Refinement-Based Context-Sensitive Points-To Analysis for Java

Manu Sridharan
University of California, Berkeley
manu_s@cs.berkeley.edu

Rastislav Bodík
University of California, Berkeley
bodik@cs.berkeley.edu

Abstract

We present a scalable and precise context-sensitive points-to analysis with three key properties: (1) filtering out of *unrealizable paths*, (2) a *context-sensitive heap abstraction*, and (3) a *context-sensitive call graph*. Previous work [21] has shown that all three properties are important for precisely analyzing large programs, *e.g.*, to show safety of downcasts. Existing analyses typically give up one or more of the properties for scalability.

We have developed a *refinement-based* analysis that succeeds by simultaneously refining handling of method calls and heap accesses, allowing the analysis to precisely analyze important code while entirely skipping irrelevant code. The analysis is *demand-driven* and *client-driven*, facilitating refinement specific to each queried variable and increasing scalability. In our experimental evaluation, our analysis proved the safety of 61% more casts than one of the most precise existing analyses across a suite of large benchmarks. The analysis checked the casts in under 13 minutes per benchmark (taking less than 1 second per query) and required only 35MB of memory, far less than previous approaches.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization

General Terms Algorithms, Languages, Performance

Keywords Refinement, context-sensitive analysis, points-to analysis, demand-driven analysis

1. Introduction

Many applications require a precise points-to analysis, but precision is often sacrificed for the sake of scalability. In static verification, imprecise points-to information generates many false positives; consequently, verifiers often ignore the effects of pointers (*e.g.*, [13]), forfeiting soundness. Automatic refactoring tools [10, 40] currently cannot reason precisely about the heap, and hence cannot soundly perform transformations such as moving instance fields between classes (to the best of our knowledge). Finally, certain IDE program understanding aids, such as object model visualizations of complex object-oriented libraries [26], would be improved by precise points-to information.

This paper presents a scalable *context-sensitive* points-to analysis. Previous work [21, 26] has shown context sensitivity to be key to computing precise points-to information for Java. Our analysis is context sensitive in three ways:

- Excludes unrealizable paths: Calls and returns are matched on interprocedural paths [31].
- Uses a context-sensitive heap abstraction: Objects allocated by the same statement in different calling contexts are distinguished.
- Constructs a context-sensitive call graph: Targets of virtual calls are computed separately for each calling context.

As shown here and in previous work [21], all three properties are essential for precise results on large programs. For example, an analysis cannot distinguish the contents of different instances of the `java.util.Vector` data structure for realistic programs without these properties. (The properties are discussed further in Section 4.) In existing context-sensitive analyses, one or more of these properties is often sacrificed for scalability. Here we present an analysis that retains all three properties while scaling to large programs and requiring far less memory than previous approaches.

We have developed a *refinement-based* approach to points-to analysis that succeeds by simultaneously refining the two key axes of precision for Java points-to analysis: handling of method calls, and handling of heap accesses, *i.e.*, reads and writes to object fields. The refinement is driven by the heap accesses; we precisely model those accesses deemed more important for a precise analysis result, while using an approximation technique for other accesses that allows for skipping inspection of much of the program. As refinement proceeds, more field accesses are modeled precisely, and hence more of the program is analyzed. Calls and returns are only modeled precisely in those parts of the program not skipped due to approximate handling of fields; refinement of field accesses exposes more calls and returns for analysis, and hence *automatically* yields refinement of context sensitivity.

Our refinement technique gauges precision using a *client-driven* approach [12], *i.e.*, based on whether the analysis client is satisfied with the result. At each stage of refinement, the client is queried to see if the current analysis result is sufficiently precise. For example, for a client aiming to prove that a downcast of variable `x` to type `T` cannot fail, a result showing that `x` can only point to `T` objects suffices. The client-driven approach allows our analysis to avoid performing excessive refinement, aiding scalability.

A key element of the success of our approach is its *demand-driven* style of analysis. A demand-driven points-to analysis [14, 37] performs only the work necessary to answer a *query*, *i.e.*, a request for a variable's points-to information from an analysis client. Through the demand-driven approach, our refinement can determine which heap accesses are key to precision for each query separately, as the statements relevant to each query are easily tracked. Furthermore, being demand-driven allows our analysis to employ expensive analysis techniques, *e.g.*, a context-sensitive heap abstraction and call graph, in a limited fashion. Note that demand-driven analysis alone does not yield scalability, as in practice too

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

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006...$5.00.

much code must still be analyzed precisely; our evaluation shows that it is the combination of refinement and demand-driven analysis that yields scalability.

Our approach is based on insights gained through a *context-free language reachability* (CFL-reachability) [29] formulation of the analysis, which makes the interplay between heap accesses and method calls clear. In previous work [37], we showed that context-insensitive points-to analysis for Java is a *balanced parentheses* CFL-reachability problem. We base the current technique on this balanced-parentheses structure, using it to (1) compute an approximate analysis with various levels of refinement and (2) help determine which code requires more precise handling.

Contributions. This paper makes the following contributions:

- We observe that *balanced parentheses* in the CFL-reachability formulation of context-insensitive Java points-to analysis [37] can be used to guide refinement of both heap access and method call handling.
- We develop a demand-driven, client-driven algorithm based on this insight that refines handling of heap accesses and method calls simultaneously and skips analysis of irrelevant code.
- We present an evaluation of our algorithm with three clients and several large benchmarks. The evaluation showed that our technique is both precise, proving 61% more casts safe than one of the best existing analysis, and scalable, checking casts in under 13 minutes per benchmark and only requiring 35MB of memory, an order of magnitude less than previous approaches.

Section 2 gives a high-level overview of our technique. Section 3 presents our CFL-reachability formulation of context-insensitive points-to analysis for Java [37], and Section 4 defines our context-sensitive formulation. Section 5 delineates our algorithm and gives a detailed example. We evaluate our technique in Section 6. Section 7 discusses related work, and Section 8 concludes.

2. Overview

This section provides an overview of our analysis technique. Section 2.1 defines the *field-sensitive*, *context-sensitive* points-to analysis problem; in the limit (*i.e.*, after full refinement), our analysis solves this problem precisely. Section 2.2 gives a formulation of this points-to analysis in the context-free language reachability framework [29], simplified to make clear the essential property of balanced parentheses. Section 2.3 presents our algorithm for a simplified version of the reachability problem, showing how we exploit its balanced-parentheses structure for refinement. Finally, Section 2.4 illustrates how the analysis works on a Java code example. We show that in typical programs our refinement explores nested data structures in a hierarchical progression, visiting only a small part of the program to obtain sufficient precision for the client.

2.1 Analysis Definition

A *points-to analysis* computes a points-to relation *pt* that maps each pointer variable to a superset of the objects that it may point to during execution. The possibly unbounded heap is modeled by a finite *heap abstraction*. We aim to compute a *control-flow insensitive* points-to analysis, *i.e.*, an analysis that treats each method as if its control-flow graph contained all possible edges. A flow-insensitive analysis typically abstracts the heap by representing all objects created at a given allocation site (*i.e.*, a `new` expression) with a single *abstract location*. So, if variable x might point to an object created at allocation site i modeled by abstract location o_i, then $o_i \in pt(x)$. Since we compute a demand-driven analysis, the work for computing $pt(x)$ is only performed when a client issues a query for x's points-to set.

We define our analysis problem as computing the *best possible* (*i.e.*, most precise) flow-insensitive points-to information given certain restrictions on the analyzed program. Namely, the analysis assumes an input program with (1) no arrays and (2) no recursive method calls, but otherwise makes no restrictions on its structure.[1] We handle the excluded program features by using approximations: arrays are discussed in Sections 2.4 and 3, and recursion in Section 4.3.

Computing the best possible flow-insensitive points-to information requires handling assignments in a *subset-based, field-sensitive* manner. A subset-based analysis models assignments with subset constraints, *e.g.*, statement `x = y` induces the constraint $pt(y) \subseteq pt(x)$. A field-sensitive analysis precisely handles the semantics of field accesses. For example, consider computing $pt(z)$ for the following program:

```
x = new Obj(); y = new Obj();
x.f = new Obj(); // o1
y.f = new Obj(); // o2
z = x.f;
```

To find $pt(z)$, the analysis must conservatively determine the possible values of the expression `x.f`. A *field-based analysis* treats each instance field as a global variable, in this case reasoning that any object written to the `f` field can be read from `x.f`, yielding $pt(z) = \{o_1, o_2\}$. In contrast, a field-sensitive analysis reasons separately about the instance fields of each abstract object. In this case, since `x` and `y` cannot be aliased, the field-sensitive analysis more precisely concludes that $pt(z) = \{o_1\}$.

A precise points-to analysis must handle method calls in a *context-sensitive* manner. A context-sensitive analysis yields results as precise as if it were computed on a modified program with all method calls inlined. Note that this definition of context sensitivity affects the heap abstraction, as inlining creates copies of allocation statements; some previous analyses did not use a context-sensitive heap abstraction (*e.g.*, [9, 43]), increasing scalability at the cost of precision. Also, potential virtual call targets must be computed context sensitively, *i.e.*, separately for each calling context, and in an on-the-fly manner, *i.e.*, using precise points-to information to compute the targets.

Note that our analysis algorithm is only fully precise in the limit, *i.e.*, with full refinement. Our algorithm often achieves sufficient precision for the client without treating all code precisely. While it is capable of providing the precision described above, the key to its scalability is that even for demanding clients, full precision is often not necessary.

