Parameterized Object Sensitivity for Points-to Analysis for Java*

Ana Milanova
Dept. Computer Science
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Atanas Rountev
Dept. Computer Science and Engineering
Ohio State University
rountev@cis.ohio-state.edu

Barbara G. Ryder
Division of Computer and Information Sciences
Rutgers University
ryder@cs.rutgers.edu

ABSTRACT

The goal of *points-to analysis* for Java is to determine the set of objects pointed to by a reference variable or a reference object field. We present *object sensitivity*, a new form of context sensitivity for flow-insensitive points-to analysis for Java. The key idea of our approach is to analyze a method separately for each of the object names that represent run-time objects on which this method may be invoked. To ensure flexibility and practicality, we propose a parameterization framework that allows analysis designers to control the tradeoffs between cost and precision in the object-sensitive analysis.

Side-effect analysis determines the memory locations that may be modified by the execution of a program statement. *Def-use analysis* identifies pairs of statements that set the value of a memory location and subsequently use that value. The information computed by such analyses has a wide variety of uses in compilers and software tools. This work proposes new versions of these analyses that are based on object-sensitive points-to analysis.

We have implemented two instantiations of our parameterized object-sensitive points-to analysis. On a set of 23 Java programs, our experiments show that these analyses have comparable cost to a context-insensitive points-to analysis for Java which is based on Andersen's analysis for C. Our results also show that object sensitivity significantly improves the precision of side-effect analysis and call graph construction, compared to (1) context-insensitive analysis, and (2) context-sensitive points-to analysis which models context using the invoking call site. These experiments demonstrate that object-sensitive analyses can achieve substantial precision improvement, while at the same time remaining efficient and practical.

1. INTRODUCTION

Points-to analysis is a fundamental static analysis used by optimizing compilers and software engineering tools to determine the set of objects whose addresses may be stored in reference variables and reference fields of objects. These *points-to sets* are typically computed by constructing one or more *points-to graphs*, which serve as abstractions of the run-time memory states of the analyzed program. (An example of a points-to graph is shown in Figure 1, which is discussed in Section 2.1.)

Optimizing Java compilers can use points-to information to perform various optimizations such as virtual call resolution, removal of unnecessary synchronization, and stack-based object allocation. Points-to analysis is also a prerequisite for a variety of other analyses—for example, *side-effect analysis*, which determines the memory locations that may be modified by the execution of a statement, and *def-use analysis*, which identifies pairs of statements that set the value of a memory location and subsequently use that value. These analyses are necessary to perform compiler optimizations such as code motion and partial redundancy elimination. In addition, such analyses are needed in the context of software engineering tools: for example, def-use analysis is needed for program slicing and data-flow-based testing. Points-to analysis is a crucial prerequisite for employing these analyses and optimizations.

Because of this wide range of applications, it is important to investigate approaches for precise and efficient computation of points-to information. The two major dimensions in the design space of points-to analysis are flow sensitivity and context sensitivity. Intuitively, *flow-sensitive* analyses take into account the flow of control between program points inside a method, and compute separate solutions for these points. *Flow-insensitive* analyses ignore the flow of control between program points, and therefore can be less precise and more efficient than flow-sensitive analyses. *Context-sensitive* analyses distinguish between the different contexts under which a method is invoked, and analyze the method separately for each context. *Context-insensitive* analyses do not separate the different invocation contexts for a method, which improves efficiency at the expense of some possible loss of precision.

Recent work [31, 43, 24, 33, 23, 7] has shown that flow- and context-insensitive points-to analysis for Java can be efficient and practical even for large programs, and therefore is a realistic candidate for use in optimizing compilers and software engineering tools. However, context insensitivity inherently compromises the precision of points-to analysis for object-oriented languages such as Java. This imprecision results from fundamental object-oriented features and

*A preliminary version of this article appeared in *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, July 2002.

programming idioms. (Section 2 presents several examples that illustrate this point.) The imprecision decreases the impact of the points-to analysis on client optimizations (e.g., virtual call resolution) and leads to less precise client analyses (e.g., def-use analysis). To make existing flow- and context-insensitive analyses more useful, it is important to introduce context sensitivity that targets the sources of imprecision that are specific to object-oriented languages. At the same time, the introduction of context sensitivity should not increase analysis cost to the point of compromising the practicality of the analysis.

In this paper we propose *object sensitivity* as a new form of context sensitivity for flow-insensitive points-to analysis for Java. Our approach uses *the receiver object at a method invocation site* to distinguish different calling contexts. Conceptually, every method is replicated for each analysis object name that represents a possible receiver object. The analysis computes separate points-to information for each replica of a local variable; the separate points-to information is valid for method invocations with the corresponding receiver object. Furthermore, the naming of objects is also object-sensitive: each object allocation site may be represented by several object names, corresponding to different receiver objects for the enclosing method.

We propose a parameterization framework that allows precision improvement through object sensitivity without incurring the cost of non-discriminatory replication for all objects and variables. The framework is parameterized in two dimensions. Analysis designers can select the degree of precision in the object naming scheme, as well as the set of reference variables for which the analysis maintains multiple points-to sets. This approach can be used to tune the cost of the analysis and to define *targeted sensitivity* for certain objects and variables for which more precise handling is likely to improve the analysis precision.

In this paper we discuss parameterized object-sensitive points-to analysis that is based on an Andersen-style points-to analysis for Java. Andersen's analysis for C [5] is a well-known flow- and context-insensitive points-to analysis. Recent work [43, 24, 33, 23] shows how to extend this analysis for Java. Although we demonstrate our technique on Andersen's analysis, parameterized object sensitivity can be trivially applied to enhance the precision of other flow- and context-insensitive analyses for Java (e.g., analyses that are based on flow- and context-insensitive points-to analyses for C [42, 38, 10]).

Modification side-effect analysis (MOD) determines, for each statement, the set of objects that may be modified by that statement. Similarly, USE analysis computes the set of objects that may be read by a statement. This information plays an important role in optimizing compilers and software productivity tools. Side-effect analysis requires the output of a points-to analysis, and the precision of the underlying points-to information directly affects the precision of the side-effect analysis. We define and evaluate a new object-sensitive MOD analysis that is based on the parameterized object-sensitive points-to analysis. Although we omit the discussion, our approach also applies to the corresponding USE analysis.

The goal of def-use analysis is to compute *def-use associations* between pairs of statements. A def-use association for a memory location l is a pair of statements (m, n) such that m assigns a value to l and subsequently n uses that value. Similarly to MOD analysis, def-use analysis requires the output of a points-to analysis. We define a new object-sensitive def-use analysis that is based on parameterized object-sensitive points-to analysis. In addition, we show how object-sensitive points-to analysis can be used to compute *contextual def-use associations* [40], a generalization of standard def-use associations defined for the purposes of data-flow-based testing.

We have implemented two instantiations of our parameterized object-sensitive analysis. We compare these instantiations with an Andersen-style flow- and context-insensitive points-to analysis. For a set of 23 Java programs, our experiments show that the cost of the three analyses is comparable. In some cases the object-sensitive analyses are actually faster than the context-insensitive analysis. We also evaluate the precision of the three analyses with respect to several client applications. MOD analyses based on object-sensitive points-to analyses are significantly more precise than the corresponding MOD analysis based on context-insensitive points-to analysis. Object sensitivity also improves the precision of call graph construction and virtual call resolution. In addition, we compare the object-sensitive analyses with a context-sensitive points-to analysis which uses the call site to distinguish different calling contexts.[1] Our experimental results show that object-sensitive analyses are capable of achieving significantly better precision than context-insensitive and call-site-based context-sensitive ones, while at the same time remaining efficient and practical. This improved precision is due to the fact that the object-sensitive analyses specifically target object-oriented features and idioms, while the precision of the other two analyses is compromised due to these features.

Contributions. The contributions of our work are the following:

- We propose object sensitivity as a new form of context sensitivity for flow-insensitive points-to analysis for Java.
- We define a parameterization framework that allows analysis designers to control the degree of object sensitivity and consequently the cost/precision tradeoffs of the analysis.
- We define a new object-sensitive side-effect analysis for Java that is based on parameterized object-sensitive points-to analysis.
- We define two new object-sensitive def-use analyses that are based on parameterized object-sensitive points-to analysis.
- We compare two instantiations of the parameterized object-sensitive analysis with an Andersen-style flow- and context-insensitive analysis and with a call-site-based context-sensitive analysis. Our experiments on a large set of programs show that the object-sensitive analyses are practical and significantly improve the

[1] This analysis is similar to the 1-1-CFA analysis from [18, 17].

```
class Y {...}
class X {
   Y f;
   void set(Y r)
        { this.f = r; }
   static void main() {
     s1: X p = new X();
     s2: Y q = new Y();
         p.set(q);
   }
}
```

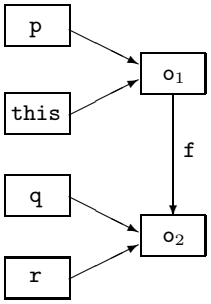

Figure 1: Sample program and its points-to graph.

precision of MOD analysis, call graph construction, and virtual call resolution.

Outline. The rest of the paper is organized as follows. Section 2 describes Andersen's analysis for Java and discusses some sources of imprecision due to context insensitivity. Section 3 defines our object-sensitive analysis. Section 4 discusses parameterized object sensitivity and Section 5 describes techniques for its efficient implementation. The new MOD analysis is defined in Section 6. Section 7 describes the def-use analyses. The experimental results are presented in Section 8. Section 9 discusses related work and Section 10 presents conclusions and future work.

2. FLOW- AND CONTEXT-INSENSITIVE POINTS-TO ANALYSIS FOR JAVA

Previous work proposes various flow- and context-insensitive analyses for Java [31, 43, 24, 33, 23, 7]. These analyses are typically derived from similar analyses for C. This section describes a flow- and context-insensitive points-to analysis for Java that is derived from Andersen's points-to analysis for C [5]; this analysis was previously described in detail in [33]. The section also illustrates how context insensitivity compromises analysis precision.

2.1 Analysis Semantics

Andersen's analysis for Java is defined in terms of three sets. Set R contains all reference variables in the analyzed program (including static variables). Set O contains names for all objects created at object allocation sites; for each allocation site s_i, there is a separate object name $o_i \in O$. Set F contains all instance fields in program classes. The analysis constructs *points-to graphs* containing two kinds of edges. Edge $(r, o_i) \in R \times O$ shows that reference variable r points to object o_i. Edge $(\langle o_i, f\rangle, o_j) \in (O \times F) \times O$ shows that field f of object o_i points to object o_j. A sample program and its points-to graph are shown in Figure 1.

For brevity, we discuss in detail only the kinds of statements listed below.[2] Other kinds of statements (e.g., calls to constructors and static methods) are handled in a similar fashion, as discussed shortly.

[2]Assumptions that the program consists of these kinds of statements are often used in the analysis literature to simplify the presentation. If necessary, temporary variables may be introduced to achieve these restrictions.

$$f(G, s_i\colon l = new\ C) = G \cup \{(l, o_i)\}$$

$$f(G, l = r) = G \cup \{(l, o_i) \mid o_i \in Pt(G, r)\}$$

$$f(G, l.f = r) = G \cup \{(\langle o_i, f\rangle, o_j) \mid o_i \in Pt(G, l) \wedge o_j \in Pt(G, r)\}$$

$$f(G, l = r.f) = G \cup \{(l, o_i) \mid o_j \in Pt(G, r) \wedge o_i \in Pt(G, \langle o_j, f\rangle)\}$$

$$f(G, l = r_0.m(r_1, \ldots, r_n)) = G \cup \{resolve(G, m, o_i, r_1, \ldots, r_n, l) \mid o_i \in Pt(G, r_0)\}$$

$$resolve(G, m, o_i, r_1, \ldots, r_n, l) = \texttt{let}\ \ m_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o_i, m)\ \ \texttt{in}\ \{(p_0, o_i)\} \cup f(G, p_1 = r_1) \cup \ldots \cup f(G, l = ret_j)$$

Figure 2: Points-to effects of program statements for Andersen's analysis.

- Direct assignment: `l = r`
- Instance field write: `l.f = r`
- Instance field read: `l = r.f`
- Object creation: `l = new C`
- Virtual invocation: `l = r0.m(r1,...,rk)`

At a virtual call, name `m` uniquely identifies a method in the program. This method is the *compile-time* target of the call, and is determined based on the declared type of r_0 [16, Section 15.11.3]. At run time, the invoked method is determined by examining the class of the receiver object and all of its superclasses, and finding the first method that matches the signature and the return type of `m` [16, Section 15.11.4].

Analysis semantics is defined in terms of *transfer functions* that add new edges to points-to graphs; no edges are ever removed from these graphs during the analysis. Each transfer function represents the semantics of a program statement. The functions for different statements are shown in Figure 2 in the format $f(G, s) = G'$, where s is a statement, G is an input points-to graph, and G' is the resulting points-to graph. $Pt(G, x)$ denotes the points-to set (i.e., the set of all successors) of x in graph G. The analysis computes the closure of the empty graph under the application of all transfer functions for program statements.

