

Extending and Evaluating Flow-insensitive and Context-insensitive Points-to Analyses for Java

Donglin Liang, Maikel Pennings, and Mary Jean Harrold
College of Computing,
Georgia Institute of Technology,
Atlanta, GA 30332, USA
{dliang,pennings,harrold}@cc.gatech.edu

ABSTRACT

This paper presents extensions to Steensgaard’s and Andersen’s algorithms to handle Java features. Without careful consideration, the handling of these features may affect the correctness, precision, and efficiency of these algorithms. The paper also presents the results of empirical studies. These studies compare the precision and efficiency of these two algorithms and evaluate the effectiveness of handling Java features using alternative approaches. The studies also evaluate the impact of the points-to information provided by these two algorithms on client analyses that use the information.

1. INTRODUCTION

Program analyses and optimizations for Java programs require reference information that determines the instances whose addresses can be stored in a reference variable. Researchers have suggested the use of points-to analysis algorithms, originally developed for analyzing C programs, for the computation of the reference information for Java programs. Java has features, however, that are not present in the C language. Without careful adaptation of these algorithms for Java programs, these features may cause the algorithms to compute incorrect information. These features may also affect the ways in which Java programs are written, and thus, may significantly affect the precision and efficiency of the algorithms on these programs.

To evaluate the effectiveness of these points-to analysis algorithms for computing reference information for Java programs, we studied two flow-insensitive, context-insensitive algorithms—Steensgaard’s [18] and Andersen’s [2]. Steensgaard’s algorithm has almost-linear time complexity and computes much more precise information than the worst-case approximation. Andersen’s algorithm, despite its cubic worst-case time complexity, can efficiently compute pointer information for large C programs [15]. In addition, studies

show that Andersen’s algorithm may compute information that is close in precision to that computed by expensive flow-sensitive, context-sensitive algorithms [12, 13].

This paper discusses extensions to Steensgaard’s and Andersen’s algorithms to handle important Java features. Our extensions account for the common ways in which Java programs are written and attempt to improve the precision and efficiency of these two algorithms in analyzing such programs. Compared to other approaches [16, 19] that extend Steensgaard’s or Andersen’s algorithms for Java, our approaches for handling `this`, collections, and maps can be more precise, and our approaches for handling fields can be more efficient.

The paper also presents several empirical studies. The studies evaluate the efficiency and the precision of Steensgaard’s and Andersen’s algorithms, and evaluate the effectiveness of using alternative approaches for handling Java features. The studies also evaluate the impact of using points-to information provided by these algorithms on virtual call resolution and escape analysis. Our studies are the first to show that, by carefully handling features such as `this`, collections, and maps, both algorithms can compute much more precise points-to information than the worst-case approximation. Our studies are also the first to show that, by simplifying field handling in Andersen’s algorithm to take advantage of encapsulation present in Java programs, the efficiency of this algorithm can be significantly improved without losing precision. Our studies also show that the use of points-to information can significantly improve the precision of virtual call resolution, and can provide useful escape information for optimization.

2. EXTENDING POINTS-TO ANALYSES FOR JAVA

This section discusses the extensions to Steensgaard’s and Andersen’s algorithms to handle several important Java features. Because of space limitation, we omit the discussion of handling other Java features, such as reflection, that must also be carefully considered when implementing a points-to analysis.

Steensgaard’s and Andersen’s Algorithms. We adapted Steensgaard’s and Andersen’s algorithms to compute points-to graphs for Java programs. In a *points-to graph*, nodes represent variables or instances, and edges represent vari-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASTE’01, June 18-19, 2001, Snowbird, Utah, USA.
Copyright 2001 ACM 1-58113-413-4/01/0006 ...\$5.00.

able references (labeled with “*”) or instance field references (labeled with field names). For efficiency, Steensgaard’s algorithm uses one node to represent the instances that may be referenced by the same variable or the same field. For example, in Figure 1(b.1), instances created at both statements 10 (h10) and 11 (h11) are represented using one node because they are referenced by `this1` (`this` of `A`’s constructor). In contrast, Andersen’s algorithm uses one node to represent one instance (see Figure 1(c.1)). In both algorithms, an instance can be represented by at most one node.

Both Steensgaard’s algorithm and Andersen’s algorithm process statements in each method in an arbitrary order. When the algorithms process a method call, they use a set of assignments to simulate the parameter passing and the return of the target method. The algorithms process only the *reference assignments* — assignments to fields or variables of reference types. After an algorithm terminates, the information contained in the points-to graph must be *safe* at each reference assignment: the set of instances referenced by the left side of the assignment subsumes the set of instances referenced by the right side of the assignment.