2.2 CFL-Reachability Formulation

Our technique is based on a formulation of points-to analysis as a context-free language reachability (CFL-reachability) problem [29], an extension of standard graph reachability that allows for filtering of uninteresting paths. Let G be a directed graph with edge labels taken from alphabet Σ, and let L be a context-free language over Σ. Each path p in G is labeled with a string $s(p)$ in Σ^*, obtained by concatenating edge labels in order. We say p is an L-path if $s(p) \in L$. Given nodes s and t, the *single-source/single-target L-path problem* asks if G contains an L-path from s to t; multiple-source or multiple-target problems are similarly defined. Hence, L characterizes the paths of interest when determining CFL-reachability. When an L-path exists from s to t,

[1] We also assume no native methods or reflection in the formulation, but they are handled by our analysis (see Section 6).

we say t is L-*reachable* from s, or simply, $s\ L\ t$; we similarly refer to S-paths and use notation $s\ S\ t$ for any non-terminal S in L's grammar.

For points-to analysis, G represents the program: its nodes model variables and abstract locations, and its edges model different types of assignments. L describes paths in G corresponding to program executions that *might* cause a variable to point to some abstract location; other paths are guaranteed not to affect the points-to relation. We define L such that if x may point to o, then $o\ L\ x$. Perhaps counter-intuitively, the L-path goes from o to x rather than from x to o, since our edges are oriented in the direction of value flow, *i.e.*, from the right-hand side of an assignment to the left-hand side. Hence, computing the points-to set of a variable x is a single-target L-path problem [29], requiring backwards reachability.

The points-to analysis of Section 2.1 can be expressed in CFL-reachability with language $L_{\mathsf{F}} \cap R_{\mathsf{C}}$, where L_{F} ensures precise handling of field accesses and R_{C} ensures context sensitivity. Our previous work [37] showed that precise handling of Java heap accesses can be formulated with L_{F} being a language of *balanced parentheses*. Intuitively, field accesses are balanced since Java instance fields cannot be accessed through lower-level pointer operations like C's dereference (`*`) and address-of (`&`) operators. So, if an object o_2 is written into $o_1.f$ through a field write "`x.f = y`", o_2 can only be read back from o_1 through some corresponding field read statement "`w = z.f`" (where z points to o_1). We model the flow of o_2 through the two statements in the reachability formulation with a pair of parenthesis symbols.

While the balanced-parentheses property of L_{F} is a recent result, context-sensitive analysis is well known to be a balanced-parentheses problem in CFL-reachability [28, 31]. In the graph, edges entering and exiting a method call are labeled with open and close parentheses specific to the call site. A path in G with mismatched call parentheses corresponds to an unrealizable control flow path [31], and we can use a balanced-parentheses language to filter out such paths.

In this section, we consider a CFL-reachability formulation of points-to analysis that is simplified in three (unsound) ways: (1) it ignores simple copy assignments (*e.g.*, `x = y`), (2) it does not properly handle paths between field parentheses (*cf.* Section 3), and (3) it does not allow partially balanced call parentheses (*cf.* Section 4.2). We simplify here to make the essential properties of the formulation clear; the full sound formulation is given in Sections 3 and 4. Let Σ_P be the alphabet of open and close brackets, respectively representing heap writes and reads, and open and close parentheses, representing method call entries and exits:

$$\Sigma_P = \{[_f\ ,\ ]_f \mid f \text{ is a field}\} \cup \{(_i\ ,\ )_i \mid i \text{ is a call site}\}$$

We compute reachability with language $L_{\mathsf{scf}} = L_{\mathsf{sf}} \cap R_{\mathsf{sc}}$ (s for simplified) over Σ_P, with L_{sf} and R_{sc} respectively representing key properties of L_{F} and R_{C}:

$$L_{\mathsf{sf}} : F \rightarrow [_f\ F\ ]_f \mid [_g\ F\ ]_g \mid \ldots \mid F\ F \mid (_i\ \mid\ )_i \mid \ldots \mid \epsilon$$
$$R_{\mathsf{sc}} : C \rightarrow (_i\ C\ )_i \mid (_j\ C\ )_j \mid \ldots \mid C\ C \mid [_f\ \mid\ ]_f \mid \ldots \mid \epsilon$$

Both L_{sf} and R_{sc} allow the parentheses of the other language to be mixed anywhere in their strings. Note that although it has a balanced-parentheses grammar, R_{sc} is regular, since we assume recursion-free programs and hence the number of open call parentheses in a string is bounded.

The worst-case time and space complexity of L_{scf}-reachability is exponential, motivating our refinement-based technique. CFL-reachability problems can be solved by a general algorithm in $O(\Gamma^3 N^3)$ time [29], where Γ is the size of L and N is the number of nodes in G. For our problem, $|R_{\mathsf{sc}}|$ (and hence, Γ) is exponential in the size of the program, since its size is proportional to the number of paths in the program's call graph. Demand-driven analysis has the same worst-case bound; we found it not to scale in practice, leading us to investigate refinement.

2.3 Refinement Algorithm

Here we present our refinement-based algorithm for solving a simplified version of a L_{scf}-reachability problem, as defined in Section 2.2. In particular, we focus on showing that a node x is *not* L_{scf}-reachable from a node y; our refinement algorithm is designed to quickly prove such unreachability properties. The key idea is to focus effort on parts of the graph likely to have unbalanced parentheses, handling the rest of the graph approximately (in fact by skipping over it entirely). We refine by approximating for less of the graph, eventually yielding a precise answer.

Simplified Problem. To focus on the key ideas of our technique, we consider the following additional simplification of the single-source/single-target L_{scf}-reachability problem (beyond using the simplified L_{sf} and R_{sc} languages): given a single path p from node o to node x with edge labels chosen from Σ_P (defined in Section 2.2), the analysis must determine if p is a L_{scf}-path, *i.e.*, if $o\ L_{\mathsf{scf}}\ x$.

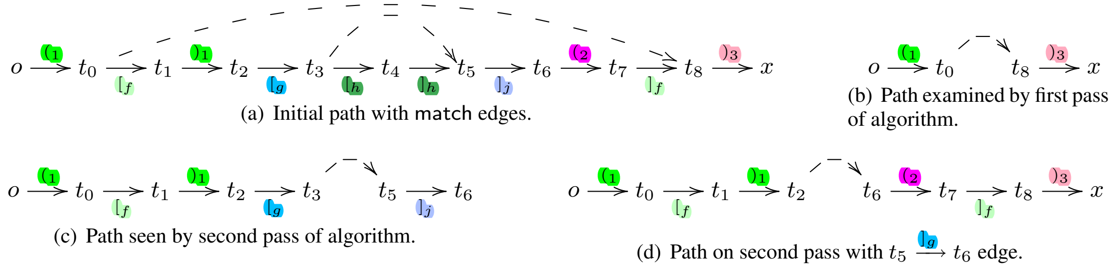
To model how refinement improves performance, we add the following optimality constraint to the problem: if p is not an L_{scf}-path, the algorithm should determine this fact while minimizing the number of edges inspected. As a trivial example, if the first edge e of p is a closed parenthesis, the algorithm should conclude that p is not an L_{scf}-path by inspecting only e. This constraint allows us to illustrate how our refinement technique can quickly show that some nodes in an approximate points-to set for a variable x are in fact unreachable from x, increasing precision. Note that our algorithm does not necessarily visit the minimal number of edges, but it often does less work than the straightforward technique of traversing p directly.

All of the ideas described here for the simplified problem generalize to points-to analysis for arbitrary programs. In the general reachability problem, we are given a variable x, and then must find all o such that $o\ L_{\mathsf{scf}}\ x$. Furthermore, for each o, we must consider all paths from o to x, not just one. For an acyclic graph, both of these generalizations can be viewed as solving multiple instances of the simplified problem. We address cyclic graphs when presenting the full algorithm in Section 5.

Figure 1(a) gives an example input path for our simplified problem (the dashed edges will be explained shortly). The path is not an L_{scf}-path, as the $[_g$ and $]_j$ and $(_2$ and $)_3$ parentheses are mismatched. We will illustrate how our analysis can discover this fact without inspecting the entire path.

Our Approach. Our technique tries to minimize work through *approximation* and *refinement*. For the simplified problem, an *approximate analysis* must answer correctly when p is an L_{scf}-path, but can answer incorrectly when it is not; for points-to analysis, this approach yields an over-approximation of the points-to relation. *Refinement* gradually removes the imprecision of this analysis, eventually yielding the correct answer when p is not an L_{scf}-path.

The key idea behind our technique is to focus analysis effort on parts of p where parentheses are likely to be unbalanced. A path p may have many unbalanced parentheses (*i.e.*, open parentheses without a balancing close parenthesis or vice-versa), but proving the existence of just one such parenthesis is sufficient to show that p is not an L_{scf}-path. In particular, for an edge e with label $[_f$ *not* deemed likely to be unbalanced, the analysis approximates by assuming that (1) some $]_f$ edge e' balances e, and that (2) the path between e and e' is from L_{scf}. These assumptions allow the analysis to skip inspection of the edges between e and e', focusing on other parts of p likely to contain unbalanced parentheses.



(a) Initial path with match edges.

(b) Path examined by first pass of algorithm.

(c) Path seen by second pass of algorithm.

(d) Path on second pass with $t_5 \xrightarrow{[_g} t_6$ edge.

Figure 1. Paths to illustrate the behavior of our refinement algorithm.

