# An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages

John Whaley    Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{jwhaley, lam}@stanford.edu

**Abstract.** We describe the design and implementation of an efficient inclusion-based points-to analysis for strictly-typed object-oriented languages. Our implementation easily scales to millions of lines of Java code, and it supports language features such as inheritance, object fields, exceptional control flow, type casting, dynamic dispatch, and reflection.
Our algorithm is based on Heintz and Tardieu's Andersen-style points-to analysis designed originally for C programs. We have improved the precision of their algorithm by tracking the fields of individual objects separately and by analyzing the local variables in a method in a flow-sensitive manner. Our algorithm represents the semantics of each procedure concisely using a sparse summary graph representation based on access paths; it iterates over this sparse representation until it reaches a fixed point solution. By utilizing the access path and field information present in the summary graphs, along with minimizing redundant operations and memory management overheads, we are able to quickly and effectively analyze very large programs. Our experimental results demonstrate that this technique can be used to compute precise static call graphs for very large Java programs.

## 1 Introduction

Points-to analysis is a fundamental technique whose results are useful for program optimizations, program comprehension tools, and software engineering aids. In particular, in the context of object-oriented programming languages like Java, points-to analysis is useful for *concrete type inferencing*. By knowing an object's concrete type statically, a compiler can replace dynamic dispatches with inlined methods to reduce the run-time overhead and enable further optimizations. Moreover, the availability of a more precise call graph is generally useful to any program comprehension tool.

Previous work on concrete type inferencing ranged from rather weak techniques such as Class Hierarchy Analysis (CHA)[3] and Rapid Type Analysis RTA[2] to more sophisticated algorithms that analyze the use of pointers in the

This material is based upon work supported in part by the National Science Foundation under Grant No. 0086160 and an NSF Graduate Student Fellowship.

program[6, 7, 9, 11]. One of the major concerns in the design of such algorithms is the trade-off between precision and speed of the analysis.

Many of the pointer alias analyses developed in the past were designed for analyzing C programs. Previously, the only pointer alias analysis deemed scalable was Steensgaard's context-insensitive and flow-insensitive algorithm[8]. With a near-linear complexity, the technique is fast but imprecise. The algorithm is unification-based; as such, pointers are either unaliased or pointing to the same set of locations. While the pointer analysis proposed by Andersen is also flow-insensitive and context-insensitive, it is more precise because it is inclusion-based[1]. That is, two aliased pointers may point to overlapping but different sets of locations. However, with an $O(n^3)$ complexity, it was considered not scalable to real programs. Recently, Heintze and Tardieu proposed an optimization technique (CLA) that allows Andersen's technique to scale to programs with a million lines of code[5].

This paper adapts and extends the CLA algorithm to the problem of pointer analysis in Java. We have developed an equally scalable points-to analysis for Java programs so that application developers needing pointer information can count on having at least Andersen's level of precision without having to be concerned with the cost of the analysis. We show that there is no reason to resort to a less precise pointer analysis for concrete type inferencing.

We have extended the CLA algorithm to handle all the Java constructs, which include virtual method invocations, type casts, exceptions, unknown and escaping pointers. In addition, we improved the CLA algorithm in three ways:

1. **A field-sensitive pointer analysis.** Most C pointer alias analyses are *field-independent*. All the fields in a structure are modeled as having the same location; thus, a write into one field writes to all the fields in the structure. Heintze and Tardieu experimented with another analysis which they termed *field-based*. Each field in each structure type is represented by a variable. Any pointer stored into a field of a particular type of structure is modeled as storing into the variable representing the field of type. They showed some preliminary empirical results indicating that the field-based approach runs much faster than the field-independent approach. Notice that since C is not type-safe, a field-based approach may not generate a conservative answer for some C programs.

   There is yet another variant on how fields can be handled. We say that a points-to analysis is *field-sensitive* if it tracks the individual fields of individual pointers. That is, a store to a field via a pointer $p$ only affects the objects pointed to by $p$ and only in that field. A field-sensitive analysis is strictly more precise than either a field-based or a field-independent analysis. In Java, the more precise the points-to analysis results are, the smaller the call graph gets. In other words, a less precise analysis may produce a much larger call graph, which would end up making the algorithm much slower. Since Java is strictly typed, the field information can be fully exploited without compromising the soundness of the algorithm. Our algorithm is field-sensitive, and our experimental results show that this added precision greatly reduces the running time of the algorithm.

2. **Flow sensitivity within a method.** Our algorithm analyzes local pointers in a method flow-sensitively. This is especially important for Java because, aside from the primitive types, all local variables in Java are pointers to heap objects. At the bytecode level, the same stack locations are used to point to different objects within the same procedure.

   The first step of our two-step algorithm converts the bytecode for each method into a sparse representation which we call a *summary graph*. A summary graph captures the semantics of the operations in a method; it records the calls made but not the semantics of the methods that are called. A summary graph contains:

   – the data accessed, expressed in terms of access paths based off the parameters and global variables,
   – the weak updates made in the procedure, and
   – an outgoing parameter map showing all the calls made and the parameters passed at each call site.

   We use a flow-sensitive data-flow algorithm to generate this representation so as to model the updates on local variables precisely. The interprocedural effects are combined in the second step of the algorithm flow-insensitively.

