Light Context-Sensitive Points-to Analysis for Java

Ana Milanova
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Abstract

We design a new context-sensitive points-to analysis for Java that targets key object-oriented features such as inheritance and encapsulation, and improves precision over the Andersen-style context-insensitive analysis. At the same time, it has cubic worst-case complexity, thus giving precision improvement for free.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms

Keywords points-to analysis, context sensitivity

1. Introduction

Points-to analysis determines the set of objects that a reference variable or a reference field may point to. This information has a wide variety of uses in software tools and optimizing compilers.

The Andersen-style points-to analysis for Java [10, 16, 7, 3, 20] is a flow- and context-insensitive analysis with cubic worst-case complexity. This analysis is relatively well-understood and there are several scalable publicly available implementations [7, 21, 9].

On the other hand, *context-sensitive points-to analysis* is not well-understood, while it is needed by many client analyses for software tools [5, 14, 17, 9]. One problem is the kind of context sensitivity needed to address imprecisions inherent in object-oriented codes. Another problem is scaling the context-sensitive analysis. Despite significant advances in BDD-based analysis [8, 21, 9], recent work [9, 17] suggests that context-sensitive points-to analysis that addresses the imprecisions in object-oriented codes while remaining scalable, is still an open problem.

We propose an approach that brings insight into the problem of context-sensitive points-to analysis. Our analysis targets key object-oriented features such as inheritance and encapsulation and improves precision over the Andersen-style context-insensitive analysis. The analysis has cubic worst-case complexity, thus giving free precision improvement.

The new analysis is explained in terms of two sets. Set R is the set of all reference variables r in the program. Set O is the set of all analysis objects; the run-time objects created at an allocation site i are represented by analysis object $o_i \in O$. The analysis is built on top of the Andersen-style context-insensitive points-to analysis. It takes as input the context-insensitive points-to graph, Pt, and constructs the *object graph* Ag which safely approximates the access relationships between objects. The Ag set of an object $o \in O$, $Ag(o) \subseteq O$, approximates the objects that o may access. Informally, o accesses o' if one of the following is true: (i) a field f of o refers to o', or (ii) a method m invoked on receiver o has a local variable that refers to o'.

We can combine Pt and Ag to compute an *object-sensitive* solution. Object-sensitive analysis is a form of context-sensitive analysis which distinguishes context by receiver object [12]. Consider a local variable r in method m and let o be a receiver of m. One can approximate the object-sensitive points-to set for r in context o by intersecting $Pt(r)$ and $Ag(o)$—in order for an object o' to be a valid member of the points-to set of r in context o, o' must appear in the Ag set of o (i.e., it must be accessed by o). Furthermore, a new and improved (context-insensitive) points-to set for r, $PtNew(r)$ can be computed by intersecting $Pt(r)$ with the union of the Ag sets of all receivers of m (i.e., taking the union of the context-sensitive points-to sets of r over all possible contexts o). By construction, the object graph avoids many imprecisions inherent in context-insensitive points-to analysis and the intersection improves the points-to result.

We have implemented the analysis as a client of the Andersen-style context-insensitive points-to analysis in Spark [7]. We evaluated its precision and cost on a set of 9 Java programs. The analysis improves precision with respect to call graph construction, a classic application of points-to information, while at the same time it has practical cost.

2. Andersen-style Points-to Analysis

The Andersen-style points-to analysis processes the following kinds of statements:

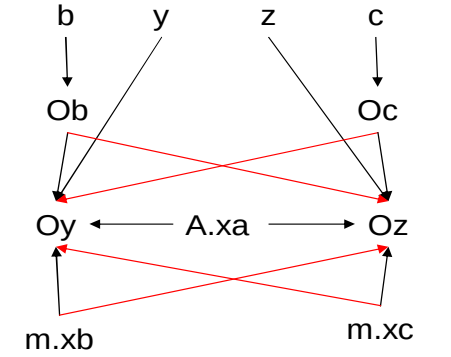
- Object creation: s_i: `l = new C`
- Direct assignment: `l = r`
- Instance field write: `l.f = r`
- Instance field read: `l = r.f`
- Virtual call: `l =` $\mathtt{r}_0$`.m(`$\mathtt{r}_1$`)`

The semantics of the analysis is well-known [16]. When the analysis processes an object creation site s_i, it creates a new points-to edge from l to o_i. When it processes statement "$l = r$" it creates new points-to edges from l to all objects pointed to by r. When it processes a virtual call, the analysis considers each object o in the points-to set of r_0. It performs dispatch based on the class of o and the compile-time target m and computes the run-time target $m_j(p_0, p_1, ret_{m_j})$ (p_0 denotes the implicit parameter `this` of m_j, p_1 denotes the formal parameter of m_j and ret_{m_j} denotes the return variable of m_j). Subsequently the analysis creates new points-to edges from p_0 to o, from p_1 to all objects pointed to by r_1, and from l to all objects pointed to by ret_{m_j}.