Refinement-based L_{sf}-reachability. We first consider computing L_{sf}-reachability using refinement, ignoring method call parentheses for now (*i.e.*, the R_{sc} language). Our algorithm selectively skips subpaths of p using so called match *edges*. A match edge, shown as a dashed edge in Figure 1(a), connects the source of some $[_f$ edge to the sink of any $]_f$ edge; initially, all possible match edges are added to the graph. (We assume appropriate data structures exist so that match edges can be added without inspecting the entire path.) When traversing p to check if it is an L_{sf}-path, following a match edge from node s to node t corresponds to assuming $s\ L_{\mathsf{sf}}\ t$, in order to focus effort elsewhere. Note that using match edges to handle accesses of field f is equivalent to a field-based handling of f (see Section 2.1) [37].

Our analysis approximates L_{sf}-reachability by computing reachability over a language L_{sfr} (r for refinement) that includes match edges:

$$L_{\mathsf{sfr}} : T \rightarrow [_f\ T\ ]_f \mid [_g\ T\ ]_g \mid \ \ldots \ \mid \mathsf{match} \mid T\ T \mid (_i \mid)_i \mid \ \ldots \ \mid \epsilon$$

L_{sfr} is identical to L_{sf} except for the additional match production. Thus, L_{sfr} is a superset of L_{sf}, and computing L_{sfr}-reachability approximates L_{sf}-reachability.

The refining L_{sfr}-reachability algorithm uses match edges to skip subpaths of p whenever possible. Given edges $t \xrightarrow{[_f} u$, $v \xrightarrow{]_f} w$, and $t \xrightarrow{\mathsf{match}} w$, our analysis *always* assumes $t\ L_{\mathsf{sfr}}\ w$, and uses the match edge to avoid inspecting the subpath from u to v. If p contains multiple $]_f$ edges, t will have multiple outgoing match edges, and the analysis must try to use each one before concluding that p is not an L_{sf}-path.

Figure 1(b) shows the L_{sfr}-path discovered by our analysis when given Figure 1(a) as input. The analysis is able to skip much of the original path using the $t_0 \xrightarrow{\mathsf{match}} t_8$ edge. However, x is not L_{sf}-reachable from o, so the analysis result with L_{sfr}-reachability is incorrect (but sound), necessitating refinement.

Refinement of the approximation of L_{sfr}-reachability is accomplished by removing match edges from the graph, forcing checking of more parentheses on the original path. Given edge $t \xrightarrow{[_f} u$, if outgoing match edges from t are removed, our algorithm must check for a path labeled $[_f\ T\ ]_f$ from t, possibly reducing the approximation caused by the match edges. Figure 1(c) shows the subpath of Figure 1(a) explored after removing the $t_0 \xrightarrow{\mathsf{match}} t_8$ edge. This removal exposes the unbalanced parenthesis $[_g$, leading the analysis to conclude that p is not an L_{sf}-path. Note that the unbalanced parenthesis is discovered without analyzing the whole path (the $t_3 \rightsquigarrow t_5$ subpath was skipped).

Refinement-based ($L_{\mathsf{sf}} \cap R_{\mathsf{sc}}$)-reachability. We now discuss how our analysis computes L_{scf}-reachability (recall $L_{\mathsf{scf}} = L_{\mathsf{sf}} \cap R_{\mathsf{sc}}$), which requires also checking for balanced method call parentheses. First, notice that this checking must be approximated when using match edges: since a match edge can skip over an arbitrary sequence of method call edges, call parentheses are only checked on subpaths with no match edges. Consider the path of Figure 1(b). While it may be tempting to conclude that $(_1$ and $)_3$ are mismatched, $(_1$ is in fact balanced on the original path. The analysis handles match edges by assuming that any possible sequence of call parentheses may appear on the skipped subpath. For Figure 1(b), the analysis assumes that $)_1(_3$ may have been skipped by the match edge, and hence approximately answers that p is an L_{scf}-path.

Removal of match edges allows for simultaneous refinement of method call and field parentheses handling, as it exposes more of both of them for checking. For instance, if we balance the field parentheses in Figure 1(a) by changing the label of the $t_5 \rightarrow t_6$ edge to $]_g$, we can still use refinement to find the unbalanced method call parenthesis $(_2$ without inspecting the whole path. While the first pass of our analysis would still discover the path of Figure 1(b) after this modification, the second pass would find the path in Figure 1(d), as the graph would now have a $t_2 \xrightarrow{\mathsf{match}} t_6$ edge. Thus, the removal of the $t_0 \xrightarrow{\mathsf{match}} t_8$ edge exposes the mismatched $(_2$ and $)_3$ parentheses, allowing the analysis to conclude that x is not L_{scf}-reachable from o. Again, the path was shown to be unbalanced without inspecting all of its edges, illustrating the performance benefits of refinement.

2.4 Refinement on Java programs

Section 2.3 showed how our algorithm is able to save work by skipping inspection of certain subpaths, with the ability to refine to a precise answer. Here, we show what these skipped paths correspond to in typical programs, *i.e.*, we show what code is typically analyzed at a particular approximation level, and what code is not visited at all. In particular, the analysis typically only applies full sensitivity for classes whose object contents must be distinguished to precisely answer a query, *e.g.*, to find the contents of a particular `Vector` object. We will show that both field and context sensitivity are necessary for sufficiently precise results, and that encapsulation allows us to analyze only a small amount of code precisely.

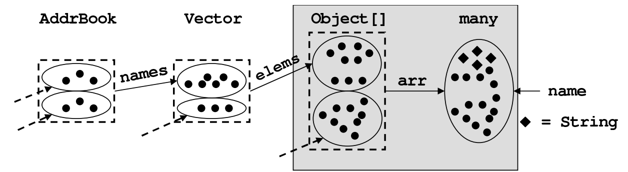
We use the example in Figure 2, a partial implementation of an `AddrBook` class with a `Vector` of names, to illustrate the effects of refinement. We consider an analysis client that aims to statically prove that downcasts cannot fail, in particular the downcast to `String` at line 27. To prove this cast safe with points-to analysis, it suffices to show that the `name` variable can only point to `String` objects.

Figures 3(a) through 3(c) give an abstract view of the analysis result at each refinement stage while computing pt(`name`). Each graph shows how the analysis computes points-to sets of the fields read to obtain the value assigned to `name`: `AddrBook.names` at line 23, `Vector.elems` at line 10, and finally `arr` (a pseudo-field for modeling array accesses) at line 11. Ovals in the graphs enclose points-to sets; pt(`name`) is shown on the right. A dashed arrow indicates a pointer from some class not shown in Figure 2. Figure 3(c)

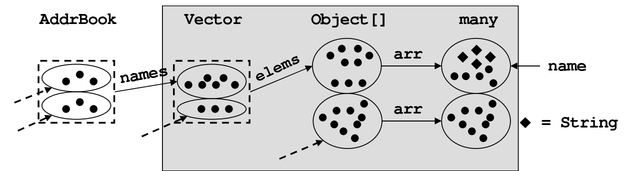
t is not declared :-(

```
1  class Vector {
2    Object[] elems; int count;
3    Vector() { t = new Object[10];
4               this.elems = t; }
5    void add(Object p) {
6      t = this.elems;
7      t[count++] = p; // writes t.arr
8    }
9    Object get(int ind) {
10     t = this.elems;
11     return t[ind]; // reads t.arr
12   } ...
13 }
14 class AddrBook {
15   private Vector names;
16   AddrBook() { t = new Vector();
17                this.names = t; }
18   void addEntry(String n, ...) {
19     t = this.names; ...;
20     t.add(n);
21   }
22   void update() {
23     t = this.names;
24     for (int i = 0; i < t.size(); i++) {
25       Object name = t.get(i);
26       // is this cast safe?
27       String nameStr = (String)name;
28       ...
29     }
30   }
31 }
32 void useVec() {
33   Vector v = new Vector();
34   Integer i1 = new Integer();
35   v.add(i1);
36   Integer i2 = (Integer)v.get(0);
37 }
```

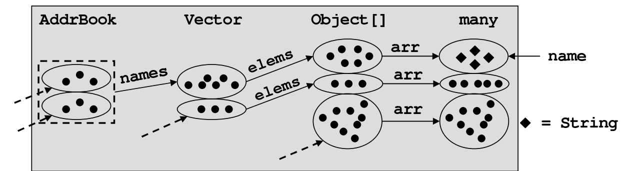
Figure 2. Example code for illustrating our algorithm.



(a) Initial analysis result.



(b) Result after distinguishing `Object[]` contents.



(c) Result after also distinguishing `Vector` contents.

Figure 3. Analysis result at different stages of approximation for proving safety of the cast at line 27 of Figure 2.

shows that after two passes of refinement, the analysis proves that pt(`name`) contains only `String` objects, shown as diamonds.

The initial analysis result, shown in Figure 3(a), is imprecise due to `match` edges, which cause merging of field contents for all fields. In the graphs of Figure 3, a dashed box indicates that field contents for objects inside the box have been merged due to `match` edges. For example, Figure 3(a) indicates that the `arr` fields for arrays from `Vector.elems` as well as those from some other data structure have been merged; fields `elems` and `names` are similarly collapsed. `match` edges cause merging since the analysis uses them to jump from a field read to all writes of the field, ignoring which object's field is being accessed. When computing pt(`name`), the analysis finds that `name` gets its value from a read of the `arr` field at line 11 of Figure 2. Through `match` edges, the analysis then concludes that any object written into `arr` can flow to `name`, essentially merging the contents of all arrays.

While imprecise, the initial analysis skips inspection of field accesses deeper than those of `arr`, thereby finishing quickly and allowing time for more precise analysis. The gray shading in the graphs of Figure 3 indicates which of the shown field dereferences are inspected by that pass of the analysis. In Figure 3(a), only accesses of the `arr` field are inspected, while fields `elems` and `names` are skipped. This occurs because when the analysis reaches the array read at line 11 of Figure 2, it can jump immediately to the array write at line 7 using a `match` edge, and to all other array writes (not shown) using other `match` edges. Through this skipping, the analysis saves times by avoiding analysis of other code in `Vector` and `AddrBook` and code that uses `AddrBook` objects.

