for o_i is given by $(\theta_2 \downarrow \phi_i) \sqcup \varGamma(o_i, f)$ where the $\downarrow$ operation is given by Definition 9. Since rules (5) and (6) for if statements and sequencing and are fairly standard, we do not describe them in detail.

4.4 Interprocedural Analysis

We now describe the instantiation of summaries at call sites. Since a key part of summary instantiation is constructing the mapping from locations in the summary to those at the call site, we first start with the rules in Figure 9 which describe the instantiation of memory locations. Informally, the rules of Figure 9 construct the mapping ζ^{-1} from Section 3. More formally, they produce judgements of the form $\mathcal{M}, \varGamma, \rho \vdash \mathit{inst\_loc}(\pi) : \theta$ where $\mathcal{M}$ maps formals to actuals, and $\varGamma$ and ρ are the abstract heap and program point associated with a call site respectively. The meaning of the judgement is that, under $\mathcal{M}, \varGamma, \rho$, location π used in the summary maps to (guarded) location set θ.

The first rule Figure 9 maps formal parameter a_i to the actual $\mathcal{M}(a_i)$. The second rule instantiates argument-derived locations of the form $\pi.f$. For this

$$\frac{\begin{array}{c}\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\mathit{lift}^{-1}(t)) : \theta \\ \phi = \mathit{has\_type}(\theta, T)\end{array}}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\text{type}(t) = T) : \phi} \qquad \frac{\begin{array}{c}\star \in \{\wedge, \vee\} \\ \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\phi_1) : \phi_1' \\ \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\phi_2) : \phi_2'\end{array}}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\phi_1 \star \phi_2) : \phi_1' \star \phi_2'}$$

Fig. 10. Rules for instantiating constraints

$$\frac{\begin{array}{c}\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\pi_1) : \theta_1 \ldots \mathit{inst\_loc}(\pi_n) : \theta_n \\ \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\phi_1) : \phi_1' \ldots \mathit{inst}_\phi(\pi_n) : \phi_n'\end{array}}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_pts}(\{(\pi_1, \phi_1), \ldots, (\pi_n, \phi_n)\}) : \sqcup_i (\theta_i \downarrow \phi_i)}$$

$$\frac{\begin{array}{c}\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\pi) : \theta' \\ \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_pts}(\theta) : \theta'' \\ \mathit{\Delta} = [(\pi_i, f) \leftarrow (\theta'' \downarrow \phi_i) \mid (\pi_i, \phi_i) \in \theta']\end{array}}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_partial\_heap}(\pi, f, \theta) : \mathit{\Delta}} \qquad \frac{\begin{array}{c}\mathit{\Delta} = \{(\pi_1, f_{11}) \mapsto \theta_{11}, \ldots, (\pi_n, f_{nk}) \mapsto \theta_{nk}\} \\ \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_partial\_heap}(\pi_1, f_{11}, \theta_{11}) : \mathit{\Delta}_{11} \\ \ldots \\ \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_partial\_heap}(\pi_n, f_{nk}, \theta_{nk}) : \mathit{\Delta}_{nk}\end{array}}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_heap}(\mathit{\Delta}) : \sqcup_{ij} \mathit{\Delta}_{ij}}$$

Fig. 11. Rules for instantiating summaries

purpose, we first instantiate prefix π to location set θ, then retrieve the points-to targets of the f field of locations in θ. The third rule instantiates access paths of the form $\pi.(f_1 \ldots f_n)^\star$. As in the previous rule, we first instantiate prefix π, which yields θ_0. Now, recall that the access path $\pi.(f_1 \ldots f_n)^\star$ describes the infinite set of access paths given by the regular expression $\pi.(f_1 + \ldots + f_n)^*$. Hence, to instantiate $\pi.(f_1 \ldots f_n)^\star$, we need to compute all locations that are reachable from θ_0 using any combination of field selectors $f_1, \ldots, f_n$. The resulting set $\bigsqcup_{i \geq 0} \theta_i$ is the reflexive transitive closure of θ_0 with respect to fields $f_1, \ldots, f_n$.

The last two rules in Figure 9 describe the instantiation of allocations and local variables. Both rules use a helper new_ctx method defined as follows:

$$\text{new\_ctx}(\rho, \boldsymbol{\rho}) = \begin{cases} \rho, \boldsymbol{\rho} \text{ if } |\boldsymbol{\rho}| \leq k \\ \boldsymbol{\rho} \quad \text{otherwise} \end{cases}$$

In other words, new_ctx appends call site ρ to context $\boldsymbol{\rho}$ if the length of $\boldsymbol{\rho}$ is less than some pre-determined threshold k. Hence, our analysis uses a k-CFA style context-sensitive heap abstraction where the value of k is configurable.