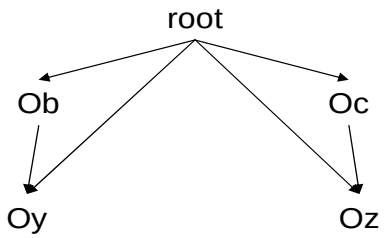
PASTE'07 June 13–14, 2007, San Diego, California, USA.

```
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(); }
}

main {
8    Y y = new Y();   // Oy
9    Z z = new Z();   // Oz
10  B b = new B(y); // Ob
11  C c = new C(z); // Oc
12  b.m();
13  c.m();
}
```



(a) Points-to Graph (Andersen)



(b) Object Graph

Figure 1. Field assignment through a superclass.

3. Motivating Examples

This section presents two cases of common object-oriented idioms and patterns for which the Andersen-style points-to analysis produces imprecise results. It also illustrates the use of the object graph to compute an improved solution.

3.1 Field Assignment Through a Superclass

Figure 1 illustrates a common idiom in object-oriented programming—field assignment through a superclass. Due to the calls to superclass constructor `A.A` at lines 2 and 5, implicit parameter `this` of `A.A` points to both o_b and o_c, and formal parameter `xa` points to o_y and o_z. Statement 1 then introduces two infeasible field edges, $(\langle o_b, f\rangle, o_z)$ and $(\langle o_c, f\rangle, o_y)$. As a result, variable `xb` in `B.m` points to both o_y and o_z and similarly, variable `xc` in `C.m` points to both o_y and o_z, and the virtual calls at lines 4 and 7 cannot be resolved.

Consider the object graph for this example. We defer the precise description of its construction for Section 4 and give only intuition here. The object graph safely approximates run-time object accesses. In Figure 1(b) edge $o_b \rightarrow o_y$ denotes that object o_b may access object o_y; for example, this access may occur when a method invoked on receiver o_b has a local variable that refers to o_y. Similarly, edge $o_c \rightarrow o_z$ denotes that o_c may access o_z. Special node `root` is used to denote the context of invocation of `main`. In this case, `root` has access to all objects. The lack of an edge from o_b to o_z means that o_b cannot access o_z. Similarly, the lack of an edge from o_c to o_y means that o_c cannot access o_y.

The new analysis combines the points-to graph (given in Figure 1(a)) and the object graph (given in Figure 1(b)) to produce a more precise solution. Consider method `B.m`; it is invoked only on receiver o_b. Therefore, if an object is a valid member of the points-to set of local `m.xb`, it must also be a member of the set of objects accessed by o_b in the object graph. The improved solution for `m.xb` is obtained as follows: $PtNew(\texttt{m.xb}) = Pt(\texttt{m.xb}) \cap Ag(o_b) = \{o_y, o_z\} \cap \{o_y\} = \{o_y\}$. Analogously, the improved solution for `m.xc` in method `C.m` is obtained as: $PtNew(\texttt{m.xc}) = Pt(\texttt{m.xc}) \cap Ag(o_c) = \{o_y, o_z\} \cap \{o_z\} = \{o_z\}$. The improved points-to set of `m.xb` is $\{o_y\}$ and the virtual call at line 4 is resolved to `Y.n`. Analogously, the improved points-to set of `m.xc` is $\{o_z\}$, and the virtual call at line 7 is resolved to `Z.n`.

3.2 Template and Factory Methods

Consider Figure 2. This example illustrates another common object-oriented idiom—the use of *template and factory methods* [6]. Class `AbstractList` defines template method `iterator` which defers the creation of the actual iterator to the concrete subclasses of `AbstractList`. Again, it illustrates typical imprecision due to context-insensitive analysis. Since the virtual call to `listIterator` at line 1 resolves to both `List1.listIterator` and `List2.listIterator`, we have that variable `itr` in method `iterator` points to both o_{i1} and o_{i2}. As a result, variable `li1` at line 7 points to both o_{i1} and o_{i2}, and the virtual calls to `hasNext` and `next` at line 7 cannot be resolved. Analogously, the calls at line 8 cannot be resolved as well.