For most statements, the effects on the points-to graph are straightforward; for example, statement $l = r$ creates new points-to edges from l to all objects pointed to by r. For virtual call sites, resolution is performed for every receiver object pointed to by r_0. Function *dispatch* uses the class of the receiver object and the compile-time target of the call to determine the actual method m_j invoked at run time. Variables $p_0, \ldots, p_n$ are the formal parameters of the method; variable p_0 corresponds to the implicit parameter `this`. Variable ret_j contains the return values of m_j (we assume that each method has a unique variable that is assigned all values returned by the method; this can be achieved by inserting auxiliary assignments).

Calls to static methods (not shown in Figure 2) are resolved immediately and the appropriate updates of parameters and return values are performed: for example, formal-

```
  class X {...}
  class Y {
      X f;
1     void set(X x) { this.f = x; } }

2   s1: X x1 = new X();
3   s2: X x2 = new X();
4   s3: Y y1 = new Y();
5   s4: Y y2 = new Y();
6     y1.set(x1);
7     y2.set(x2);
```

Figure 3: Imprecision due to field encapsulation.

actual bindings are treated as direct assignments. We assume that object creation is represented by "`l = new C`" followed immediately by a call "`l.C(..)`" to the appropriate constructor. The constructor call is treated as an instance call for which there is only one target method. The analysis ignores statements that have no effect on the flow of reference values—for example, assignments to variables of primitive types such as `int`, `boolean`, etc.

2.2 The Imprecision of Context-Insensitive Analysis

This section presents several examples of basic object-oriented features and programming idioms for which context-insensitive analysis produces imprecise results.

2.2.1 Encapsulation

Figure 3 illustrates the typical situation when an encapsulated field is written through a modifier method. At the call site at line 6, y_1 points to o_3 and x_1 points to o_1. After the analysis applies the transfer function for the virtual call (as shown in Figure 2), the implicit parameter `this` of method `set` points to o_3 and formal parameter `x` points to o_1. After the analysis processes the call at line 7, `this` points to o_4 and `x` points to o_2. Thus, at statement `this.f=x` at line 1, the analysis erroneously infers points-to edges $(\langle o_3, f\rangle, o_2)$ and $(\langle o_4, f\rangle, o_1)$.

This imprecision can be avoided if the analysis distinguishes invocations of `set` on o_3 from invocations of `set` on o_4. This distinction could be achieved if the analysis were able to associate *multiple* points-to sets with `this` and with `x`, one for each of the objects on which `set` is invoked. This would allow statement `this.f=x` to be analyzed separately for each of the receiver objects, and would avoid creating spurious points-to edges. In Section 3 we show how object-sensitive analysis achieves this goal.

During context-insensitive analysis, there is a single copy of every method for all possible invocations. Therefore, field `f` of *each* receiver object will point to *all* objects passed as arguments to the method which sets the value of `f`. In object-oriented languages, encapsulation and information hiding are strongly supported, and fields are almost always accessed indirectly through method invocations. As a result, context-insensitive analysis can incur significant imprecision.

2.2.2 Inheritance

Consider the example in Figure 4. At line 2, which is executed after the constructor at line 10 is invoked, `B.this` points to o_3 and `B.xb` points to o_1.[3] After the analysis processes the call to the superclass constructor, `A.this` and `A.xa` point to o_3 and o_1, respectively. Because of the call at line 5, which occurs due to the constructor call at line 11, `A.this` will point to o_4 and `A.xa` will point to o_2. Thus, at statement `this.f=xa` at line 1, spurious edges $(\langle o_3, f\rangle, o_2)$ and $(\langle o_4, f\rangle, o_1)$ are added to the graph. The imprecision propagates further, as the analysis infers that `xb` at line 3 points to both o_1 (of class `Y`) and o_2 (of class `Z`). Therefore, it appears that the possible targets of the virtual call at line 4 are `Y.n` and `Z.n` (the same problem also occurs at line 7). As a result, the calls at lines 4 and 7 cannot be devirtualized using the solution computed by the context-insensitive analysis. The imprecision is due to statement `this.f=xa` in the constructor of superclass `A`, which merges the information for all possible receiver objects.

```
  class X { void n() {...} }
  class Y extends X { void n() {...} }
  class Z extends X { void n() {...} }

  class A {
      X f;
1     A(X xa) { this.f = xa; } }

  class B extends A {
2     B(X xb) { super(xb); ...  }
      void m() {
3         X xb = this.f;
4         xb.n(); } }

  class C extends A {
5     C(X xc) { super(xc); ...  }
      void m() {
6         X xc = this.f;
7         xc.n(); } }

8   s1: Y y = new Y();
9   s2: Z z = new Z();
10  s3: B b = new B(y);
11  s4: C c = new C(z);
12    b.m();
13    c.m();
```

Figure 4: Field assignment through a superclass.

In the presence of inheritance, instance fields are often located in superclasses and are written through invocations of superclass constructors or methods. During context-insensitive analysis, fields of subclass instances point to objects intended for instances of other subclasses. Thus, in the presence of wide and deep inheritance hierarchies, context insensitivity can lead to substantial imprecision.

2.2.3 Collections and Maps

Consider the example in Figure 5. There is a single object

[3] We use `C.m` to refer to method/constructor `m` in class `C`. Similarly, we use `C.m.v` to refer to local variable or formal parameter `v` in method/constructor `m` in `C`; sometimes `m.v` is used for brevity.

```
  class Container {
      Object[] data;
      Container(int size) {
1   s1:   Object[] data_tmp = new Object[size];
2         this.data = data_tmp; }
      void put(Object e,int at) {
          Object[] data_tmp = this.data;
3         data_tmp[at] = e; }
      Object get(int at) {
4         Object[] data_tmp = this.data;
5         return data_tmp[at]; } }

6   s2: Container c1 = new Container(100);
7   s3: Container c2 = new Container(200);
8   s4: X x = new X();
9       c1.put(x,0);
10  s5: X y = new Y();
11      c2.put(y,1);
```

Figure 5: Simplified container class.

name o_1 which represents the `data` arrays of both instances of `Container`. Therefore, objects stored in individual containers appear to be shared between the two containers. In order to avoid this imprecision, the `data` array of every instance of `Container` should be represented by a distinct object name. In addition, the analysis should be able to assign distinct points-to sets to `put.this` and `put.e` for every possible receiver object of `put`.

Context insensitivity causes data that is stored in one instance of a collection or a map to be retrieved from every other instance of the same class, and very likely from all instances of its subclasses. Since collections (e.g., `Vector`) and maps (e.g., `Hashtable`) are commonly used in Java, context insensitivity can seriously compromise analysis precision.

3. OBJECT-SENSITIVE ANALYSIS

In context-sensitive analysis, a method is analyzed separately for different calling contexts. We define a new form of context-sensitive points-to analysis for Java which we refer to as *object-sensitive analysis*; this approach was first introduced in [26, 25]. With object sensitivity, each instance method (i.e., non-static method) and each constructor is analyzed separately for each object on which this method/constructor may be invoked. More precisely, the analysis uses a set of *object names* to represent objects allocated at run time. If a method/constructor may be invoked on run-time objects represented by object name o, the object-sensitive analysis maintains a separate contextual version of that method/constructor that corresponds to invocation context o.

Our object-sensitive analysis is based on Andersen's analysis for Java from Section 2.1. However, the same approach can be trivially applied to other flow- and context-insensitive analyses for Java (e.g., analyses derived from flow- and context-insensitive points-to analyses for C [42, 38, 10]). Section 3.1 defines the semantics of the object-sensitive analysis. Section 3.2 discusses why object sensitivity is appropriate for flow-insensitive analysis of object-oriented programs, and compares this approach with other context-sensitive analyses.

3.1 Analysis Semantics

Our object-sensitive analysis is defined in terms of five sets. Recall from Section 2.1 that set R contains all reference variables in the analyzed program (including static variables), and set F contains all instance fields in program classes. Set S contains all object allocation sites in the program. We also use a set of object names O' and a set of replicas of reference variables R'; both sets will be discussed shortly.

To simplify the presentation, we define a relation α which shows that a method or a constructor m may be invoked on instances of a given class C. Suppose that m is defined in some class D. Relation $\alpha(C, m)$ holds if and only if C and D are the same class or C is a subclass of D. Note that $\alpha(C, m)$ should hold even if m is overridden somewhere on the inheritance chain between D and C, because m could still be invoked on instances of C through `super`. We extend the notation to object names: for any $o \in O'$ which represents instances of class C, $\alpha(o, m)$ if and only if $\alpha(C, m)$.

3.1.1 Object Names

The analysis uses a set of object names $O' \subseteq S \cup S^2 \cup \ldots \cup S^k$, where $k \geq 1$ is a parameter of the analysis. We will use $o_{ij\ldots pq}$ to denote the sequence of allocation sites $(s_i, s_j, \ldots, s_p, s_q)$. Consider an allocation site $s_q \in S$ in method m. If m is a static method, the run-time objects allocated at s_q are represented by a single object name o_q. If m is an instance method or a constructor, the run-time objects allocated at s_q are represented by a set of object names from O' of the form $o_{ij\ldots pq}$.

A particular name $o_{ij\ldots pq}$ represents all run-time objects that were created by s_q when the enclosing instance method or constructor was invoked on an object represented by name $o_{ij\ldots p}$ which was created at allocation site s_p. This context-sensitive naming scheme allows the analysis to distinguish among different objects created by the same allocation site. (In contrast, Andersen's analysis uses a single object name per allocation site.) For example, allocation site s_1 in Figure 5 appears in the constructor of `Container`. Sites s_2 and s_3 create instances of `Container`; thus, there are two object names o_{21} and o_{31} that correspond to s_1.

The formal definition of O' is as follows:

- $o_q \in O'$ for each $s_q \in S$ located in a static method
- if $o_{ij\ldots p} \in O'$ and $s_q \in S$ is located in an instance method or a constructor m such that $\alpha(o_{ij\ldots p}, m)$, then
  1. if $|ij\ldots p| < k$, then $o_{ij\ldots pq} \in O'$
  2. if $|ij\ldots p| = k$, then $o_{j\ldots pq} \in O'$

This definition ensures that each object name corresponds to a sequence of at most k allocation sites, where k is an analysis parameter. Continuing with our `Container` example, if k=1, allocation sites 2 and 3 are dropped from sequences o_{21} and o_{31} respectively, leaving object name o_1 to represent all objects created at allocation site s_1.

$$F(G, s_q\colon l = new\ C) = G \cup \bigcup_{c \in \mathcal{C}_m} \{(l^c, c \oplus_k s_q)\}$$

$$F(G, l = r) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c = r^c)$$

$$F(G, l.f = r) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c.f = r^c)$$

$$F(G, l = r.f) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c = r^c.f)$$

$$F(G, l = r_0.m(r_1, \ldots, r_n)) =$$
$$G \cup \bigcup_{c \in \mathcal{C}_m} \{resolve(G, m, o, r_1^c, \ldots, r_n^c, l^c) \mid o \in Pt(G, r_0^c)\}$$

$$resolve(G, m, o, r_1^c, \ldots, r_n^c, l^c) =$$
$$\texttt{let}\ \ c' = o$$
$$m_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o, m)\ \ \texttt{in}$$
$$\{(p_0^{c'}, o)\} \cup f(G, p_1^{c'} = r_1^c) \cup \ldots \cup f(G, l^c = ret_j^{c'})$$

Figure 6: Object-sensitive points-to effects of statements in instance methods and constructors. $\mathcal{C}_m$ is the set of possible contexts for the enclosing method.

3.1.2 Context Sensitivity

Set $\mathcal{C} = O' \cup \{\epsilon\}$ represents the space of all possible contexts for our object-sensitive analysis. A static method is always analyzed under the empty context ϵ. Any instance method or constructor m is separately analyzed for each context $o \in O'$ for which $\alpha(o, m)$ holds. This separation is achieved by maintaining *multiple replicas* of reference variables for each possible context. The set of replicas of reference variables R' is defined by a partial injective function $map : R \times \mathcal{C} \rightarrow R'$. If $r \in R$ is a static variable or a local variable in a static method, the pair (r, ϵ) is mapped to r. If r is a local variable or a formal parameter of an instance method or a constructor m, the pair (r, o) is mapped to a "fresh" variable r^o for every context $o \in O'$ for which $\alpha(o, m)$ holds. For the rest of the paper we will refer to the elements of R' as *context copies*. For example, in Figure 4 we have $\alpha(o_3, \texttt{A.A})$ and $\alpha(o_4, \texttt{A.A})$, and there are two copies of `A.this` corresponding to contexts o_3 and o_4. Similarly, there are two copies of `A.xa`. For the example in Figure 5, we have $\alpha(o_2, \texttt{Container.put})$ and $\alpha(o_3, \texttt{Container.put})$; therefore there are context copies of `put.this, put.data_tmp`, and `put.e` corresponding to contexts o_2 and o_3.

