Points-to Analysis for Java Based on Annotated Constraints

Atanas Rountev Ana Milanova Barbara G. Ryder
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
{rountev,milanova,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. In this paper we present a points-to analysis for Java based on Andersen's points-to analysis for C [5].

Andersen's analysis can be implemented efficiently by using systems of set-inclusion constraints and by employing several techniques for constraint representation and resolution. We extend these techniques to efficiently represent and solve systems of *annotated* inclusion constraints. The annotations play two roles in our analysis. *Method annotations* are used to model precisely and efficiently the semantics of virtual calls. *Field annotations* allow us to distinguish between different fields of an object. In addition, our analysis keeps track of all reachable methods and avoids analyzing irrelevant library code.

We evaluate the performance of the analysis on a large set of realistic Java programs. Our experiments show that the analysis runs in practical time and space, and has significant impact on call graph construction, virtual call resolution, synchronization removal, stack-based object allocation, and object read-write information. The results show that our analysis is a realistic candidate for a relatively precise, practical, general-purpose points-to analysis for Java.

1 Introduction

Although the Java programming language has gained widespread popularity, the performance of Java software is typically worse than the performance of software written in languages such as C and C++. Improving performance through the use of compiler technology is crucial for making Java a viable choice for production-strength software. In addition, the development of large Java software systems requires strong support from software engineering tools for program understanding, restructuring, and testing.

Optimizing compilers and software engineering tools employ various static analyses to determine properties of run-time program behavior. One fundamental static analysis is *points-to analysis*. For Java, points-to analysis determines the set of objects whose addresses may be stored in a given reference variable or a reference object field. By computing such *points-to sets* for variables and fields, the analysis constructs an abstraction of the run-time memory states of the analyzed program. This abstraction is typically represented by one or more *points-to graphs*. (An example of a points-to graph is shown in Figure 1, which is discussed later.)

Optimizing compilers for Java can use points-to analysis to perform a variety of popular optimizations such as virtual call resolution, removal of unnecessary synchronization, and stack-based object allocation.[1] Typically, each of these optimizations is based on a *specialized* analysis designed for the purpose of this specific optimization. Thus, compilers that employ multiple optimizations need to implement many different analyses. In contrast, using a single *general-purpose* points-to analysis can enable several different optimizations. An important advantage of this approach is its practicality: the cost of the analysis is amortized across many client optimizations, and the costly development effort to implement the optimizations is significantly reduced.

Another advantage of using points-to analysis is that it enables 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. Such analyses are needed by compilers to perform well-known optimizations such as code motion and partial redundancy elimination. These analyses are also important in the context of software engineering tools: for example, def-use analysis is needed for program slicing and data-flow-based testing. Such analyses and optimizations have been extensively investigated for other languages, but there has been little work on using them for Java. Points-to analysis is a crucial prerequisite for employing these analyses and optimizations, and will allow them to play an increasingly important role in optimizing compilers and software engineering tools for Java.

To make points-to analysis a useful component in realistic Java compilers and tools, the main challenge is to design analyses with the right balance between cost and precision. Fortunately, there is extensive previous experience in designing points-to analyses for C; this analysis expertise can be leveraged by adapting existing C analyses for Java. In this paper we define and evaluate a points-to analysis for Java based on Andersen's points-to analysis for C [5]. Andersen's analysis is a relatively precise flow- and context-insensitive analysis[2] with cubic worst-case complexity. Previous work has shown that constraint-based implementations of this

[1] For example, for virtual call resolution, the compiler can use the points-to solution to identify virtual calls that have only one possible target method. Any such call can be replaced with a direct call, and subsequently the target method can be inlined.

[2] A flow-insensitive analysis ignores the flow of control between program points. A context-insensitive analysis does not distinguish between different invocations of a procedure.

analysis can be very efficient [17, 38]. These implementations are based on set-inclusion constraints of the form $L \subseteq R$, where L and R are points-to sets. Their performance depends on several techniques for efficient constraint representation and constraint resolution.

The previous work on constraint-based implementations of Andersen's analysis for C is insufficient for analysis of Java. First, the analysis must be adapted to model precisely the semantics of virtual calls by performing dispatch based on the class of the receiver object. In addition, since object fields in Java often point to other objects, the analysis must distinguish between the different fields of an object. This is analogous to separate tracking of the different fields of a C structure, which is not done by the existing constraint-based implementations of Andersen's analysis for C. Our empirical results show that the previous constraint-based techniques do not perform well for Java unless augmented with more precise handling of object fields.

We extend and modify the existing constraint-based technology to efficiently represent and solve systems of *annotated* inclusion constraints. The annotated constraints are of the form $L \subseteq_a R$, where a is an annotation. The annotations play two roles in our analysis. *Method annotations* are used to model precisely and efficiently the semantics of virtual calls, by representing the relationships between a virtual call, its receiver objects, and its target methods. *Field annotations* allow separate tracking of the flow of values through the different fields of an object. By using constraint annotations, we have been able to perform practical and precise points-to analysis for Java.

One disadvantage of Andersen's analysis is the implicit assumption that all code in the program is executable. Java programs contain large portions of unused library code; including such dead code can have negative effects on analysis cost and precision. In our points-to analysis, we keep track of all methods reachable from the entry points of the program, and only analyze reachable methods.

We have implemented our analysis and evaluated its performance on a large set of realistic Java programs. On 16 out of the 23 data programs, analysis time is less than a minute. Even on large programs, the analysis runs in a few minutes and uses less than 200Mb of memory. To the best of our knowledge, these are the first published empirical results showing that a relatively precise points-to analysis can run in practical time and space for large Java programs. This practicality makes the analysis a realistic candidate for a relatively precise general-purpose points-to analysis for Java.

We evaluate the impact of the analysis on some of its possible client applications. Our experiments show significant improvement in the precision of the program call graph. Through profiling experiments, we also show that in many cases our analysis allows near-perfect resolution of virtual calls. Using the points-to solution, we can determine that on average 48% of the object allocation sites correspond to objects for which synchronization is unnecessary, and 29% of the sites correspond to objects that can be stack-allocated instead of heap-allocated. Our profiling experiments confirm that a large number of the objects created at run-time can be characterized as not needing synchronization or being allocatable on the stack. Finally, our measurements show very good analysis precision in determining the objects that can be read or written by program statements; this *object read-write information* is a crucial prerequisite for clients such as side-effect analysis and def-use analysis.

```
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);
  }
}
```

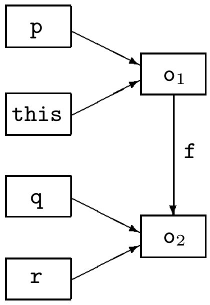


Figure 1: Sample program and its points-to graph.

Contributions The contributions of our work are the following:

- We define a general-purpose points-to analysis for Java based on Andersen's points-to analysis for C.
- We define *annotated* inclusion constraints and present techniques for efficient representation and resolution of such constraints.
- We show how to implement our points-to analysis using annotated inclusion constraints. The implementation models virtual calls and object fields precisely and efficiently, and only analyzes reachable methods.
- We evaluate the analysis on a large set of realistic Java programs. Our results show that the analysis runs in practical time and space, and has significant impact on call graph construction, virtual call resolution, synchronization removal, stack-based object allocation, and object read-write information.

Outline The rest of the paper is organized as follows. Section 2 defines the semantics of our points-to analysis. Section 3 discusses the applications of points-to analysis for Java. Section 4 describes the general structure of our annotated inclusion constraints, and Section 5 contains the details of our constraint-based points-to analysis. The experimental results are presented in Section 6. Section 7 discusses related work, and Section 8 presents conclusions and future work.

2 Semantics of Points-to Analysis for Java

In this section we define the semantics of our points-to analysis for Java by extending the semantics of Andersen's points-to analysis for C. Section 5 describes the implementation of the analysis with annotated inclusion constraints.