3. **Optimizations to the CLA algorithm.** The second step of the algorithm applies an extended CLA algorithm on the summary graphs representing the methods to compute the interprocedural points-to information. The algorithm starts by analyzing the `main` method and analyzes only the methods that are reachable from `main`. It iterates over the summaries and the points-to relations until a fixpoint is reached. We introduce two optimizations to speed up the computation: we reuse the cache of intermediate results between iterations to minimize the storage management overhead, and we minimize the application of the expensive set union operation.

This paper presents empirical results demonstrating the efficiency of the proposed technique. Our algorithm successfully analyzes over a million lines of Java source code on the order of minutes, using only 512MB of heap space. We apply the points-to results to the problem of concrete type inferencing. The results indicate that the call graph generated is very precise: for the same call sites, it is about twice as precise as rapid type analysis. Many other applications are also made possible with a fast inclusion-based points-to analysis.

## 2 Sparse Intra-Method Summaries

Rather than operate on the program source directly, our algorithm operates on a sparse "summary" representation generated from the source. This has a number of advantages. First, it reduces the amount of code that the interprocedural algorithm must deal with, thus improving the speed and the scalability of the analysis. Second, because each method is small, we can afford to use a flow-sensitive technique to generate the method summary. This allows the flow-insensitive interprocedural algorithm to take advantage of some local flow sensitivity, which is essential for accurately keeping track of updates to local variables.

A method summary is a flow-insensitive summary that encodes all stores (writes) performed by the method, and all calls made by the method, along with their parameters. The summary also includes information about return values, and thrown and caught exceptions. For simplicity, we do not describe constants, exceptions, type casts, or invocations of native methods here; our implementation correctly handles these features, however.

### 2.1 Program Representation

Let us start with an example of a method summary. As shown in Figure 1 is a summary graph for an `add` method for a linked list. The implicit *this* parameter and the *o* parameter are represented by the nodes marked *this* and *o*, respectively. The dashed edges are access path edges, marked with field names. The solid edges are write edges, which record the possible updates that the method can make. The node marked List is a concretely-typed node that represents the object created at the `new` statement. The call to `equals` is also recorded in the summary graph, with dotted links to the nodes that represent each of the parameters.

```
class List {
  Object data; List next;
  boolean add(Object o) {
    List p = this;
    for (;;) {
      if (o.equals(p.data)) return false;
      if (p.next == null) break;
      p = p.next;
    }
    p.next = new List(); p.next.data = o;
    return true;
  }
}
```

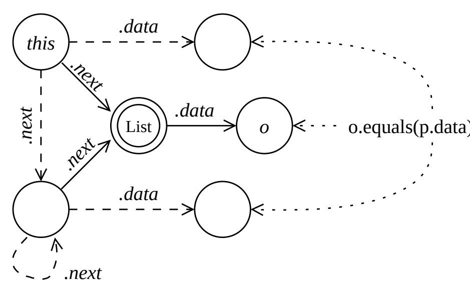



**Fig. 1.** Example code and summary graph.

A program is represented as follows. There is a set $\mathtt{l} \in L$ of local variables and a global variable $g$. The analysis models static variables as fields of the global object. Together, the local variables and the global variable form the set $v \in V = L \cup \{g\}$ of variables.

There is a set of class types $cl \in CL$, which refer to the classes in the program. There is also a set $f \in F$ of object fields, which are accessed using syntax of the form $v.f$. Static variables are accessed using syntax of the form $g.f$. There is also a set $op \in OP$ of methods. Each method has a formal parameter list $p_0,...,p_k$. Parameter $p_0$ refers to the receiver object.

Each method is represented by a control flow graph. The nodes are statements $st \in ST$. All statements are in one of the following forms:

- A copy statement $\mathtt{l} = v$
- An object creation statement $\mathtt{l} = \mathit{new}\ \mathit{cl}$
- A load statement $\mathtt{l}_1 = \mathtt{l}_2.f$
- A store statement $\mathtt{l}_1.f = \mathtt{l}_2$
- A method invocation statement of the form $\mathtt{l} = \mathtt{l}_0.op(\mathtt{l}_1,...,\mathtt{l}_k)$
- A return statement `return l`
- An enter statement $\mathtt{enter}_{op}$
- An exit statement $\mathtt{exit}_{op}$

The control flow graph for each method *op* starts with an enter statement $\mathtt{enter}_{op}$ before any other statements, and ends with an exit statement $\mathtt{exit}_{op}$ after any other statements.

### 2.2 Object Representation

The analysis represents the objects that the program manipulates using a set $n \in N$ of nodes. Nodes fall into six types:

- Set of concretely-typed nodes $N_{\mathrm{C}}$. A concretely-typed node represents an object that is known to have a concrete, well-known type. There is one concretely-typed node $n_{\mathrm{C}}$ for each object creation statement in the program. Each concretely-typed node has an associated type $cl \in CL$, denoted by $\mathrm{cl}(n_{\mathrm{C}})$.
- Set of parameter nodes $N_{\mathrm{P}}$. There is one parameter node $n_{\mathrm{P}}$ for each formal parameter $p_k$ in the program; it represents the object passed in as that parameter into the method currently being analyzed. $\mathrm{k}(n_{\mathrm{P}})$ refers to the index of the parameter referred to by the parameter node $n_{\mathrm{P}}$.
- Set of return value nodes $N_{\mathrm{RV}}$. There is one return value node $n_{\mathrm{RV}}$ for each method invocation statement *st* in the program; it signifies the objects that can be returned by the invoked method.
- Set of local return value nodes $N_{\mathrm{LRV}}$. There is one local return value node $n_{\mathrm{LRV}}$ for each method *op* in the program; it signifies the values that can be returned by that method.
- Global node, $G$, which holds references to static variables.
- Set of load nodes $N_{\mathrm{L}}$, which represent entries in access paths off of other nodes. Each load node $n_{\mathrm{L}}$ has an associated field $\mathrm{f}(n_{\mathrm{L}}) \in F$, a set of base nodes $\mathrm{base}(n_{\mathrm{L}}) \subseteq N$, and an associated load statement $\mathrm{st}(n_{\mathrm{L}}) \in ST$.