Consider the object graph for this example. As mentioned earlier, this graph safely approximates the accesses between objects. The lack of an edge from o_{L1} to o_{i2} means that o_{L1} cannot access o_{i2}. We have that `l1` in `main` points to o_{L1} only; therefore, the object that is returned at call `li1=l1.iterator()` must appear in the set of objects accessed by o_{L1}. The new points-to set of `li1` is obtained as the intersection of the Pt set of `li1` and the Ag set of o_{L1}; we have $Pt(\texttt{li1}) \cap Ag(o_{L1}) = \{o_{i1}, o_{i2}\} \cap \{o_{i1}\} = \{o_{i1}\}$. As a result, the virtual calls at line 7 can be resolved to `List1.hasNext` and `List1.next`. Analogously, the new points-to set of variable `li2` equals $\{o_{i2}\}$ and the virtual calls at line 8 can be resolved as well.

```
interface Iterator {
  boolean hasNext();
  Object next(); }
class List1Iterator implements Iterator {
  boolean hasNext() {...}
  Object next() {…} }
class List2Iterator implements Iterator {
  boolean hasNext() {...}
  Object next() {…} }

abstract class AbstractList {
  Iterator iterator() {
1    Iterator itr = listIterator();
2    return itr; }
  abstract Iterator listIterator();
 }
class List1 extends AbstractList {
  Iterator listIterator() {
3    return new List1Iterator(); }        //Oi1
}
class List2 extends AbstractList {
  Iterator listIterator() {
4    return new List2Iterator(); }        //Oi2
}
main {
5  List1 l1 = new List1();   //OL1
6  List2 l2 = new List2();   //OL2
7  for (Iterator li1 = l1.iterator(); li1.hasNext();) { li1.next(); }
8  for (Iterator li2 = l2.iterator(); li2.hasNext();) { li2.next(); }
}
```

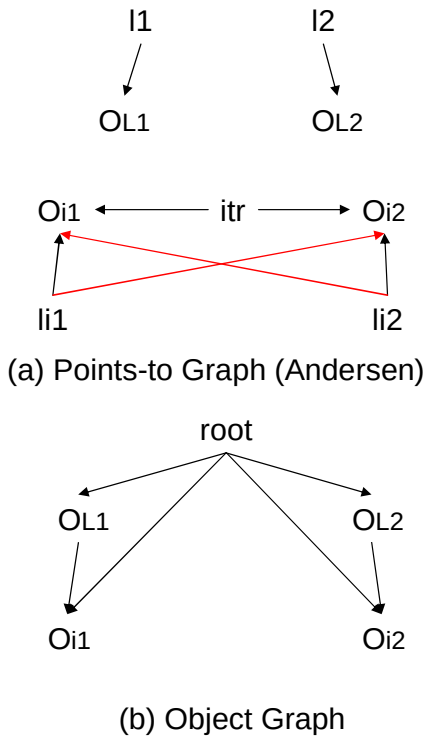


Figure 2. Template method and hook factory methods.

4. Light Context-Sensitive Points-to Analysis

4.1 Construction of the Object Graph

The *run-time object graph*, originally proposed in work on ownership types [4, 15], represents object accesses during an execution of the program:

- There is an edge $o^r \rightarrow o_1^r$ in the run-time object graph if at some point of program execution a reference field f of run-time object o^r refers to run-time object o_1^r.[1]
- There is an edge $o^r \rightarrow o_1^r$ if at some point of program execution an instance method invoked on receiver o^r, has a local variable r that refers to o_1^r.
- There is an edge $o^r \rightarrow o_1^r$ if at some point of program execution, a static method called on a sequence of static calls, from an instance method invoked on receiver o^r, has a local variable r that refers to o_1^r.
- Let special node `root` represent method `main`, the start of program execution. There is an edge `root` $\rightarrow o^r$ if at some point of execution, `main`, or a static method called on a sequence of static calls from `main`, has a local variable r that refers to o^r.