The analysis can be easily extended to analyze static methods separately for the contexts of their invoking instance methods or constructors. In Java programs static methods rarely change the state of the invoking object, and we expect that analyzing static methods for different contexts will only lead to relatively small gains in precision while making the analysis more complex. Therefore, we have decided to treat static methods in an intuitive manner and to analyze them only under the special context ϵ. The same approach is used for naming the objects allocated by static methods: the runtime objects created at an allocation site s_q are represented by a single object name o_q.

3.1.3 Transfer Functions

The object-sensitive analysis constructs points-to graphs in which the nodes are elements of R' and O'. Analysis semantics can be defined by transfer functions that add new edges to these points-to graphs. For statements that are located inside static methods, the transfer functions are identical to those in Figure 2. For statements located in instance methods and constructors, the transfer functions are presented in Figure 6. For example, the transfer function for an allocation site creates points-to edges from context copies l^c to the appropriate object names. Operation $c \oplus_k s_q$ adds s_q to the end of c and (if necessary) removes allocation sites from the beginning of c to ensure that the length of the resulting name does not exceed k.

The effects of $F(G, s)$ are essentially equivalent to applying the corresponding $f(G, s)$ from Figure 2 for each context from the set $\mathcal{C}_m = \{o \in \mathcal{C} \mid \alpha(o, m)\}$, where m is the method in which s is located. For simplicity, we present the semantics as if *all* elements of $\mathcal{C}_m$ are possible contexts. As discussed in Section 5, analysis implementations only need to consider contexts that actually occur at calls to m.

The correctness of these transfer functions can be established by considering a small-step operational semantics for Java. We can define an abstraction relation that encodes the correspondence between entities in the semantics (e.g., stack variables, static fields, heap objects, and points-to relationships) and nodes/edges in the points-to graphs. It can be shown that the semantic effects of each kind of statement are modeled correctly by the corresponding transfer function from Figure 6: if the abstraction relation holds before the statement, it also holds after the statement is executed. Based on this property, a proof by induction can be used to demonstrate that any points-to relationship that can occur at run time is represented by an appropriate points-to edge in the analysis solution.

3.1.4 Example

Consider the set of statements in Figure 4. Since $\alpha(\texttt{B}, \texttt{B.B})$ and $\alpha(\texttt{B}, \texttt{A.A})$, we have

$$\{\texttt{B.this}^{o_3},\ \texttt{B.xb}^{o_3},\ \texttt{A.this}^{o_3},\ \texttt{A.xa}^{o_3}\} \subseteq R'$$

Similarly, we have

$$\{\texttt{C.this}^{o_4},\ \texttt{C.xc}^{o_4},\ \texttt{A.this}^{o_4},\ \texttt{A.xa}^{o_4}\} \subseteq R'$$

At line 2, $\texttt{B.this}^{o_3}$ points to o_3 and $\texttt{B.xb}^{o_3}$ points to o_1. When the analysis processes the call to `A.A` at line 2, `A.this` and `A.xa` are mapped to the context copies corresponding to o_3, and points-to edges ($\texttt{A.this}^{o_3}$, o_3) and ($\texttt{A.xa}^{o_3}$, o_1) are added to the graph. Similarly, because of line 5, $\texttt{A.this}^{o_4}$ points to o_4 and $\texttt{A.xa}^{o_4}$ points to o_2. Statement `this.f=xa` at line 1 occurs in the context of o_3 and o_4. Thus

$$\texttt{A.this}^{o_3}\texttt{.xa}^{o_3} \qquad \texttt{A.this}^{o_4}\texttt{.xa}^{o_4}$$

which produces edges $(\langle o_3, f\rangle, o_1)$ and $(\langle o_4, f\rangle, o_2)$. Since $\alpha(\texttt{B}, \texttt{B.m})$ and $\alpha(\texttt{C}, \texttt{C.m})$, we have

$$\{\texttt{B.m.this}^{o_3},\ \texttt{B.m.xb}^{o_3},\ \texttt{C.m.this}^{o_4},\ \texttt{C.m.xc}^{o_4}\} \subseteq R'$$

When the analysis processes the statement at line 3, `B.m.xb` and `B.m.this` will be mapped to the context copies corresponding to o_3. Since $\texttt{B.m.this}^{o_3}$ points to o_3 and $\langle o_3, f\rangle$ points only to o_1, the statement at line 3 produces edge $(\texttt{B.m.xb}, o_1)$. Similarly, the statement at line 6 produces edge $(\texttt{C.m.xc}, o_2)$.

3.2 Advantages of Object Sensitivity

In object-oriented languages such as Java, one of the primary roles of instance methods is to access or modify the state of the objects on which they are invoked. Instance methods typically work on encapsulated data, using the implicit parameter `this` to modify or retrieve data from the object structure rooted at the receiver object. If points-to analysis does *not* distinguish the different receiver objects of instance methods, the states of these objects are essentially merged and any access/modification of the state of one object is propagated to all other objects. Therefore, it is crucial to distinguish the different objects pointed to by `this` and to analyze instance methods separately for different receiver objects. Similarly, the role of a constructor is to create the initial object state. To avoid merging the initial states of all objects pointed to by `this`, points-to analysis should distinguish the different objects on which a constructor is invoked.

Context sensitivity mechanisms of finer granularity than a receiver object may create redundant contextual versions. For example, one of the most popular mechanisms for context sensitivity is the *call string* approach, which represents invocation context using a string of k enclosing call sites [39]. For $k = 1$, a method is analyzed separately for each call site that invokes that method. For many statements, it is redundant to distinguish between *distinct* call sites that have the *same* receiver object. For example, if statement `this.f=formal` were analyzed separately for distinct call sites that have the same receiver object, the effect would be the same as if it were analyzed once for that object: field f of the receiver would point to all objects in the points-to sets of the corresponding actual parameters at all call sites. Clearly, because of the flow insensitivity of the analysis, the effects of the distinct per-call-site versions of the statement are merged. The same kind of redundancy also occurs for statements that read the value of any field of the receiver object (e.g., `l=this.f`), as well as for certain method invocations on the receiver (e.g., `l=this.m()`). These redundancies cause the call string approach to incur increased analysis cost for such cases, without any gain in precision. On the other hand, object-sensitive analysis performs exactly the necessary amount of work for such statements.

In certain cases, distinguishing calling context by a chain of enclosing call sites can be less precise than distinguishing context per receiver object. To illustrate such a case, recall the set of statements from Figure 4. Suppose that the following new statement is added at line 14:

```
14  s5 : C c2 = new C(y);
```

If calling context is distinguished per call site ($k = 1$), the effects of constructor `A.A` invoked at line 5 are merged for receivers o_4 and o_5. Thus, there are redundant points-to edges $(\langle o_4, f\rangle, o_1)$ and $(\langle o_5, f\rangle, o_2)$. The imprecision propagates and affects both the points-to analysis and its clients; for example, the virtual call at line 7 cannot be resolved.

In other cases, distinguishing context by a chain of enclosing call sites can be more precise than distinguishing context per receiver object. Suppose that there is an instance method that returns its formal parameter:

```
X m(X param) { return param; }
```

If there are two different call sites of m with the same receiver, and if the actual parameters passed at the two call sites are different, the object-sensitive analysis propagates redundant information to the left-hand side of each call. An analysis that distinguishes calling context per call site infers correctly that the information passed at each call site does not flow to the left-hand side of the other call site. In general, object sensitivity (an instance of the functional approach to context sensitivity [39]) and call chain context sensitivity (an instance of the call string approach) are incomparable in terms of precision.

Our observations of many Java programs indicate that situations in which object sensitivity is more beneficial than call chain context sensitivity occur frequently in practice. We have examined the output of two client analyses: (1) the side-effect analysis described in Section 6, and (2) an analysis that computes test coverage requirements for error recovery code in Java web services applications [15]. We encountered many situations for which using the simple and practical object-sensitive analysis produces precise results; in the same time, the call chain approach would require a call chain of length at least four to achieve the same level of precision (call chains of length greater than one are typically considered impractical). We are yet to find a situation in a Java program for which distinguishing context per call site will produce more precise results with respect to some client analysis, compared to distinguishing context per receiver object. The empirical results in Section 8 also indicate that in practice, object sensitivity has more significant impact on precision than context sensitivity based on call chains.

4. PARAMETERIZED OBJECT SENSITIVITY

In this section we define a parameterized framework for object-sensitive analysis. The least precise and least costly instantiation of the framework is the context-insensitive Andersen's analysis, while the most precise and most expensive instantiation is the object-sensitive analysis described in Section 3.

The framework is parameterized in two dimensions. First, the analysis designer can select the degree of precision in the naming scheme for object names. This is accomplished by defining a separate *context depth* k_q for each allocation site s_q, instead of having a single depth k for the entire analysis. Second, the analysis designer can specify the set of reference variables for which multiple points-to sets should be maintained. The analysis replicates only these selected variables.

The goal of the parameterization is to enhance the flexibility of the object-sensitive analysis. By varying the object naming scheme and the set of variables that are replicated, the analysis designer can control directly the size of the points-to graph and the cost of the analysis. Furthermore, the parameterization allows *targeted* context sensitivity. Instead of using the global non-discriminatory replication presented in Section 3, the analysis designer can choose objects and variables for which keeping more precise information is likely to improve the precision of the points-to solution (e.g., implicit parameters `this`, formal parameters, return variables, sub-objects of composite objects, etc.).

The parameterization for object names is based on a separate context depth k_q for each allocation site s_q. For every object name of the form $o_{ij\ldots pq}$, the analysis ensures that $|ij\ldots pq| \leq k_q$. For the boundary case $k_q = 1$, there is a single object name for the allocation site (similarly to Andersen's analysis). For example, if $k_1 = 2$ for allocation site s_1 in Figure 5, the analysis will maintain two separate object names o_{21} and o_{31} for s_1. In the case when $k_1 = 1$, the analysis will maintain a single object name o_1 corresponding to s_1. It is straightforward to modify the definition of O' from Section 3.1 to accommodate this parameterization. A similar change can be made to the first transfer function in Figure 6: instead of $\oplus_k$, it should use $\oplus_{k_q}$.

The parameterization for reference variables is based on a set $R^* \subseteq R$ which contains all variables that should be replicated during the analysis. For the boundary case $R^* = \emptyset$, there is no replication and analysis behavior is similar to Andersen's analysis. Function $map : R \times \mathcal{C} \rightarrow R'$ constructs R' as follows: if $r \in R^*$ is a local variable in an instance method or a constructor m, r is mapped to a "fresh" variable r^o for every context $o \in O'$ such that $\alpha(o, m)$. Any other variable is mapped to itself. Thus, *map* replicates variables in R^* for all applicable contexts, and preserves variables not in R^* (i.e., $map(r, c) = r$ for any $r \notin R^*$). For example, if `A.xa` and `A.this` in Figure 4 are in R^*, the analysis will maintain two separate context copies for each variable, one for context o_3 and one for context o_4. If `A.xa` and `A.this` are not in R^*, there are just single variables; as a result, statement `this.f=xa` at line 1 is analyzed context-insensitively and introduces spurious points-to edges. The transfer functions in the parameterized analysis are identical to the ones from Figure 6, except for the use of the modified function *map* based on parameter set R^*.

Clearly, there are other possible dimensions of parameterization. For example, the analysis designer may choose to analyze a sequence of constructor calls context-sensitively; or the designer may choose to analyze parts of the program per class, instead of per object, etc. The parameterizations for object names and reference variables are simple to formulate and intuitive to understand. Due to their simplicity, these dimensions of parameterization were chosen to formalize the idea of targeted context sensitivity. We have started experimental studies of other dimensions of parameterization and plan to continue work in this direction in the future.

5. IMPLEMENTATION TECHNIQUES

A typical implementation of Andersen's flow- and context-insensitive analysis for Java uses a *statement processing routine* which processes different kinds of program statements, and a *virtual dispatch routine* which models the semantics of virtual calls. The parameterized object-sensitive analysis can build on any existing implementation I of Andersen's analysis for Java. This can be achieved by (1) implementing function $map(v, c)$, (2) augmenting the statement processing routine in I to process each statement once for every possible context in accordance with the rules from Figure 6, and (3) augmenting the virtual dispatch routine in I to map the formal parameters and return variable of the invoked method to the corresponding invocation context.

Let I' be an implementation of the parameterized analysis which augments I with function *map* and alters the statement processing routine and the virtual dispatch routine. Any such I' can be optimized in several ways.

First, the semantics in Figure 6 implicitly assumes that all possible contexts of a method m are actually used at calls to that method—that is, m is invoked with every context o for which $\alpha(o, m)$ holds. Clearly, I' can keep track of which contexts actually occur at calls to m. Thus, I' would take into account the effects of a statement in m for context o if and only if m can be invoked with receiver object o according to the current analysis solution.

Second, whenever the points-to set of a replica $\texttt{this}^o$ is needed, the analysis can return the singleton set $\{o\}$. Thus, I' can avoid storing replicas $\texttt{this}^o$ and redundant points-to edges as well as retrieving the points-to set of $\texttt{this}^o$.