The analysis treats an array as a node with a special field *elements*. The *elements* field represents all of the elements of the array. Note that a single node can correspond to multiple objects. Likewise, due to aliasing, multiple nodes may correspond to a single object at run time. Because we only perform weak updates to pointer variables and because dereferences are not resolved until the interprocedural phase, the effect is a safe, conservative approximation of the effect of the method in all possible contexts.

### 2.3 Summary Graphs

A summary graph consists of a set of access path (read) edges $E_{\mathrm{R}} \subseteq N \times F \times N_{\mathrm{L}}$, a set of write edges $E_{\mathrm{W}} \subseteq (V \cup N) \times F \times N$, and an outgoing parameter map $P \subseteq N \times \mathcal{N} \times ST$, where $\mathcal{N}$ represents the set of natural numbers.

Read edges are created by load statements and they refer to references that may have been created outside of the current method. Write edges are created by store statements and refer to references created by this method. Following a sequence of edges from a parameter or global node to a load node gives us the access path to that load node. The outgoing parameter map is a mapping between nodes and method call parameters. It records which nodes are passed as which parameters to the called methods.

We denote the edges in set $S$ from a node $n$ as $\mathrm{edgesFrom}(S, n)$. The source of an edge $e$ is $\mathrm{from}(e)$, and the target of an edge $e$ is $\mathrm{to}(e)$. We denote the targets of read edges from node $n$ on field $f$ as $E_{\mathrm{R}}(n, f)$. Likewise, the targets of write edges from node $n$ on field $f$ are denoted as $E_{\mathrm{W}}(n, f)$. The union of $E_{\mathrm{R}}$ and $E_{\mathrm{W}}$ is $E$. We denote an edge from node $n_1$ to node $n_2$ on field $f$ as $n_1 \xrightarrow{f} n_2$; accesses of scalar variables are unlabeled. For a method call statement $st$ and parameter number $j$, we denote the set of nodes that can be passed as that parameter to that method call statement as $P(st, j)$. We denote an edge from the node $n$, representing the $j$th actual parameter of the call statement $st$ as $n \xrightarrow{j} st$.

### 2.4 Data-Flow Analysis

The algorithm uses a data-flow analysis to generate a summary graph at every point in the method. Initially, there are no read edges or outgoing parameters, and, there are implicit write edges between the global variable $g$ and the global node $G$, and between the first $n$ local variables and the parameters, where $n$ is the number of parameters. That is, for a method with formal parameters $p_0, ..., p_n$, the initial summary graph for the method has read edges $E_{\mathrm{R}} = \emptyset$, write edges $E_{\mathrm{W}} = \{g \rightarrow G\} \cup \{\langle \mathtt{l}_k \rightarrow n_{p_k} \rangle \mid n_{p_k} \in N_{\mathrm{P}}\}$, and outgoing parameter map $P = \emptyset$.

Once the algorithm has generated the initial summary graph, it propagates summary graphs through the statements of the method's control flow graph, applying transfer functions and computing joins, until the result stabilizes. We use a standard work-list solver to solve the data flow. The final result is the computed summary graph after the exit statement, $\mathtt{exit}_{op}$.

A transfer function $\langle E'_{\mathrm{R}}, E'_{\mathrm{W}}, P' \rangle = [st](\langle E_{\mathrm{R}}, E_{\mathrm{W}}, P \rangle)$ models the effect of a statement $st$ on the summary graph. The following gives the transfer function for each statement; unless specified otherwise, we assume:

$E'_{\mathrm{R}} = E_{\mathrm{R}} \quad E'_{\mathrm{W}} = E_{\mathrm{W}} \quad P' = P$

**Copy statements.** A copy statement of the form $\mathtt{l} = v$ makes $\mathtt{l}$ point to the object that $v$ points to. The transfer function updates $E_{\mathrm{W}}$ to reflect this change by killing the current set of edges from $\mathtt{l}$, then generating write edges from $\mathtt{l}$ to all of the nodes that $v$ points to.

$$\begin{aligned} \mathrm{Kill}_{E_W} &= \mathrm{edgesFrom}(E_W, \mathtt{l}) \\ \mathrm{Gen}_{E_W} &= \{\mathtt{l} \rightarrow n \mid n \in E_W(v)\} \\ E'_W &= (E_W - \mathrm{Kill}_{E_W}) \cup \mathrm{Gen}_{E_W} \end{aligned}$$

**Object creation statements.** An object creation statement $\mathtt{l} = \mathit{new}\ \mathit{cl}$ allocates a new object of type $\mathit{cl}$ and makes $\mathtt{l}$ point to the object. All objects that are allocated at a specific creation statement are represented by a single, concretely-typed node $n_C$. The transfer function kills all edges from $\mathtt{l}$, then generates a write edge from $\mathtt{l}$ to $n_C$.