We now turn to the instantiation of constraints, summarized in Figure 10. To translate a constraint type$(t) = T$, we map t to its corresponding location set θ by using *inst_loc*. The function $\mathit{has\_type}(\theta, T)$ then yields the condition under which some element in θ has dynamic type T (recall Definition 5).

Using these ingredients, Figure 11 shows how to instantiate an abstract heap $\mathit{\Delta}$. Given location π_i and field f_j from the callee heap, *inst_partial_heap* instantiates all points-to edges from π_i labeled with f_j and yields instantiated partial heap $\mathit{\Delta}_{ij}$. The instantiation of $\mathit{\Delta}$ is obtained by taking the join over all $\mathit{\Delta}_{ij}$'s.

Finally, Figure 12 describes the analysis of method calls. First, consider a static call to m with corresponding summary $\mathit{\Delta}$ (rule (1)). To analyze it, we construct the formal-to-actual mapping $\mathcal{M}$ and perform a least fixed-point computation that instantiates the summarized heap $\mathit{\Delta}$ until we obtain an overapproximation of the set $\mathit{\Delta}'$ of m's side effects. The abstract heap after the method call is obtained by taking the union of the existing heap $\mathit{\Gamma}$ and the new "edges" $\mathit{\Delta}'$.

$$(1)\ \frac{\begin{array}{c}\Upsilon(T, m) = \Delta \\ \mathcal{M} = [a_1 \mapsto v_1, \ldots, a_n \mapsto v_n] \\ \mathcal{M}, \Gamma \sqcup \Delta', \rho \vdash \mathit{inst\_heap}(\Delta) : \Delta'\end{array}}{\Upsilon, \Gamma \vdash m^{\rho}@T(v_1, \ldots, v_n) : \Gamma \sqcup \Delta'} \qquad (2)\ \frac{\begin{array}{c}\text{static\_type}(v_0) = T \quad T_1 <: T, \ldots, T_n <: T \\ \phi_i = \text{has\_type}(\Gamma(v_0), T_i) \\ \Upsilon, \Gamma \vdash m^{\rho}@T_1(v_0, \ldots, v_k) : \Gamma_1 \\ \ldots \\ \Upsilon, \Gamma \vdash m^{\rho}@T_n(v_0, \ldots, v_k) : \Gamma_n\end{array}}{\Upsilon, \Gamma \vdash v_0.m^{\rho}(v_1, \ldots, v_k) : \sqcup_i(\Gamma_i \downarrow \phi_i)}$$

Fig. 12. Analysis of method calls

The second rule of Figure 12 describes the analysis of virtual calls. Here, we first overapproximate the call's targets and then use the previous rule for analyzing static method calls to obtain heap Γ_i assuming the called method is $T_i :: m$. Now, since the target of the virtual call is $T_i :: m$ under the assumption that v_0 has dynamic type T_i, we generate the constraint $\phi_i = \text{has\_type}(\Gamma(v_0), T_i)$. Then, the final abstract heap after the call is obtained as $\sqcup_i(\Gamma_i \downarrow \phi_i)$.

5 Implementation and Extensions

We implemented the proposed algorithm in a tool called Scuba (`http://www.cs.utexas.edu/~yufeng/scuba.html`) which is built on top of Chord [16]. Scuba performs analysis on the Quad representation of Joeq [25] and obtains an initial callgraph by running the context-insensitive pointer analysis implemented in Chord. It also uses the Z3 SMT solver [5] for checking satisfiability of constraints.

Our implementation performs several optimizations over the core algorithm described here. One optimization is memoizing instantiation results. For example, consider an access path $a_0.\epsilon.f.g.h$ used in m's summary. Since this access path may be instantiated many times when analyzing a call site of m, our analysis maintains a cache per callsite that records instantiation results. A second optimization concerns constraint generation for virtual method calls. Consider a call $v.m(...)$ where the static type of v is T_0. Further, suppose T_0 has a large number of subclasses $T_1, \ldots, T_n$ all of which inherit T_0's m method except T_n. Assuming v points to heap object o, we need to introduce constraints of the form $\bigvee_{0 \leq i < n} \text{type}(o) = T_i$. Since such constraints can be very large, our implementation allows subtyping constraints and translates them to linear inequalities by assigning integer identifiers to types in reverse topological order.

6 Evaluation

We evaluated Scuba on ten large Java applications from the DaCapo and Ashes benchmark suites [2, 1]. These applications range between 92615 and 227507 lines of statements in the Quad IR and contain between 4634 and 9653 reachable methods. To evaluate our algorithm, we compared Scuba against the k-CFA and k-object-sensitive algorithms implemented in Chord [16]. All analyses are Anderson-style flow-insensitive pointer analyses that allow customizing the value of k. Chord also allows customizing the context-sensitivity associated with heap objects using a value h. For example, a 2-obj-1-h analysis uses the abstract allocation site of the receiver as a context up to depth 2, and it also differentiates heap allocations with different contexts up to depth 2.