Third, whenever I' processes a statement s which contains only non-replicated variables, there is no need to analyze s multiple times for different contexts. Similarly, if a replicated variable l is assigned *only* at statements of the form `l=r` or `l=r.f` where $r \notin R^*$, these statements can be analyzed only once. In this case, all context copies of l have the same values, and the analysis needs to keep only one copy for l. In other words, transfer function $F(G, s)$ from Figure 6 can be replaced with $f(G, s)$ from Figure 2. For the rest of the paper we refer to statements that can be analyzed only once as *context-independent*, and to statements that need to be analyzed multiple times for different contexts as *context-dependent*.

Other kinds of statements can also be analyzed once instead of multiple times. For example, consider the following statements that occur in an instance method m, and suppose that $p \in R^*$ and $x, y \notin R^*$:

```
void m(X p) { x=p.f; y=p; }
```

Suppose that the analysis keeps a non-replicated variable p' such that the points-to set of p' is the *union* of the points-to sets of all context copies of p. It is easy to see that instead of analyzing statements `x=p.f` and `y=p` multiple times, the analysis algorithm can analyze statements $\texttt{x=p'.f}$ and $\texttt{y=p'}$ only once without affecting the computed points-to sets.

This observation leads to an implementation technique that recognizes statements which contain replicated variables but may be processed context-independently (i.e., processed once instead of multiple times). The technique does not affect analysis correctness or precision. Suppose l is a replicated variable. The simplification creates a new non-replicated variable l' and a new (context-dependent) statement $\texttt{l'=l}$. Clearly, the points-to set of l' is the union of the points-to sets of all context copies of l. Consider assignments of the form `l=p`, `l.f=p`, `l=p.f`, `p=l`, `p=l.f`, and `p.f=l` for which $p \notin R^*$. If the left-hand side is a replicated variable (as in `l=r` and `l=p.f`), no changes can be made because if l is replaced by l', context-sensitive information may be lost. For the remaining four kinds of statements, l can be replaced by l' and the statement can be analyzed context-independently. To see that there is no loss of precision, consider `l.f=p`. If it is processed context-dependently, for every context c the analysis will get an object from the points-to set of l^c and will set its f field to refer to the objects pointed to by p. If $\texttt{l'.f=p}$ is processed context-independently, the analysis will process exactly the same objects and will create exactly the same points-to edges.

Similar reasoning can be used to determine that l can be replaced by l' in certain calls. Consider a virtual call `x=l.m(p1,..,pn)` such that $p_i \notin R^*$ for every i and $x \notin R^*$. This call can be analyzed context-independently without loss of precision, because in both cases the analysis would examine the same set of receiver objects and would process the same set of formal-actual assignments (recall that the points-to set of l' is the union of the points-to sets of l^c for every context c). Similarly, if $r \notin R^*$ and $x \notin R^*$, call `x=r.m(p1,..,l,..,pn)` can be processed context-independently. Since r is not replicated, the same set of target methods is invoked for all contexts of the call. If the call is analyzed context-dependently, the analysis will process several assignments $param^o = l^{c_1}, \ldots, param^o = l^{c_k}$, where *param* is a formal parameter of method $m' = target(o, m)$. If the call is analyzed context-independently, the analysis will process only one assignment $param^o = l'$ which has equivalent effect. For ease of explanation, in the rest of this section we will use C_{ci} to denote the set of all (context-independent) calls in these two categories.

We would like to analyze context-independently even call sites c_i: `x=l.m(l1,..,ln)` for which l and some of the l_i's are replicated. For this, we precompute certain information and provide it as input to the points-to analysis. The computation uses a conservative call graph (e.g., computed by CHA [11]). Site c_i can be analyzed context-independently by replacing l with l', l_1 with l'_1, etc. and by modifying the transfer function to assign the values of the actuals to the non-replicated p'_i corresponding to the formals p_i. This change occurs if the values of p_i are not used in a context-dependent manner by c_i's callees, which can be guaranteed under the following two conditions.

First, assignments that contain only replicated variables should not be reachable from c_i in the conservative call graph—that is, the value of a replicated formal will not be used in context-dependent manner by assignments in the callees. For example, for the sample method m shown earlier, p is only used in statements that contain non-replicated variables; therefore, calls to m satisfy this condition. Second, any call site c_j reachable from c_i (including c_i) must be either monomorphic, or must belong to the set C_{ci} defined earlier. Otherwise, the information at c_j may be propagated imprecisely. For example, consider the sample method m and a call site c_i: `l.m(l1)` that invokes m for some context o_1 and another method m' for context o_2. Here only the value of $l_1^{o_1}$ should be propagated to m, not the value of l'_1. If the two conditions hold for c_i, all formal-actual pairs can be treated context-independently. Finding such call sites can be done in a single pass through the conservative call graph. In order to have access to the points-to sets of context copies p^c of formals, additional context-dependent assignments `p=q` need to be added at c_i for formal-actual pair (p, q). As described below, in many cases such assignments can be removed before the points-to analysis.

One technique that increases the number of context-independent calls takes into account a sequence of constructor calls $c_1, c_2, \ldots c_n$, where c_i invokes a constructor of some class X and c_{i+1} invokes a constructor of the superclass of X. Such sequences are common when new objects are created. Whenever one of the constructors contains an assignment `this.f=param`$_i$ where `param`$_i$ is a formal, this assignment is "inlined" at the call site c_j to that constructor and if possible at call sites c_{j-1}, c_{j-2}, etc. For example, statement 1 in Figure 4 can be inlined in constructors `B.B` and `C.C`, and subsequently at the constructor calls at lines 10 and 11. As a result, the calls to `A.A` at lines 2 and 5, to `B.B` at line 10, and to `C.C` at line 11 become context-independent and therefore inexpensive to process.

Depending on the intended uses of the points-to analysis, the introduction of a non-replicated variable l' can be used to remove completely the context copies of a replicated variable l. Suppose that for the purposes of the analysis clients (e.g., call graph construction), it is sufficient to compute the union of the points-to sets of replicas l^c. In this case, all l^c can be eliminated by changing function *map* to map l to l' for all contexts. To perform this change, all occurrences of l in statements from the original program must have been replaced by l' using the techniques discussed earlier. In this case, statements `l=q` introduced at call sites (as described above) can be removed.

input *Stmt*: set of statements *map*: $R \times \mathcal{C} \to R'$
Methods: set of methods *Pt*: $R' \to \mathcal{P}(O')$
output *Mod*: $Stmt \times \mathcal{C} \to \mathcal{P}(O')$
initialized to $\emptyset$ for all $(s, c) \in Stmt \times \mathcal{C}$
declare *MMod*: $Methods \times \mathcal{C} \to \mathcal{P}(O')$
initialized to $\emptyset$ for all $(m, c) \in Methods \times \mathcal{C}$

```
[1]  foreach instance field write s: p.f = q ∈ Stmt do
[2]    foreach context c in which s appears do
[3]      Mod(s,c) := Pt(map(p,c))
[4]      add Mod(s,c) to MMod(EnclMethod(s),c)
[5]  while changes occur in Mod or MMod do
[6]    foreach virtual call s: l = r.m(...) ∈ Stmt do
[7]      foreach context c in which s appears do
[8]        foreach object o∈Pt(map(r,c)) do
[9]          add MMod(target(o,m),o) to Mod(s,c)
[10]       add Mod(s,c) to MMod(EnclMethod(s),c)
[11]   foreach static call s: l = C.m(...) ∈ Stmt do
[12]     foreach context c in which s appears do
[13]       Mod(s,c) := Mod(s,c) ∪ MMod(m,ε)
[14]       add Mod(s,c) to MMod(EnclMethod(s),c)
```

Figure 7: Object-sensitive MOD analysis. $\mathcal{P}(X)$ denotes the power set of X.

6. SIDE-EFFECT ANALYSIS

In this section we present a MOD analysis based on object-sensitive points-to analysis. For each statement s and context c of the method enclosing s, our MOD algorithm computes the set $Mod(s, c)$ of objects that could be modified by executing s when in c. The algorithm is shown in Figure 7. $Pt(x)$ denotes the set of objects pointed to by context copy x. We say that statement s *appears* in context c if $\alpha(c, m)$ holds for the method m enclosing s. $MMod(m, c)$ stores the sets of objects modified by each contextual version of a method (i.e., objects that are modified when m is invoked with context c). For virtual calls (lines 6–10) the target methods are determined for each receiver object o in context c, based on the class of o and the compile-time target m. In addition, object o determines which set of modified

objects associated with the target method will be added to the *Mod* set at line 9. For static calls (lines 11-14) we use ϵ to denote the special empty context in which the statements in those methods appear.

Although formulated with respect to an underlying object-sensitive analysis, the MOD analysis can be easily generalized to account for an arbitrary method of context sensitivity. For example, consider a points-to analysis that distinguishes context per call site. A MOD algorithm based on this points-to analysis will compute $Mod(s, c)$ sets for each statement s and for each call site of the enclosing method of s. For virtual calls (lines 6 through 10), the target methods will be determined for each receiver object o in context c, based on the class of o and the compile-time target m. The update of $Mod(s, c)$ at line 9 will be altered to add the set $MMod(target(o, m), c)$ of objects associated with the target method under the context represented by call site c.

In the case of context-insensitive MOD analysis, there will be a single *Mod* set for each statement and a single *MMod* set for each method. For example, the treatment of virtual calls becomes

```
[6]  foreach virtual call s: l = r.m(...) ∈ Stmt do
[7]    foreach object o∈Pt(r) do
[8]      Mod(s) := Mod(s) ∪ MMod(target(o, m))
[9]      add Mod(s) to MMod(EnclMethod(s))
```

Example. Consider the example in Figure 4. MOD analysis based on context-insensitive points-to analysis erroneously determines that the *Mod* sets for statements 2 and 5 are $\{o_3, o_4\}$. Consider a MOD analysis based on the object-sensitive points-to analysis from Section 3. The statement at line 1 appears in two contexts: o_3 and o_4. Therefore, $MMod(A.A, o_3)$ is $\{o_3\}$ and $MMod(A.A, o_4)$ is $\{o_4\}$. The receiver for the call statement at line 2 is o_3; therefore the MOD analysis infers that $Mod(2, o_3)$ is $\{o_3\}$. Similarly, $Mod(5, o_4)$ is $\{o_4\}$.

The MOD algorithm from Figure 7 can be easily adapted to exploit the implementation techniques from Section 5. Recall that if a client analysis for the points-to analysis does not need the individual points-to sets for the context replicas of a variable $v \in R^*$, in some cases all replicas of v can be eliminated. The MOD analysis can be modified so that the points-to sets of context copies of certain formal parameters are not necessary as part of the MOD analysis input. In particular, this is possible for each formal parameter that is not assigned inside the body of its method; in reality, formals are almost never assigned inside methods. Let *Param* denote the set of explicit formal parameters *param* such that (1) *param* is never assigned in the body of its method, and (2) $param \in R^*$. The modified MOD analysis introduces additional sets *PtParam*: $Param \times \mathcal{C} \rightarrow \mathcal{P}(O')$. Set $PtParam(p, o)$ contains the points-to sets of context copies p^o of a formal parameter $p \in Param$. All such sets are initialized to $\emptyset$ in the beginning of the MOD analysis. The modified part of the analysis becomes

```
[5]  while changes occur in Mod, MMod or PtParam do
[6]    foreach instance field write s: param.f = q do
[7]      foreach context c in which s appears do
[8]        Mod(s, c) := PtParam(param, c)
[9]        add Mod(s, c) to MMod(EnclMethod(s), c)
[10]   foreach virtual call s: l = r.m(r1, ..., rn) do
[11]     foreach context c in which s appears do
[12]       foreach object o∈Pt(r^c) do
[13]         add MMod(target(o, m), o) to Mod(s, c)
[14]         foreach param_i where 1 ≤ i ≤ n do
[15]           add Pt(r_i^c) to PtParam(param_i, o)
[16]       add Mod(s, c) to MMod(EnclMethod(s), c)
```

Lines 6–9 update the set $Mod(s, c)$ for indirect write statements s through $param \in Param$ whenever the corresponding set $PtParam(param, c)$ is modified. Note that in most cases indirect writes are done through implicit parameters `this`; thus, one would expect that the analysis will perform work due to lines 6–9 infrequently. Lines 14–15 propagate the points-to set of the context copy of the actual to the set $PtParam(param_i, o)$. If $r_i \in Param$, $Pt(r_i^c)$ is $PtParam(r_i, c)$; otherwise $Pt(r_i^c)$ is $Pt(map(r_i, c))$. In most programs chains of parameters are rare (i.e., typically $r_i \notin Param$) and the amount of work due to statements 14–15 is expected to be small.

In addition, the implementation of the MOD analysis can take advantage of the fact that some variables are not in R^*. For some statements the analysis does not need to maintain multiple *Mod* sets; the set of such statements is denoted by S_{ci}. Similarly, for some methods the analysis does not need to maintain multiple *MMod* sets; the set of such methods is denoted by M_{ci}. Consider an assignment s_i: `p.f=q` where $p \notin R^*$, and a call s_j: `x.m(..)` where $x \notin R^*$. Clearly, $Mod(s_i, c') = Pt(p)$ for all contexts c' of s_i and $Mod(s_j, c'') = \cup\{MMod(target(o, m), o) \mid o \in Pt(x)\}$ for all contexts c'' of s_j. Thus, the analysis needs to maintain only a single *Mod* set for each such statement. If all statements in a method m are in S_{ci}, then m is in M_{ci}. Furthermore, call statements can be added repeatedly to S_{ci} as follows: whenever all possible targets of a call become members of M_{ci}, the call itself can be added to S_{ci}. Using these techniques, the analysis can avoid storing and processing redundant context-sensitive *Mod* and *MMod* sets.