$$\begin{aligned} \mathrm{Kill}_{E_W} &= \mathrm{edgesFrom}(E_W, \mathtt{l}) \\ \mathrm{Gen}_{E_W} &= \{\mathtt{l} \rightarrow n_C\} \\ E'_W &= (E_W - \mathrm{Kill}_{E_W}) \cup \mathrm{Gen}_{E_W} \end{aligned}$$

**Load statements.** A load statement $\mathit{st}$ of the form $\mathtt{l}_1 = \mathtt{l}_2.f$ makes $\mathtt{l}_1$ point to the object that $\mathtt{l}_2.f$ points to. A load node represents the abstract location accessed by $\mathtt{l}_2.f$. For each node $n_2$ pointed to by $\mathtt{l}_2$, we create a load node $n_L$ and add an access path (read) edge $n_2 \xrightarrow{f} n_L$.

We must be careful when creating load nodes on cyclic access paths to avoid infinite expansion of cyclic access paths. For each node $n_2$, we compute the transitive closure of predecessor base nodes, $\mathrm{bases}(n_2)$. If a node in $\mathrm{bases}(n_2)$ comes from the same load statement, we use that node rather than creating a new node.

$$\begin{aligned} n_L(n_2) &= n' \text{ where } n' \in \mathrm{bases}(n_2), \mathrm{st}(n') = \mathit{st}, \text{ if } n' \text{ exists} \\ &\quad\ n'' \text{ where } n'' \in N_L \text{ is a fresh node, otherwise.} \end{aligned}$$

$$\begin{aligned} \mathrm{Gen}_{E_R} &= \{n_2 \xrightarrow{f} n_L(n_2) \mid n_2 \in E_W(\mathtt{l}_2)\} \\ \mathrm{Kill}_{E_W} &= \mathrm{edgesFrom}(E_R, \mathtt{l}_1) \\ \mathrm{Gen}_{E_W} &= \{\mathtt{l}_1 \rightarrow n_L(n_2) \mid n_2 \in E_W(\mathtt{l}_2)\} \\ E'_W &= (E_W - \mathrm{Kill}_{E_W}) \cup \mathrm{Gen}_{E_W} \\ E'_R &= E_R \cup \mathrm{Gen}_{E_R} \end{aligned}$$

**Store statements.** A store statement of the form $\mathtt{l}_1.f = \mathtt{l}_2$ makes the $f$ field of the object referred to by $\mathtt{l}_1$ refer to the same object referred to by $\mathtt{l}_2$. The analysis models this by finding the set of nodes that $\mathtt{l}_1$ points to, then generating write edges from all of these nodes to the nodes that $\mathtt{l}_2$ points to.

$$\begin{aligned} \mathrm{Gen}_{E_W} &= \{n_1 \xrightarrow{f} n_2 \mid n_1 \in E_W(\mathtt{l}_1), n_2 \in E_W(\mathtt{l}_2)\} \\ E'_W &= E_W \cup \mathrm{Gen}_{E_W} \end{aligned}$$

**Method invocation statements.** Method invocation statements invoke a method with the given parameters and return a value, which is stored into the local variable $\mathtt{l}$. To handle return values, every method invocation statement has a return value node $n_{RV}$, which signifies the possible return values of the method invocation. The transfer function kills all edges from $\mathtt{l}$, then generates a write edge from $\mathtt{l}$ to $n_{RV}$.

Our intra-method analysis also records the sets of nodes that are passed in as each parameter. This information is later used in the inter-method analysis phase. For a method invocation statement $\mathit{st}$ of the form $\mathtt{l} = \mathtt{l}_0.\mathit{op}(\mathtt{l}_1,\ldots,\mathtt{l}_k)$, we record the set of nodes passed into the $j$th parameter, $0 \leq j \leq k$.

$$\begin{aligned}
\mathrm{Gen}_P &= \{n \xrightarrow{j} st \mid n \in E_{\mathrm{W}}(\mathtt{l}_j)\} \\
P' &= P \cup \mathrm{Gen}_P \\
\mathrm{Kill}_{E_{\mathrm{W}}} &= \mathrm{edgesFrom}(E_{\mathrm{W}}, \mathtt{l}) \\
\mathrm{Gen}_{E_{\mathrm{W}}} &= \{\mathtt{l} \rightarrow n_{\mathrm{RV}}\} \\
E'_{\mathrm{W}} &= (E_{\mathrm{W}} - \mathrm{Kill}_{E_{\mathrm{W}}}) \cup \mathrm{Gen}_{E_{\mathrm{W}}}
\end{aligned}$$

**Return statements.** A return statement `return l` returns the object pointed to by `l` to the caller method. We model this in the transfer function by generating write edges from the local return value node $n_{\mathrm{LRV}}$ to the nodes pointed to by `l`.

$$\begin{aligned}
\mathrm{Gen}_{E_{\mathrm{W}}} &= \{n_{\mathrm{LRV}} \rightarrow n \mid n \in E_{\mathrm{W}}(\mathtt{l})\} \\
E'_{\mathrm{W}} &= E_{\mathrm{W}} \cup \mathrm{Gen}_{E_{\mathrm{W}}}
\end{aligned}$$

**Exit statements.** The exit statement, $\mathtt{exit}_{op}$, removes all write edges from local variables $\mathtt{l} \in L$ and the global variable $g$, as they are no longer relevant outside of the context.