Steensgaard’s algorithm processes a reference assignment by merging the node that represents the instances referenced by the left side of the assignment with the node that represents the instances referenced by the right side of the assignment. In this way, after the assignment is processed, the information contained in the points-to graph remains safe at this assignment. Therefore, Steensgaard’s algorithm processes each reference assignment only once. In contrast, Andersen’s algorithm processes a reference assignment by adding edges to the points-to graph so that the set of instances referenced by the left side of the assignment subsumes the set of instances referenced by the right side of the assignment. However, because the set of instances referenced by the right side may change after the assignment is processed, Andersen’s algorithm may have to revisit the assignment and add more edges to ensure that the points-to information remains safe at this assignment. Such processing continues until no new edge is added to the points-to graph. This approach causes Andersen’s algorithm to be more expensive but more precise than Steensgaard’s algorithm.

Virtual Method Calls. We present two approaches for handling virtual method calls in a points-to analysis. The first approach computes the set of invocable methods at each virtual method-call using algorithms such as class-hierarchy analysis (CHA) [8] or rapid type analysis (RTA) [3]. This approach is simple and requires minor modification to the points-to analysis. The second approach starts the analysis from `main()` and discovers, during the analysis (on the fly), the targets of each virtual method call that can be reached from `main()` or from class initialization methods.¹ This approach may be more expensive but may compute more precise information than the first approach.

To use the second approach, a points-to analysis computes, for each reference variable v , a set of virtual method calls that invoke methods through v (for ease of explanation, we assume that the program is formatted in such a way that virtual method calls are invoked through variables). During

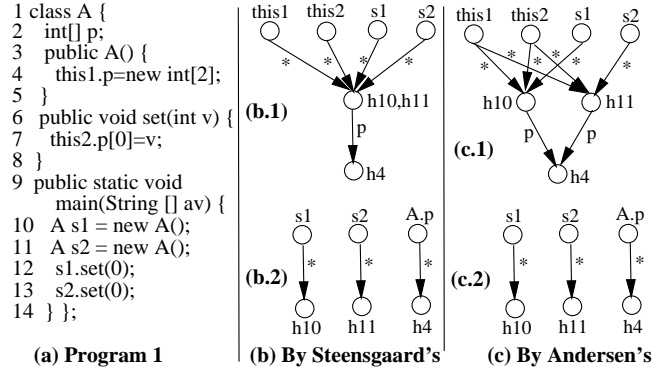


Figure 1: Java example and its points-to graphs.

the analysis, when a new instance o of type T is added to v ’s *points-to set* (the set of instances that may be referenced by v), if o is the first instance of type T in the set, then the algorithm reprocesses each virtual call that invokes methods through v . If the algorithm discovers a new target at a call, it binds the actual parameters of the call to the formal parameters of the new target. If the new target has not been processed previously, the algorithm processes each statement in the new target.

Fields. We discuss two approaches for handling fields in points-to analysis. The first approach treats the fields of instances using an approach similar to those used for fields of structures in C, and computes a points-to set for each instance field [16, 19]. To do this, within a method $m()$ of class `A`, the analysis treats `this` as a formal parameter. If a field f of `A` is accessed in $m()$, the analysis treats such access as an access to `this.f`. At a method call $r.m()$ (method call $m()$ is treated as `this.m()`), the analysis binds r to `this` to pass the receiver instance into $m()$. Figures 1(b.1) and 1(c.1) depict the points-to graphs constructed for Program 1 by Steensgaard’s and Andersen’s algorithms, respectively, using this approach. The graphs show that Steensgaard’s algorithm may compute very imprecise information because instances pointed to by `this` of a method m are forced to be represented by one node.

We propose a second approach that computes one points-to set for each class field. At each statement in which a field of an instance is accessed using expression $r.f$ (f is declared in class `A`), the algorithm treats such an instance field access as class field access `A.f` and drops r .² Such simplification can improve the efficiency of both algorithms. The simplification also lets the algorithms avoid computing the points-to set for `this` in the analysis if such a points-to set is not needed for the computation of the points-to sets of other variables.³ The points-to set of `this` is needed in a method m for the computation of the points-to sets of other variables in the following cases: (1) `this` is used in m as the right side of an assignment or an actual parameter; or (2) m calls another method m_1 through a *direct* method call—a call in the format of $m_1()$ or `this.m_1()`—and the points-to set of `this` is needed in m_1 or the points-to analysis will

²Reference [20] uses a similar approach to handle field accesses.

³The points-to set for `this` in method m can be computed, after the analysis, by collecting the receiver instances at each call to m .