7. DEF-USE ANALYSIS

The goal of def-use analysis is to compute *def-use associations* between pairs of statements. A def-use association for a memory location l is a pair of statements (m, n) such that m assigns a value to l and subsequently n uses that value. For procedural languages such as Fortran and C, there are well-known algorithms (e.g., [3, 20, 32]) for computing *intraprocedural* associations in which m and n are in the same procedure, as well as *interprocedural* associations in which m and n are in different procedures. This information has a wide variety of uses in optimizing compilers (e.g. for dependence analysis) and in software tools (e.g. for slicing and for data-flow-based testing). In the presence of pointers, def-use analyses must use the output of a pointer analysis to disambiguate indirect definitions and indirect uses. Typically, this analysis is followed by a reaching definitions analysis which determines sets of definitions that may reach each program statement, in order to identify the def-use associations.

For object-oriented languages, such def-use analyses can be defined in a similar manner. For Java, there are three different categories of memory locations that need to be considered. *Local variables* (including formals) in Java cannot

have aliases, and therefore only intra-method def-use associations can exist for them. This is true both for variables of primitive types and for variables of reference types. Traditional intraprocedural def-use analyses can be trivially applied in this situation. *Global variables* (i.e., `static` fields) also cannot have aliases, but their def-use associations may cross method boundaries. Such associations can be identified with traditional interprocedural def-use analyses.

Def-use associations for *object fields* can be computed in a manner similar to associations in procedural languages with pointers. Points-to analysis must be used to determine which objects may be accessed by expressions of the form $p.f$. For each object o_i in the points-to set of p, memory location $o_i.f$ is added to the *DEF* or *USE* set for the corresponding statement.

7.1 Standard Def-Use Analysis

In this subsection we present a def-use analysis for object fields that directly instantiates standard techniques for interprocedural def-use analysis [32]. This analysis takes as input the solution computed by the parameterized object-sensitive points-to analysis, and computes a set of def-use associations. The analysis input also contains the interprocedural control-flow graph (ICFG) of the program [39]. The ICFG is a directed graph with nodes representing statements and edges representing flow of control between statements. Each method has associated a single *entry node* and a single *exit node*. Each call statement is represented by a pair of nodes, a *call node* and a *return node*. As described later, the ICFG contains edges from a call node to the entry nodes of the methods invoked by the call site (for virtual calls there may be multiple outgoing edges). The graph also contains matching edges from the exit nodes of the called methods to the return node at the call site.

Without loss of generality, we make the simplifying assumption that any statement that writes an object field is of the form $p.f = q$ and any statement that reads an object field has the form $q = p.f$. Therefore, the *DEF* set of an ICFG node n can be defined as

$$DEF(n) = \{o \mid n \text{ occurs in context } c \wedge o \in Pt(p^c)\}$$

if n has the form $p.f = q$, and as $DEF(n) = \emptyset$ for all other nodes. $USE(n)$ can be defined similarly. For example, for the program in Figure 4, the non-empty sets are $DEF(1) = \{o_3.f, o_4.f\}$, $USE(3) = \{o_3.f\}$, and $USE(6) = \{o_4.f\}$.

For each node n, the analysis computes a set of *reaching definitions* $RD(n)$. Each definition is a pair of the form $(m, o.f)$, where m is an ICFG node. If $(m, o.f) \in RD(n)$ and $o.f \in USE(n)$, the analysis reports the def-use association $(m, n, o.f)$. For example, the computed associations for Figure 4 are $(1, 3, o_3.f)$ and $(1, 6, o_4.f)$.

The analysis can be defined as an instance of a more general framework for interprocedural analysis due to Reps et al. [32]. The system of equations in Figure 8 defines the analysis semantics. For each method m, the set $MDEF(m)$ contains the definitions that are created in m and in all direct or indirect callees of m. All such definitions are propagated back to the return nodes of calls to m. This is due to the fact that the analysis cannot perform kills of definitions, since it has only "may-point-to" information.

Interprocedural propagation is based on sets $Callees(n)$ and $CallNodes(n)$. If n is the call node for a virtual call $p.m(\ldots)$, $Callees(n)$ is the set of methods $\{target(o, m) \mid o \in Pt(p^c)\}$ for each context c in which n occurs. This set encodes the (call,entry) edges at the call site. For non-virtual instance calls and for static calls, the set contains only the compile-time target method. A similar set $Callees(m)$ can be defined for a method m, as the union of the corresponding sets for all call sites inside m. For an entry node n, set $CallNodes(n)$ contains all call nodes n' such that $Callees(n')$ contains the enclosing method of n.

input $Stmt$: set of statements $Pt : R \to \mathcal{P}(O)$
output DU : subset of $(Stmt \times Stmt \times O \times F)$

For each method m:

$$DEF(m) = \bigcup_{n \in m} \{(n, o.f) \mid o.f \in DEF(n)\}$$
$$MDEF(m) = DEF(m) \cup \bigcup_{m' \in Callees(m)} MDEF(m')$$

For each node n that is not an entry node or a return node:

$$RD(n) = \bigcup_{n' \in Pred(n)} RD(n') \cup \{(n', o.f) \mid o.f \in DEF(n')\}$$

For each return node n with a corresponding call node n':

$$RD(n) = RD(n') \cup \bigcup_{m \in Callees(n')} MDEF(m)$$

For each entry node n:

$$RD(n) = \bigcup_{n' \in CallNodes(n)} RD(n')$$

Def-use solution:

$$DU = \bigcup_{n \in Stmt} \{(m, n, o.f) | (m, o.f) \in RD(n) \wedge o.f \in USE(n)\}$$

Figure 8: Standard def-use analysis.

Example. Consider the statements in Figure 4. Since $DEF(1) = \{o_3.f, o_4.f\}$, $MDEF(B.B) = \{(1, o_3.f), (1, o_4.f)\}$. Therefore, these two definitions reach the bottom of statement 10 and are propagated to statement 12. As a result, $RD(3) = \{(1, o_3.f), (1, o_4.f)\}$, and since $USE(3) = \{o_3.f\}$, the analysis reports def-use association $(1, 3, o_3.f)$. Similarly, $RD(6) = \{(1, o_3.f), (1, o_4.f)\}$, which results in association $(1, 6, o_4.f)$.

7.2 Object-Sensitive Def-Use Analysis

Any points-to analysis can be used to construct sets *DEF* and *USE* in order to compute def-use associations for object fields. However, a straightforward application of standard def-use analyses may introduce imprecision. For the example in Figure 4, both $(1, o_3.f)$ and $(1, o_4.f)$ reach the bottom of node 10, while in reality only the first definition is feasible for that program point. For this particular example the imprecision does not create infeasible def-use associations. However, in general an arbitrary number of infeasible associations may be introduced.

Example. Consider the set of statements in Figure 9. In

```
  class X { ...  }
  class A {
      X f;
1     void m() { X z = this.f; }
2     void n(X x) { this.f = x; }
      static void main() {
3        s1: X x1 = new X();
4        s2: X x2 = new X();
5        s3: A p = new A();
6        s4: A q = new A();
7           p.m();
8           p.n(x1);
9           q.m();
10          q.n(x2); }   }
```

Figure 9: Imprecision of standard def-use analysis.

this case $USE(1) = DEF(2) = \{o_3.f, o_4.f\}$. Therefore, the definitions reaching the bottom of statement 8 are $(2, o_3.f)$ and $(2, o_4.f)$, and due to statement 9 they are propagated to the body of method `m`. As a result, the analysis reports def-use associations $(2, 1, o_3.f)$ and $(2, 1, o_4.f)$. However, both associations are infeasible because field f is null in both cases when $m()$ is called.

A potential source of imprecision in the standard def-use analysis is the fact that it treats uniformly all definitions created within a method. As a result, such definitions are propagated back to *all* callers of the method. In the example above, definition $(2, o_4.f)$ is valid only if the calling context is o_4. Therefore, this definition should not be propagated back to statement 8, because the calling context at 8 is o_3. This problem can be solved if the def-use analysis uses the output of an object-sensitive points-to analysis. In this case, the def-use analysis itself can be object-sensitive, as described below.

For each node n that has the form $p.f = q$, we define several *DEF* sets as follows:

$$DEF(n, c) = \{o \mid n \text{ occurs in context } c \wedge o \in Pt(p^c)\}$$

For all other nodes, $DEF(n, c) = \emptyset$. Sets $USE(n, c)$ can be defined similarly. For example, for the program in Figure 9, the non-empty sets are $USE(1, o_3) = \{o_3.f\}$, $USE(1, o_4) = \{o_4.f\}$, $DEF(2, o_3) = \{o_3.f\}$, and $DEF(2, o_4) = \{o_4.f\}$.

There are also multiple *RD* sets for each node: $RD(n, c)$ contains pairs of the form $(m, o.f)$ for definitions that reach n when the calling context for the enclosing method is c. If $(m, o.f) \in RD(n, c)$ and $o.f \in USE(n, c)$, the analysis reports the association $(m, n, o.f)$. For an entry node n and a context c, the analysis uses a set $CallNodes(n, c)$ defined as follows: if the enclosing method of n is called under context c from a call site node n' when n' occurs in context c', (n', c') is in $CallNodes(n, c)$. This set can be constructed easily from the object-sensitive points-to solution. During the def-use analysis, for each pair (n', c') in this set, the definitions reaching n' in context c' are propagated to entry node n for context c.

The semantics of the object-sensitive def-use analysis is defined by the equations in Figure 10. Whenever information is propagated into a callee or back to a caller, the propagation takes into account the available information about the calling context. For example, if a definition reaches a call site n under some context c for n's enclosing method, the definition is propagated only into callees that are potentially invoked by n under context c. Similarly, a definition created inside a callee under some context is propagated back only to callers that introduce this context (this restriction is encoded by the definition of *CallMDEF*).

For the example in Figure 9, the set of definitions reaching the bottom of node 8 is $CallMDEF(8, \epsilon)$. Since $Pt(p^\epsilon) = \{o_3\}$, this is equivalent to $MDEF(n, o_3)$, which contains only $(2, o_3.f)$. Because the object-sensitive analysis keeps track of the calling context for n, it avoids propagating the spurious definition $(2, o_4.f)$ back to the bottom of node 8. In the final solution, $RD(1, o_3) = \emptyset$ and $RD(1, o_4) = \{(2, o_3.f)\}$. Since $USE(1, o_4) = \{o_4.f\}$, the object-sensitive analysis correctly concludes that there are no def-use associations involving nodes 1 and 2. In contrast, the standard def-use analy-

input $Stmt$: set of statements Pt : $R' \to \mathcal{P}(O')$
output DU : subset of $(Stmt \times Stmt \times O' \times F)$

For each method m:

$$DEF(m, c) = \bigcup_{n \in m} \{(n, o.f) \mid o.f \in DEF(n, c)\}$$

$$MDEF(m, c) = DEF(m, c) \cup \bigcup_{\text{call site } n \in m} CallMDEF(n, c)$$

$$CallMDEF(p.x(..), c) = \bigcup_{o \in Pt(p^c)} MDEF(target(o, x), o)$$

$CallMDEF(x(..), c) = MDEF(x, \epsilon)$ for a static call

For each node n that is not an entry node or a return node:

$$RD(n, c) = \bigcup_{n' \in Pred(n)} RD(n', c) \cup \{(n', o.f) | o.f \in DEF(n', c)\}$$

For each return node n with a corresponding call node n':

$$RD(n, c) = RD(n', c) \cup CallMDEF(n', c)$$

For each entry node n:

$$RD(n, c) = \bigcup_{(n', c') \in CallNodes(n, c)} RD(n', c')$$

$CallNodes(n, c)$ contains all (n', c') such that n' is a call node that occurs in context c' and
- $\diamond$ if n' is $p.x(..)$, then $c \in Pt(p^{c'})$ and n is in $target(c, x)$
- $\diamond$ if n' is $x(..)$, then $c = \epsilon$ and n is in x

Def-use solution:

$$DU = \bigcup_{n \in Stmt} \{ (m, n, o.f) \mid n \text{ occurs in context } c \wedge (m, o.f) \in RD(n, c) \wedge o.f \in USE(n, c)\}$$

Figure 10: Object-sensitive def-use analysis.

sis presented in Figure 8 reports two spurious associations: $(2, 1, o_3.f)$ and $(2, 1, o_4.f)$.

7.3 Contextual Def-Use Analysis

Object-oriented programs typically manipulate object state through the invocation of instance methods. Therefore, for the purposes of data-flow-based testing (e.g., for test construction), standard def-uses may provide insufficient information because they do not include information about the calling context enclosing definition and use statements. To address this problem, Souter and Pollock propose *contextual def-use associations*, a generalization of standard def-uses which includes information about the context enclosing field writes and field reads [40].