$$\begin{aligned}
\mathrm{Kill}_{E_{\mathrm{W}}} &= \textstyle\bigcup_{v \in V} \mathrm{edgesFrom}(E_{\mathrm{W}}, v) \\
E'_{\mathrm{W}} &= E_{\mathrm{W}} - \mathrm{Kill}_{E_{\mathrm{W}}}
\end{aligned}$$

**Control-flow join points.** The confluence operator $\sqcup$ for points-to graphs at control-flow joins is:

$$\bigsqcup(\langle E_{\mathrm{R}_1}, E_{\mathrm{W}_1}, P_1\rangle, \langle E_{\mathrm{R}_2}, E_{\mathrm{W}_2}, P_2\rangle) = \langle E_{\mathrm{R}_1} \cup E_{\mathrm{R}_2}, E_{\mathrm{W}_1} \cup E_{\mathrm{W}_2}, P_1 \cup P_2\rangle$$

## 3 Inclusion-based Points-to Analysis

Our inclusion-based points-to analysis algorithm is essentially the algorithm for C programs due to Heintze and Tardieu, which is based on Andersen's points-to analysis. We have extended their algorithm to handle modern language features such as object fields, strict typing, virtual methods, type casts, and exceptions. We have also optimized the algorithm to use type information, reuse caches between iterations, and to reduce the number of expensive set union operations.

### 3.1 Review: Andersen's Points-to Analysis

An inclusion-based analysis, such as Andersen's algorithm, models points-to information as inclusion relations. That is, an assignment such as $x = y$ implies the constraint that $x$ points to all the objects pointed to by $y$, written $x \supseteq y$. The assignment is uni-directional: information only flows from $y$ to $x$, and not vice-versa. Heintze and Tardieu showed that Andersen's $\mathrm{O}(n^3)$ algorithm could be implemented efficiently for C programs using a graph-based algorithm with caching and online cycle detection. Our analysis and formulation are based on the presentation by Heintze and Tardieu, which was originally based on a presentation style by McAllester.

Consider a simple language $L$ where all expressions take the form:

$$e \ ::= \ x \ \mid \ x.f \ \mid \ new_x$$

$$\frac{x \Longrightarrow new_y}{new_y.f \Longrightarrow e} \quad (\text{if } x.f = e \text{ in } Q) \quad [\textsc{Store}]$$

$$\frac{x \Longrightarrow new_y}{e \Longrightarrow new_y.f} \quad (\text{if } e = x.f \text{ in } Q) \quad [\textsc{Load}]$$

$$\frac{}{e_1 \Longrightarrow e_2} \quad (\text{if } e_1 = e_2 \text{ in } Q) \quad [\textsc{Copy}]$$

$$\frac{e_1 \Longrightarrow e_2, e_2 \Longrightarrow e_3}{e_1 \Longrightarrow e_3} \quad [\textsc{Trans}]$$

**Fig. 2.** Deduction rules for Andersen's analysis on program $Q$

Programs are sequences of assignments of the form $e_1 = e_2$, where $e_1$ cannot be of the form $new_x$. $new_x$ corresponds to the object returned at object creation site $x$. Given some program $Q$, we use the deduction rules in Figure 2 as the constraints of the system. Conceptually, an inclusion edge $\Longrightarrow$ refers to an *inclusion relationship* between lvalues; that is, $x \Longrightarrow y$ represents the relation $x \supseteq y$.

The STORE rule specifies that if a store instruction of the form $x.f = e$ exists in the program and there is an inclusion edge from $x$ to $new_y$, then add an inclusion edge from $new_y.f$ to $e$. The LOAD rule specifies that if a load instruction of the form $e = x.f$ exists in the program and there is an inclusion edge from $x$ to $new_y$, then add an inclusion edge from $e$ to $new_y.f$. The COPY rule adds an inclusion edge from $e_1$ to $e_2$ for all assignments $e_1 = e_2$ in the program. The TRANS rule is transitive closure.

### 3.2 Graph-based Andersen's analysis

As a flow-insensitive and context-insensitive analysis, Andersen's algorithm is imprecise and the resulting points-to sets of each location can potentially be very large. Another problem is that computing the full transitive closure of the graph is very expensive. Most of the work on speeding up Andersen's analysis goes into avoiding explicitly representing all points-to sets in the program, and avoiding computing the full transitive closure of the graph. Our algorithm is based on the pre-transitive graph algorithm by Heintze and Tardieu, which gathers its savings from intelligent caching and online cycle detection and collapsing.

Our algorithm operates on the sparse method summaries described in Section 2. It builds up a set of inclusion edges between nodes in that representation. Like in Heintze and Tardieu's algorithm, we keep the inclusion edge graph in pre-transitive form. The read and write edges in the sparse method summaries correspond to the LOAD and STORE statements in language $L$, respectively. To capture the interprocedural effects, as the targets of the method invocations become resolved, our algorithm creates the equivalent of COPY statements that copy the actual parameters to the corresponding formal parameters, and from the returned values to the variables receiving the result.