¹Similar approaches have been used to resolve indirect calls through function pointers in points-to analysis of C programs (e.g., [10, 11]).

discover the targets of the direct method call on the fly. In other cases, the points-to set of `this` in `m` is not needed. Given a call `r.m()`, if the points-to set of `this` in `m()` is not needed, then the points-to analysis ignores `r` and considers only the bindings between actuals and formals. Otherwise, the analysis treats `this` as a formal parameter and also binds `r` to `this`.

In many cases, by avoiding the computation of points-to sets for `this`, Steensgaard’s algorithm using the second approach may be more efficient and more precise than using the first approach. For example, the graph in Figure 1(b.2) computed by Steensgaard’s algorithm using the second approach is smaller but contains more precise information than the graph in Figure 1(b.1). Andersen’s algorithm using the second approach may be more efficient but less precise than using the first approach. The loss of precision is caused by treating the same field of different instances in the same way. However, if the reference fields of each instance are always modified within the methods of the instance, the information computed by Andersen’s algorithm using the second approach is equivalent to that computed using the first approach. For example, the points-to graph in Figure 1(c.2) computed by Andersen’s algorithm using the second approach contains information that is equivalent to the information contained in the graph in Figure 1(c.1). In Java programs where encapsulation is strongly encouraged, we expect to see little difference in the precision of information computed by Andersen’s algorithm using either approach.

Collections and Maps. Besides arrays, a Java program frequently uses a collection (e.g., `Vector`) or a map (e.g., `Hashtable`) to store and retrieve data. Because methods for a collection or a map may be provided in native code, it is difficult to analyze these methods. Even if the methods are provided in byte code, analyzing these methods not only adds extra cost to the analysis, but also causes a context-insensitive points-to analysis to be less precise. For example, data stored in one `Vector` instance may be returned by invoking `elementAt()` on another `Vector` instance because these data are referenced by the return statement of `elementAt()`, which is shared by all instances of `Vector`.

We solve this problem with user-provided models. A model instructs the algorithm to make a conservative assumption when processing calls to methods of a collection instance or a map instance. A model describes different slots where data can be stored in an instance. For example, in a hash table instance, keys are stored in a key slot and values are stored in a value slot. The model also describes a set of operations that store data into or retrieve data from slots. Given such models, when a new instance of a collection or a map is instantiated, the slots associated with the instance are also created. When the algorithm processes a method call on an instance, the algorithm first determines, using user-provided information, the set of operations that can simulate the actions of the method. The algorithm then processes these operations on the slots associated with this instance. Because the slots for different instances are separated, the algorithm can distinguish the data stored in different instances.

For example, Figure 2(a) shows the description of a model for the collection. The description is very similar to the definition of a class. Besides a slot and five operations, the model also consists of two instances, “`iterator`” and “`array`”, and a sequence of initialization assignments described in “`INIT()`”. “`iterator`” is used to represent the instance re-

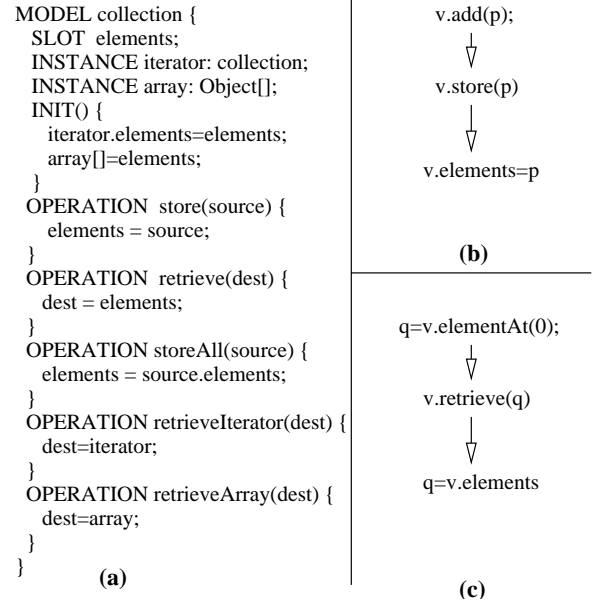


Figure 2: The model for collection.

turned by a method call to `iterator()` (the algorithm treats “`iterator`” as a collection). “`array`” is used to represent the array instance returned by a method call to `toArray()`. The initialization assignments are used to copy data from slot “`elements`” into “`iterator.elements`” and “`array`”. When the algorithm processes an instantiation statement that creates a collection instance, the algorithm also creates an instance for “`iterator`” and an instance for “`array`”. The algorithm then processes the initialization assignments.