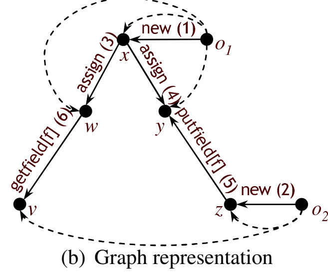
Our analysis refines by removing all `match` edges for all fields of some class `T`, with the goal of distinguishing the contents of different instances of `T`. For the example, we remove `match` edges for accesses to `arr` after the first pass, yielding the result in Figure 3(b). The dashed box around `Object` arrays has disappeared, and the analysis now distinguishes the `arr` field of arrays stored in `Vector.elems` (*i.e.*, the internal arrays of `Vector`s), from other arrays in the program. The `elems` field is still merged across `Vector`s because of `match` edges; hence, the analysis concludes that `name` can point to any object stored in the internal array of any `Vector`, still too imprecise a result for the cast-checking client. However, the `match` edges on `elems` allow the analysis to skip inspection of accesses to `names` and code that uses `AddrBook`s, again saving time and allowing for more refinement.

In its third pass, our analysis succeeds in showing safety of the downcast by removing `match` edges on `elems`, which exposes calls to the methods of `Vector` for context-sensitive handling. With `match` edges on `elems` present, the analysis would exit `Vector`'s methods on a `match` edge (*e.g.*, from the read of `elems` at line 10 in `get()` to the write at line 4 in `Vector()`), skipping an unknown sequence of calls and returns and hence forcing approximation of context sensitivity. Context sensitivity for calls to these methods is required to distinguish contents of different `Vector` instances (see Section 4 for details). After removing `match` edges on `elems`, the analysis yields the result in Figure 3(c), showing that `name` only gets its value from a `Vector` stored in `AddrBook.names`; since such `Vector`s only contain `String`s, this is sufficient to show the downcast at line 27 cannot fail, and the analysis terminates.

Our refinement technique exploits encapsulation in object-oriented code for better performance. In Figure 2, the `names` field is *encapsulated*, *i.e.*, the field and the `Vector` it points to cannot be directly accessed outside the `AddrBook` class; similarly, `Vector` encapsulates its `elems` field. When fields are encapsulated, `match`

```
1   x = new Obj(); // o1
2   z = new Obj(); // o2
3   w = x;
4   y = x;
5   y.f = z;
6   v = w.f;
```

(a) Code example



(b) Graph representation

Figure 4. A small code example and its graph representation for CFL-reachability-based points-to analysis. Line numbers from (a) are given on corresponding edges in (b). Dashed edges in (b) indicate the existence of a *flowsTo*-path from the source to the sink.

edges for those fields can only connect accesses in the same class, limiting the scope of our analysis. For example, with match edges present for `elems` in Figure 3(b), the analysis processed code in `Vector`, but not code in `AddrBook` that uses a `Vector`. Furthermore, encapsulated fields allow the refinement to explore data structures in a hierarchical progression: in the example, we explore the array, then the `Vector` pointing to the array, and finally the `AddrBook` pointing to the `Vector`. Note that our analysis exploits encapsulation without any code annotations, instead discovering encapsulation automatically. When fields are not encapsulated, our analysis can still provide precise results, but it may run slower, as more code needs to be analyzed with full precision.

3. Context-Insensitive Points-To Analysis

In this section, we formulate field-sensitive, context-insensitive points-to analysis for Java in CFL-reachability, adapted from [37]. Section 2.2 presented a simplified language L_{sf} for this problem; here we present a language L_{F} that differs primarily by soundly handling paths between parentheses. We first describe how to construct a graph G representing the pointer-manipulating statements of a program P. Then, we define L_{F} such that computing L_{F}-reachability over G yields the desired points-to analysis. We extend the formulation to context-sensitive points-to analysis in Section 4.

Program representation. We first describe how given a program P, the graph G for points-to analysis is constructed. Nodes in G represent variables and abstract locations, while edges represent four canonical assignment forms: (1) allocation statements `x = new T()`, (2) copy statements `x = y`, (3) heap reads `x = y.f`, and (4) heap writes `x.f = y`. We represent these statement types in G with the following edges:

$$
\begin{aligned}
\texttt{x = new T()} &\implies o \xrightarrow{\mathsf{new}} x \\
\texttt{x = y} &\implies y \xrightarrow{\mathsf{assign/assignglobal}} x \\
\texttt{x = y.f} &\implies y \xrightarrow{\mathsf{getfield}[f]} x \\
\texttt{x.f = y} &\implies y \xrightarrow{\mathsf{putfield}[f]} x
\end{aligned}
$$

Note that all edges are oriented in the direction of value flow. We use the assignglobal label for copy assignments where either `x` or `y` is a global variable (*i.e.*, a static field), and the assign label otherwise; both are treated as assign edges here, but the context-sensitive analysis handles assignglobal edges specially (see Section 5). The source of a new edge is the corresponding abstract location node. For getfield[f] and putfield[f] edges, the field f is part of the edge label. Loads and stores to array elements are modeled by collapsing all array elements into a field *arr*; for example, `x.a[i]=y` is translated to `tmp=x.a; tmp.arr=y;`. Figure 4(b) gives G for the program of Figure 4(a).

We model method calls in our graph to handle interprocedural flow. For each method m, G has a node for each of m's formal parameters and a special ret_m node for m's `return` statements. At call site i of m, we add param[i] edges from each actual parameter to the appropriate formal parameter and a return[i] edge from the ret_m node to the appropriate caller's variable. In this subsection, which presents context-insensitive analysis, param[i] and return[i] edges are treated as assign edges, allowing unrealizable paths; the context-sensitive analysis of Section 4.2 filters out such paths.

To soundly handle virtual calls, the analysis must determine possible virtual call targets using a conservative call graph [11]. An inexpensive analysis, typically based on declared types (*e.g.*, [4]), can be used to compute an approximation of virtual call targets prior to points-to analysis. However, this approach can cause significant imprecision in the subsequent points-to analysis [20, 21, 26]. A more precise solution is to compute targets of virtual calls *on the fly*, *i.e.*, as the relevant points-to information is computed; for a context-sensitive analysis, this technique yields a context-sensitive call graph. We elide our formulation of on-the-fly call graph construction in CFL-reachability for lack of space. Our analysis computes an on-the-fly call graph by raising recursive points-to queries for virtual calls, as described in Section 5. For clarity's sake, the remainder of this section and Section 4 are presented assuming the existence of some prior conservative call graph.

The Language L_{F}. We now define the language L_{F} used to compute context-insensitive points-to analysis. Recall from Section 2.2 that we want to define L_{F} such that x is L_{F}-reachable from o iff $o \in pt(x)$. We first consider programs without field accesses, corresponding to graphs restricted to new and assign edges. For such graphs, L_{F} is defined by the grammar below (*flowsTo* is the start non-terminal):

$$
\mathit{flowsTo} \rightarrow \mathsf{new}\ (\ \mathsf{assign}\ )^*
$$

Intuitively, an object can flow to a variable from an allocation site only through a new edge followed by a (possibly empty) sequence of assign statements. For example, in Figure 4(b), the path $o_1 \xrightarrow{\mathsf{new}} x \xrightarrow{\mathsf{assign}} w$ is a *flowsTo*-path witnessing $o_1 \in pt(w)$.

We now extend L_{F} to track value flow through the heap via putfield[f] and getfield[f] statements. Recall that we seek a precise (*i.e.*, field-sensitive) handling of field accesses, as described in Section 2.1. We define field-sensitivity more formally in terms of *may-aliasing*:

Definition 3.1. *Two pointers x and y are* may-aliased *iff* $pt(x) \cap pt(y) \neq \emptyset$.

Definition 3.2. *Given a heap read statement* `a = b.f` *with base pointer b, a heap write statement* `c.f = d` *to the same field* `f` *with base pointer c, and $o \in pt(d)$, a* field-sensitive *analysis concludes $o \in pt(a)$ iff b and c are may-aliased.*

To precisely track flow through the heap, we extend the *flowsTo* production:

$$
\mathit{flowsTo} \rightarrow \mathsf{new}\ \ (\ \mathsf{assign}\ \mid\ \mathsf{putfield}[f]\ \ \mathit{alias}\ \ \mathsf{getfield}[f]\ )^*
$$

This *flowsTo* production assumes the existence of an *alias* language (to be defined shortly) that captures may-aliasing: x *alias* $y \Leftrightarrow pt(x) \cap pt(y) \neq \emptyset$. The *alias* path connects the base variables of the field accesses, matching the may-aliasing in Definition 3.2.

We have now reduced the problem of defining L_{F} to defining the *alias* language. Observe that we can check for aliasing of x and y by means of *flowsTo*-paths: x and y are may-aliased iff there is an object o such that o *flowsTo* x and o *flowsTo* y, *i.e.*, $o \in pt(x) \cap pt(y)$. Unfortunately, reasoning about may-aliasing in terms of two *flowsTo*-paths is unsuitable for CFL-reachability, which can only check language membership of strings on a *single* path. Since these two *flowsTo*-paths cannot be concatenated to

form a single path from x to y, we need to extend our graph representation, as presented in [29] for the C language, to complete the CFL-reachability formulation.