The source and destination nodes in the summaries can represent *abstract locations*, for example, parameter nodes, load nodes, etc. We need to find the

```
main(){
    add summaries of root methods to methodSet
    iterationCount = 1
    do {
        noChange = true // set to false if any set changes
        foreach method summary ms in methodSet do
            foreach (write) edge e in E_W with field f in ms do
                foreach n in getConcreteNodes(from(e)) do
                    if n contains field f
                        add write edge n.f → to(e) to E_W
            foreach (read) edge e in E_R with field f in ms do
                foreach n in getConcreteNodes(from(e)) do
                    add inclusion edge to(e) ⟹ E_W(n, f)
            foreach method call st in ms
                S = { }
                foreach n in getConcreteNodes(P(st, 0)) do
                    S = S ∪ targets(st, cl(n))
                foreach method op in S do
                    add method summary of op, ms′, to methodSet
                    addCallMappings(ms, st, ms′)
        iterationCount++
    } until noChange
}
addCallMappings(caller, st, callee) {
    if already mapped (st, callee) return
    foreach parameter node n_P in callee do
        foreach node n in P(st, k(n_P)) in caller do
            add inclusion edge n_P ⟹ n
    foreach node n in E_W(n_LRV) in callee do
        add inclusion edge n_RV(st) ⟹ n
}
```

**Fig. 3.** Main algorithm

concrete nodes that these abstract nodes refer to in order to find what locations the write edges (store statements) could be writing to, and likewise what locations that read edges (load statements) could be reading from. Similarly, method calls may have abstract nodes as their receiver objects: we need to find the corresponding concrete nodes to find the possible class types and thereby the possible target methods of the method call.

We use the inclusion edge graph to find these concrete nodes. However, because we keep the graph in pre-transitive form, we need to perform a graph reachability computation whenever we want to know the reachable concrete nodes. The *getConcreteNodes* method, described in Section 3.3, implements this graph reachability computation.

Figure 3 presents the main algorithm. We control iteration through a global *noChange* flag. This flag is set to false if any edges are added in each iteration of the algorithm. (Note that edges are only added and therefore sets can only grow. The number of nodes is finite and so the algorithm is guaranteed to converge.) We have a set of method summaries that the algorithm iterates over; this set is initialized to the root set of the program and contains the summaries of the methods in the reachable call graph at any given point in the algorithm. We also keep track of the iteration number; this is used in versioning of the caches, as described in Section 3.3.

For each method summary, we visit every write edge, every read edge, and every method invocation. For each write edge, we find the concrete nodes that correspond to the source node of the edge. Then, we add write edges on the same field from each of the concrete nodes to the target of the edge. However, we do not add write edges from concrete nodes that do not have the named field. This is one example where having a strictly-typed language allows us to refine the points-to information.

Java contains many virtual method invocations that depend on the type of the receiver object. We use the points-to information and the strict type information to find the possible targets of virtual method invocations and bind their parameters and return values. For each method call in the summary, we build up a set of the class types of the concrete nodes of the receiver object. We use that set to find the possible target methods of that method invocation statement and add the target methods to the method set. The function *targets*(*st*, *cl*) returns the set of possible methods invoked by statement *st* in the case the receiver node is of class *cl*. The *addCallMappings* function is used to insert inclusion edges due to method invocations. *addCallMappings* skips call site and target pairs that have already been mapped. It adds inclusion edges from each formal parameter node in the callee to the actual parameters from the parameter map in the caller. It also adds inclusion edges from the return value node of that method invocation in the caller to the returned nodes of the callee.

### 3.3 Reachability Computation

We adopted the online cycle detection mechanism of Heintze and Tardieu's algorithm in our reachability computation on the inclusion edge graph, as shown in Figure 4. *getConcreteNodes* is a wrapper function for *getConcreteNodes**, which takes two arguments (a node to explore from and a list of nodes that defines the path being explored) and returns two results (the set of concrete nodes that are reachable from the first argument and a boolean value stating whether or not the set of nodes has changed since the last iteration). *getConcreteNodes* passes **null** as the path to the recursive *getConcreteNodes** function and returns the set of reachable concrete nodes to its caller.

To detect cycles in the path, each node has a boolean flag called *onPath* to mark if the node is already on the path being explored. When a cycle is detected, *getConcreteNodes** unifies the nodes in the cycle. The function *unify* unifies two nodes. Node unification is implemented by an optional *skip* field for each node. Two nodes $n_1$ and $n_2$ are unified by setting $n_1.skip$ to $n_2$, and adding all outgoing inclusion edges from $n_1$ to $n_2$. Subsequently, whenever node $n_1$ is

```
getConcreteNodes(n) {
    (r, changed) = getConcreteNodes*(n, null)
    return r
}
getConcreteNodes*(n, path) {
    if (n ∈ N_C) return ({n}, false)
    if (n.onPath) { // cycle detected
        while (car(path) ≠ n)
            unify(car(path), n); path = cdr(path)
        remove cyclic edges from n
        return ({ }, false)
    } else {
        (r, changed, ncount) = cache.get(n)
        if (r == null) // does not exist in cache
            r = { }
        else if (ncount == iterationCount)
            return (result, changed)
        path = cons(n, path)
        n.onPath = true; changed = false
        foreach inclusion edge n → n' do
            (r_2, changed_2) = getConcreteNodes*(n', path)
            if (changed_2 || n → n' never visited)
                r = r ∪ r_2; if r grows, changed = true
        n.onPath = false
        cache.put(n, r, changed, iterationCount)
        return(r, changed)
    }
}
```

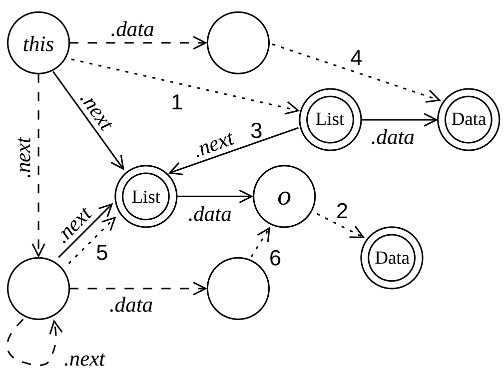
**Fig. 4.** *getConcreteNodes*: graph reachability

accessed, we follow its *skip* pointer. After unifying a set of nodes, we eliminate transitive self-edges. This method of cycle elimination is very low cost, yet it finds and collapses all cycles in the parts of the graph that we traverse.