Figure 2(b) shows the steps taken by the algorithm to process statement `v.add(p)` that stores an instance referenced by `p` into the `Vector` referenced by `v`. The algorithm uses “`store`” operation to simulate the action of `add()` method, and replaces “`store`” operation with its body. Thus, `v.add(p)` will be treated as assignment `v.elements=p`. Figure 2(c) shows the steps taken by the algorithm to process statement `q=v.elementAt(0)` that retrieves an instance from `Vector` referenced by `v`. The algorithm uses “`retrieve`” operation to simulate the action of `elementAt()`, and replaces “`retrieve`” operation with its body. Thus, the algorithm treats `q=v.elementAt(0)` as assignment `q=v.elements`. These two examples show that instances stored into different `Vector` instances can be distinguished.

Casting. Andersen’s algorithm for Java can benefit from the fact that casting is checked at runtime. Given a reference assignment `p=(A)q`, only instances of type `A` or a subtype of `A` can be returned by the casting. Thus, Andersen’s algorithm propagates only the instances whose type is `A` or a subtype of `A` from `q`’s points-to set to `p`’s points-to set. This approach improves both the precision and the efficiency of the algorithm.

Exceptions. We propose an approach that uses assignments to simulate the passing of exception instances from the `throw` statements to the corresponding `catch` statements. Our approach utilizes the control-flow information provided by Sinha and Harrold’s algorithm [17]. This algorithm provides information about the possible types of the raised ex-

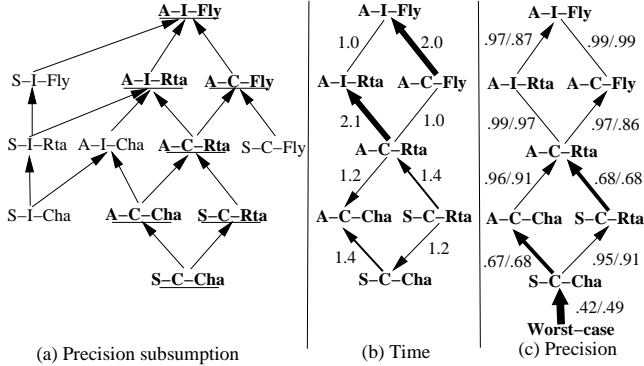


Figure 3: Intuitive comparison of algorithms.

ception at a `throw` and information about where the control flows after an exception of a specific type is raised. The information can be used to create assignments that assign the exception reference at a `throw` to the exception reference at a `catch`. A type filter is also associated with each assignment to specify the types of the exceptions that can be passed from the `throw` to the corresponding `catch`.

3. EMPIRICAL STUDIES

We implemented Steensgaard’s and Andersen’s algorithms using Java Architecture for Bytecode Analysis (JABA)⁴ that analyzes the control flow and exceptions, simulates the changes in the operand stack, and builds an abstract-syntax tree representation for a Java program from the byte code. Our implementation of the points-to analysis simulates the effects of method calls on instances of classes in `java.lang`, `java.util`, and `java.io`, and thus, avoids analyzing the byte code of these classes. Our implementation also carefully handles reflection using user-provided information.

Our implementation of points-to analysis for Java can be instantiated into 12 algorithms depending on the approaches for handling fields and virtual method calls. Figure 3(a) shows the 12 algorithms (nodes) and the precision subsumption relations (edges). The first letter in an algorithm’s name indicates whether the algorithm is Steensgaard’s (S) or Andersen’s (A). The second letter in the name indicates whether the algorithm computes information for class fields (C) or instance fields (I). The last three letters in the name indicate how the algorithm handles virtual method calls: using CHA, using RTA, or discovering targets on the fly (Fly). The target of each edge is at least as precise as the source of the edge. Note that algorithms S-I-* are not comparable with algorithms S-C-*, and S-C-Fly is not comparable with the other S-C-* because of the difference in treating `this`.

We conducted several empirical studies on a set of Java programs to evaluate the performance of these algorithms. In some of the studies, we also collected the execution traces by running each subject program on a typical input. The traces are gathered using a profiler that is implemented using Java Virtual Machine Profiler Interface (JVMPi).⁵ Table 2(I) shows the sizes of the subjects (classes in libraries excluded). All data are collected on a Sun Ultra-30 with 640Mb physical memory. Because of the space limitation,

⁴<http://www.cc.gatech.edu/aristotle/>, Georgia Tech.