The analysis is defined in terms of three sets. Set R contains all reference variables in the analyzed program. Set O contains names for all objects created at object allocation sites; for each allocation site s_i, we use a separate object name $o_i \in O$. Set F contains all instance fields in program classes. Analysis semantics is expressed as manipulations of *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.

We assume that the program is represented by statements of the following form:

- Direct assignment: `l = r`

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

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

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

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

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

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

Figure 2: Points-to effects of program statements.

- Instance field write: `l.f = r`
- Instance field read: `l = r.f`
- Object creation: `l = new C`
- Virtual invocation: $\mathtt{l = r_0.m(r_1,\ldots,r_k)}$

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 $\mathtt{r_0}$ [22, 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` [22, Section 15.11.4].

Analysis semantics is defined in terms of rules for adding new edges to points-to graphs. Each rule represents the semantics of a program statement. The rules for different program statements are shown in Figure 2 in the format $\langle s, G\rangle \Rightarrow G'$, where s is a statement, G is the 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 solution computed by the analysis is a points-to graph that is the closure of the empty graph under the edge-addition rules.

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 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 value of m_j.

3 Applications of Points-to Analysis for Java

Points-to analysis has a wide range of client applications, which makes it a useful component of optimizing compilers and software engineering tools for Java. The cost of the analysis can be amortized across many client applications. Once implemented, the analysis can be reused by various clients at no additional development cost; such reusability is an important practical advantage. In this section we briefly discuss several specific applications of points-to analysis for Java. In our experiments, we have evaluated the impact of the analysis on some of these applications; the results from these experiments are presented in Section 6.

Call Graph Construction and Virtual Call Resolution The points-to solution can be used to determine the target methods of a virtual call by examining the classes of all possible receiver objects. The set of target methods is needed to construct the *call graph* for the analyzed program; this graph is a fundamental prerequisite for all interprocedural analyses and optimizations used in Java compilers and tools. If the call has only one target method, it can be *resolved* by replacing the virtual call with a direct call; this optimization eliminates the run-time overhead of virtual dispatch. In addition, virtual call resolution allows subsequent inlining of the target method, potentially enabling additional optimizations within the caller.

Synchronization Removal Synchronization in Java allows safe access to shared objects in multi-threaded programs. Each object has an associated lock which is used to ensure mutual exclusion. Synchronization operations on locks can have considerable run-time overhead; this overhead occurs even in single-threaded programs, because the standard Java libraries are written in a thread-safe manner.

Static analysis can be used to detect properties that allow the removal of unnecessary synchronization. For example, no synchronization is necessary when accessing an object that may never "escape" its creating thread (i.e., a *thread-local* object).

Definition 1 *Let o be an object and T be the thread that created o. The object is thread-local if no thread $T' \neq T$ can access o.*

Some escape analyses [11, 7, 8, 43] have been used to identify thread-local objects and to remove the synchronization constructs associated with such objects. Points-to analysis can be used as an alternative to escape analysis in detecting thread-local objects. Consider an object o_i and suppose that in the points-to graph computed by the analysis, o_i is not reachable from (i) static (i.e., global) reference variables, or (ii) objects of classes that implement interface `java.lang.Runnable`[3]. It can be proven that in this case o_i is not accessible outside the thread that created it; thus, we can identify such thread-local objects by traversing the points-to graph.

Stack Allocation In some cases, an object can be allocated on a method's stack frame rather than on the heap. This transformation reduces garbage collection overhead and enables additional optimizations such as object reduction [21]. Similarly to synchronization removal, static analysis can be used to detect properties that allow stack-based allocation. For example, stack allocation is possible for an object that may never "escape" the lifetime of its creating method (i.e., a *method-local* object).

Definition 2 *Let o be an object and M be the method that created o. The object is method-local if o can only be accessed during the lifetime of M.*

Some escape analyses [11, 7, 43] can detect method-local objects; clearly, such objects can be allocated on the stack frames of their creating methods. Points-to analysis can be used as an alternative to escape analysis in identifying method-local objects. Let object o_i be thread-local according to the conditions described above. Suppose that in the

[3]The `run` methods of such objects are the starting points of new threads.

points-to graph computed by the analysis, o_i is not reachable from the formal parameters or the return variable of the method that created o_i. In this case, it can be proven that o_i is method-local; thus, we can identify such method-local objects by traversing the points-to graph.

Object Read-Write Information Points-to analysis can be used to determine which objects are read and/or written by every program statement. This information is required for a variety of other static analyses. For example, for the purposes of side-effect analysis, points-to information can be used to answer questions like *"Can statement $p.f = x$ modify the f field of any object pointed to by q?"*. As another example, points-to information is needed to answer questions like *"Can statement $z = q.f$ read any memory locations that are written by statement $p.f = x$?"*, which are fundamental for def-use analysis.

Analyses that require read-write information are used in compilers to perform various advanced optimizations such as intraprocedural and interprocedural code motion and partial redundancy elimination. In addition, such analyses play a fundamental role in a variety of software engineering tools (e.g., in the context of program slicing or data-flow-based testing), thus enabling tool support for complex development tasks. Despite the extensive previous work on such analyses and optimizations for other languages, there has been little work on using them in Java compilers and tools. Practical and precise points-to analysis is crucial for enabling the use of these analyses and optimizations for Java.

4 Annotated Inclusion Constraints

This section describes the general structure of the annotated inclusion constraints used in our points-to analysis for Java. The details about the specific kinds of constraints and annotations are discussed in Section 5.

Previous work on Andersen's analysis for C [17, 38] is based on non-annotated inclusion constraints and uses several techniques for efficient constraint representation and resolution. We extend this work by introducing annotations that allow us to model object fields and virtual calls in Java.

4.1 Constraint Language

We consider annotated set-inclusion constraints of the form $L \subseteq_a R$, where a is chosen from a given set of annotations. We assume that one element of this set is designated as the empty annotation ϵ, and will use $L \subseteq R$ to denote constraints labeled with it. In our analysis, the annotations are used to model the flow of values through fields of objects, as well as the flow of values between a virtual call site and the run-time target methods of the call.

L and R are expressions representing sets, defined by the following grammar:

$$L, R \to v \mid c(v_1, \ldots, v_n) \mid \mathit{proj}(c, i, v) \mid 0 \mid 1$$

Here v and v_i are set variables, $c(\ldots)$ are constructed terms and $\mathit{proj}(\ldots)$ are projection terms. Each *constructed term* is built from an n-ary constructor c. A constructor is either *covariant* or *contravariant* in each of its arguments; the role of this variance in constraint resolution will be explained shortly. Constructed terms may appear on both sides of inclusion relations. 0 and 1 represent the empty set and the universal set; they are treated as nullary constructors. *Projections* of the form $\mathit{proj}(c, i, v)$ are terms used to select the i-th argument of any constructed term $c(.., v_i, ..)$.

$$c(v_1, ..., v_n) \subseteq_a c(v'_1, ..., v'_n) \Rightarrow \begin{cases} v_i \subseteq_a v'_i & \text{if } c \text{ is covariant in } i \\ v'_i \subseteq_a v_i & \text{if } c \text{ is contravariant in } i \end{cases}$$

$$c(v_1, ..., v_n) \subseteq_a \mathit{proj}(c, i, v) \Rightarrow \begin{cases} v_i \subseteq_a v & \text{if } c \text{ is covariant in } i \\ v \subseteq_a v_i & \text{if } c \text{ is contravariant in } i \end{cases}$$

Figure 3: Resolution rules for structural constraints.

Projection terms may appear only on the right-hand side of an inclusion.

4.2 Annotated Constraint Graphs

Systems of constraints from the above language can be represented as directed multi-graphs. This representation is a natural extension of the graph representation for non-annotated constraints used to implement Andersen's analysis for C [17]. Constraint $L \subseteq_a R$ is represented by an edge from the node for L to the node for R; the edge is labeled with the annotation a. There could be multiple edges between the same pair of nodes, each with a different annotation.