To allow for *alias* paths, we extend our graph by introducing *inverse paths*. With inverse paths, the $(x\ \textit{alias}\ y)$-path can be constructed by concatenating the inverse of the $(o\ \textit{flowsTo}\ x)$-path with the $(o\ \textit{flowsTo}\ y)$-path. We invert the *flowsTo*-path using *inverse edges*: for each edge $x \to y$ in G labeled t, there is an inverse edge $y \to x$ in G labeled with $\bar{\mathsf{t}}$, following the notation of [29]. Given a path p, the inverse path $\overline{p}$ is then constructed using inverse edges in the obvious way. So, an $(x\ \textit{alias}\ y)$-path can be now defined as a path $x\ \overline{\textit{flowsTo}}\ o\ \textit{flowsTo}\ y$, for some node o. The *alias* language is defined by the following grammar:

$$\begin{array}{rcl} \textit{alias} & \to & \overline{\textit{flowsTo}}\ \textit{flowsTo} \\ \overline{\textit{flowsTo}} & \to & (\,\overline{\mathsf{assign}}\ \mid\ \overline{\mathsf{getfield}}[f]\ \ \textit{alias}\ \ \overline{\mathsf{putfield}}[f]\,)^*\ \ \overline{\mathsf{new}} \end{array}$$

Note the absence of the $\overline{\textit{alias}}$ non-terminal symbol; we use *alias* instead because the two generate the same language: $\overline{\textit{alias}} \to \overline{\overline{\textit{flowsTo}}\ \textit{flowsTo}} = \overline{\textit{flowsTo}}\ \overline{\overline{\textit{flowsTo}}} = \overline{\textit{flowsTo}}\ \textit{flowsTo} = \textit{alias}$. L_{F} is described in greater detail in [37].

As mentioned in Section 2.2, L_{F} is a language of balanced parentheses. Note that because of inverse edges, we have two pairs of matched parentheses for each field f, (putfield[f], getfield[f]) and ($\overline{\text{getfield}}$[f], $\overline{\text{putfield}}$[f]). We use these parentheses to guide our approximation and refinement, as discussed in Sections 2.3 and 5.

Example. Let us derive a *flowsTo*-path from o_2 to v in Figure 4(b). First, we derive y *alias* w using statements 1, 3, and 4.

$$\begin{array}{rl} & y\ \ \overline{\mathsf{assign}}\ \ x\ \ \overline{\mathsf{new}}\ \ o_1\ \ \mathsf{new}\ \ x\ \ \mathsf{assign}\ \ w \\ \to & y\ \ \overline{\textit{flowsTo}}\ \ o_1\ \ \textit{flowsTo}\ \ w \\ \to & y\ \ \textit{alias}\ \ w \end{array}$$

With this *alias* path, we can derive o_2 *flowsTo* v using statements 2, 5 and 6:

$$\begin{array}{rl} & o_2\ \ \mathsf{new}\ \ z\ \ \mathsf{putfield}[f]\ \ y\ \ \textit{alias}\ \ w\ \ \mathsf{getfield}[f]\ \ v \\ \to & o_2\ \ \textit{flowsTo}\ \ v \end{array}$$

□

4. Context-Sensitive Points-To Analysis

This section extends the points-to analysis of Section 3 with context-sensitivity. Section 4.1 defines context-sensitive points-to analysis and discusses its precision benefits. Section 4.2 formulates the analysis in CFL-reachability by adapting previous techniques [27, 31]; the key difference from R_{sc} in Section 2.2 is handling of partially balanced parentheses. Finally, Section 4.3 discusses how our analysis handles programs with recursion.

4.1 Context-Sensitive Analysis Problem

A context-sensitive points-to analysis precisely models calls and returns. As in Section 2.1, we assume the input program P to be recursion-free; this assumption is relaxed in Section 4.3. Let P' be the program constructed by inlining all method calls in P. The *context-sensitive flow-insensitive points-to analysis problem* is defined as computing the result of the context-insensitive analysis of Section 3 on the (call-free) program P'. Note that the actual algorithm need not work over P'; it must only compute the same points-to relation.

To satisfy context-insensitive analysis clients, we must define a method for projecting the context-sensitive analysis result for P' back to P. The points-to relation pt' computed on P' refers to copies of variables and abstract locations from P created during inlining. A context-insensitive analysis client, *e.g.*, the downcast safety checker of Section 2.4, raises queries for variables from P; we must define the context-sensitive result for variables in P for use with such clients. We denote a (local) variable in P' as $\langle v, c\rangle$, where v is the corresponding variable in P and c is the complete sequence of call sites (or *call string*) that was inlined to create the copied variable, from the root of the call graph for P to the method declaring v. The (copied) allocation sites of P' have analogously named abstract locations. Thus, pt' maps "context-refined" variables to "context-refined" abstract locations: $\langle o, c'\rangle \in pt'(\langle x, c\rangle)$. This result is projected back to P in the natural manner:

$$pt(x) \equiv \{\ o \mid \exists\, c, c'\ .\ \langle o, c'\rangle \in pt'(\langle x, c\rangle)\}$$

This definition of context-sensitive points-to analysis yields two of the three desired analysis properties discussed in Section 1. The analysis filters out unrealizable paths [31], as P' has no method calls, and hence no unrealizable paths. Analyzing P' yields a context-sensitive heap abstraction; each inlined copy of an allocation site from P is represented with its own abstract location, and hence the corresponding allocated objects are distinguished. Since (for simplicity) the definition assumes the existence of a conservative call graph (to inline methods), it does not require the construction of a context-sensitive call graph. Nonetheless, our analysis constructs such a call graph, as discussed in Sections 4.3 and 5.

To see the relevance of these properties to computing precise results, consider the downcast checking example of Section 2.4, which required distinguishing the contents of two `Vector` objects. If the analysis did not filter out unrealizable paths, the parameters and return values of the calls to `Vector.get()` at lines 25 and 36 of Figure 2 would be conflated, preventing the analysis from proving that the two calls can return distinct objects. The context-sensitive heap abstraction is required to distinguish the internal arrays of the two `Vectors`, both allocated at line 3. A context-insensitive heap abstraction represents both of these arrays with a single abstract object, and hence merges the contents of the `Vector` objects. The need for a context-sensitive call graph stems from spurious recursion in context-insensitive call graphs, and is discussed along with our handling of recursion in Section 4.3.

4.2 Context-Sensitive Analysis in CFL-Reachability

Here we show how to extend the analysis formulation from Section 3 to compute a context-sensitive points-to analysis. We will show how to answer both a projected query "is $o \in pt(x)$?" and a context-sensitive query "is $\langle o, c'\rangle \in pt'(\langle x, c\rangle)$?". We achieve context sensitivity by filtering out *flowsTo*-paths that correspond to unrealizable paths [31], without explicitly constructing P'.

To filter out unrealizable *flowsTo*-paths, we develop a regular language R_{C} that describes all realizable paths in G; *flowsTo*-paths are then filtered by checking if they are also R_{C}-paths. A *flowsTo*-path p corresponds to a realizable control-flow path iff after entering a method m from call site i, it exits from m back to call site i. Matching these call entries and exits is a well-known balanced parentheses problem [29].

Because of inverse paths, defining call entries and exits for *flowsTo*-paths is slightly tricky. In the absence of inverse paths, a realizable *flowsTo*-path traverses a method in the direction of value flow, entering through a param[i] edge and exiting through a return[i] edge. (Here, i is a unique identifier of a call site. The two edges are defined in Section 3.) However, when a realizable *flowsTo*-path p contains inverse $\overline{\textit{flowsTo}}$ subpaths, p might enter a method through a $\overline{\text{return}}$[i] edge and/or exit through a $\overline{\text{param}}$[i] edge. Hence, we define call entries and exits through the following non-terminals *callEntry*[i] and *callExit*[i]:

$$\begin{array}{rcl} \textit{callEntry}[i] & \to & \mathsf{param}[i]\ \mid\ \overline{\mathsf{return}}[i] \\ \textit{callExit}[i] & \to & \mathsf{return}[i]\ \mid\ \overline{\mathsf{param}}[i] \end{array}$$

$$S \xrightarrow{callEntry[i]} S.i \qquad S \xrightarrow{\mathsf{assignglobal}} \epsilon$$
$$\epsilon \xrightarrow{callExit[i]} \epsilon \qquad S \xrightarrow{\overline{\mathsf{assignglobal}}} \epsilon$$
$$S.i \xrightarrow{callExit[i]} S \qquad S \xrightarrow{\mathsf{match}} \epsilon$$
$$S.j \xrightarrow{callExit[i]} error \quad (i \neq j) \qquad S \xrightarrow{\overline{\mathsf{match}}} \epsilon$$

Figure 5. State transitions in the FSM for language R_{C}. The transitions for match and assignglobal edges are discussed in Section 5.

Figure 5 defines the transitions in the finite-state machine for R_{C}, where each state is a finite stack configuration corresponding to *callEntry*[i] edges (*i.e.*, the open parentheses). We discuss the match and assignglobal edge transitions in Section 5. Though it checks for balanced parentheses, R_{C} is a regular language because we assume the input program is recursion-free, and hence there are a finite number of possible stack configurations. Transitions in the FSM manipulate the stack in the usual way. The initial state is an empty stack configuration. All states except the *error* state are accept states, and all transitions not shown are self-transitions (*i.e.*, the state machine ignores edges processed by L_{F}, such as assign and putfield[f]).

R_{C} allows *partially balanced parentheses* since a realizable path need not start and end in the same procedure. For example, paths to a formal parameter are realizable, though they end with an unmatched *callEntry*[i] edge. To model this scenario, an R_{C}-path p is allowed to contain a prefix with unbalanced closed parentheses, due to the $\epsilon \xrightarrow{callExit[i]} \epsilon$ transition, and a suffix with unbalanced open parentheses, as only mismatched *callExit*[i] edges cause a transition to $error$. (This is not specific to our formulation of context-sensitive analysis [27, 29].)