A contextual def-use association is a tuple of the form $(CDEF, CUSE, o.f)$ where $CDEF$ is a chain of call sites enclosing a statement $p.f = q$ which writes the field f of location o; similarly, $CUSE$ is a chain of call sites enclosing a statement $r = s.f$ that reads location o. Unlike standard def-use analysis which assigns non-empty DEF sets *exclusively* to statements of the form $p.f = q$, the contextual def-use analysis assigns $CDEF$ sets to call sites; the call sites lead directly or indirectly to a definition statement $p.f = q$. For example, for the set of statements in Figure 4, the contextual analysis associates a $CDEF$ tuple (10-2-1,$o_3.f$) with statement 10 and a $CUSE$ tuple (12-3,$o_3.f$) with statement 12; instead of computing def-use association (1,3,$o_3.f$), this analysis computes association (10-2-1,12-3,$o_3.f$).

Contextual def-use associations can be computed using object-sensitive points-to analysis. The computation consists of two phases. In the first phase, the analysis determines the $CDEF$ and $CUSE$ tuples associated with program statements. In the second phase, it uses this information to construct the contextual def-use associations.

7.3.1 Construction of *CDEF* and *CUSE* Tuples

Object-sensitive points-to analysis can be used to determine the $CDEF$ and $CUSE$ sets for program statements. The algorithm for $CDEF$ construction, parameterized by an object-sensitive points-to analysis, is described in Figure 11 ($CUSE$ construction is essentially the same). For brevity, we only show calls to instance methods; calls to static methods can be handled similarly. The algorithm produces object-sensitive sets $CDEF(n, c)$ for any node n that represents a field write or a call; for all other nodes, the $CDEF$ sets are empty.

Similarly to the propagation of standard DEF sets from callees to callers (Figure 10), statements of the form $p.f = q$ along with the modified location are propagated to the callers. This propagation is expressed through the equations for statements of the form $p.x(...)$, which compute the $CallMDEF$ and $CDEF$ sets. The call site at the propagation point is attached to the chain which forms the context for the definition. The backward propagation is defined in terms of auxiliary function $Escapes$. For a method m and an object o, $Escapes(m, o)$ holds if and only if o is reachable in the points-to graph from the formal parameters of m, the return variable of m, or a global (static) variable. Intuitively, if the object escapes, it is passed from the caller or escapes from the callee to the caller; therefore, the call site affects the definition and becomes part of the contextual chain [40]. For example, accesses through `this` are always propagated to the caller.

Appropriate mechanisms can be used to prevent infinite call chains: for example, decomposition of the call graph into strongly connected components [40], or a k-limit on the length of the call chains. Limiting the call chains to the last call site provides a practical approximation of the precise contextual def-use analysis.

Example. Consider the example in Figure 4. Using the algorithm in Figure 11 we have

$$EDEF(1, o_3) = EDEF(\texttt{A.A}, o_3) = \{(1, o_3.f)\}$$

$$EDEF(1, o_4) = EDEF(\texttt{A.A}, o_4) = \{(1, o_4.f)\}$$

Propagating these definitions to the callers of `A.A` and attaching the corresponding call sites results in

$$MDEF(\texttt{B.B}, o_3) = \{(2\text{-}1, o_3.f)\}$$

$$MDEF(\texttt{C.C}, o_4) = \{(5\text{-}1, o_4.f)\}$$

The algorithm computes the following $CDEF$ sets:

$$CDEF(10, \epsilon) = \{(10\text{-}2\text{-}1, o_3.f)\}$$

input $Stmt$: set of statements $Pt : R' \to \mathcal{P}(O')$
output $CDEF : Stmt \times \mathcal{C} \to \mathcal{P}(Stmt^k \times O' \times F)$

For each node n of the form $p.f = q$:

$$CDEF(n, c) = \{(n, o.f) \mid n \text{ occurs in context } c \\ \wedge\ o \in Pt(p^c) \wedge \neg Escapes(EnclMethod(n), o)\}$$

$$EDEF(n, c) = \{(n, o.f) \mid n \text{ occurs in context } c \\ \wedge\ o \in Pt(p^c) \wedge Escapes(EnclMethod(n), o)\}$$

For each node n not of the form $p.f = q$:

$$CDEF(n, c) = \emptyset \text{ if } n \text{ is not a call site}$$

$$EDEF(n, c) = \emptyset$$

For each method m:

$$EDEF(m, c) = \bigcup_{n \in m} EDEF(n, c)$$

$$MDEF(m, c) = EDEF(m, c) \cup \bigcup_{call\ site\ n \in m} CallMDEF(n, c)$$

For each node n corresponding to a call site $p.x(..)$:

$$CallMDEF(n, c) = \bigcup_{o \in Pt(p^c)} \{(n\text{-}n', o'.f) | (n', o'.f) \in \\ MDEF(target(o, x), o) \wedge Escapes(EnclMethod(n), o')\}$$

$$CDEF(n, c) = \bigcup_{o \in Pt(p^c)} \{(n\text{-}n', o'.f) | (n', o'.f) \in \\ MDEF(target(o, x), o) \wedge \neg Escapes(EnclMethod(n), o')\}$$

Figure 11: Algorithm for $CDEF$ computation.

For each node n that is not a return node or an entry node:

$$RD(n,c) = \bigcup_{n' \in Pred(n)} RD(n',c) \cup CDEF(n',c)$$

For each return node n with a corresponding call node n':

$$RD(n,c) = RD(n',c) \cup CDEF(n',c)$$

For each entry node n:

$$RD(n,c) = \emptyset$$

Figure 12: Contextual reaching definitions.

$$CDEF(11,\epsilon) = \{(11\text{-}5\text{-}1, o_4.f)\}$$

Similarly, the contextual uses are $CUSE(12,\epsilon)$=$\{(12\text{-}3, o_3.f)\}$ and $CUSE(13,\epsilon)$=$\{(13\text{-}6, o_4.f)\}$.

As another example, consider again the statements in Figure 4 and suppose that class `X` has an integer field `g` and method `Z.n` contains a statement `this.g=0` at line 33. Also, suppose that the following statement is added after line 13:

```
14  int h = z.g;
```

In this case, the contextual def tuple for field g of object o_2 is $(13\text{-}7\text{-}33, o_2.g)$, and the corresponding use tuple is $(14, o_2.g)$.

7.3.2 Construction of Def-Use Associations

In the second phase, the analysis computes reaching definitions for each node by propagating *CDEF* tuples intraprocedurally on the control flow graph of the method enclosing the *CDEF* tuple. The algorithm for *RD* computation is shown in Figure 12; it computes essentially the same information as computed by the tuple construction algorithm from [40].

The algorithm propagates to a return node the reaching definitions for its corresponding call node $n' \in m$, and the *CDEF* tuples associated with n'. Note that no definition is propagated backwards from the callee. Every statement of the form n: $p.f = q$ that modifies an object visible in method m is propagated during the *CDEF* construction phase of the analysis (shown in Figure 11). Therefore, n appears with the appropriate context either in $CDEF(n',c)$ or in a *CDEF* tuple in some caller of m. Similarly, no definition is propagated to the entry of a method, because the corresponding uses have already been propagated backwards with the appropriate context during the first phase of the analysis.

Contextual def-use tuples are constructed by examining $RD(n,c)$ and $CUSE(n,c)$ for every node n. For each contextual definition $cd \in RD(n,c)$ that writes $o.f$ and for each matching $cu \in CUSE(n,c)$, the algorithm computes a def-use association $(cd, cu, o.f)$. If n is a call node, the algorithm also considers each $cd \in CDEF(n,c)$, and creates def-use associations with matching $cu \in CUSE(n,c)$.

Example. For the example in Figure 4, contextual definition $(10\text{-}2\text{-}1, o_3.f)$ reaches nodes 11, 12, and 13. Similarly, $(11\text{-}5\text{-}1, o_4.f)$ reaches nodes 12 and 13. At node 12, $(10\text{-}2\text{-}1, o_3.f)$ matches with the corresponding use $(12\text{-}3, o_3.f)$, which results in contextual def-use tuple $(10\text{-}2\text{-}1, 12\text{-}3, o_3.f)$. Similarly, at node 13, $(11\text{-}5\text{-}1, o_4.f)$ is matched with $(13\text{-}6, o_4.f)$ which leads to the construction of $(11\text{-}5\text{-}1, 13\text{-}6, o_4.f)$. If the example is augmented with an integer field `g` as described earlier, tuple $(13\text{-}7\text{-}33, 14, o_2.g)$ will also be created.

7.3.3 Imprecision of Context-Insensitive Analysis

Clearly, *CDEF* and *CUSE* tuples can be computed context-insensitively. However, ignoring the context in the backward propagation in Figure 11 can introduce substantial imprecision because all modified locations are propagated uniformly to the callers, regardless of the invocation context.

For example, consider the set of statements in Figure 4. If the algorithm in Figure 11 does not take into account object context, it will compute sets $EDEF(1) = EDEF(\texttt{A.A}) = \{(1, o_3.f), (1, o_4.f)\}$. Propagating these definitions to the callers of `A.A` and attaching the corresponding call sites results in

$$MDEF(\texttt{B.B}) = \{(2\text{-}1, o_3.f), (2\text{-}1, o_4.f)\}$$
$$MDEF(\texttt{C.C}) = \{(5\text{-}1, o_4.f), (5\text{-}1, o_3.f)\}$$

which leads to the infeasible contextual definitions $(10\text{-}2\text{-}1, o_4.f) \in CDEF(10)$ and $(11\text{-}5\text{-}1, o_3.f) \in CDEF(11)$. This imprecision results in infeasible contextual def-use associations $(10\text{-}2\text{-}1, 13\text{-}6, o_4.f)$ and $(11\text{-}5\text{-}1, 12\text{-}3, o_3.f)$.

8. EMPIRICAL RESULTS

We chose to implement two particular instantiations of the parameterized object-sensitive points-to analysis. In the first instantiation (denoted by *ObjSens1*) we keep context-sensitive information for implicit parameters `this` and formal parameters of instance methods and constructors. In the second instantiation (denoted by *ObjSens2*) we keep context-sensitive information for implicit parameters `this`, formal parameters, and return variables of instance methods and constructors. In both cases, we use context depth 1 for allocation sites. Since methods and constructors in Java are usually short, keeping precise information for formals (including `this`) and for return variables has the potential to improve considerably the points-to solution without significant increase in analysis cost. Other potentially beneficial instantiations of the object-sensitive framework may use context depth of 2 for allocation sites in container classes—for example, the array of hash entries in `Hashtable`, the array of objects in `Vector`, etc. Such increased precision in the object naming will allow to avoid "sharing" of objects stored in different containers. More generally, object naming with context depth ≥ 2 may be beneficial for sub-objects of composite objects: for example, for allocations sites in constructors which create new objects that are immediately assigned to instance fields of the object that is being constructed.

In addition to the two object-sensitive analyses, we implemented a context-sensitive analysis based on an instantiation of the popular call string approach to context sensitivity for call string length $k = 1$. This analysis distinguishes context *per call site* and is conceptually similar to the 1-1-CFA algorithm from [18, 17]. In order to allow comparison with the two object-sensitive analyses, the context replication is performed for `this`, formal parameters, and return variables in instance methods and constructors. As discussed in Section 3.2, this approach to context sensitivity is theoretically incomparable with the object-sensitive analyses.

Program	User Class	Size (Kb)	Whole-program		
			Class	Method	Stmt
proxy	18	56.6	565	3283	58837
compress	22	76.7	568	3316	60010
db	14	70.7	565	3339	60747
jb-6.1	21	55.6	574	3393	60898
echo	17	66.7	577	3544	62646
raytrace	35	115.9	582	3451	62755
mtrt	35	115.9	582	3451	62760
jtar-1.21	64	185.2	618	3583	65112
jlex-1.2.5	25	95.1	578	3381	65437
javacup-0.10	33	127.3	581	3564	66463
rabbit-2	52	157.4	615	3770	68277
jack	67	191.5	613	3573	69249
jflex-1.2.2	54	198.2	608	3692	71198
jess	160	454.2	715	3973	71207
mpegaudio	62	176.8	608	3531	71712
jjtree-1.0	72	272.0	620	4078	79587
sablecc-2.9	312	532.4	864	5151	82418
javac	182	614.7	730	4470	82947
creature	65	259.7	626	3881	83454
mindterm1.1.5	120	461.1	686	4420	90451
soot-1.beta.4	677	1070.4	1214	5669	92521
muffin-0.9.2	245	655.2	824	5253	94030
javacc-1.0	63	502.6	615	4198	102986

Table 1: Characteristics of the data programs. First two columns show the number and bytecode size of user classes. Last three columns include library classes.