The nodes in the graph can be classified as variables, sources, and sinks. *Sources* are constructed terms that occur on the left-hand side of inclusions. *Sinks* are constructed terms or projections that occur on the right-hand side of inclusions. The graph only contains edges that represent *atomic constraints* of the following forms: $\mathit{Source} \subseteq_a \mathit{Var}$, $\mathit{Var} \subseteq_a \mathit{Var}$, or $\mathit{Var} \subseteq_a \mathit{Sink}$. If the constraint system contains a *structural* (non-atomic) constraint, the resolution rules from Figure 3 are used to generate new atomic constraints, as described in Section 4.3.

We use annotated constraint graphs based on the *inductive form* representation [3]. Inductive form is an efficient sparse representation that does not explicitly represent the transitive closure of the constraint graph. The graphs are represented with adjacency lists $\mathit{pred}(n)$ and $\mathit{succ}(n)$ stored at each node n. Edge (n_1, n_2, a), where a is an annotation, is represented either as a predecessor edge by having $\langle n_1, a\rangle \in \mathit{pred}(n_2)$, or as a successor edge by having $\langle n_2, a\rangle \in \mathit{succ}(n_1)$, but not both. $\mathit{Source} \subseteq_a \mathit{Var}$ is always a predecessor edge and $\mathit{Var} \subseteq_a \mathit{Sink}$ is always a successor edge. $\mathit{Var} \subseteq_a \mathit{Var}$ is either a predecessor or a successor edge, based on a fixed total order $\tau : \mathit{Vars} \to \mathcal{N}$. Edge (v_1, v_2, a) is a predecessor edge if and only if $\tau(v_1) < \tau(v_2)$. The order function is typically based on the order in which variables are created as part of building the constraint system [38].

4.3 Solving Systems of Annotated Constraints

Every system of annotated inclusion constraints can be represented by an annotated constraint graph in inductive form. The system is solved by computing the closure of the graph under the following transitive closure rule:

$$\left.\begin{array}{l} \langle L, a\rangle \in \mathit{pred}(v) \\ \langle R, b\rangle \in \mathit{succ}(v) \\ \mathit{Match}(a, b) \end{array}\right\} \Rightarrow L \subseteq_c R \qquad (\textsc{Trans})$$

The closure rule can be applied locally, by examining

$pred(v)$ and $succ(v)$. The new transitive constraint is created *only if* the annotations of the two existing constraints "match"—that is, only if $Match(a, b)$ holds, where *Match* is a binary predicate on the set of annotations. Intuitively, the TRANS rule uses the annotations to filter out some flow of values in the constraint system. The *Match* predicate is defined as follows:

$$Match(a, b) = \begin{cases} \text{true} & \text{if } a \text{ or } b \text{ is the empty annotation } \epsilon \\ \text{true} & \text{if } a = b \\ \text{false} & \text{otherwise} \end{cases}$$

The annotation c of the new constraint is

$$c = \begin{cases} a & \text{if } b = \epsilon \\ b & \text{if } a = \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

Intuitively, an annotation is propagated until it is matched with another instance of itself, after which the two instances cancel out.

If the new constraint generated by the TRANS rule is atomic, a new edge is added to the graph. Otherwise, the resolution rules from Figure 3 are used to transform the constraint into several atomic constraints and their corresponding edges are added to the graph.

The closure of a constraint graph under the TRANS rule is the *solved inductive form* of the corresponding constraint system. The least solution of the system is not explicit in the solved inductive form [3], but is easy to compute by examining all predecessors of each variable. For constraint graphs without annotations, the least solution $LS(v)$ for a variable v is

$$LS(v) = \{c(\ldots) \mid c(\ldots) \in pred(v)\} \cup \bigcup_{u \in pred(v)} LS(u)$$

In this case, $LS(v)$ can be computed by transitive acyclic traversal of all predecessor edges [17]. For an annotated constraint graph, the traversal is done similarly, but the annotations are used as in rule TRANS:

$$\begin{aligned} LS(v) = \{\langle c(\ldots), a\rangle \mid \langle c(\ldots), a\rangle \in pred(v)\} \cup \\ \{\langle c(\ldots), z\rangle \mid \langle u, x\rangle \in pred(v) \wedge \langle c(\ldots), y\rangle \in LS(u) \\ \wedge\, Match(x, y)\} \end{aligned}$$

Here annotation z is computed from annotations x and y as in the TRANS rule.

5 Constraint-based Points-to Analysis for Java

In this section we show how to implement the points-to analysis from Section 2 using annotated inclusion constraints. Recall that the analysis is defined in terms of the set R of all reference variables and the set O of names for all objects created at object allocation sites. Every element of $(R \cup O)$ is essentially an abstract memory location representing a set of run-time memory locations.

To model the analysis with annotated inclusion constraints, we extend a technique developed for Andersen's analysis for C [17, 38]. For each abstract location x, a set variable v_x represents the set of abstract locations pointed to by x. The representation of each location is through a trinary constructor *ref* which is used to build constructed terms of the form $ref(x, v_x, \overline{v_x})$. The last two arguments are the same variable, but with different variance—the overline notation is used to denote a contravariant argument. Intuitively, the second argument is used to read the values of locations pointed to by x, while the last argument is used to update the values of locations pointed to by x. Given a reference variable $r \in R$ and an object variable $o \in O$, constraint

$$ref(o, v_o, \overline{v_o}) \subseteq v_r$$

shows that r points to o.

We use *field annotations* to model the flow of values through fields of objects. Field annotations are unique identifiers for all instance fields defined in program classes. For any two object variables o_1 and o_2, constraint

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$$

shows that field f in object o_1 points to object o_2.

$$\langle l = new\ o_i\rangle \Rightarrow \{ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq v_l\}$$

$$\langle l = r\rangle \Rightarrow \{v_r \subseteq v_l\}$$

$$\langle l.f = r\rangle \Rightarrow \{v_l \subseteq proj(ref, 3, u),\ v_r \subseteq_f u\},\ u \text{ fresh}$$

$$\langle l = r.f\rangle \Rightarrow \{v_r \subseteq proj(ref, 2, u),\ u \subseteq_f v_l\},\ u \text{ fresh}$$

Figure 4: Constraints for assignment statements.

s_1: `p = new X();`
s_2: `q = new Y();`
`p.f = q;`
`r = p.f;`

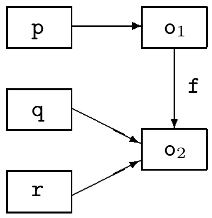


Figure 5: Accessing object fields.

5.1 Constraints for Assignment Statements

For every program statement, our analysis generates annotated inclusion constraints representing the semantics of the statement. Figure 4 shows the constraints generated for assignment statements. The first two generation rules are straightforward. The rule for $l.f = r$ uses the first constraint to access the points-to set of l, and the second constraint to update the values of field f in all objects pointed to by l. Similarly, the last rule uses two constraints to read the values of field f in all objects pointed to by r.

Example Consider the statements in Figure 5 and their corresponding points-to graph. After processing the statements, our analysis creates the following constraints:

$$\begin{array}{ll} ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p & ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_q \\ v_p \subseteq proj(ref, 3, u) & v_q \subseteq_f u \\ v_p \subseteq proj(ref, 2, w) & w \subseteq_f v_r \end{array}$$

where u and w are fresh variables. For the purpose of this example we assume that the variable order τ (defined in Section 4.2) is $\tau(v_p) < \tau(v_q) < \tau(v_r) < \tau(v_{o_1}) < \tau(v_{o_2}) < \tau(u) < \tau(w)$. Consider the indirect write in `p.f = q`. Since we have

$$ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p \subseteq proj(ref, 3, u)$$

we can use the Trans rule and the resolution rules from Figure 3 to generate a new constraint $u \subseteq v_{o_1}$. Thus,

$$v_q \subseteq_f u \subseteq v_{o_1}$$

and using rule Trans we generate $v_q \subseteq_f v_{o_1}$. Intuitively, this new constraint shows that some of the values of field f in object o_1 come from variable q. Now we have

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_q \subseteq_f v_{o_1}$$

Since both constraint edges are predecessor edges, we cannot apply rule Trans. Still, in the least solution of the constraint system (as defined in Section 4.3), we have the constraint $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$, which shows that field f of o_1 points to o_2.