The results of the calls to *getConcreteNodes** are cached for efficiency. The reachability of a node is guaranteed to be computed accurately only if it is the first such request in an iteration of the main algorithm. The same iteration is likely to need the reachability information of the same node over and over again, and the true answer may change during the iteration as more edges are added. For efficiency, *getConcreteNodes** returns the cached result, if one has been computed for the current iteration, and relies on future iterations to propagate the correct answer.

Heintze and Tardieu's algorithm also caches the results, but deletes them after every iteration. We noticed from our experiments that many of the cached results did not change between iterations and that a significant amount of time was spent reallocating and rebuilding these caches on every iteration. Therefore, we keep the cache entries across iterations, but store along with each entry the iteration number in which it was last computed and whether the last recomputation changed its value. If the iteration number on the entry is not current, its value is recomputed. The function $cache.get(n)$ returns the triple (concrete node set, changed flag, iteration number). Likewise, *cache.put* updates the cache. We use soft references for the cache entries so that old entries can be flushed when memory is low.

Set union operations are expensive. We minimize the number of set union operations performed by skipping the $r = r \cup r_2$ step whenever we can determine that $r_2$ has not changed since the last computation of $r$. More specifically, we check if $r_2$'s value in the cache is current, i.e. it has not changed since the last iteration, and that we have visited that edge before. We cache the results, along with the updated iteration number and whether the set has changed.




Here is the result after one iteration of our algorithm operating on the summary graph from Figure 1, when the `add` method is called on a single-element list. The three rightmost concrete nodes correspond to the single-element list and the element to add. Inclusion edges are marked as dotted edges. Edges are added by the algorithm in numerical order. Edges 1 and 2 are from parameter mapping, edge 3 is from the callee write edge, and edges 4-6 are from callee read edges.

**Fig. 5.** Example of interprocedural algorithm on the graph from Figure 1.

## 4 Experimental Results

The inclusion-based points-to analysis described above was implemented in the joeq system[12] in about 800 lines of source; the implementation is available as part of the open-source joeq project. The experiments were performed on a PC with a 2GHz Pentium 4 and 2 GB of memory, running Redhat Linux 7.2 and Java HotSpot VM build 1.3.1_01. We ran all tests with a heap size of 512MB, even though only some of the benchmarks ended up needing that much memory.

Figure 6 shows the results of our analysis on a variety of benchmarks. The first nine benchmarks are from the standard SpecJVM v1.03 benchmark suite. The next six benchmarks are components of the reference implementation of Java 2 Enterprise Edition version 1.3 (J2EE). With over 900,000 lines of code,

| Benchmark | Classes | Methods | Byte-codes | Calls | Ave. Call Targets RTA | Points-to | Time (s) Opt | No opt | Space (MB) | Itera-tions |
|---|---|---|---|---|---|---|---|---|---|---|
| check | 202 | 1342 | 112K | 9428 | 1.79 | 1.02 | 14 | 47 | 18 | 22 |
| compress | 92 | 304 | 27K | 1408 | 2.32 | 1.01 | 5 | 49 | 3 | 16 |
| db | 95 | 360 | 30K | 1730 | 2.47 | 1.01 | 5 | 52 | 4 | 13 |
| jack | 139 | 573 | 57K | 3468 | 2.05 | 1.01 | 3 | 59 | 1 | 15 |
| javac | 262 | 1015 | 89K | 5955 | 2.58 | 1.17 | 13 | 115 | 14 | 18 |
| jess | 306 | 1654 | 129K | 10723 | 1.85 | 1.02 | 16 | 69 | 27 | 22 |
| mpegaudio | 243 | 1529 | 158K | 9852 | 1.79 | 1.02 | 16 | 61 | 25 | 22 |
| mtrt | 84 | 254 | 22K | 1141 | 3.00 | 1.01 | 4 | 45 | 2 | 15 |
| raytrace | 84 | 254 | 22K | 1140 | 3.00 | 1.01 | 4 | 45 | 2 | 15 |
| admintool | 283 | 1563 | 108K | 6628 | 2.73 | 1.22 | 19 | 155 | 28 | 18 |
| appclient | 847 | 5037 | 342K | 26888 | 2.29 | 1.18 | 318 | 960 | 285 | 29 |
| deploytool | 1231 | 8194 | 532K | 40156 | 2.38 | 1.43 | 744 | 1788 | 507 | 33 |
| j2eeserver | 1021 | 6039 | 413K | 32564 | 2.43 | 1.39 | 516 | 2479 | 494 | 34 |
| packager | 355 | 1869 | 143K | 10748 | 2.04 | 1.05 | 28 | 481 | 26 | 26 |
| verifier | 753 | 4750 | 336K | 25810 | 1.99 | 1.07 | 74 | 1101 | 76 | 24 |
| joeq | 1229 | 9890 | 450K | 40151 | 2.63 | 1.34 | 136 | 372 | 193 | 12 |
| cloudscape | 859 | 8166 | 483K | 35493 | 1.95 | 1.03 | 211 | 612 | 70 | 22 |
| jedit | 1371 | 10365 | 691K | 51656 | 2.36 | 1.30 | 614 | 1953 | 472 | 24 |

**Fig. 6.** Analysis results on the benchmarks

J2EE is a sizable framework for building Java Enterprise applications. joeq is the joeq virtual machine, upon which the analysis was implemented. It has about 75,000 lines of code. Cloudscape is a database server that is shipped with J2EE. The source code is not available. jEdit, containing over 100,000 lines of code, is a full-featured editor written in Java.