Our analysis constructs Ag, an approximation of this run-time object graph: if there is a run-time access edge $o_1^r \rightarrow o_2^r$ for some execution, then there is an edge in Ag from the representative of o_1^r to the representative of o_2^r. The nodes in Ag are taken from the set of analysis objects O and the edges represent the access relationships. Figure 3 outlines the construction of Ag. It takes as input the points-to graph Pt and the set of methods $PtReach$, reachable according to the points-to analysis. Notation $\mathcal{RC}_m$ stands for the receivers of method m; sets $\mathcal{RC}_m$ are computed as follows. If m is an instance method, $\mathcal{RC}_m$ equals to the points-to set of the implicit parameter `this` of m. If m is a static method, $\mathcal{RC}_m$ includes the points-to sets of all implicit parameters `this` of instance methods n reachable backwards from m on a chain of static calls; if `main` is reachable backwards from m on a chain of static calls, $\mathcal{RC}_m$ includes the node `root`.

Lines 1-2 process statements that account for flow due to object creation. New edges are added to Ag from each receiver of the enclosing method m (i.e., $o \in \mathcal{RC}_m$), to the analysis name o_i that represents the newly created object. Intuitively, at object allocation sites (i.e., constructor calls), the newly created object becomes accessible to the receiver of the caller m. Lines 3-4 process statements that account for flow from other objects to the receiver of m. For example, at an instance call not through `this`, new edges are added from each receiver of m (i.e., $o \in \mathcal{RC}_m$) to each returned object o_j. Intuitively, the returned object becomes accessible to the receiver of m. Lines 5-6 process statements that account for flow from m into other objects. For example, at an instance call $l.n(r)$, edges are added from each object o_i in the points-to set of l, to each object o_j in the points-to set of reference argument r. Intuitively, the object passed as actual argument becomes accessible to the receiver of the call. Finally, lines 7-8 take into account that an object may become accessible to itself through a leak of implicit parameter `this`.

[1] We use notation o^r to denote run-time objects, and notation o to denote analysis objects (i.e., the analysis representatives of the run-time objects).

input $ReachPt$: reachable methods Pt: $R \cup O \rightarrow \mathcal{P}(O)$
output $Ag : O \rightarrow \mathcal{P}(O)$
[1] **foreach** statement s in method $m \in ReachPt$
$s_i : l = new\ C(...)$
[2] add $\{o \rightarrow o_i \mid o \in \mathcal{RC}_m\}$ to Ag
// flow into the receiver of m due to object creation
[3] **foreach** statement s in method $m \in ReachPt$
$s : l = r.n(...)$ s.t. $r \neq$ `this`,
$s : l = r.f$ s.t. $r \neq$ `this`
[4] add $\{o \rightarrow o_j \mid o \in \mathcal{RC}_m \wedge o_j \in Pt(l)\}$ to Ag
// flow from callees into the receiver of m
[5] **foreach** statement s in method $m \in ReachPt$
$s : l = new\ C(r)$,
$s : l.n(r)$ s.t. $l \neq$ `this`,
$s : l.f = r$ s.t. $l \neq$ `this`
[6] add $\{o_i \rightarrow o_j \mid o_i \in Pt(l) \wedge o_j \in Pt(r)\}$ to Ag
// flow into the receiver of the callee from m
[7] **foreach** statement s in method $m \in ReachPt$
$s : l =$ `this`,
$s :$ `this`$.n($`this`$)$
$s :$ `this`$.f =$ `this`
[8] add $\{o_i \rightarrow o_i \mid o_i \in Pt($`this`$)\}$ to Ag
// self-access through a leak of `this`

Figure 3. Construction of Ag. $\mathcal{P}(X)$ denotes the power set of X. Ag is initially empty.

Consider the code in Figure 1. Edges `root` $\rightarrow o_y$, `root` $\rightarrow o_z$, `root` $\rightarrow o_b$, and `root` $\rightarrow o_c$ in the object graph are due to code lines 8-11 (lines 1-2 in the algorithm in Figure 3). Edges $o_b \rightarrow o_y$ and $o_c \rightarrow o_z$ are due to code lines 10 and 11 respectively (and lines 5-6 in the algorithm in Figure 3). Note that the algorithm considers only *external* statements—those that may cause new objects to become accessible. Code lines 1,2,3,5 and 6 are not considered because they account for field accesses and method calls through `this`. Intuitively, no new object becomes accessible to the receiver due to such *internal* statements. For example, consider the instance field write `this.f=xa` at line 1. The object pointed to by `xa` is assigned to field f of the receiver; however, this object must have already been made accessible to the receiver through an external statement—in this case, it is made accessible either through statement 10 or 11.