To model indirect reads, we use the second argument of the *ref* constructor. For example, for the constraints above we have

$$ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p \subseteq proj(ref, 2, w)$$

and therefore $v_{o_1} \subseteq w \subseteq_f v_r$, which through Trans generates $v_{o_1} \subseteq_f v_r$. This new constraint shows that the value of r comes from field f of object o_1. Now we have

$$v_q \subseteq_f v_{o_1} \subseteq_f v_r$$

Since the annotations of the two constraints match—that is, they represent accesses to the same field—we generate $v_q \subseteq v_r$ to represent the flow of values from q to r. Thus, in the least solution of the system we have

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_r$$

which shows that reference variable r points to o_2. This example illustrates how field annotations allow us to model flow of values through object fields.

5.2 Handling of Virtual Calls

For every virtual call in the program, our analysis generates a constraint according to the following rule:

$$\langle l = r_0.m(r_1, \ldots, r_k)\rangle \Rightarrow \\ \{v_{r_0} \subseteq_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)\}$$

The rule is based on a *lam* (lambda) constructor. The constructor is used to build a term that encapsulates the actual arguments and the left-hand-side variable of the call. The annotation on the constraint is a unique identifier of the *compile-time* target method of the call. This annotation is used during the analysis to find all appropriate *run-time* target methods.

To model the semantics of virtual calls as defined in Section 2, we separately perform virtual dispatch for every receiver object pointed to by r_0. In order to do this efficiently, we use a precomputed *lookup table*. For a given receiver object at a virtual call site, the lookup table is used to determine the corresponding run-time target method, based on the class of the receiver object.[4] Such a table is straightforward to precompute by analyzing the class hierarchy; the table is essentially a representation of the *dispatch* function from Section 2.

[4]Every object is tagged with its class; this tag is used when performing lookups.

Given the class of the receiver object and the unique identifier for the compile-time target of the virtual call, the lookup table returns a lambda term of the form

$$lam(\overline{v_{p0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret})$$

Here p_i are the formal parameters of the run-time target method; p_0 corresponds to the implicit parameter `this`. We assume that each method has a unique variable *ret* that is assigned the value returned by the method. At the beginning of the analysis, lambda terms of this form are created for all non-abstract methods in the program and are stored in the lookup table.

To model the effects of virtual calls, we define an additional closure rule Virtual. This rule encodes the semantics of virtual calls described in Section 2 and is used together with the Trans rule to obtain the solved form of the constraint system. Virtual is applied whenever we have two constraints of the form

$$ref(o, v_o, \overline{v_o}) \subseteq v \qquad v \subseteq_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

As described in Section 4.2, the edge from the *ref* term is a predecessor edge, and the edge to the *lam* term is a successor edge. Thus, the Virtual closure rule can be applied locally, by examining sets $pred(v)$ and $succ(v)$. Whenever two such constraints are detected, the lookup table is used to find the lambda term for the run-time method corresponding to object o and compile-time target method m. The result of applying Virtual are two new constraints:

$$\begin{array}{rcl} ref(o, v_o, \overline{v_o}) & \subseteq & v_{p0} \\ lam(\overline{v_{p0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) & \subseteq & lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l) \end{array}$$

The first constraint creates the association between parameter `this` of the invoked method and the receiver object. The second constraint immediately resolves to $v_{r_i} \subseteq v_{p_i}$ (for $i \geq 1$) and $v_{ret} \subseteq v_l$, plus the trivial constraint $0 \subseteq v_{p0}$. These new atomic constraints model the flow of values from actuals to formals, as well as the flow of return values to the left-hand side variable l used at the call site.

Example Consider the set of statements in Figure 6. For the purpose of this example, assume that $\tau(v_a) < \tau(v_c) < \tau(v_b)$. Since the declared type of `b` is `A`, at call site c_1 the compile-time target method is `A.n`[5]; thus, we have

$$v_b \subseteq_{A.n} lam(\overline{0}, v_x)$$

When rule Virtual is applied as shown in (1), the lookup for receiver object o_2 and compile-time target `A.n` produces run-time target `B.n`. The resolution with the *lam* term for `B.n` creates the two new constraints shown in (1).

The declared type of `c` is `B`, and for call site c_2 we have

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_c \qquad v_c \subseteq_{B.n} lam(\overline{0}, v_y)$$

where the first constraint is obtained through the Trans rule.[6] By applying Virtual, we create the constraints shown in (2). For call site c_3, the receiver object can be either o_1 or o_2. As shown in (3) and (4), separate lookup and resolution is performed for each receiver.

[5]We use `X.z` to denote method `z` defined in class `X`.

[6]Note that if $\tau(v_b) < \tau(v_c)$, instead of propagating the *ref* term to v_c we would propagate the *lam* term to v_b.

```
class A { X n() { ...  return rA; } }
class B extends A
        { X n() { ...  return rB; } }
s1:  A a = new A();
s2:  A b = new B();
     B c = b;
c1:  X x = b.n();
c2:  X y = c.n();
     if (...)  a = b;
c3:  X z = a.n();
```

$$(1)\quad ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_b \subseteq_{A.n} lam(\overline{0}, v_x) \Rightarrow$$
$$\{ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_{B.n.this},\ v_{rB} \subseteq v_x\}$$

$$(2)\quad ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_c \subseteq_{B.n} lam(\overline{0}, v_y) \Rightarrow$$
$$\{ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_{B.n.this},\ v_{rB} \subseteq v_y\}$$

$$(3)\quad ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_a \subseteq_{A.n} lam(\overline{0}, v_z) \Rightarrow$$
$$\{ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_{A.n.this},\ v_{rA} \subseteq v_z\}$$

$$(4)\quad ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_a \subseteq_{A.n} lam(\overline{0}, v_z) \Rightarrow$$
$$\{ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_{B.n.this},\ v_{rB} \subseteq v_z\}$$

Figure 6: Example of virtual call resolution.

5.3 Correctness

For every program statement, our analysis generates constraints representing the semantics of the statement. This initial constraint system is solved by closing the corresponding constraint graph under closure rules TRANS and VIRTUAL. Let A^* be the solved inductive form of the constraint system. Recall that the least solution of the system is not explicit in A^* and can be obtained through additional traversal of predecessor edges, as described in Section 4.3.

Let G^* be the points-to graph computed by the algorithm in Section 2. Consider a reference variable r and an object variable o such that $(r, o) \in G^*$. It can be proven that the least solution constructed from A^* contains the constraint $ref(o, v_o, \overline{v_o}) \subseteq v_r$. Similarly, consider two object variables o_i and o_j such that $(\langle o_i, f\rangle, o_j) \in G^*$; it can be proven that the least solution contains $ref(o_j, v_{o_j}, \overline{v_{o_j}}) \subseteq_f v_{o_i}$.

The proof of the these claims depends on the following restriction on the variable order τ: all variables v_r, where $r \in R$, should have lower order than the rest of the constraint variables. We enforce this restriction as part of building the constraint system. Given this restriction, it can be proven that the least solution of the constraint system represents all points-to pairs from G^*. The proof of this claim is outlined in Appendix A.

5.4 Cycle Elimination and Projection Merging

Cycle elimination [17] and projection merging [38] are two important techniques for reducing the cost of Andersen's analysis for C. In this section we show how these techniques can be adapted for the annotated constraints used in our analysis for Java.

The idea behind cycle elimination is to detect a set of variables that form a cycle in the constraint graph:

$$v_1 \subseteq v_2 \subseteq \ldots \subseteq v_k \subseteq v_1$$

Clearly, all such variables have equal solutions and can be replaced with a single variable. Whenever a cycle is detected during the resolution process, one variable from the cycle is chosen as a witness variable, and the rest of the variables are redirected to the witness. This transformation has no effect on the computed solution, but can significantly reduce the cost of the analysis.