The first four columns of numbers in the table specify the number of classes, methods, bytecodes, and call sites that were analyzed. Our algorithm analyzes only those methods reached through the root set, including code in the class libraries; the numbers reflect this reachable code. We found that the algorithm was successful in greatly reducing the amount of code that need to be dealt with.

The next two columns give the average number of targets per call site when using a basic RTA analysis [2] and when using our analysis. Our analysis is effective in reducing the number of targets to almost 1, in many cases. The next two columns give the run time of our algorithm, with and without our optimizations described in Section 3.3. By reducing the number of potentially $O(n^2)$ set union operations for large points-to sets, the analysis speeds up significantly, by a factor between 2.4 and 18.3. The last two columns give the amount of heap space used for the analysis and the number of iterations taken, respectively. The algorithm uses a reasonable amount of memory and converges fairly quickly. Our algorithm analyzed over 1200 classes, 8000 methods, and 500KB of byte codes in 12 minutes, in Java nonetheless.

We also ran our analysis in a field-independent manner (i.e. by ignoring all field names.) When in field-independent mode, the run times skyrocketed. Compress, one of the smallest benchmarks in the suite, took more than two hours to analyze. We did not attempt any of the larger benchmarks. This shows that field sensitivity is essential when running the algorithm on Java code.

## 5 Related Work

We have adapted and extended Heintze and Tardieu's Andersen-style points-to analysis designed originally for C programs to work on Java programs. The most significant difference is that our algorithm is field-sensitive. By tracking the different fields in an object separately, our algorithm is both faster and more precise than a field-independent technique. In addition, our algorithm is locally flow-sensitive, allowing local pointers to be destructively updated. Our algorithm handles all of the Java constructs which include virtual method invocations, type casts, exceptions, unknown and escaping pointers. Finally, we have made a couple of practical optimizations to improve the algorithm's efficiency.

Among previous work that extends Andersen's algorithm to Java[6, 7, 9], the algorithm by Rountev et al. appears to be the fastest and the most precise[7]. Like our algorithm, their technique is field-sensitive. They adapted and extended a constraints-based algorithm[4, 10] also originally designed for C to Java. Of all the experimental results reported, `javac` took the longest to run, requiring 6 minutes on a 360-Mhz Sun Ultra60 machine with 512 MB of memory. Our algorithm is more precise because it is locally flow-sensitive. By using a CLA-based technique, our algorithm is also faster, taking 12 seconds on `javac` on a 2GHz PC. We have also reported results on much larger programs. Liang et al. experimented with a number of different context-insensitive and flow-insensitive techniques based on Steensgaard's and Andersen's algorithms[6]. Like Heintze and Tardieu, they experimented with a field-independent and a field-based analysis. They observed context insensitivity does not provide very accurate results when applied to collections and maps. They suggested representing these objects with special models that allow some degree of context sensitivity. They recommended the use of a field-based Andersen algorithm as the best tradeoff between time and precision. In view of our results, there is no reason not to use a more precise field-sensitive technique.

## 6 Conclusion

We have presented an efficient inclusion-based points-to analysis for Java, based on the Heintze and Tardieu's CLA algorithm originally designed for C. Our algorithm handles all the Java language features including inheritance, object fields, exceptional control flow, type casting, dynamic dispatch, and reflection.

We have extended the CLA algorithm in two ways to improve its precision. First, our analysis is field-sensitive. Fields in an object are treated distinctly from each other. Second, our analysis handles local variables in a method flow-sensitively; this is important because locations on the stack are often reused in Java bytecodes to hold different objects. Our algorithm uses a data-flow analysis to compute a summary graph for each program point in the program. Only weak updates are allowed on non-local variables; the summary graph at the exit of a method is a flow-insensitive representation of the local effect of a method. By improving the algorithm's precision, these two techniques also improve its efficiency due to the smaller call graphs produced.

We have also proposed two techniques to improve the efficiency of the CLA algorithm. We reuse the data in the cache to minimize the memory management overhead, and we avoid expensive set union operations if the operands have not changed. This paper presents empirical results demonstrating that our algorithm is scalable to extremely large code bases without sacrificing accuracy.

## References


1. L. Andersen. *A Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, 1994.
2. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 324–341, 1996.
3. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*, pages 77–101, 1995.
4. M. Fahndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1998.
5. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 146–161, 2001.
6. D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 73–79, 2001.
7. A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java based on annotated constraints. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 43–55, 2001.
8. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.
9. M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, University of Passau, Sept. 2000.
10. Z. Su, M. Fahndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 81–95, 2000.
11. V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 264–280, 2000.
12. J. Whaley. joeq virtual machine. http://joeq.sourceforge.net, 2001.