As another example, consider the code in Figure 2. Edges `root` $\rightarrow o_{L1}$ and `root` $\rightarrow o_{L2}$ are due respectively to code lines 5 and 6. Statement `li1=l1.iterator()` contributes edges `root` $\rightarrow o_{i1}$ and `root` $\rightarrow o_{i2}$ due to the fact that in the points-to graph `li1` points to both o_{i1} and o_{i2}. Statement `li2=l2.iterator()` contributes (redundantly) the same two edges. The rest of the statements in `main` do not contribute new edges. Edge $o_{L1} \rightarrow o_{i1}$ is due to line 3 and edge $o_{L2} \rightarrow o_{i2}$ is due to line 4.

4.2 Intersection of the Points-to Graph and Object Graph

We extend the Ag notation for a method m as follows: $Ag(m) = \bigcup_{o \in \mathcal{RC}_m} Ag(o)$. $Ag(m)$ is the union of the Ag sets of all receivers of m; intuitively, this is the set that may be accessible to all receivers of m and thus it is a superset of all objects accessible in m. We extend Ag for a local variable r as well: $Ag(r) = \bigcup_{o \in Pt(r)} Ag(o)$; it is the union of the Ag sets of all objects in the points-to set of a local r.

Let l be a local variable (different from `this`) in a method m. To simplify our notation we assume that there is a single assignment to l (the analysis can be trivially extended to the case when there are multiple assignments to l). There are two cases. The first case considers when the assignment to l is internal—that is, we have $l = r$, $l =$ `this`$.f$, or $l =$ `this`$.n(...)$. Since the analysis does not track internal flow we cannot do better but intersect the old points-to set of l with the Ag set of m: $PtNew(l) = Pt(l) \cap Ag(m)$. Consider Figure 1 and variable `xb` in method `B.m`. The assignment to `xb` at line 3 is internal and therefore we have $PtNew($`xb`$) = Pt($`xb`$) \cap Ag($`B.m`$) = \{o_y, o_z\} \cap \{o_y\} = \{o_y\}$. The second case considers when the assignment to l is external—that is, we have $l = r.f$ or $l = r.n(...)$ where $r \neq$ `this`. Since this is the only assignment to l, the object that flows to l must be contained in the Ag set of r. Thus we have $PtNew(l) = Pt(l) \cap Ag(m) \cap Ag(r)$. Figure 2 illustrates the benefit of this refinement. Consider variable `li1`. Without the refinement we have $PtNew($`li1`$) = Pt($`li1`$) \cap Ag($`root`$)$. Since $Ag($`root`$)$ contains both o_{i1}, and o_{i2}, this intersection does not improve the points-to set: we have $Pt($`li1`$) \cap Ag($`root`$) = \{o_{i1}, o_{i2}\}$. However, `li1` is assigned only at statement `li1=l1.iterator()`, and therefore the objects in its points-to set must be in the Ag set of `l1` as well. Since $Pt($`l1`$) = \{o_{L1}\}$, we have that $Ag($`l1`$) = Ag(o_{L1}) = \{o_{i1}\}$. Thus, we have $PtNew($`li1`$) = \{o_{i1}, o_{i2}\} \cap \{o_{i1}\} = \{o_{i1}\}$.

4.3 Complexity

Let n be the size of the program—that is, the number of statements, the number of variables, the number of methods, and the number of object names is of order n. The underlying Andersen's points-to analysis has complexity of $O(n^3)$ (under certain assumptions). The light context-sensitive points-to analysis is cubic as well. To see this, consider the construction of the object graph in Figure 3. It is dominated by the processing of statements at lines 3-4 and 5-6. For each such statement the analysis performs at most $O(n^2)$ work. For example, at lines 5-6 there are at most $O(n)$ objects in the points-to set of l and at most $O(n)$ objects in the points-to set of r; therefore, processing a statement at lines 5-6 is of order $O(n^2)$. Since there are $O(n)$ statements in the program, the complexity of the object graph construction is $O(n^3)$. The computation of sets $Ag(m)$ and $Ag(r)$ is $O(n^3)$, and thus the computation of the intersections is $O(n^3)$ as well.

5. Empirical Results

The analysis is implemented in Java using the Soot 2.2.3 [19] and Spark [7] frameworks. Specifically, it is implemented as a client of the Andersen-style points-to analysis provided by Spark. It constructs the object graph as described in Section 4.1 and computes the intersections as described in Section 4.2. For the rest of this section, the Andersen-style analysis by Spark is denoted by *Andersen* and the light context-sensitive analysis is denoted by *LightSens*.