Cycle detection is performed every time a new edge is added between two variables v_i and v_j. The detection algorithm essentially performs depth-first traversal of the constraint graph and tries to determine whether v_i is reachable from v_j. Cycle detection is partial and does not detect all cycles. Nevertheless, for Andersen's analysis for C this technique has significant impact on the running time of the analysis (see [17] for more details).

Cycle elimination cannot be reused in a straightforward manner for the annotated constraint systems presented in this paper. If we performed the standard cycle detection, we would discover cycles in which some edges have field annotations; however, the variables in such cycles do not have the same solution, and cannot be replaced by a single witness variable. To guarantee the correctness of our analysis for Java, we use a restricted form of cycle elimination. The cycle detection algorithm is the same as in [17], but is invoked only when a new edge is added between two reference variables—that is, when the new edge is (v_{r_i}, v_{r_j}), where $r_i, r_j \in R$. It can be proven that in this case, the detected cycle contains only reference variables, and all edges in the cycle have empty annotations. This guarantees that all variables on the cycle have identical points-to sets, and therefore replacing the cycle with a single variable preserves the points-to solution.

Projection merging is a technique for reducing redundant edge additions in the constraint system [38]. It combines multiple projection constraints for the same variable into a single projection constraint. For example, constraints $v \subseteq proj(c, i, u_1)$ and $v \subseteq proj(c, i, u_2)$ are replaced by

$$v \subseteq proj(c, i, w) \qquad w \subseteq u_1 \qquad w \subseteq u_2$$

where w is a special projection variable. For points-to analysis for C, constraints of the form $w \subseteq u_i$ are represented only as successor edges; this restriction guarantees a bound on the number of projection variables w. The analysis ensures the restriction by assigning to w a high index in the variable order τ. Moreover, projection merging is beneficial only when coupled with cycle elimination (see [38] for details).

In our annotated constraint systems, projection merging does not interact with cycle elimination. In our case, the high indices from [38] (which necessitate the interaction) are not required. The high indices become unnecessary because the bound on the number of special projection variables is ensured by the variable ordering restriction from Section 5.3. Thus, the special projection variables can be treated similarly to the rest of the variables in the constraint system. This *decoupled* form of projection merging has significant impact on the running time of the analysis for Java.

5.5 Tracking Reachable Methods

Andersen's analysis implicitly assumes that all code in the program is executable. Since Java programs heavily use libraries that contain many unused methods, we have augmented our analysis to keep track of all reachable methods, in order to avoid analyzing dead code. Thus, we take into account the effects of statements in a method body only if the method has been shown to be reachable from one of the entry methods of the program. The set of entry methods M_0 contains (i) the `main` method of the starting class, (ii) the methods invoked at JVM startup (e.g., `initializeSystemClass`), and (iii) the class initialization methods `<clinit>` containing the initializers for static fields [28, Section 3.9].

We augment the semantic rules from Section 2 to include this reachability computation. The original rules are of the form $\langle s, G\rangle \Rightarrow G'$, and define the effects of statement s on points-to graph G. The augmented rules have the form $\langle s, G, M\rangle \Rightarrow (G', M')$, where M and M' are sets of reachable methods. If the method that contains s is not in M, we have $G' = G$ and $M' = M$. Otherwise, G' is defined as in Figure 2, and M' contains all elements of M plus all methods that become reachable when s is a call statement. Any call to a constructor also generates a corresponding call to the appropriate `finalize` method. For multi-threaded programs, a call to `Thread.start` is treated as a call to the corresponding `run` method.

We implement this augmented semantics by maintaining a list of reachable methods, initialized with M_0. Whenever a method becomes reachable, all statements in its body are processed and the appropriate constraints are introduced in the constraint system. It is easy to show that this approach produces a safe solution with respect to the augmented semantics.

5.6 Analysis Implementation

We use the Soot framework (`www.sable.mcgill.ca`), version 1.0.0, to process Java bytecode and to build a typed intermediate representation [42]. The constraint-based analysis uses Bane (Berkeley ANalysis Engine) [2]. Bane is a toolkit for constructing constraint-based program analyses. The public distribution of Bane (`bane.cs.berkeley.edu`) contains an efficient constraint-solving engine that employs inductive form [3], cycle elimination [17], and projection merging [38]. We modified the constraint engine to attach annotations to the constraints, to implement the appropriate resolution and closure rules, and to perform cycle elimination and projection merging as described in Section 5.4. The analysis works on top of the constraint engine, by processing newly discovered reachable methods and generating the appropriate constraints. The points-to 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 [25, 19, 40, 41].

6 Empirical Results

All experiments were performed on a 360MHz Sun Ultra-60 machine with 512Mb 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. We used 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

Program	User Class	Size (Kb)	Whole-program Class	Whole-program Method	Whole-program 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.

classes, after using class hierarchy analysis (CHA) [13] to filter out irrelevant classes and methods.[7] 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.

Analysis Cost Our first set of experiments measured the cost of the analysis, as shown in Table 2. The first two columns show the running time of the analysis and the amount of memory used. For 16 out of the 23 programs, the analysis runs in less than a minute. For all programs, the running time is less than six minutes and the memory usage is less than 180Mb. To the best of our knowledge, these are the first published empirical results showing that a relatively precise points-to analysis can run in a few minutes on large realistic Java programs, using reasonable amounts of memory. Empirical results for more expensive points-to analyses [23, 9] suggest that they do not scale; for example, the 1-1-CFA analysis from [23] runs in about 2000 seconds on `javacup` and runs out of memory on `javac`.

The empirical results indicate that our analysis is practical in terms of running time and memory usage, as evidenced on a large set of Java programs. This practicality shows that the analysis is a realistic candidate for a relatively precise general-purpose points-to analysis for Java. Performance can be improved further by techniques such as variable substitution [31], which have proven effective in reducing the cost of Andersen's analysis for C; we intend to investigate this approach in our future work. In addition, analysis cost can be reduced if the library code is analyzed in advance. This would allow analysis information about the Java libraries to be computed once and subsequently used multiple times for different client programs; this is another promising direction for future work.

We also investigated a version of our analysis in which

[7] CHA is an inexpensive analysis that determines the possible targets of a virtual call by examining the class hierarchy of the program.

Program	Time (sec)	Memory (Mb)	Time-nf (sec)	Memory-nf (Mb)
proxy	6.5	38.9	29.8	45.1
compress	22.2	45.3	68.4	77.2
db	23.2	46.8	63.2	80.7
jb	13.3	40.7	28.1	45.6
echo	41.5	61.8	231.6	184.3
raytrace	26.1	51.2	99.6	100.4
mtrt	26.1	49.0	89.7	100.2
jtar	44.7	58.8	125.8	113.3
jlex	17.7	45.7	137.7	83.6
javacup	32.0	54.1	80.5	83.8
rabbit	27.9	53.3	74.3	82.7
jack	48.7	63.1	5871.4	134.8
jflex	56.1	73.1	208.6	191.8
jess	41.8	64.9	202.7	207.7
mpegaudio	28.1	51.8	227.1	226.3
jjtree	24.6	52.5	57.4	76.0
sablecc	287.2	151.9	815.4	280.9
javac	350.0	151.5	396.8	334.1
creature	176.4	101.0	821.4	319.7
mindterm	94.6	95.4	407.7	341.0
soot	239.5	176.4	401.8	308.5
muffin	243.7	163.8	–	–
javacc	190.5	125.5	167.8	167.7

Table 2: Running time and memory usage of the analysis. Last two columns show the cost when object fields are not distinguished.