⁵<http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/>, Sun Microsystems, Inc.

program	(I)Steens		(II)Andersen				
	C-Cha	C-Rta	C-Cha	C-Rta	C-Fly	I-Rta	I-Fly
JavaSim	0.13	0.13	32.0	0.22	0.27	0.31	0.27
antlr	31.5	30.4	32.0	24.2	6.72	51.4	9.08
jar	0.25	0.20	0.36	0.27	0.30	0.36	0.31
javacup	2.94	2.13	9.38	8.87	9.48	10.4	11.3
javac	96.1	96.3	136.	123.	158.	1489	1480
jbf	1.50	1.63	1.94	2.84	2.11	2.58	2.52
jess	35.1	26.0	53.1	39.5	48.9	294.	344.
jfe	20.0	7.58	18.1	5.98	6.19	10.2	7.70
jlex	0.62	0.69	1.25	1.41	1.48	1.77	1.79
jtarg	1.18	0.83	1.70	1.58	1.45	2.38	1.65
kawa	62.4	20.2	119.	23.1	25.8	95.8	102.
raja	0.81	0.76	0.15	0.17	0.14	0.18	0.19
sablecc	19.4	21.5	48.7	52.4	81.7	218.	305.
toba	1.59	1.79	2.58	3.58	2.09	3.30	2.32

Table 1: (I) Time in seconds for Steensgaard’s, (II) Time in seconds for Andersen’s.

this paper shows only the results for the seven algorithms underlined in Figure 3(a).

Efficiency. In this study, we compared the time required to run each of the seven algorithms. Table 1 shows the results. The time in the table excludes the time required to build the abstract syntax tree from the bytecode using JABA. To see the trends, we summarize the results in Figure 3(b). In the figure, the number associated with each edge from A_1 to A_2 shows the geometric mean of the ratio of the time required by A_2 to the time required by A_1 . Note that when we compute the numbers for the edge from A-I-Fly to A-I-Rta and the edge from A-C-Fly to A-C-Rta, we exclude `antlr` because it is an outlier: by discovering virtual call targets on the fly, A-I-Fly and A-C-Fly analyze many fewer methods, and thus, run 3-5 times faster than A-I-Rta and A-C-Rta on `antlr` whereas the difference in running time on other subject programs are much smaller. The results show that the biggest efficiency gap appears between algorithms that compute points-to sets for class fields and algorithms that compute points-to sets for instance fields (about 10 times for `javac`). This is not surprising because the latter must compute many more points-to sets than the former. The results further show that, except for A-I-Rta and A-I-Fly, the algorithms can efficiently compute points-to information for large programs, and thus, can be used in practice.

Precision. In this study, we compared the precision of the algorithms. We also compared them to the worst-case approximation: a reference variable of type T may refer to any instance that is created in the program and whose type is T or a subtype of T . For each algorithm A , we measured $I_C[A]$, the average number of receiver instances for each method call, and $C_I[A]$, the average number of method calls that are invoked on each instance computed using information provided by A . These measurements reflect the impact of the points-to information on subsequent data-flow analyses: the larger the number, the less precise a subsequent data-flow analysis will be. To compare the results for different algorithms, when we compute $I_C[A]$, we consider only the virtual method calls in the code analyzed by all seven algorithms, and when we compute $C_I[A]$, we consider only the instances created by statements in the code analyzed by all seven algorithms. Table 2(II,III,IV) shows the results. In the table, each pair of numbers for algorithm A shows $I_C[A]$ and $C_I[A]$, respectively.

Figure 3(c) summarizes the results. In the pair of num-

program	(I) Subject Size†			(II) Worst case	(III) Steens		(IV) Andersen				
	Nodes	Cls	Methods		C-Cha	C-Rta	C-Cha	C-Rta	C-Fly	I-Rta	I-Fly
JavaSim	3305	37	242	6.24/12.6	2.11/7.16	2.11/7.16	1.91/6.5	1.91/6.5	1.84/6.05	1.91/6.5	1.84/6.05
antlr	44065	148	1858	11.8/98.4	8.53/82.9	8.53/81.0	1.93/27.9	1.93/27.9	1.76/10.4	1.84/21.8	1.70/10.2
jar	2264	8	89	6.45/16.3	1.87/5.25	1.53/4.19	1.44/4.08	1.40/3.86	1.40/3.83	1.40/3.83	1.40/3.83
javacup	12778	35	372	5.85/50.3	3.18/25.4	3.18/25.4	2.54/20.5	2.54/20.5	2.53/20.5	2.50/20.3	2.50/20.3
javac	31346	151	1404	55.9/344.	53.8/335.	53.8/335.	47.0/291.	47.0/291.	47.0/289.	47.1/291.	47.0/289.
jbf	8730	45	548	13.0/69.8	5.53/32.9	5.53/32.5	4.71/27.7	4.71/27.7	4.71/27.7	4.71/27.7	4.71/27.7
jess	23412	207	1132	57.3/258.	48.6/227.	44.4/186.	40.3/190.	36.3/152.	32.8/136.	36.3/152.	31.9/133.
jfe	31019	310	1837	28.7/148.	4.82/37.4	3.90/29.9	2.00/19.4	1.94/16.3	1.88/8.73	1.94/15.3	1.87/8.16
jlex	6629	20	134	4.31/46.6	1.39/13.8	1.39/13.8	1.19/11.8	1.19/11.8	1.19/11.7	1.19/11.8	1.19/11.7
jtarg	6489	40	202	4.32/12.1	1.38/6.36	1.38/6.36	1.07/5.08	1.07/5.08	1.07/4.56	1.07/5.08	1.07/4.56
kawa	33388	319	1989	33.2/153.	20.3/125.	16.1/73.0	13.0/83.2	8.61/39.3	8.07/35.3	8.58/38.6	8.06/35.3
raja	6351	65	391	9.25/42.8	2.62/10.6	2.62/10.6	1.33/3.60	1.33/3.60	1.33/3.60	1.33/3.60	1.33/3.60
sablecc	28232	295	2025	23.7/143	12.5/76.3	12.5/76.3	7.29/44.2	7.29/44.2	7.16/43.4	7.28/44.1	7.15/43.3
toba	10376	26	196	6.31/29.9	1.41/13.5	1.41/13.5	1.37/13.3	1.37/13.3	1.37/13.3	1.37/13.3	1.37/13.3