Benchmark	# methods	# statements	CIPA	2-CFA	2-obj	Scuba-2	Scuba-3	Scuba-4
antlr	5411	112831	29	1380	355	30	37	50
hedc	4967	103066	25	1337	446	25	28	31
avrora	5230	104948	24	1328	336	53	55	59
polyglot	4634	92615	21	608	284	14	17	16
toba-s	4702	101501	24	930	299	15	18	17
weblech	5816	115937	27	1657	506	41	35	39
xalan	6405	131332	27	1100	3600	173	180	211
hsqldb	6767	137947	33	2474	1348	63	65	77
luindex	6157	127451	28	2525	532	93	147	315
sunflow	9653	227507	66	T/O	T/O	411	405	521

Table 1. Analysis time in seconds. Runs exceeding the time-limit of 3600s are labeled T/O.

Benchmark	Alias pairs	CIPA	2-CFA	2-obj	Scuba-2	Scuba-3	Scuba-4
antlr	6839	0	1082	2785	3219	3231	3231
hedc	1728	0	725	962	1025	1055	1055
avrora	1182	0	406	687	738	741	745
polyglot	165	0	59	103	128	128	128
toba-s	5118	0	3354	3350	3589	3589	3595
weblech	1417	0	662	654	656	681	763
xalan	124	0	24	24	24	24	24
hsqldb	5254	0	2746	2724	3318	3318	3426
luindex	4649	0	1326	1420	1353	1400	1400
sunflow	4303	0	N/A	N/A	339	339	339

Table 2. May-alias results. The bigger the better.

Before describing the results, we first explain how our algorithm relates to k-CFA and k-obj-sensitive analyses. Similar to k-CFA, Scuba uses *call sites* rather than *receiver objects* as contexts. However, unlike k-CFA, we do not impose a fixed value of k since our algorithm instantiates method summaries differently for each call site. On the other hand, we can customize the context-sensitivity associated with heap objects by varying the parameter k used in the new_ctx function from Section 4.4. Hence, for a given value of k in the new_ctx function, Scuba is *roughly* comparable to a ∞-CFA-k-h analysis. In what follows, we write Scuba-k to refer to different configurations of Scuba for different values of parameter k used in the new_ctx function from Section 4.4.

Table 1 compares the running times of Scuba-k (for $2 \leq k \leq 4$) against the context-insensitive(CIPA), 2-CFA, and 2-obj-sensitive analyses using h value of 1. While we also tried comparing Scuba against k-CFA and k-obj-sensitive analyses for $k = 3$ and $k = 4$, these analyses did not complete within an hour for most of the benchmarks; hence, we do not include these results in Table 1. We also note that the running times shown in Figure 1 include the analysis time for libraries (e.g., JDK, Swing, Sun Security) as well as the application code (i.e., we did not use manually provided stub methods for analyzing libraries). As the results in Table 1 show, Scuba-k is significantly faster compared to k-CFA and k-obj analyses.

To compare the precision of Scuba against k-CFA and k-obj analyses, we used two typical pointer analysis clients, namely *may-alias* and *downcast* analyses. Table 2 compares the precision of different analysis configurations in the

Benchmark	# downcasts	CIPA	2-CFA	2-obj	Scuba-2	Scuba-3	Scuba-4
antlr	76	18	21	48	45	52	52
hedc	28	5	6	23	23	23	23
avrora	21	0	0	8	15	18	18
polyglot	13	2	3	9	13	13	13
toba-s	59	23	23	34	37	37	37
weblech	48	16	23	33	38	38	38
xalan	14	7	10	13	13	13	13
hsqldb	45	22	24	32	28	28	35
luindex	213	104	106	180	177	177	177
sunflow	81	19	N/A	N/A	25	52	52

Table 3. Downcast results. Bigger numbers indicate higher precision.

context of the may-alias client. The column labeled "Alias pairs" shows the number of variable pairs that are queried by the may-alias client. To generate these pairs, we first ran a context-insensitive pointer analysis to identify potential may-alias in the application code. From these variables, we further filtered those pairs that are "obviously" aliases (e.g., due to a direct assignment). Columns 3-11 in Table 2 show the number of variables proven not to be aliases according to each analysis configuration. Hence, a higher number indicates better precision. Observe that every configuration of Scuba-k yields better precision *on average* compared to 2-CFA and 2-obj analyses [3].

Table 3 shows the precision of each analysis in the context of the downcast client. Here, the second column labeled "# downcasts" shows the total number of downcasts in the application, and the subsequent columns show the number of downcasts that can be proven safe. The number of downcasts shown in Table 3 only include the downcasts performed in the application code rather than in external libraries [4]. According to the results shown in Table 3, Scuba-k has better precision on average compared to both 2-CFA and 2-obj.