Object-sensitive analyses *ObjSens1* and *ObjSens2* were compared with Andersen's context-insensitive analysis (denoted by *And*) and with the call string context-sensitive analysis (denoted by *CallSite*). The three context-sensitive analyses were built on top of an existing constraint-based implementation of Andersen's analysis [33]. In our implementation we use the techniques described in Sections 5 and 6 in order to identify context-independent statements and to eliminate unnecessary replication of replicated variables. The Soot framework (`www.sable.mcgill.ca`) was used to process Java bytecode and to build a typed intermediate representation [47]. The points-to analysis implementation used the BANE toolkit (`bane.cs.berkeley.edu`) for constraint-based program analysis [4].

All experiments were performed on a 900MHz Sun Fire-280R shared machine with 4Gb physical memory. The reported times are the median values out of three runs. We used 23 publicly available data programs, ranging in size from 56Kb to about 1Mb of bytecode. The same set of programs was used in our previous work on Andersen's analysis [33]. The set includes programs from the SPEC JVM98 suite, other benchmarks used in previous work on analysis for Java, as well as programs from an Internet archive (`www.jars.com`) of popular publicly available Java applications.

Table 1 shows some characteristics of the data programs. The first two columns show the number of user (i.e., non-library) classes and their bytecode size. The next three columns show the size of the program, including library classes, after using class hierarchy analysis (CHA) [11] to filter out irrelevant classes and methods. CHA is an inexpensive analysis that determines the possible targets of a virtual call by examining the class hierarchy of the program. The number of methods is essentially the number of nodes in the call graph computed by CHA. The last column shows the number of statements in Soot's intermediate representation. In these size measurements and in the subsequent points-to analyses, the effects of JVM startup code and native methods (for JDK 1.1.8) are encoded in stubs included in the analysis input. Dynamic class loading (e.g., through `Class.forName`) and reflection (e.g., calls to `Class.newInstance`) are resolved manually; similar approaches are typical for static whole-program compilers and tools [22, 14, 45, 46].

8.1 Analysis Cost

The measurements of analysis cost are presented in Table 2. The first two columns show the running time and memory usage of Andersen's analysis. The next columns show the cost of *CallSite*, *ObjSens1*, and *ObjSens2*. The running times are for the executions of the constraint-based analyses, not including the construction of the Soot intermediate representation.

The empirical results demonstrate that the object-sensitive analyses are practical in terms of running time and memory consumption. For the majority of programs they have comparable performance to Andersen's analysis. In certain cases (e.g., `sablecc` and `creature`) the cost of the object-sensitive analyses is significantly lower than the cost of the context-insensitive analysis. There are two factors that could explain these results. First, the improved precision produces smaller points-to sets, which results in less work and reduced memory consumption for the analysis. In the case when the points-to sets are significantly smaller, *ObjSens1* and *ObjSens2* can actually run faster than *And*, as observed for some of our data programs. Second, even if the points-to sets were the same, for many statements *And*, *ObjSens1*, and *ObjSens2* perform comparable amount of work. One might expect that because the object-sensitive analyses analyze context-dependent statements multiple times (once for each context), they would be more expensive. However, for any statement s that accesses the receiver object (e.g., any s containing `this`), there are as many different contextual versions as the number of receivers of the enclosing method. When *And* processes s, it has to consider all of the possible receivers. The amount of work that *And* has to perform for one receiver roughly corresponds to the amount of work that *ObjSens1* and *ObjSens2* perform for one contextual version. Therefore, for this statement *And* and the object-sensitive analyses have comparable cost. Given that many statements in instance methods and constructors access the receiver object, this may explain why the analyses exhibit comparable costs.

For the majority of programs, adding return variables to R^* does not result in increased analysis cost. The reason is that in most cases assignments to return variables can be analyzed as context-independent (recall from Section 5 that context-independent statements can be analyzed only once, regardless of the calling contexts). If this is not the case (e.g., for statement `ret_var=this.f`), the statement typically involves implicit parameter `this`; for such statements *And*, *ObjSens1*, and *ObjSens2* perform a comparable amount of work. In certain cases the replication of return

Program	*And*		*CallSite*		*ObjSens1*		*ObjSens2*	
	Time [sec]	Mem [Mb]	Time [sec]	Mem [Mb]	Time [sec]	Mem [Mb]	Time [sec]	Mem [Mb]
proxy	4.4	40	6.9	39	5.9	40	6.1	40
compress	12.0	46	12.0	47	8.8	46	13.2	46
db	12.2	47	11.4	47	11.7	48	12.2	46
jb	7.5	43	7.3	43	7.1	43	7.0	43
echo	27.3	60	24.8	59	24.3	59	27.2	59
raytrace	13.9	50	12.6	51	13.6	50	13.8	50
mtrt	15.4	50	12.9	51	15.2	50	15.6	50
jtar	23.3	58	19.9	56	21.4	56	20.6	56
jlex	8.5	46	9.0	46	8.8	46	8.7	46
javacup	13.1	57	17.3	56	16.8	53	15.1	56
rabbit	16.1	52	14.5	52	13.9	52	13.9	52
jack	38.4	62	38.1	62	37.5	62	37.6	62
jflex	20.2	71	20.1	70	19.7	70	19.7	70
jess	24.6	67	26.1	67	24.3	66	26.9	67
mpegaudio	15.4	52	14.1	54	16.2	52	13.2	54
jjtree	12.4	53	11.8	53	11.2	53	8.7	51
sablecc	68.7	115	40.1	94	62.3	113	35.9	94
javac	427.6	121	430.7	123	430.2	120	416.9	120
creature	85.2	100	61.1	86	55.7	88	57.6	86
mindterm	44.5	91	44.9	89	49.9	88	42.6	92
soot	80.8	130	92.6	132	69.8	128	89.7	132
muffin	110.0	144	108.4	135	99.6	132	101.1	133
javacc	76.2	112	82.3	112	81.4	116	80.1	112

Table 2: Running time and memory usage of the analyses.

variables has substantial effect on the analysis. For example, `sablecc` declares a pair of methods (`set` and `get`) at the root of a wide and deep inheritance hierarchy. These methods write and read a field of the receiver object, and are invoked frequently on receivers of different classes. *ObjSens1* is able to separate the object context of an invocation for `set` and the object fields are assigned correctly. However, when the `get` method is invoked, the results are merged over all possible receivers because return variables are not distinguished for different contexts. By replicating return variables *ObjSens2* avoids this imprecision, which results in significant reduction in the running time. Although the most substantial benefits from object sensitivity are due to parameter replication, replicating return variables does not increase analysis cost and in some cases results in significant precision and cost improvements. Therefore, it is beneficial to replicate return variables in addition to formal parameters.

Similarly to the object-sensitive analyses, the call string context-sensitive analysis achieves practical cost. This is due to two factors. First, it is implemented using optimization techniques analogous to the ones described in Section 5. Second, *CallSite* also benefits from increased precision over context-insensitive analysis. However, for the nine largest programs in our experiments, the running times of *CallSite* are slower than *ObjSens2*; this may be explained by the fact that *CallSite* is less precise than *ObjSens2*.

8.2 Analysis Precision

We evaluated the precision improvements of *ObjSens2* over the context-insensitive analysis and the call string analysis with respect to MOD analysis, call graph construction, and virtual call resolution.

8.2.1 MOD Analysis

Using the MOD algorithm described in Section 6, we performed measurements for *ObjSens2*, *CallSite*, and *And* in order to estimate the impact of the analyses on MOD analysis. More precise points-to analyses produce a smaller number of modified objects per statement.

We considered all methods that both *ObjSens2* and *CallSite* reported as potentially executable (i.e., methods that are reachable from `main`, static initializers, and JVM start-up methods). For all statements in such methods, we computed (1) *Mod* sets according to the algorithm from Figure 7, (2) *Mod* sets using *CallSite* and the corresponding version of the algorithm from Figure 7 which distinguishes context per invocation site, and (3) *Mod* sets using *And* and the corresponding context-insensitive version of the algorithm from Figure 7. In order to compare the output of *CallSite* with the output of *And*, we merged the *Mod* sets for *CallSite* for different contexts to obtain a single *Mod* set. Similarly, we merged the *Mod* sets for *ObjSens2* for different object contexts. For example, the aggregate *ObjSens2*-based *Mod* set for line 1 in Figure 4 is $\{o_3, o_4\}$, which is the union of $Mod(1, o_3)$ and $Mod(1, o_4)$. Analogously, the *CallSite*-based *Mod* set for 1 is $\{o_3, o_4\}$, which is the union of $Mod(1, 2)$ and $Mod(1, 5)$.

Table 3 shows the distribution of the number of modified objects per program statement for the three analyses. Only statements that modify at least one object were considered for these results. Each column corresponds to a specific

Program	*And*			*CallSite*			*ObjSens2*		
	1-3	4-9	≥10	1-3	4-9	≥10	1-3	4-9	≥10
proxy	19%	6%	75%	25%	7%	68%	75%	14%	11%
compress	23%	4%	73%	27%	5%	67%	67%	9%	24%
db	20%	4%	76%	24%	4%	72%	48%	25%	27%
jb	15%	5%	80%	20%	5%	75%	67%	20%	13%
echo	25%	6%	69%	30%	5%	65%	63%	11%	26%
raytrace	23%	5%	72%	28%	6%	66%	66%	9%	25%
mtrt	23%	5%	72%	28%	6%	66%	66%	9%	25%
jtar	18%	8%	74%	24%	7%	69%	61%	15%	24%
jlex	17%	4%	79%	20%	4%	76%	56%	34%	10%
javacup	14%	3%	83%	21%	4%	75%	53%	38%	9%
rabbit	18%	5%	77%	23%	6%	71%	47%	13%	40%
jack	17%	3%	80%	20%	3%	77%	53%	8%	39%
jflex	19%	4%	77%	23%	5%	72%	54%	34%	12%
jess	15%	5%	80%	25%	3%	72%	60%	9%	31%
mpegaudio	23%	4%	73%	28%	4%	68%	65%	9%	26%
jjtree	8%	2%	90%	10%	2%	88%	32%	26%	42%
sablecc	20%	3%	77%	32%	4%	64%	52%	15%	33%
javac	14%	4%	82%	18%	6%	76%	37%	5%	58%
creature	18%	3%	79%	27%	3%	70%	54%	13%	33%
mindterm	20%	8%	73%	25%	7%	68%	55%	16%	29%
soot	16%	4%	80%	25%	8%	67%	43%	15%	42%
muffin	16%	4%	80%	24%	4%	72%	45%	7%	48%
javacc	10%	1%	89%	11%	1%	88%	29%	49%	22%
Average	18%	4%	78%	23%	5%	72%	54%	18%	28%

Table 3: Number of modified objects for program statements. Each column shows the percentage of statements whose number of modified objects is in the corresponding range.

range of numbers. For example, the first column corresponds to statements that may modify one, two or three objects, while the last column corresponds to statements that may modify at least 10 objects. Each column shows what percentage of the total number of statements corresponds to the particular range of numbers of modified objects.

The measurements in Table 3 show that object sensitivity significantly improves analysis precision. For MOD analysis based on *ObjSens2*, on average 54% of the statements modify at most three objects. In contrast, for MOD analysis based on *And* this percentage is 18%. It is also significant to note that for *And* nearly 80% of the statements modify at least 10 objects. This indicates substantial imprecision, that can be reduced significantly by using *ObjSens2*.

The results from MOD analysis based on *ObjSens1* are not shown because they are essentially the same, with the exception of `sablecc` for which *ObjSens2* is slightly more precise than *ObjSens1*. Most modifications in object-oriented programs occur through implicit parameter `this` and *ObjSens1* benefits from object sensitivity because it separates `this` for different object contexts.

The experiments demonstrate that distinguishing context per invocation site does not substantially improve MOD analysis precision over *And*. For MOD analysis based on *CallSite*, on average 23% of the statements modify at most three objects. This is substantially worse than 54% for *ObjSens2*, and slightly better than 18% for *And*. In addition, for *CallSite* more than 70% of the statements modify at least 10 objects which indicates substantial imprecision.