Program	(a) Removed Targets		(b) Resolved Call Sites	
	Points-to	RTA	Points-to	RTA
proxy	8.7	5.8	50%	9%
compress	6.4	2.4	58%	18%
db	6.0	2.2	61%	18%
jb	8.5	5.8	56%	13%
echo	3.4	1.4	45%	19%
raytrace	6.0	2.3	58%	21%
mtrt	6.0	2.3	58%	21%
jtar	5.3	3.2	39%	17%
jlex	9.3	6.2	63%	12%
javacup	6.3	3.9	63%	14%
rabbit	8.6	4.1	54%	14%
jack	3.0	0.9	83%	12%
jflex	6.5	3.5	45%	12%
jess	5.7	2.0	57%	15%
mpegaudio	7.1	2.2	53%	17%
jjtree	8.4	5.9	54%	14%
sablecc	7.1	1.5	32%	7%
javac	2.8	1.1	32%	15%
creature	3.8	2.3	50%	25%
mindterm	2.5	1.3	43%	24%
soot	4.5	1.0	41%	1%
muffin	5.7	2.0	48%	17%
javacc	5.0	2.8	79%	10%
Average	5.9	2.9	53%	15%

Table 3: Improvements for CHA-unresolved virtual call sites using RTA and the points-to analysis. (a) Average reduction in the number of target methods per call site. (b) Percentage of call sites uniquely resolved by the analyses.

no field annotations are used, and therefore individual object fields are not distinguished. In essence, this *no-fields* version of the analysis is a straightforward extension of the best existing constraint-based implementations of Andersen's analysis for C (recall that these implementations do not distinguish structure fields) [17, 38]. In this version we used the original cycle elimination and projection merging instead of the modified versions from Section 5.4. These experiments investigated the performance achievable through simple reuse of the already existing constraint-based techniques, without introducing our field annotations.

The last two columns in Table 2 show the cost of the *no-fields* version of the analysis. The running time is between 88% and 12056% (average 892%, median 384%) of the running time of the annotations-based analysis; the memory usage is between 112% and 437% (215% on average). Typically, the *no-fields* version is significantly more expensive than our analysis; for one of the larger programs, it even ran out of memory.

We draw two conclusions from these experiments. First, simple reuse of the constraint-based techniques for C is not sufficient: good performance for C *does not* guarantee good performance for Java. Second, it is crucial to distinguish object fields: the improved precision produces smaller points-to sets, which significantly reduces analysis cost. By using field annotations, we have been able to distinguish object fields in a simple and practical manner.

Call Graph Construction and Virtual Call Resolution To measure the precision of the analysis with respect to call graph construction and virtual call resolution, we compared our points-to analysis with Rapid Type Analysis (RTA) [6]. RTA is an inexpensive and widely used analysis for call graph construction. It performs a reachability computation on the call graph generated by CHA; by keeping track of the classes that have been instantiated, RTA computes a more precise call graph than CHA.

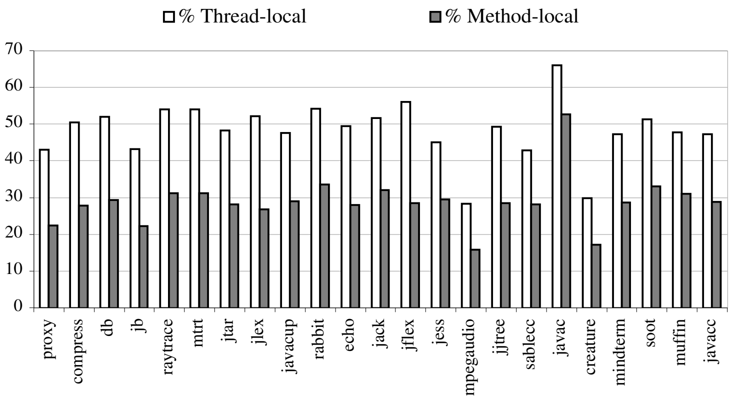
Both our analysis and RTA improve the call graph computed by CHA by identifying sets of methods reachable from the entry points of the program; this reachability computation reduces the number of nodes in the call graph. For brevity, we summarize this reduction without explicitly showing the number of nodes for each program. The average reduction in the number of nodes is 54% for our analysis and 47% for RTA. On average, the call graph computed by our analysis has 14% less nodes than the call graph computed by RTA. This reduction allows subsequent analyses and optimizations to safely ignore large portions of the program.

To determine the improvement for call graph edges, 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 by our analysis as reachable. For our data programs, the size of V is between 7% and 44% (22% on average) of all virtual call sites in reachable methods. For each site from V, we computed the difference between the number of target methods according to CHA and the number of target methods according to RTA and our analysis. The average differences are shown in the first section of Table 3. On average, our analysis removes more than twice as many targets as RTA; this improved precision is beneficial for reducing the cost and improving the precision of subsequent interprocedural analyses.

The second section of Table 3 shows the percentage of call sites from V that were resolved to a single target method. Our points-to analysis performs significantly better than RTA—on average, 53% versus 15% of the virtual call sites are resolved. The increased precision allows better removal of run-time virtual dispatch and additional method inlining.

We performed additional experiments to estimate the potential performance impact of analysis precision on virtual

(a) Object allocation sites



(b) Run-time objects

Program	Objects	Thread-local	Method-local
compress	456	99.3%	39.0%
db	154325	0.03%	0.01%
mtrt	6457298	99.9%	85.0%
jlex	7350	50.9%	31.6%
jack	1340919	86.7%	77.0%
jess	7902221	17.9%	17.9%
mpegaudio	2025	12.4%	12.4%
sablecc	420494	24.9%	13.7%
javac	3738777	27.6%	21.2%
javacc	43265	65.7%	45.8%

Figure 7: (a) Thread-local and method-local allocation sites. (b) Number of objects created at run-time: total number, percentage of thread-local objects, and percentage of method-local objects.

Program	(a) Num Calls	(b) Resolved		(c) Run-time Monomorphic
		RTA	Points-to	
compress	0	—	—	—
db	10550782	0%	100%	100%
mtrt	2833913	0%	0%	0%
jlex	1336	11.0%	99.9%	100%
jack	2663305	5.9%	98.6%	98.6%
jess	1511933	0.03%	0.3%	50.2%
mpegaudio	6528736	0%	0%	0.5%
sablecc	1005390	0.1%	36.5%	46.7%
javac	20198864	0%	19.2%	71.0%
javacc	44322	0.02%	85.8%	85.9%

Table 4: Dynamic execution counts for virtual call sites. (a) Total count for CHA-unresolved call sites. (b) Percentage of (a) contributed by resolved sites. (c) Percentage of (a) contributed by sites with a single run-time target.

call resolution. These experiments used a subset of our data programs for which we had representative input data. For each program, we instrumented the user classes (i.e., non-library classes) and measured the number of times each call site was executed during a profile run of the program. Column (a) in Table 4 shows the total number of invocations of CHA-unresolved call sites. This number is an indicator of the run-time overhead of virtual dispatch, as well as the missed opportunities for performance improvement through inlining. We also measured what percentage of this total number was contributed by call sites that are uniquely resolved by RTA and by our analysis. These percentages are shown in column (b) in Table 4. Higher percentages indicate higher potential for performance improvement if the analyses are used during the compilation and optimization of the user classes.

The results from this profiling experiment indicate that RTA has little potential for improving the run-time performance over CHA. Our analysis shows significantly higher potential, and for several programs it allows the resolution of the majority of run-time virtual calls. In addition, we used the profile to determine what CHA-unresolved call sites had only one run-time target. Column (c) shows the contribution of such sites to the total count from column (a). This number is an upper bound on the number of invocations that could be resolved by a static analysis. By comparing columns (b) and (c), it is clear that in many cases our analysis achieves performance close to the best possible performance.

Synchronization Removal and Stack Allocation In addition to call graph construction and virtual call resolution, points-to analysis produces information that is useful for a variety of other applications: for example, synchronization removal, stack allocation, side-effect analysis, and def-use analysis. To investigate the impact of our analysis on synchronization removal and stack allocation, we identified all object allocation sites that correspond to thread-local and method-local objects, as described in Section 3. Figure 7(a) shows what percentage of all allocation sites in reachable methods were identified as thread-local or method-local.