†The statistics may differ from that reported in other works for a subject because (1) other interfaces, as well as the classes implementing collections and maps (e.g. in `sablecc`), are excluded, (2) different versions are used.

Table 2: (I) Subject size, (II) Worst case precision, (III) Precision of Steensgaard’s, and (IV) Precision of Andersen’s.

bers associated with an edge from A_1 to A_2 in the figure, the first is the geometric mean of the ratio $I_C[A_2]/I_C[A_1]$, and the second is the geometric mean of the ratio $C_I[A_2]/C_I[A_1]$. The study shows that Steensgaard’s algorithm computes significantly more precise information than the worst-case approximation. This result agrees with the comparison of the worst-case approximation to Steensgaard’s algorithm on C [12].⁶ However, this result is quite different from that reported in Reference [19], in which the precision of Steensgaard’s is close to the worst-case approximation. Although the precision is measured differently, another major factor that may contribute to the difference is that our implementation of Steensgaard’s computes more precise information: our implementation avoids computing points-to sets for `this` and handles collections and maps more precisely. The study also shows that Andersen’s algorithm may compute significantly more precise information than Steensgaard’s. However, as expected, there is no significant difference between algorithms that compute information for class fields and algorithms that compute information for instance fields. This result suggests that the high cost of computing information for instance fields in Andersen’s algorithm is not worthwhile.

Virtual Call Resolution. In this study, we compared the effectiveness of resolving virtual method calls using the points-to information provided by the seven algorithms we considered. Given a virtual method call `r.m()`, we can obtain the set of invocable methods in two steps. We first collect the types of the instances in the points-to set of `r`. We then search for methods with signature `m()` in the types that we collect in the first step. These methods are the invocable methods for the method call. A virtual method call is *resolved* if there is only one invocable method.

Let C be the set of virtual method calls that we consider, and R_x be the set of virtual calls that are resolved using approach x (x can be CHA, RTA, or one of the seven algorithms). Given an algorithm A , we measured the percentage of virtual calls that can be solved by A but not by CHA ($|R_A - R_{CHA}| * 100 / |C|$). We also compared the results with that computed using RTA. To compare different algorithms, we considered only the method calls that appear in the code that is analyzed by all seven algorithms. The left-side of Table 3 shows the results. In each pair of numbers, the first number is computed by counting the number of

⁶Results are more dramatic on C because types are unsafe in C.

virtual method call statements in the code, and the second number is computed by counting the number of virtual call invocations in a trace generated by the profiler. The table shows that, for several programs, resolving virtual method calls using points-to information can be significantly more precise than resolving virtual method calls using CHA or RTA. This is consistent with the findings in Reference [16]. However, except for `antlr` and `jar`, the absolute difference for the results computed with various points-to analysis algorithms is insignificant. This suggests that Steensgaard’s algorithm is good enough for virtual call resolution. Note that for `toba`, because a call statement that reads bytes from a stream has been executed many times, resolving such a call greatly increases the percentage of resolved virtual method invocations.

Escape Analysis. In this study, we compared the effectiveness of computing escape information using the points-to information provided by the seven algorithms.⁷ We measured the percentage of instances that are local to a method (an instance I is *local* to a method m if I is instantiated in m and cannot be returned to m ’s callers). To compare different algorithms, we considered only the instances that are explicitly created using `new` statements in the code that is analyzed by all seven algorithms.

The right side of Table 3 shows the results of this study. In each pair of numbers, the first is computed by examining the `new` statements that create local instances and the second one is computed by examining the instances created at those statements at runtime. The table shows that points-to information computed by these algorithms can effectively support escape analysis. The table also shows that, for a few programs (e.g., `raja`), Andersen’s algorithm computes significantly better results than Steensgaard’s. However, among different versions of Andersen’s algorithm, the differences in the results are insignificant. This suggests that any version of Andersen’s algorithm can be used for escape analysis.