7 Related Work

Top-down pointer analysis. Most existing context-sensitive pointer analysis algorithms are top-down [12, 14, 26, 24, 7]. Generally speaking, top-down context-sensitivity comes in two flavors: *call-site sensitivity* [22] (k-CFA) and *k-object-sensitivity* [14]. Specifically, CFA-based algorithms use method call sites as the context, while object-sensitive approaches use the receiver's abstract allocation site. The recent work described in [9] has proposed selectively combining k-CFA and k-object sensitivity to achieve better precision. Several papers have used BDD-based methods for top-down context-sensitive pointer analysis [26, 31, 11]. The use of BDDs exposes commonalities between different contexts and allows the technique to scale better. The Chord framework [16] used in our experimental evaluation also uses BDDs to exploit equivalences between calling contexts.

Bottom-up pointer analysis. While not as widely-studied as top-down algorithms, several papers propose bottom-up pointer analysis. However, many

[3] We manually inspected a randomly selected subset of the may-alias queries that could only be discharged by Scuba and confirmed that these are not false negatives.

[4] Since most benchmarks use the same libraries, this strategy avoids double counting. Furthermore, clients are typically interested in finding defects in the application.

of these approaches are unification-based [10, 18, 13]. The algorithm proposed in [28] is also bottom-up but uses a combined equality- and subset-based approach. By contrast, our algorithm is subset-based and therefore more precise.

The algorithm described in [4] presents a subset-based, partially bottom-up pointer analysis for C. Unlike our approach, it incorporates both top-down and bottom-up phases where the top-down phase is used for precise handling of function pointers. Also unlike our approach, it tracks alias pairs as opposed to an explicit heap model and is meant for C rather than Java. Another subset-based pointer analysis for C that combines top-down and bottom-up phases is based on the observation that context-insensitivity does not result in a loss of precision if function side effects are accounted for [17]. In contrast to our approach, that technique handles SCCs in a context-insensitive way and employs a top-down phase that removes callee side effects. The algorithms described in [3, 6] perform bottom-up pointer analysis for C++ programs. Unlike the method presented in this paper, they perform case splits on possible aliasing patterns. Since the approach of [6] performs strong updates, it is more precise but less scalable compared to our technique.

Another related work is the algorithm described in [27], which describes a compositional pointer and escape analysis for Java. While this analysis is flow-sensitive and applies strong updates, it assumes that parameters do not alias and generates a summary that is valid under this assumption. However, if this assumption is violated at a call site, the analysis corrects the summary through a complex mechanism that involves merging of memory locations. Another difference is that [27] does not precisely handle virtual method calls.

Summarization. Many papers describe general frameworks for interprocedural analysis [19–21]. The work described in [29], [30] compute polymorphic summaries for dataflow problems but both rely on global points-to sets.

8 Conclusion

We described a new bottom-up, summary-based pointer analysis for Java. The experimental evaluation demonstrates that our algorithm runs significantly faster than top-down pointer analyses with comparable precision. We believe that Scuba is able to scale better because the cost of instantiating a method summary is smaller compared to the cost of re-analyzing the function.

References

1. Ashes benchmark suite. http://www.sable.mcgill.ca/software/#ashessuitecollection.
2. Dacapo benchmarks. http://www.dacapobench.org/.
3. Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference, 1999.
4. Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *PLDI*, 2000.
5. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS'08*.
6. Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, 2011.

7. Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. *PLDI'00*, 2000.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
9. George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, pages 423–434, 2013.
10. Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.
11. Ondrej Lhoták. *Program analysis using binary decision diagrams*. PhD thesis, McGill University, 2006.
12. Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *CC*, pages 47–64, 2006.
13. Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *FSE*, pages 199–215, 1999.
14. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA*, 2002.
15. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *TOSEM*, 2005.
16. Mayur Naik. Chord framework. http://pag.gatech.edu/chord.
17. Erik M Nystrom, Hong-Seok Kim, and W Hwu Wen-mei. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS'04*. 2004.
18. Robert O'Callahan. *Generalized aliasing as a basis for program analysis tools*. PhD thesis, Carnegie Mellon University, 2001.
19. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
20. Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TAPSOFT'95*, 1996.
21. Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
22. Olin Shivers. Control-flow analysis of higher-order languages. Technical report, 1991.
23. Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, 2011.
24. Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *PLDI*, pages 387–400, 2006.
25. John Whaley. Joeq: A virtual machine and compiler infrastructure. In *IVME*, pages 58–66. ACM, 2003.
26. John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.
27. John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.
28. Guoqing Xu and Atanas Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA*, 2008.
29. Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *POPL*, pages 221–234, 2008.
30. Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. In *PLDI*, page 28, 2014.
31. Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *PLDI*, 2004.