We performed whole-program analysis with the Sun JDK 1.4.1 libraries. All experiments were done on a 900MHz Sun Fire 380R with 4GB of RAM. The implementation which includes Soot and Spark was run with a max heap size of 1GB. Native methods are handled by utilizing the models provided by Soot. Reflection is handled by specifying the dynamically loaded classes which Spark uses to appropriately resolve reflection calls. This approach is used in other whole-program analyses based on Soot and Spark [9, 17].

Our benchmark suite includes `soot-c` and `sablecc-j` from the Ashes suite [1], the `polyglot` Java front-end, and several benchmarks from the DaCapo benchmark suite version beta051009 [2]. These benchmarks (but earlier versions) were used in recent Soot and Spark-based analyses [9, 17]. The suite is described in Table 1.

Analysis times are given in Table 2. *LightSens* includes the time for *Andersen* (it includes the time for Soot and Spark), plus the time to compute the object graph and the intersections. It takes on average twice as much time as *Andersen*. Performance can likely be further improved by a more careful implementation of the client.

We evaluated the precision improvement of *LightSens* over *Andersen* with respect to a classic application of points-to information, call graph construction. We considered all call sites in meth-

(1)Program	(2)Description	(3) #Reachable methods by Spark
soot-c	An analysis framework for Java	6041
sablecc-j	A Java parser generator	7965
polyglot-1.3.2	A framework for Java language extensions	7444
antlr	A parser and lexical analyzer generator	5097
bloat	A Java bytecode optimizer	6397
chart	A graph plotting toolkit and PDF renderer	8755
jython	A Python interpreter	5601
pmd	A Java source code analyzer	8648
ps	A postscript interpreter	5391

Table 1. Information about the Java benchmarks.

(1)Program	*Andersen*	*LightSens*
soot-c	143	+123 (1.9)
sablecc-j	186	+198 (2.1)
polyglot-1.3.2	577	+247 (1.4)
antlr	144	+108 (1.8)
bloat	157	+199 (2.3)
chart	298	+427 (2.4)
jython	136	+147 (2.1)
pmd	253	+256 (2.1)
ps	140	+201 (2.4)
Average		2.1×*Andersen*

Table 2: Analysis cost in seconds.

(1)Program	#Polymorphic calls		#Polymorphic call targets	
	Andersen	*LightSens*	*Andersen*	*LightSens*
soot-c	1403	1335 (5%)	16193	15767 (3%)
sablecc-j	1410	1308 (7%)	21207	15057 (30%)
polyglot-1.3.2	1347	1251 (7%)	10636	10145 (5%)
antlr	1265	1229 (3%)	5510	5355 (3%)
bloat	1533	1429 (7%)	17302	16787 (3%)
chart	1402	1326 (5%)	9595	9188 (4%)
jython	805	758 (6%)	5665	5374 (5%)
pmd	2065	2011 (3%)	11155	10931 (2%)
ps	774	708 (9%)	16554	16209 (2%)
Average		6%		6%

Table 3: Call graph improvement.

ods reachable by *Andersen* that were determined to be polymorphic by *Andersen*. The first column of Table 3, "#Polymorphic calls", shows the number of polymorphic call sites by *Andersen* and by *LightSens*. *LightSens* resolves from 3 to 9% (6% on average) of the original polymorphic call sites. The second column of Table 3, "#Polymorphic call targets", shows the total number of targets at polymorphic call sites by *Andersen* and by *LightSens*. *LightSens* removes from 2 to 30% (6% on average) of the original targets.

We believe that these results are promising. To the best of our knowledge, *LightSens* is the least expensive general-purpose context-sensitive points-to analysis, at least in terms of worst-case complexity. It improves the precision of the call graph, in some cases significantly. Note that with respect to this application, *Andersen* already is very precise; it is difficult to improve the call graph even with much more powerful (and more expensive) context-sensitive analyses [9].

6. Related Work

This work draws upon our work on object-sensitive points-to analysis [12, 11, 13]. The light context-sensitive points-to analysis uses the receiver object as context and is therefore object-sensitive—it can compute for each reference variable r a set $PtNew(r, o)$ which approximates the points-to set of r when the enclosing method of r is invoked on receiver object o. Unlike the standard object-sensitive analysis which keeps an object-sensitive points-to set for each reference variable, the current analysis keeps a set per method receiver (thus merging the object-sensitive information across all variables within methods with the same receiver). In terms of precision, the light context-sensitive analysis is more precise than Andersen and less precise than the standard object-sensitive analysis; it is incomparable to call-site-based (CFA) context-sensitive analyses.