Summary. Our studies suggest that A-C-Rta or A-C-Fly perform the best among the evaluated algorithms: they are more efficient than, but as precise as, A-I-Rta and A-I-Fly, and they compute more precise information than Steensgaard’s algorithm. Because A-C-Rta and A-C-Fly handle

⁷Reference [16] discusses the details about computing escape information using points-to information.

program	Resolved by Cha	% resolved by points-to analysis but not by CHA†						% of method local instances‡				
		Rta	S-C-Cha	S-C-Rta	A-C-Cha	A-I-Rta	A-I-Fly	S-C-Cha	S-C-Rta	A-C-Cha	A-I-Rta	A-I-Fly
JavaSim	100/100	0/0	0/0	0/0	0/0	0/0	0/0	51/0.	51/0.	51/0.	51/0.	51/0.
antlr	60.7/67.1	13.9/30.5	14.3/30.5	14.3/30.5	19.7/31.2	19.9/31.5	20.0/31.9	47/2.	47/2.	49/2.	50/4.	59/39
jar	85.7/58.0	2.85/0	6.79/0	13.5/41.9	9.70/41.9	13.5/41.9	13.5/41.9	48/56	53/63	76/85	76/85	76/85
javacup	94.1/96.3	0.42/0.00	0.43/0.00	0.43/0.00	0.53/0.11	0.53/0.11	0.53/0.11	24/33	24/33	24/35	24/35	24/35
javac‡	78.6/-	0.38/-	0.43/-	0.43/-	2.06/-	2.09/-	2.09/-	33/-	33/-	40/-	40/-	40/-
jbf	92.7/43.0	0/0	3.61/24.1	3.61/24.1	4.16/32.3	4.16/32.3	4.16/32.3	61/26	61/26	69/27	69/27	69/27
jess	69.0/95.7	3.78/1.47	3.83/1.47	3.96/1.54	5.13/1.57	5.36/1.64	6.70/2.24	34/10	34/10	35/10	35/10	35/10
jfe	91.0/95.6	7.06/4.25	7.21/4.25	7.21/4.25	7.21/4.25	7.21/4.25	7.25/4.25	61/55	61/55	67/55	67/55	68/56
jlex	99.2/99.9	0.38/0.02	0.57/0.03	0.57/0.03	0.57/0.03	0.57/0.03	0.57/0.03	30/56	30/56	32/57	32/57	32/57
jtar	96.9/99.9	1.02/0	1.02/0	1.02/0	1.02/0	1.02/0	1.02/0	60/23	60/23	70/27	70/27	70/27
kawa	83.7/95.6	2.43/0	2.57/0	2.75/0	3.10/1.47	3.33/1.47	3.80/1.47	37/2.	37/2.	41/2.	41/2.	42/2.
raja	70.3/61.4	14.8/14.0	29.6/38.5	29.6/38.5	29.6/38.5	29.6/38.5	29.6/38.5	11/15	11/15	42/65	42/65	42/65
sablecc	77.2/89.0	5.32/5.16	7.94/9.05	7.94/9.05	7.97/9.65	7.97/9.65	11.2/9.65	27/13	27/13	36/13	36/13	36/13
toba	97.4/73.5	0.89/26.3	1.07/26.4	1.07/26.4	1.07/26.4	1.07/26.4	1.07/26.4	78/32	78/32	79/36	79/36	79/36

†Data for A-C-Rta and A-C-Fly are not shown because they are identical to the data for A-I-Rta and A-I-Fly.

‡Dynamic data are unavailable for javac because our profiler ran out of memory.

Table 3: Left: Percentage of resolved virtual method calls. Right: Percentage of method-local instances.

mostly one-level references (only references to arrays, collections, and maps are multi-level), these algorithms may be implemented, without losing much precision, using an efficient approach similar to the one presented in Reference [7]. Our future work will evaluate such an optimization.

The lack of difference between the information computed by A-I-* and that computed by A-C-* can be interpreted in two ways: (1) encapsulation present in Java programs helps to simply and improve Andersen’s algorithm; or (2) encapsulation lowers the precision of Andersen’s algorithm because the accesses to the same field of different instances cannot be distinguished. Our future work will investigate the impact of encapsulation on other points-to analyses.