Examination of several examples extracted from our benchmarks clearly reveal that the precision of the *CallSite*-based MOD analysis is compromised by inherent object-oriented features such as encapsulation and inheritance. For example, benchmark `jb` defines a class `Namedhash` which extends `java.util.Hashtable` and overrides `Hashtable.put`. It uses the following call sequence: `Namedhash.put` contains a call site c_2: `super.put(...)` and `Hashtable.put` contains a call site c_1: `this.rehash(...)`. Method `Hashtable.rehash` allocates a new array of hash entries and assigns the array to a field of the receiver (i.e., it contains an instance field write statement `this.table=...`). Since c_1 is the only call site in the program that invokes `rehash`, the *CallSite*-based MOD analysis computes a single set $MMod(Hashtable.rehash, c_1)$ for `rehash`. `Hashtable.put` is typically invoked on every instance of class `Hashtable` (and also on every instance of its subclasses); therefore, the set $MMod(Hashtable.rehash, c_1)$ contains *every* direct or indirect instance of `Hashtable`. The *CallSite*-based MOD analysis propagates this set to every call site that invokes `Hashtable.put`, including c_2; subsequently, it propagates the set to every call to `Namedhash.put`. Therefore, the *CallSite*-based MOD analysis determines that each call statement to `Namedhash.put` modifies at least 10 objects due to the instance field write in `rehash`, while in fact it modifies exactly one object (namely, the receiver object at the call to `Namedhash.put`). In contrast, the object-sensitive MOD analysis computes *MMod* sets for `rehash`, `Hashtable.put` and `Namedhash.put` for each possible receiver object; it precisely determines that each call to `Namedhash.put`

Program	*CallSite*		*ObjSens1*		*ObjSens2*	
	(a) Resolved	(b) Removed	(a) Resolved	(b) Removed	(a) Resolved	(b) Removed
proxy	10%	2%	12%	3%	12%	3%
compress	10%	8%	19%	13%	19%	13%
db	9%	8%	17%	14%	17%	14%
jb	42%	5%	45%	5%	45%	5%
echo	6%	9%	10%	13%	10%	13%
raytrace	10%	9%	18%	15%	18%	15%
mtrt	10%	9%	18%	15%	18%	15%
jtar	31%	6%	39%	7%	39%	7%
jlex	32%	5%	40%	5%	40%	5%
javacup	23%	5%	26%	5%	26%	5%
rabbit	19%	8%	31%	11%	31%	11%
jack	3%	7%	5%	12%	5%	12%
jflex	21%	3%	23%	3%	23%	3%
jess	7%	8%	17%	14%	17%	14%
mpegaudio	12%	12%	20%	17%	20%	17%
jjtree	48%	6%	48%	6%	48%	6%
sablecc	21%	183%	10%	1%	24%	183%
javac	3%	8%	6%	10%	7%	10%
creature	8%	3%	21%	5%	21%	5%
mindterm	2%	6%	9%	9%	9%	9%
soot	4%	1%	5%	1%	5%	1%
muffin	1%	5%	3%	6%	3%	7%
javacc	14%	4%	15%	4%	15%	4%
Average	15%	14%	20%	8%	21%	16%

Table 4: Improvements over context-insensitive analysis. (a) Increase in the number of resolved call sites. (b) Improvement in the number of removed target methods.

modifies exactly one object (i.e., the receiver at that call) due to the instance field write statement `this.table=...` in `rehash`.

In object-oriented programs, instance fields are often written by a sequence of intraclass method invocations through the receiver (i.e., through implicit parameter `this`). Thus, context-insensitive MOD analysis is likely to incur substantial imprecision, since it merges the information about indirect modifications over all possible receivers. Call string context-sensitive analysis is also likely to incur substantial imprecision unless the length of the call string is sufficiently large. In the above `Hashtable` example, the analysis needs a call string of length at least 3 in order to determine precisely that each call to method `Namedhash.put` modifies exactly one object due to the instance field write in `rehash`. However, a call string length greater than one is usually considered impractically expensive for large programs.

The above empirical results show that object-sensitive analysis is a promising candidate for producing useful side-effect information. It is able to capture side-effects that arise due to fundamental object-oriented features such as encapsulation and inheritance. In contrast, the context-insensitive analysis and the call string context-sensitive analysis appear to be ill-suited for side-effect analysis of object-oriented languages. The precise information computed by the object-sensitive analysis is important for (1) implementing advanced optimizations in aggressive optimizing compilers, and (2) improving the precision of software productivity tools, with the corresponding reduction in human time and effort spent on software understanding, restructuring, and testing.

8.2.2 *Virtual Call Resolution and Call Graph Construction*

One application of points-to analysis is to determine the potential target methods at virtual call sites. This information can be used to construct the program call graph (which is a prerequisite for all interprocedural analyses) and to identify virtual call sites that can be resolved to a single target method. We performed measurements to evaluate the improvement of *CallSite*, *ObjSens1* and *ObjSens2* over *And* for virtual call resolution and call graph construction. (Andersen's analysis itself already produces relatively precise call graph results [33].)

To determine the improvement in the number of resolved calls, we considered call sites that could not be resolved to a single target method by CHA. Let V be the set of all CHA-unresolved call sites that occur in methods identified as executable by both *ObjSens2* and *CallSite*. We computed the number of sites from V that were resolved to a single target method according to *And*, *CallSite*, *ObjSens1*, and *ObjSens2*. The improvement in the number of resolved call sites for *CallSite*, *ObjSens1* and *ObjSens2* over *And* is shown in columns (a) in Table 4. On average, *CallSite* resolves 15% more call sites than *And*, *ObjSens1* resolves 20%, and *ObjSens2* resolves 21% more call sites. The increased

precision allows better removal of redundant run-time virtual dispatch and enables additional method inlining.

We also computed the total number of removed target methods over all sites in V according to *And*, *CallSite*, *ObjSens1*, and *ObjSens2*. The improvement in the total number of removed target methods for *CallSite*, *ObjSens1*, and *ObjSens2* over *And* is shown in columns (b) in Table 4. On average, *CallSite* removes 14% more target methods than *And*, *ObjSens1* removes 8%, and *ObjSens2* removes 16% more target methods. This improved precision is beneficial for reducing the cost and improving the precision of subsequent interprocedural analyses.

The three context-sensitive analyses are able to improve precision over the context-insensitive analysis. The object-sensitive analysis produces more precise results than the call string context-sensitive analysis because it handles object-oriented features more precisely. Consider the example in Figure 4. If statement s_5 : `C c2 = new C(y)` is added at line 14, *CallSite* will be unable to resolve the virtual call at line 7. In the presence of encapsulation and inheritance, instance fields are often written through a sequence of invocations. A call string analysis will typically need an impractically long call string in order to separate writes to fields of instances of different classes or writes to fields of different instances of the same class. The object-sensitive analysis is able to separate such field writes precisely, which improves the precision of client analyses such as call graph construction and virtual call resolution.

The precision experiments confirm that substantial benefits from object sensitivity are due to parameter replication. Replicating return variables in addition to parameters does not appear to increase analysis cost, and sometimes can result in significant improvements in cost and precision. Therefore, *ObjSens2* is a better candidate than *ObjSens1* for use in optimizing compilers and software tools.

9. RELATED WORK

Flow-insensitive context-sensitive alias analysis for Java has been developed by Ruf [34] in the context of a specialized algorithm for synchronization removal. Ruf's analysis uses method summaries to model context sensitivity and, unlike our analysis, requires bottom-up traversal of the call graph (i.e., a called method is analyzed before or together with its callers). Our analysis is based on Andersen's analysis, which has cubic time worst case complexity [5]; in contrast, Ruf's algorithm is based on the almost-linear Steensgaard's points-to analysis for C [42]. Other context-sensitive points-to analyses for Java are presented in [18, 8]. The algorithm in [8] uses method summaries to model context sensitivity, while reference [18] uses the call string approach. These analyses are more precise and significantly more costly than ours. Flow-insensitive context-insensitive points-to analyses for Java are described in [31, 43, 24, 33, 23, 7]. Other work that is based on Andersen's analysis is the points-to analysis described in [48], which is context-insensitive and intraprocedurally flow-sensitive. Dimensions of precision for reference analysis of object-oriented languages are discussed in [35].

Class analysis for object-oriented languages computes a set of classes for each program variable; this set approximates the classes of all run-time values for this variable. Typical clients of this information are call graph construction and virtual call resolution. Various practical context-insensitive class analyses are presented in [28, 13, 6, 12, 46, 44]. Different mechanisms for context sensitivity have been studied in the context of class analysis [27, 1, 30, 2, 18]; these methods typically use some combination of the parameter types to abstract context. The work in [27, 1, 2] presents class analyses for Smalltalk and Self. Similarly to our analysis, these analyses use information about the receiver object in order to create and select contextual method versions. Unlike our analysis, they use additional information (e.g., the method invocation site). The idea of object sensitivity is to use only the receiver object as context; we believe that for the purposes of flow-insensitive points-to analysis for Java, using invocation sites or other information may be redundant in most cases. The non-parameterized object-sensitive analysis from Section 3 can be expressed in the general framework for context-sensitive class analysis presented in [18]; however, it is neither identified nor studied in [18].

Conceptually, our MOD analysis is based on similar MOD analyses for C [37, 21, 36]. Razafimahefa [31] presents algorithms for side-effect analysis for Java that are based on context-insensitive information. The more precise of the algorithms is based on context-insensitive points-to analysis for Java derived from Steensgaard's analysis for C [42]. Clausen [9] investigates side-effect analysis for Java in the context of a Java bytecode optimizer. Clausen's side-effect analysis does not use points-to information (i.e., a modification through field f is assumed to write *all* objects whose class contains field f). This may result in less precise side-effect information.

Previous work by Pande et al. [29] defines a def-use analysis for C programs with single-level pointers. The analysis is based on a flow- and context-sensitive pointer analysis, which allows the def-use analysis also to be context-sensitive. The def-use analyses presented in Section 7 are based on a similar idea: they take advantage of the context sensitivity of the points-to analysis. However, our work targets a different language and is based on a different notion of context sensitivity that is appropriate for object-oriented software.

Work on data-flow-based testing for object-oriented programs includes [19, 41, 40]. Harrold and Rothermel [19] describe techniques for data-flow-based testing of classes. Their work focuses on def-use pairs for instance variables within a class and does not address interclass interactions, polymorphism, and aliasing. Souter and Pollock [40] develop contextual def-use associations, a generalization of existing data-flow-based testing techniques (e.g., [19, 41]); their work addresses interclass interactions, polymorphism, and aliasing. The computation of def-use pairs in [40] is based on the points-to escape analysis from [49]. Our work shows that the object-sensitive points-to analysis can be used for the purposes of computing contextual def-uses as a more practical alternative to the context- and flow-sensitive analysis from [49].

10. CONCLUSIONS AND FUTURE WORK

We present a framework for parameterized object-sensitive points-to analysis, as well as side-effect and def-use analyses based on it. The basic idea of our approach is to distin-

guish among the different receiver objects of a method. We show that object-sensitive analysis is capable of achieving significantly better precision than context-insensitive analysis, while at the same time remaining efficient and practical. Thus, object-sensitive analysis is a better candidate for a relatively precise, practical, general-purpose points-to analysis for Java.

In our future work we plan to investigate other instantiations of our framework, especially instantiations that involve more precise object naming schemes with targeted context sensitivity. For example, it would be interesting to consider more precise naming for sub-objects of composite objects (i.e., when an object is associated with a single enclosing object). Using the parameterization framework, we plan to focus on instantiations that significantly improve the precision with acceptable increase in analysis cost. This is especially important for software productivity tools in which imprecision may result in wasted time and effort for a tool user.

This work presents experimental comparison of object-sensitive analyses with a context-sensitive analysis that distinguishes context by one enclosing call site (i.e., an instance of the call string approach). It would be interesting to perform theoretical and empirical comparisons between object sensitivity and other instances of the functional approach to context sensitivity (e.g., [8, 34]).

We plan to further investigate applications of points-to, side-effect, and def-use analyses in the context of software productivity tools (e.g., tools for program understanding and testing). The tradeoff between cost and precision is an important issue in such tools, and we intend to focus our work on this problem.

Acknowledgments. We would like to thank the reviewers for their helpful comments. This research was supported in part by NSF grant CCR-9900988.

11. REFERENCES

[1] O. Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis Symposium*, LNCS 864, pages 78–100, 1994.

[2] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming*, 1995.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *International Workshop on Types in Compilation*, 1998.

[5] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[6] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[7] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDD's. In *Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[8] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[9] L. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[10] M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[11] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[12] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[13] A. Diwan, J. B. Moss, and K. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, 1996.

[14] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, Mar. 2000.

[15] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of Java web services for robustness. In *International Symposium on Software Testing and Analysis*, 2004.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[17] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, 1994.

[18] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, 1997.

[19] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Symposium on the Foundations of Software Engineering*, 1994.

[20] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, Mar. 1994.

[21] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[22] IBM Corporation. *High Performance Compiler for Java*, 1997. `http://www.alphaWorks.ibm.com/formula`.

[23] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on*

Compiler Construction, LNCS 2622, pages 153–169, 2003.

[24] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.

[25] A. Milanova. *Precise and Practical Flow Analysis of Object-Oriented Software*. PhD thesis, Rutgers University, Aug. 2003. Available as Techical Report DCS-TR-539.

[26] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[27] N. Oxhoj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *European Conference on Object-Oriented Programming*, pages 329–349, 1992.

[28] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.

[29] H. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[30] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.

[31] C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, Dec. 1999.

[32] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[33] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, Oct. 2001.

[34] E. Ruf. Effective synchronization removal for Java. In *Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[35] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, LNCS 2622, pages 126–137, 2003. invited paper.

[36] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, Mar. 2001.

[37] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, LNCS 1302, pages 16–34, 1997.

[38] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[39] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[40] A. Souter and L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering*, 29(11):1005–1018, Nov. 2003.

[41] A. Souter, L. Pollock, and D. Hisley. Inter-class def-use analysis with partial class representations. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 47–56, 1999.

[42] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[43] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, U. Passau, Sept. 2000.

[44] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.

[45] F. Tip, C. Laffra, P. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, 1999.

[46] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000.

[47] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, 2000.

[48] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis Symposium*, pages 180–195, 2002.

[49] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.