Whaley and Lam [21], and Zhu and Calman [22] describe context-sensitive points-to analyses that use the string of the k enclosing calls as context; k is finite because the analyses collapse sets of strongly connected components. The analyses are exponential and appear not to handle object-oriented features well [9]. In contrast, the light context-sensitive points-to analysis is cubic and it is designed to handle object-oriented features and idioms. Lhotak and Hendren [9] present a study of several inclusion-based context-sensitive points-to analyses, including call-site-based context-sensitive analysis, object-sensitive analysis [12, 11, 13], and the analyses by Whaley and Lam [21] and Zhu and Calman [22]. These analyses are substantially more precise and substantially more expensive than ours. Sridharan and Bodik [17] describe a demand-driven points-to analysis based on CFL-reachability queries. The analysis is demand-driven—that is it computes only information needed by a specific client (e.g., if the client is downcast safety, the points-to analysis compute context-sensitive solutions only for variables related to downcast expressions). In contrast, our analysis is a general-purpose analysis — it computes context-sensitive solutions for all variables, and the solution can be used by arbitrary clients of points-to information. Theoretically the analysis in [17] is incomparable to ours because it is call-site-based, while ours is object-sensitive. However, we expect that in practice it will produce better results with respect to a client, because it tracks information more precisely than our analysis.

Finally, while the analyses in [12, 11, 13, 21, 22, 9] and [17] are algorithmically complex and/or require substantial infrastructure such as BDDs, the light context-sensitive analysis is simple and can be easily built on top of any implementation of Andersen's analysis.

7. Conclusions and Future Work

We present a new context-sensitive points-to analysis for Java that targets key object-oriented features such as inheritance and encapsulation and improves precision over the Andersen-style context-

insensitive analysis. At the same time, it has cubic worst-case complexity, thus giving precision improvement for free.

We believe that our approach and results bring insight into the difficult problem of context-sensitive points-to analysis. In the future we will address the following questions.

First, the object graph may be improved by cutting edges due to innocuous flow. Innocuous flow occurs when an object flows to another object temporarily and the flow does not affect the points-to solution. For example, in `void m(x) { if (x == y) ... }` formal parameter `x` is used only in the address comparison; clearly, the objects that flow through `x` do not reach any points-to set in `m`. Our current implementation does filter this simple case of innocuous flow; however there are many other cases of innocuous flow that will additionally reduce the object graph. Furthermore, the object graph construction can track flow more precisely; for example, it can compute object sets for methods (i.e., $Ag(o, m)$) in the sense of Tip and Palsberg's family of analyses [18].

Second, the scalability of the analysis can be improved. Most clients of points-to analysis do not need information about the standard libraries. When constructing the object graph, one can approximate the flow from standard libraries and avoid the processing of large amounts of library code.

Finally, we will apply the light context-sensitive analysis on clients such as side-effect analysis [12, 13] and def-use analysis [5]. We expect that it will have impact on these clients and on other clients of points-to information.

References

[1] Ashes suite collection. http://www.sable.mcgill.ca/software.

[2] Dacapo benchmark suite. http://www-ali.cs.umass.edu/dacapo/gcbm.html.

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

[4] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 48–64, 1998.

[5] C. Fu, A. Milanova, B. G. Ryder, and D. Wannacott. Robustness testing of Java server applications. *IEEE Transactions on Software Engineering*, 31(4):292–311, Apr. 2005.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[8] O. Lhoták and L. Hendren. JEDD: A BDD-based relational extension of Java. In *Conference on Programming Language Design and Implementation*, pages 158–169, 2004.

[9] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *International Conference on Compiler Construction*, pages 47–64, 2006.

[10] 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, 2001.

[11] A. Milanova. *Precise and Practical Flow Analysis of Object-Oriented Software*. PhD thesis, Rutgers University, Aug. 2003.

[12] 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.

[13] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[14] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[15] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-oriented Programming*, pages 158–185, 1998.

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

[17] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

[18] 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.

[19] 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, pages 18–34, 2000.

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

[21] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[22] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Conference on Programming Language Design and Implementation*, pages 145–157, 2004.