4. RELATED WORK

Rountev et al. [16] extend Andersen’s algorithm to Java using annotated inclusion constraints that let the algorithm compute information for instance fields and discover the targets of virtual calls on the fly. The algorithm uses stubs for native methods and analyzes methods both in the application and in library. The efficiency of the algorithm and the impact on call graph construction, virtual call resolution, and escape analysis have also been evaluated. Streckenbach and Snelting [19] extend both Steensgaard’s and Andersen’s algorithms to Java using a framework. The framework computes information for instance fields and uses condition constraints to discover the targets of virtual calls on the fly. They also propose a conservative approach to approximate the effect of unanalyzed code. The approach puts all the instances that are passed into unanalyzed code into one set. When a statement in analyzed code calls a method in unanalyzed code, the approach assumes that all instances that are in the set and have appropriate type may be returned. When a method in analyzed code can be called from unanalyzed code, the approach assumes that all instances that are in the set and that have appropriate types may be passed through formals. They also report empirical studies that evaluate the efficiency, precision, and impact on virtual call resolution and KABA, another client analysis.

Our work differs from these existing works in several aspects. First, we consider other alternatives to handle fields and virtual calls. Our empirical evaluation of these alternatives reveals that simplifying field handling in Andersen’s algorithm significantly improves its efficiency without losing precision. It also reveals that finding targets using RTA is almost as good as finding the target on the fly. Second,

we propose a more precise way to handle collections and maps. Third, we propose an approach that avoids, if possible, computing points-to set for `this`. This approach may help Steensgaard’s algorithm to compute more precise information than Streckenbach and Snelting’s approach.

Various flow-insensitive type inference algorithms (e.g., [1, 3, 9, 14, 20, 21]) have been developed to determine the types of the objects that may be pointed to by a pointer in an object-oriented program. Such type information can be used for resolving virtual method calls. Because the type information can be inferred from the points-to information, our work offers effective approaches that compute safe type information for Java programs. In addition, because pointer analysis distinguishes instances of the same class, using information provided by pointer analysis may compute more precise type information than using a type inference algorithm.

Recently, several escape analysis algorithms for Java [4, 5, 6, 22] have been developed. Our work offers another alternative for computing escape information. Our empirical studies show that this alternative may be efficient and effective.

5. CONCLUSION

This paper discussed various approaches for extending two flow-insensitive, context-insensitive pointer-analysis algorithms — Steensgaard’s and Andersen’s — to handle important features of Java programs. The paper also presented a set of empirical studies that evaluate the effectiveness of these approaches. Our studies show that, by careful handling various Java features, Steensgaard’s algorithm and Andersen’s algorithm can efficiently compute much more precise information than the worst-case assumption. The studies also show that object-oriented programming style may significantly affect the performance of points-to analysis algorithms.

Our future work will include more extensive experimentation with these algorithms and their extensions. Given the results of the empirical studies presented in this paper and our previous experience with developing efficient points-to analysis algorithms, we believe that both the precision and the efficiency of these algorithms on Java programs can be further improved. Our future work will also involve the development of new algorithms for computing points-to information for Java programs.

Acknowledgments

This work was supported by grants to Georgia Tech from Boeing Aerospace Corporation, by NSF awards CCR-9988294, CCR-0096321, and EIA-0196145, and by the State of Georgia under the Yamacraw Mission. We thank Michael Hind from IBM and other anonymous reviewers who made many helpful suggestions that improved the presentation of the paper.

6. REFERENCES

- [1] O. Agesen. The Cartesian product algorithm. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 2–26, Aug. 1995.
- [2] L. Andersen. Program analysis and specialization for the C programming language. Technical Report 94-19, University of Copenhagen, 1994.
- [3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Notices*, 31(10):324–341, Oct. 1996.
- [4] B. Blanchet. Escape analysis for object oriented languages. application to Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 20–34, Oct. 1999.
- [5] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 35–46, Oct. 1999.
- [6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 1–19, Oct. 1999.
- [7] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of 2000 Conference on Programming Language Design and Implementation*, June 2000.
- [8] 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.
- [9] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *25th ACM Symposium on Principles of Programming Languages*, pages 222–236, Jan. 1998.
- [10] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of 1994 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
- [11] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.
- [12] M. Hind and A. Pioli. Which pointer analysis should i use? In *ISSSTA'00*, pages 113–123, Aug. 2000.
- [13] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *7th ESEC/FSE*, pages 199–215, Sept. 1999.
- [14] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Nov. 1991.
- [15] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of 2000 Conference on Programming Language Design and Implementation*, pages 47–56, June 2000.
- [16] 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 (to appear)*, Oct. 2001.
- [17] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Trans. on Soft. Eng.*, 26(9):849–871, Sept. 2000.
- [18] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.
- [19] M. Streckenbach and G. Snelting. Points-to for java: A general framework and an empirical comparison. Technical report, University Passau, Nov. 2000.
- [20] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Valle-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, Oct. 2000.
- [21] 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, Oct. 2000.
- [22] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 187–206, Oct. 1999.