Across all programs, the analysis detects a significant number of allocation sites for thread-local objects—on average, about 48% of all allocation sites. These results indicate that the analysis can be very useful in detecting and eliminating unnecessary synchronization in Java programs. The analysis also discovers a significant number of sites for method-local objects—on average, about 29% of all sites. This result indicates that there are many opportunities for stack-based object allocation that can be detected with our analysis.

As with virtual call resolution, we performed additional profiling experiments to obtain better estimates of the potential impact on run-time performance. Using the same set of programs with instrumented user classes and the same data input sets, we measured the number of run-time objects created at each object allocation site. The total number of created objects is shown in the first column of Figure 7(b); the other two columns show what percentage of these objects were identified by our analysis as thread-local or method-local.

The results from this experiment indicate that our analysis has good potential for improving the run-time performance through synchronization removal and stack-based object allocation. The results in Figure 7(b) are similar to the results obtained through more expensive flow- and context-sensitive escape analyses [11, 43]. Even though direct comparison with this previous work is not possible (due to differences in the infrastructure and the data programs), the results indicate that our analysis may be a viable alternative to these more expensive analyses.

Program	1	2	3	4–5	6–9	≥10
proxy	52%	18%	7%	9%	6%	8%
compress	56%	12%	12%	7%	6%	7%
db	54%	12%	13%	7%	7%	7%
jb	56%	20%	6%	5%	6%	7%
echo	54%	15%	9%	8%	5%	9%
raytrace	50%	12%	11%	6%	11%	10%
mtrt	50%	12%	11%	6%	11%	10%
jtar	46%	12%	8%	10%	9%	15%
jlex	88%	5%	1%	2%	2%	2%
javacup	64%	11%	4%	4%	8%	9%
rabbit	47%	24%	11%	6%	6%	6%
jack	55%	10%	8%	6%	4%	17%
jflex	60%	12%	10%	4%	3%	11%
jess	45%	14%	14%	13%	8%	6%
mpegaudio	55%	15%	14%	6%	5%	5%
jjtree	51%	20%	8%	4%	11%	6%
sablecc	67%	13%	3%	2%	3%	12%
javac	45%	14%	8%	9%	4%	20%
creature	36%	50%	1%	1%	3%	9%
mindterm	67%	6%	10%	7%	2%	8%
soot	66%	12%	2%	6%	4%	10%
muffin	59%	16%	5%	5%	4%	11%
javacc	70%	5%	2%	3%	7%	13%

Table 5: Number of accessed objects for indirect access expressions. Each column shows the percentage of indirect access expressions whose number of accessed objects is in the corresponding range.

Object Read-Write Information We performed measurements to estimate the potential impact of our analysis on clients of object read-write information (e.g., side-effect analysis and def-use analysis). In particular, we considered all expressions of the form $p.f$ occuring in statements in reachable methods. For each such *indirect access expression*, the points-to set of p contains all objects that may be read or written by the corresponding statement. More precise points-to analyses produce smaller numbers of accessed objects, which in turn results in better precision for the clients of the read-write information. Thus, to estimate the potential impact of our analysis, we measured the number of accessed objects for each indirect access expression; similar metrics have been traditionally used for points-to analysis for C.

Table 5 shows the distribution of the number of accessed objects; each column corresponds to a specific range of numbers. For example, the first column corresponds to expressions that may only access a single object, while the last column corresponds to expressions that may access at least 10 objects. Each column shows what percentage of all indirect access expressions corresponds to the particular range of numbers of accessed objects.

The measurements in Table 5 indicate that our analysis produces precise read-write information. Typically, more than half of the indirect accesses are resolved to a single object (which is the lower bound for this metric), and the majority of accesses are resolved to at most three objects. These results show that our analysis is a promising candidate for producing read-write information for clients such as side-effect analysis and def-use analysis.

7 Related Work

Points-to analysis for object references in Java is clearly related to pointer analysis for imperative languages such as C. There are various pointer analyses for C with different tradeoffs between cost and precision [26, 24, 16, 5, 33, 44, 36, 35, 27, 20, 12, 18, 10]. The closest related work from this category are the efficient constraint-based implementations of Andersen's analysis [17, 38], which use the inductive form representation together with efficient resolution techniques such as cycle elimination and projection merging. Our work extends this approach by introducing constraint annotations and by modifying the graph representation and the resolution procedure. Field annotations allow us to track object fields separately; this is not possible with the constraints from [17, 38]. Method annotations allow us to model the semantics of virtual calls. In addition, we avoid analyzing dead library code by including a reachability computation in the analysis.

Constraint indices and constraint polarities [18] have been used to introduce context-sensitivity in unification-based flow analysis. This work has similar flavor to our use of annotations for tracking flow of values through object fields. Conceptually, in both cases the annotations model restricted value flow in constraint systems—either for unification-based constraints in [18], or for inclusion constraints in our case.

Recent work [37], which postdates our initial report [32], describes points-to analysis for Java based on Andersen's analysis for C. Analysis running times are several times slower than ours, and memory usage is significantly higher. We believe that our low memory consumption is due to the use of annotations for tracking object fields—this representation is inherently more efficient than the standard way of tracking fields through distinct set variables. Another example of points-to analysis for object-oriented languages is due to Chatterjee et al. [9]. This flow- and context-sensitive analysis is more precise than ours, but it is unclear whether it will scale for programs as large as our data programs. Points-to analyses with different degrees of precision have been proposed in the context of a framework for call graph construction in object-oriented languages [23]. The closest to our work is the 1-1-CFA algorithm, which incorporates flow- and context- sensitive points-to analysis. The data set from [23] contains few Java programs, and the analysis times suggest that 1-1-CFA is not practical. Our empirical results on a large set of realistic Java applications show that our analysis is a practical alternative to the more expensive analyses from [23, 9].

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. The traditional client applications of class analysis are call graph construction and virtual call resolution. In general, points-to analysis subsumes class analysis; practical implementations of points-to analysis can be used for a variety of applications, including call graph construction and virtual call resolution. DeFouw et al. [14] present a family of practical interprocedural class analyses. Other work in this area considers more precise and costly analyses with some degree of context- or flow-sensitivity [29, 1, 30, 15, 23], as well as less precise but inexpensive analyses such as RTA [6, 41, 39].

There is a large body of work on synchronization removal and stack-based object allocation [4, 11, 7, 8, 43, 21, 34]. Gay and Steensgaard [21] present a unification-based analysis for stack allocation. Ruf [34] describes an unification-based algorithm for synchronization removal. Aldrich et al. [4] develop several algorithms for synchronization removal based on 1-1-CFA; the scalability and usability of these analyses can be significantly improved if our points-to analysis is used instead of 1-1-CFA. Previous work on escape analysis for Java [11, 7, 8, 43] also investigates synchronization removal and stack allocation; the scalability of these approaches remains unclear, since only [7] reports analysis times. In contrast to these specialized analyses, we have a practical general-purpose points-to analysis that can be used by a variety of client applications, including synchronization removal and stack allocation.

8 Conclusions and Future Work

Points-to analysis has a wide range of client applications, which makes it a useful component of optimizing compilers and software engineering tools for Java. We present a points-to analysis for Java based on Andersen's points-to analysis for C. Our work extends existing constraint-based techniques by introducing constraint annotations. Method annotations are used to model precisely and efficiently the semantics of virtual calls. Field annotations allow us to distinguish between different fields of an object; this analysis feature is crucial for reducing analysis cost. On a large set of realistic Java programs, our experiments show that the analysis cost is practical. Through static and profiling measurements, we show that the points-to solution has significant impact on call graph construction, virtual call resolution, synchronization removal, stack-based object allocation, and object read-write information. Our results show that the analysis is a realistic candidate for a relatively precise, practical, general-purpose points-to analysis for Java.