R_{C} provides a context-sensitive heap abstraction by treating abstract location nodes identically to variable nodes. Since new and $\overline{\mathsf{new}}$ are not mentioned in Figure 5, R_{C} has self-transitions from all states on both symbols. Hence, the R_{C} stack is maintained at abstract location nodes, yielding a context-sensitive heap abstraction in an elegant manner.

Given R_{C}, a context-sensitive points-to analysis algorithm can be stated concisely. The answer to a projected query "is $o \in pt(x)$?" for program P, represented by graph G, is found by checking for the existence of a *flowsTo*-path p from o to x in G such that p is also an R_{C}-path. In other words, we compute CFL-reachability on G with language $L_{\mathsf{CF}} = L_{\mathsf{F}} \cap R_{\mathsf{C}}$, where L_{F} was developed in Section 3. The answer to the context-sensitive query "is $\langle o, c' \rangle \in pt(\langle x, c \rangle)$?" is obtained by modifying R_{C}: we set the initial state of the state machine for R_{C} to c and make c' the only accepting state, thereby mapping the nodes o and x to the appropriate inlined copies $\langle o, c \rangle$ and $\langle x, c' \rangle$. Note that while the call strings of Section 4.1 must start at the root of the call graph, our analysis allows queries with partial calling contexts, since they are valid R_{C} states. Since R_{C} *only* filters paths with mismatched call entries and exits, computing points-to information for program P using L_{CF}-reachability is equivalent to the projected result of computing L_{F}-reachability for fully inlined program P', as desired.

L_{CF}-reachability can be computed by tracking the state of R_{C} for each explored path while computing L_{F}-reachability; paths which cause R_{C} to reach its *error* state are excluded. This technique essentially explodes the input graph for R_{C} [31], creating a node (x, s) when node x is reached with state s of R_{C}. As $|R_{\mathsf{C}}|$ is exponential in the size of the program (due to the exponential number of paths in the call graph), this algorithm has worst-case exponential time complexity. In practice the algorithm does not scale, motivating our refinement approach.

4.3 Handling Recursion

Until this point, we have assumed that the input programs for our analysis are recursion-free; here, we discuss how we handle recursive programs. We leverage our demand-driven approach to only approximate recursive calls in the program subset relevant to a query, reducing the number of calls that must be handled imprecisely.

As context-sensitive and field-sensitive analysis for programs with recursion has been shown undecidable [30], any analysis with both properties must approximate to guarantee termination. One approximation approach is to use k-limiting, *i.e.*, tracking at most k levels of calling context [35]; however, this approach approximates more than what is necessary to achieve decidability. A less drastic approximation is to treat calls within strongly-connected components (SCCs) of the call graph as gotos [17, 43]. This approach essentially re-labels param[i] and return[i] edges within an SCC with assign, collapsing the SCC into a single method and making the program recursion-free, yielding decidability.

With a context-insensitive call graph, the SCC-collapsing approximation leads to a large precision loss. As shown in [21], for large programs a context-insensitive call graph typically has an SCC with more than 1000 methods, including methods whose handling is critical to analysis precision, *e.g.*, those of `Vector`. A more precise call graph is necessary to avoid approximate handling of all calls within this large SCC.

Our analysis only approximates handling of method calls that are recursive in a call graph computed during the demand-driven analysis. As our analysis only touches edges in G relevant to the current query, the recursive cycles it encounters are a subset of all the recursive cycles in the call graph. Consider a query for the objects possibly returned by a call to this simplified version of `Vector.elementAt()` from the Java standard library:

```
Object elementAt(int index) {
  if (index >= numElements) {
    throw new OutOfBoundsException(index + " too big"));
  }
  return elems[index];
}
```

Our analysis considers only the array access and the read of the `elems` field in the `return` statement, ignoring the calls to the `OutOfBoundsException` constructor and `String` and `StringBuffer` methods from the string concatenation. In a context-insensitive call graph that considers all control flow, these ignored calls lead to `elementAt()` being included in the aforementioned large SCC. Our analysis approximates less due to recursion since it constructs a context-sensitive call graph in the program subset relevant to each query, ignoring many potentially recursive calls.

5. The Refinement-Based Algorithm

In this section, we present the details of our refinement-based points-to analysis algorithm. In Section 2.3, we presented the algorithm for the problem of checking a single path's string for membership in a simplified version of our reachability languages. Here, we add support for determining L_{CF}-reachability in arbitrary graphs.

We use the `AddrBook` example of Figure 2 to illustrate the algorithm, again considering a client trying to prove safety of the cast of `name` to `String` at line 27, which requires computing $pt(\mathtt{name})$. Figure 6 shows the relevant part of the graph representation for the code in Figure 2. To prove the cast safe, the analysis must discover that there is no $(L_{\mathsf{F}} \cap R_{\mathsf{C}})$-path from o_{34}, an `Integer` object, to the $\mathtt{name}_{\mathtt{update}}$ node. The columns in Figure 6 indicate how the analysis explores the graph during refinement. The initial pass of the algorithm visits the left-most column, and each subsequent pass

Figure 6. Relevant portion of graph for code in Figure 2. Solid edges represent program statements, with single edges for intraprocedural statements and double edges for call entry and exit statements. Variables are subscripted with the name of the enclosing method, and line numbers in labels refer to call sites or allocation sites. The dashed edges are match edges. For space, getfield and putfield are abbreviated gf and pf.

additionally visits the next column to the right, until the safety of the cast is proven.

Approximation. As discussed in Section 2.3, we use match edges between corresponding field accesses to enable approximation. L_{F} (defined in Section 3) has two types of matched parentheses pairs, (putfield[f],getfield[f]) and ($\overline{\text{getfield}}$[f],$\overline{\text{putfield}}$[f]); we add match edges for the former type (shown as dashed edges in Figure 6, and $\overline{\text{match}}$ edges for the latter type. The match edges *define* the columns of Figure 6, as the analysis uses match edges to limit its scope in each pass. This correspondence was shown previously in Figure 3, where in each analysis pass, the gray shading (indicating analyzed code) stops at the first field whose accesses still have match edges.

Our language for computing reachability with match edges is L_{FR} (R for refinement), with start symbol *flowsToR*:

$$\begin{array}{lll} \mathit{flowsToR} & \rightarrow & \textsf{new}\ (\textsf{assign} \mid \textsf{putfield}[f]\ \mathit{aliasR}\ \textsf{getfield}[f] \\ & & \mid \textsf{match})^* \\ \mathit{aliasR} & \rightarrow & \overline{\mathit{flowsToR}}\ \mathit{flowsToR} \end{array}$$

Note that similar to L_{F}, L_{FR} treats param[i], return[i], and assignglobal edges as assign edges. Section 2.3 described how the algorithm uses match edges to "skip" checking for a valid path between matched parentheses. Here, match edges allow the analysis to skip from a putfield[f] source to a getfield[f] sink without finding a connecting *aliasR*-path. In Figure 6, the algorithm skips from $\texttt{ret}_{\texttt{get}}$ to $\texttt{p}_{\texttt{add}}$ using a match edge in its first pass, never leaving the first column of the graph. However, this leads the analysis to conclude that o_{34} is in the points-to set, necessitating refinement. Note that the analysis cannot filter out the param[35] edge in this pass, as it reaches `add()` through a match edge and hence cannot rule out call site 35.

Refinement. Selective refinement is accomplished by *removing* match edges from the graph, as in Section 2.3. Removal of match edges forces the analysis to search for more *aliasR*-paths, as it can no longer skip between the corresponding field accesses; this search can yield a more precise analysis result. In the example, the second pass of our analysis refines by removing the $\texttt{ret}_{\texttt{get}} \xleftarrow{\text{match}} \texttt{p}_{\texttt{add}}$ in Figure 6. This removal forces the analysis to handle the read and write of `arr` by checking for an *aliasR*-path from $\texttt{t}_{\texttt{get}}$ to $\texttt{t}_{\texttt{add}}$. Since match edges in the second column remain, the analysis only explores the first two columns of the graph. In this case, an *aliasR*-path is found by connecting $\overline{\mathit{flowsToR}}$-path $\texttt{t}_{\texttt{get}} \xrightarrow{\overline{\text{match}}} \texttt{t}_{\texttt{Vector}} \xrightarrow{\overline{\text{new}}} o_3$ with *flowsToR*-path $o_3 \xrightarrow{\text{new}} \texttt{t}_{\texttt{Vector}} \xrightarrow{\text{match}} \texttt{t}_{\texttt{add}}$. Again, the analysis cannot filter the path using the param[35] edge, as `add()` is reached through a match edge. So, further refinement is needed.

Context Sensitivity. As in Section 2.3, a match edge can skip over an arbitrary sequence of method calls and returns, requiring approximation of context sensitivity. As shown in Figure 5, R_{C} "forgets" calling context information when "traversing" match edges using the transition $S \xrightarrow{\text{match}} \epsilon$ and treats $\overline{\text{match}}$ analogously. We transition to ϵ since R_{C} accepts any *callExit*[i] symbol in that state. Each pass of our analysis algorithm computes $(L_{\mathsf{FR}} \cap R_{\mathsf{C}})$-reachability, using R_{C} to filter unrealizable *flowsToR*-paths as much as possible, *i.e.*, by checking that subpaths between match edges are R_{C}-paths. As previously noted, the first two passes of the analysis of the example were unable to disregard the param[35] edge, as R_{C} state was cleared across a match edge before reaching `add()`.

Removal of match edges can lead to filtering of paths because it exposes more call entry and exit edges for analysis. After the second pass, our analysis further refines field-sensitivity by removing match edges on the `elems` field, *i.e.*, the $\texttt{t}_{\texttt{Vector}} \xrightarrow{\text{match}} \texttt{t}_{\texttt{get}}$ and $\texttt{t}_{\texttt{Vector}} \xrightarrow{\text{match}} \texttt{t}_{\texttt{add}}$ edges in the second column of Figure 6. So, in the third pass the analysis must find *aliasR*-paths from $\texttt{this}_{\texttt{get}}$ to $\texttt{this}_{\texttt{Vector}}$ and from $\texttt{this}_{\texttt{Vector}}$ to $\texttt{this}_{\texttt{add}}$. Clearly, the `this` parameters of the `Vector` methods can be aliased; hence, the key to precision will be to use the state of R_{C} tracked along the *aliasR*-paths between the `this` parameters to do filtering.

The tracking of R_{C} state in the third pass of our analysis yields a result precise enough for proving the downcast safe in the example. When searching for an *aliasR*-path from $\texttt{this}_{\texttt{get}}$ to $\texttt{this}_{\texttt{Vector}}$, the analysis starts with R_{C} call stack $\langle 25 \rangle$, a call site for `Vector.get()` in Figure 2. The *aliasR*-path to $\texttt{this}_{\texttt{Vector}}$ combines the $\overline{\mathit{flowsToR}}$-path $\texttt{this}_{\texttt{get}} \xrightarrow{\overline{\text{param}[25]}} \texttt{t}_{\texttt{update}} \xrightarrow{\overline{\text{match}}} \texttt{t}_{\texttt{AddrBook}} \xrightarrow{\overline{\text{new}}} o_{16}$ and the *flowsToR*-path $o_{16} \xrightarrow{\text{new}} \texttt{t}_{\texttt{AddrBook}} \xrightarrow{\text{param}[16]} \texttt{this}_{\texttt{Vector}}$. Tracking R_{C} state along this *aliasR*-path yields call stack $\langle 16 \rangle$ at $\texttt{this}_{\texttt{Vector}}$; essentially, the analysis has concluded that the `get()` call at line 25 of Figure 2 must be on a `Vector` allocated at line 16, *i.e.*, a `Vector` used by an `AddrBook`.

Similarly, after finding an *aliasR*-path from $\texttt{this}_{\texttt{Vector}}$ to $\texttt{this}_{\texttt{add}}$, the analysis reaches $\texttt{this}_{\texttt{add}}$ with R_{C} call stack $\langle 20 \rangle$, concluding that `add()` must be entered from the call site at line 20. The analysis can then filter out the `add()` call at line 35, thereby filtering o_{34} from $pt(\texttt{name})$ and yielding the desired result, *i.e.*, that $\texttt{name}_{\texttt{update}}$ is not $(L_{\mathsf{F}} \cap R_{\mathsf{C}})$-reachable from o_{34}. Notice that the result is computed without touching the fourth column of Figure 6, illustrating how our technique can compute a sufficiently precise result while limiting analysis scope.

Refinement Policy. We have yet to specify how we choose match edges to remove after each analysis pass. We remove match edges

```
checkingAlias : Set of ((Node, Context), Node)

FINDPOINTSTO(x, c, visited)
 1  if (x, c) ∈ visited then return ∅; ADDTO(visited, {(x, c)})
 2  pointsTo ← ∅
 3  for each edge x <-new- o do ADDTO(pointsTo, {(o, c)})
 4  for each edge x <-assign- y
 5      do ADDTO(pointsTo, FINDPOINTSTO(y, c, visited))
 6  for each edge x <-assignglobal- y
 7      do ADDTO(pointsTo, FINDPOINTSTO(y, ϵ, visited))
 8  for each edge x <-getfield[f]- y
 9      do for each edge q <-putfield[f]- p
10            do if edge x <-match- p exists
11                then ADDTO(pointsTo,
                          FINDPOINTSTO(p, ϵ, visited))
12                     continue
13               if ((y, c), q) ∈ checkingAlias then continue
14               ADDTO(checkingAlias, {((y, c), q)})
15               yPointsTo ← FINDPOINTSTO(y, c, ∅)
16               yAlias ← ∅
17               for each (o, c′) ∈ yPointsTo
18                   do ADDTO(yAlias,
                          FINDFLOWSTO(o, c′, ∅))
19               for each (r, c′′) ∈ yAlias
20                   do if q = r
21                       then ADDTO(pointsTo,
                                 FINDPOINTSTO(p, c′′, visited))
22               REMOVEFROM(checkingAlias, {((y, c), q)})
23  for each edge x <-return[i]- y
24      do ADDTO(pointsTo,
              FINDPOINTSTO(y, PUSH(c, i), visited))
25  for each edge x <-param[i]- y
26      do if c = ϵ or PEEK(c) = i
27           then ADDTO(pointsTo,
                     FINDPOINTSTO(y, POP(c), visited))
28  return pointsTo
```

Figure 7. Pseudocode for the core of our algorithm. We elide (1) the FINDFLOWSTO procedure, which is analogous to FINDPOINTSTO but traverses edges in the opposite direction; (2) The outer refinement loop, which checks if the result is sufficiently precise for the client and removes match edges according to the refinement policy; (3) code for constructing the context-sensitive call graph, described at the end of Section 5; and (4) code for collapsing call graph SCCs, described in Section 4.3. The PUSH and POP procedures return a new stack corresponding to the respective operation on their argument(s), while ADDTO and REMOVEFROM mutate the set passed as the first argument.

with the goal of distinguishing the field contents of different objects of some class `T`; as shown in Figure 3, match edges for accesses of field f cause merging of the contents of f across objects. Our method for choosing `T` is straightforward, but empirically effective: we choose the enclosing class for the field corresponding to the first match edge encountered in the previous analysis pass, and then remove match edges on all fields of this class.[2] In the example of Figure 6, we encounter a match edge on `arr` in the first pass and `elems` in the second pass, leading to removal of match edges on those fields. Removing these match edges allows the analysis to distinguish the contents of the internal `Object` array of a particular `Vector`, as desired.

[2] We also remove match edges for fields in superclasses and inner classes of `T`, as they also tend to be relevant.

Pseudocode. Figure 7 gives pseudocode for the core of our algorithm. Given a variable x and a call stack c, the procedure call FINDPOINTSTO$(x, c, \emptyset)$ returns the points-to set of x in context c, containing pairs (o, c') (see Section 4.1). The algorithm traverses the graph looking for incoming *flowsToR*-paths to x, filtering those paths using the tracked call stack state for R_{C}. Note that since the procedure traverses edges in the inverse direction, an incoming return[i] edge (*i.e.*, a $\overline{\text{return}}$[i] edge) requires pushing on the call stack (line 24) and an incoming param[i] edge (a $\overline{\text{param}}$[i] edge) requires popping (line 27). The check for $c = \epsilon$ at line 26 allow for partially balanced call parentheses.

The algorithm clears the R_{C} state across match edges as expected (line 11) and also across assignglobal edges (line 7). We model accesses to globals in our graph with direct assignments, using assignglobal and $\overline{\text{assignglobal}}$ labels to distinguish such accesses from assignments between locals. Figure 5 shows that R_{C} has transitions $S \xrightarrow{\text{assignglobal}} \epsilon$ (analogously for $\overline{\text{assignglobal}}$), since like match edges, these edges also "skip" the sequence of calls and returns between the reads and writes of the corresponding global.

The only non-trivial aspect of the algorithm in Figure 7 is its handling of graph cycles. For cyclic paths with no getfield[f] or putfield[f] edges, the *visited* set ensures that we only visit each node once per call stack. A more complex situation arises for a cyclic path with field accesses, for example caused by a statement like `x = x.next`. The analysis may need to process this statement multiple times, for example to discover paths of the form $p \xrightarrow{\text{pf[next]}} q \xrightarrow{\text{pf[next]}} r \xrightarrow{aliasR} x \xrightarrow{\text{gf[next]}} x \xrightarrow{\text{gf[next]}} x$. We ensure termination using the *checkingAlias* set, which holds the pairs $((y, c), q)$ such that the algorithm is currently searching for an *aliasR*-path p from y to q in context c. Line 13 ensures that we do not recursively repeat the search for p; this test is sound since if the existence of p depends solely on p itself existing, then p cannot exist. The algorithm terminates since the sizes of *visited* and *checkingAlias* are bounded.

Call Graph Construction. We construct a context-sensitive call graph, *i.e.*, a call graph where targets of virtual calls are computed separately for each calling context, by raising call graph queries on the fly. Say that, during analysis, we reach a virtual call `x.m()` with R_{C} call stack c. To handle the call, we recursively query the analysis for $pt(x)$ in context c, use this result to determine the possible callees of the `x.m()` call, and then continue our original query in these callees. There may be cases where targets of virtual calls cyclically depend on each other; we handle such cases by tracking pending queries and re-propagating them as new call targets become known, a standard technique in demand-driven analyses [31].

6. Evaluation

Our experiments validated the following three experimental hypotheses:

- **Some clients need context sensitivity.** We confirmed, as shown previously [21], that context-insensitive analysis does not have enough precision for the cast-checking client, as it could only prove 7.8% of the downcasts in our benchmarks safe.
- **Our refinement approach is precise.** Our refinement algorithm proved 61% more casts safe on average than one of the most precise existing algorithms [21], and refinement was critical for this precision gain. Also, our algorithm proved a disjointness property of objects allocated in some factory methods, requiring precision beyond that of the existing algorithm.
- **Our refinement approach is scalable.** With the analysis budget we chose, our algorithm checked all application downcasts in under 13 minutes on all benchmarks. Furthermore, our algorithm required no more than 35MB of memory for any of the

benchmarks, an order of magnitude less than the memory requirements for existing comparable analyses.