One direction of future work is to investigate techniques for further reducing the cost of the analysis. For example, various kinds of approximations could reduce the cost without significant (or any) loss of precision (e.g., through techniques such as variable substitution [31]). In addition, analysis cost can be reduced if the library code is analyzed in advance. This would allow analysis information about the Java libraries to be computed once and subsequently used multiple times for different client programs.

Another direction of work is to investigate the impact of the analysis solution on traditional client analyses such as def-use analysis and side-effect analysis, which in turn are necessary for various well-known optimizations (e.g., code motion and partial redundancy elimination). Such analyses and optimizations have been extensively investigated in other languages, and will play an increasingly important role in aggressive optimizing compilers for Java.

Finally, it would be interesting to investigate applications of points-to analysis in the context of software engineering tools: for example, def-use analysis for program slicing and data-flow-based testing. The functionality provided by such tools will be an important advantage in the development of production-strength Java software systems.

9 Acknowledgments

This research was supported by NSF grant CCR-9900988.

References

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

[2] 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.

[3] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, June 1993.

[4] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis Symposium*, LNCS 1694, pages 19–38, 1999.

[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] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20–34, 1999.

[8] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 35–46, 1999.

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

[10] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[11] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

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

[13] 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.

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

[15] 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.

[16] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Conference on Programming Language Design and Implementation*, pages 242–257, 1994.

[17] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[18] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

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

[20] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, LNCS 1824, pages 175–198, 2000.

[21] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction*, LNCS 1781, 2000.

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

[23] 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.

[24] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Trans. Programming Languages and Systems*, 21(4):848–894, May 1999.

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

[26] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[27] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 199–215, 1999.

[28] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

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

[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] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Conference on Programming Language Design and Implementation*, pages 47–56, 2000.

[32] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated inclusion constraints. Technical Report DCS-TR-417, Rutgers University, July 2000. (Initial report superseded by this work).

[33] E. Ruf. Context-insensitive alias analysis reconsidered. In *Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

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

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

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

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

[38] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[39] 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.

[40] 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.

[41] 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.

[42] R. Vallée-Rai and L. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, McGill University, July 1998.

[43] 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.

[44] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

A Proof of Correctness

Let G be a points-to graph as defined in Section 2 and A be an annotated constraint graph in inductive form as described in Section 5. We will define a *representation relation* α between A and the edges in G. If α holds between A and all edges in G, we will write $\alpha(A, G)$.

Consider a reference variable r and an object variable o such that $e = (r, o)$ is an edge in G. We have $\alpha(A, e)$ if and only if A contains a path

$$ref(o, v_o, \overline{v_o}) \to v_1 \to \ldots \to v_n \to v_r$$

such that all v_i correspond to reference variables and all edges are predecessor edges with empty annotations.

Similarly, consider two object variables o_i and o_j such that $e = (\langle o_i, f\rangle, o_j)$ is an edge in G. We have $\alpha(A, e)$ if and only if at least one of the following two conditions is true. The first condition is that A contains a path

$$ref(o_j, v_{o_j}, \overline{v_{o_j}}) \to v_1 \to \ldots \to v_n \xrightarrow{f} v_{o_i}$$

such that all v_i $(1 \le i \le n)$ correspond to reference variables, all edges are predecessor edges, and the only non-empty annotation on the path is f on edge (v_n, v_{o_i}). The second condition is that A contains a path

$$ref(o_j, v_{o_j}, \overline{v_{o_j}}) \to v_1 \to \ldots \to v_n \xrightarrow{f} u \to v_{o_i}$$

such that all v_i $(1 \le i \le n)$ correspond to reference variables and u is a "fresh" variable created due to field access (see Figure 4). In this path, all edges are predecessor edges, and the only non-empty annotation is f on edge (v_n, u).

Let G^* be the final points-to graph computed by the algorithm in Section 2, and A^* be the solved inductive form of the corresponding annotated constraint system. The least solution of the constraint system is obtained through additional traversal of predecessor edges in A^*, as described in Section 4.3.

Suppose that $\alpha(A^*, G^*)$. It is easy to show that in this case G^* is contained in the set of points-to pairs extracted from the least solution of the constraint system. For example, for every edge $(\langle o_i, f\rangle, o_j) \in G^*$, the least solution contains $ref(o_j, v_{o_j}, \overline{v_{o_j}}) \subseteq_f v_{o_i}$. To prove that $\alpha(A^*, G^*)$, we use the following theorem:

Theorem 1 *Let $\langle s, G\rangle \Rightarrow G'$ as described in Figure 2. If $\alpha(A, G)$, there exists a sequence of applications of closure rules and resolution rules such that A can be transformed into A' for which $\alpha(A', G')$.*

The proof of this claim requires case-by-case analysis of all statement kinds.

Case 1. Consider the assignment $l = r$. In G', there are new edges of the form (l, o_i), where $(r, o_i) \in G$. In A, we have a path from $ref(o_i, v_{o_i}, \overline{v_{o_i}})$ to v_r containing only predecessor edges with empty annotations. Graph A also contains $v_r \subseteq v_l$, which is represented either as a predecessor edge or as a successor edge (depending on the order of v_r and v_l). If $v_r \in \mathit{pred}(v_l)$, then we have the needed path from $\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}})$ to v_l. If $v_l \in \mathit{succ}(v_r)$, then a sequence of applications of rule Trans creates the needed path.

Case 2. Consider the assignment $l = r.f$. In G', there are new edges of the form (l, o_j), where $(r, o_i) \in G$ and $(\langle o_i, f\rangle, o_j) \in G$. In A we have a path $\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) \rightarrow \ldots \rightarrow v_r$ containing only predecessor edges with empty annotations. There are two possible paths representing edge $(\langle o_i, f\rangle, o_j)$; for brevity, we only discuss the case when A contains a path

$$\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \xrightarrow{f} v_{o_i}$$

such that all v_i ($1 \leq i \leq n$) correspond to reference variables, all edges are predecessor edges, and the only non-empty annotation on the path is f on edge (v_n, v_{o_i}). A also contains constraints $v_r \subseteq \mathit{proj}(\mathit{ref}, 2, u)$ and $u \subseteq_f v_l$ (see Figure 4), which are represented as successor edges. Therefore we have

$$\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) \rightarrow \ldots \rightarrow v_r \rightarrow \mathit{proj}(\mathit{ref}, 2, u)$$

and after a sequence of applications of Trans followed by the resolution rules for structural constraints shown on Figure 3, we have $v_{o_i} \subseteq u$.

In the case when $\tau(v_{o_i}) < \tau(u)$, we have $v_{o_i} \in \mathit{pred}(u)$ and $\langle v_l, f\rangle \in \mathit{succ}(u)$, which results in an f-annotated successor edge that represents constraint $v_{o_i} \subseteq_f v_l$. Combining this edge with the f-annotated predecessor edge on the path from $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}})$ to v_{o_i} results in the needed path. Similarly, if $\tau(v_{o_i}) > \tau(u)$, we have $\langle v_n, f\rangle \in \mathit{pred}(v_{o_i})$ and $u \in \mathit{succ}(v_{o_i})$, which results in an f-annotated predecessor edge that represents constraint $v_n \subseteq_f u$. Combining this edge with the f-annotated successor edge representing $u \subseteq_f v_l$ creates the needed path.

Note that the variable ordering restriction described in Section 5.3 ensures that the appropriate kinds of edges (predecessor or successor) are created when transforming A. For example, suppose that $\tau(u) < \tau(v_l)$ for Case 2 from above. Transforming A according to the applicable rules would create a path from $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}})$ to v_l which contains edges with non-empty annotations; in contrast, the needed path from $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}})$ consists of predecessor edges with empty annotations. Therefore $\alpha(A', G')$ cannot be guaranteed without the variable ordering restriction from Section 5.3.

The rest of the cases in the proof are handled in a similar manner. Given the above theorem, it is trivial to show that α holds between A^* and G^*.