6.1 Experimental Configuration

Implementation We implemented our analysis using the Soot 2.2.1 [41] and Spark [20] frameworks. For our graph representation, we augment the pointer assignment graph built by Spark with param[i] and return[i] edges for context sensitivity. We analyzed the Sun JDK 1.3.1_01 libraries, as Soot provides models of this version's native methods. Unmodeled native methods and reflection calls are handled by conservatively answering that the queried variable can point to any abstract location. All experiments were performed on a machine with a Xeon 2.4GHz processor and 2GB RAM, running Fedora Core 1 Linux.

Our implementation adds two optimizations to the algorithm described in Figure 7. The first is to cache the results of the FINDPOINTSTO and FINDFLOWSTO procedures when possible, as they are often invoked repeatedly with the same arguments for a single query. The second is to sometimes use an alternate strategy for finding *aliasR*-paths from y to q: rather than computing the set of all variables V may-aliased with y and then checking if $q \in V$ (as in Figure 7), the analysis finds $pt(y)$ and $pt(q)$, and then checks if $pt(y) \cap pt(q) \neq \emptyset$, tracking R_{C} state appropriately. The latter strategy sometimes requires far less graph traversal, and the implementation uses it when the former strategy fails.

We give experimental results for the following analyses:

DemRef: our demand-driven, refinement-based algorithm.

Full: our demand-driven algorithm configured to treat all code with full precision, rather than refining.

1H: a 1-limited object-sensitive analysis [23] (*i.e.*, limited to 1 level of object sensitivity) with a (1-limited) context-sensitive heap abstraction and call graph, provided as part of the Paddle framework for BDD-based analysis [19].

The 1H algorithm was chosen because in recent work [21], it was shown to be the most precise of a set that included the Zhu and Calman and Whaley and Lam algorithms [43, 46] and a call string approach [35]. We were unable to run the 1H algorithm on the chart benchmark within 2GB of RAM; the result for chart in Table 2 is taken from [21], as its results for other benchmarks exactly matched our observations.

To compare with an analysis that handles assignments with equality constraints, we also implemented data structure analysis [17], a context-sensitive analysis for C that we adapted to Java. We implemented the analysis both with and without on-the-fly context-sensitive call graph construction. We found that without a context-sensitive call graph, the analysis was much too imprecise for our clients; *e.g.*, it could not prove any casts safe in most benchmarks. This imprecision stemmed from the collapsing of call graph SCCs by the analysis (see Section 4.3), which are large in a context-insensitive call graph [21]. We were unable to sufficiently scale the algorithm variant with context-sensitive call graph construction to analyze our benchmarks; the most similar published analysis for Java [26] had similar scalability issues.

Configuration. We configure our analysis to refine the precision of context-insensitive field-sensitive Andersen's analysis with an on-the-fly call graph, as implemented in Spark [20]. We use the context-insensitive analysis to answer queries that require less precision, and to rule out certain paths in our analysis. For example, if we are trying to prove a cast of x to type T safe, we do not traverse to nodes y where the context-insensitive analysis shows that all locations in $pt(y)$ are subtypes of T. The analysis is scalable, analyzing all benchmarks in under 3.5 minutes, including the time required to construct the graph representation. We include the context-insensitive analysis time in all presented running times for our analysis.

Benchmark	Methods	Statements
compress	2722	36690
db	2741	37243
jack	2996	42729
javac	3916	77619
jess	3354	47645
mpegaudio	2927	41009
mtrt	2873	39180
soot-c	4979	90355
sablecc-j	8853	164056
polyglot	6227	120634
antlr	4021	77934
bloat	5415	106629
chart	7323	110594
jython	4560	69026
pmd	7388	115857
ps	5320	106718

Table 1. Information about our benchmarks. We include the SPECjvm98 suite, soot-c and sablecc-j from the Ashes suite [1], several benchmarks from the DaCapo suite version beta050224 [2], and the Polyglot Java front-end [25]. The "Statements" column gives the number of edges in the graph representation. The numbers include the reachable portions of the Java library, determining using a call graph constructed on the fly with Andersen's analysis [3] by Spark [20].

Our refinement analysis is best run with a *budget*: after some fixed amount of time for each query, the analysis terminates and returns a conservative result to the client [37]. This budget prevents the analysis from running excessively long on queries it cannot hope to answer precisely, *e.g.*, those that require flow-sensitive precision. For our experiments, we configured our analysis to traverse at most 75000 nodes per query, divided evenly among a maximum of 10 refinement iterations, a sweet spot for the tested clients; doubling the budget yielded a negligible precision gain.

Benchmarks. Our benchmark suite is described in Table 1. We use the same suite as that of [21], to compare with its object-sensitive analysis. The size of the benchmarks are comparable to those used in other recent Java points-to analysis studies [20, 43].

Clients. We evaluated our analysis using three clients. The first was a client that checked the safety of downcasts in application code; as in [21], library casts were excluded to make benchmark differences clear, but the library was still analyzed when necessary for application casts. As illustrated in Section 2.4, downcast checking is an exacting test of points-to analysis precision, especially of the ability to distinguish the contents of different data structures; an analysis that fares poorly at proving downcast safety is unlikely to satisfy other demanding clients.

We also experimented with a client that tries to prove disjointness of the contents of objects allocated in *factory methods*, *i.e.*, methods that return a newly-allocated object for each call. For example, an iterator() method typically allocates a new Iterator object for each call. The client looks for factory methods using simple pattern matching, and then tries to prove disjointness of method return values for objects allocated in different calls to these methods. For iterator(), the client tries to show that calls to next() on Iterator objects allocated by different calls to iterator() can return distinct objects. Proving such disjointness properties could be important, *e.g.*, to reduce false positives for a verification client. Furthermore, this client requires greater precision than the 1H algorithm of [21] can provide (since it requires at least 2 levels of

Benchmark	Casts	DemRef Time (s)	DemRef Safe	Full Safe	1H Safe
`compress`	6	44.8	33.3	33.3	0.0
`db`	24	44.4	79.2	37.5	25.0
`jack`	135	62.8	52.6	23.0	31.1
`javac`	315	150.3	20.6	12.4	13.3
`jess`	76	63.7	72.4	6.6	57.9
`mpegaudio`	12	58.8	25.0	25.0	33.3
`mtrt`	10	47.4	50.0	40.0	40.0
`soot-c`	906	387.8	28.0	14.1	8.3
`sablecc-j`	362	315.8	18.5	5.5	11.9
`polyglot`	3482	750.3	88.1	6.8	72.5
`antlr`	281	118.2	50.9	2.8	21.7
`bloat`	1217	472.6	12.6	5.2	6.7
`chart`	535	283.5	38.5	9.0	30.5
`jython`	464	84.9	8.8	2.8	6.5
`pmd`	1135	571.7	15.1	10.0	11.2
`ps`	659	131.1	6.2	5.5	41.0

Table 2. Results for the cast safety client. The "Casts" column gives the number of downcasts that context-insensitive analysis cannot prove safe; these numbers differ from those in [21] because we exclude casts of non-pointers (*e.g.*,`float` to `int`), as they cannot cause a runtime exception. The three rightmost columns respectively give the percentage of these casts proven safe by our refinement algorithm ("DemRef"), our demand-driven algorithm configured to treat all code precisely ("Full"), and the object-sensitive analysis of [21] ("1H"). The "DemRef Time" column gives the running time for the refinement algorithm in seconds.

object-sensitivity), and hence illustrates the benefits of having a more precise analysis.

Finally, to further test performance, we ran a client that queried the DemRef analysis for all application variables where the 1H analysis yielded a more precise result than context-insensitive analysis, representing a client that requires near-exhaustive points-to information. Due to space constraints, full data for this client and the factory method client are given in a technical report [36]; here we give an overview of their results.

6.2 Experimental Results

Imprecision of context-insensitive analysis. We found context-insensitive Andersen's analysis (from Spark [20]) to be insufficient for proving downcasts safe in our benchmarks. The analysis could prove an average of only 7.8% of casts safe, ranging from 0% for `compress` to 31.7% for `sablecc-j`. This result is consistent with previous work [21], and shows the client's need for more precise analysis.

Precision for cast-checking. Table 2 shows that our refinement algorithm provides more precision for the cast-checking client than the 1H algorithm. The refinement technique proved an average of 1.61x as many casts safe as the 1H algorithm (excluding `compress` where 1H proved no casts safe), ranging from 0.15x for `ps` to 3.39x for `soot-c`. The large precision benefit for the `soot-c` benchmark stems primarily from precise handling of iterators. Given code "`Iterator i = x.iterator(); o = (Foo)i.next();`", proving the cast to `Foo` safe is beyond the capabilities of the 1H algorithm.[3]

The refinement algorithm is significantly more precise for cast checking (within the same budget) than a demand-driven analysis that treats all code precisely (the "Full" algorithm), as shown in Table 2. Given the analysis budget of 75000 nodes, the algorithm with refinement proved 4.25x more casts safe than without refinement, ranging from 1x for `mpegaudio` to 17.88x for `antlr`; doubling the analysis budget had a negligible impact on this result. The algorithm without refinement is often even less precise than then 1H algorithm, showing the importance of using both the demand-driven approach and refinement.

There are several reasons why some casts cannot be proven safe by our analysis. DemRef was less precise than 1H for the `ps` benchmark due to 181 casts of objects read from an operator stack mutated in many parts of the program; the large amount of relevant code led to DemRef choosing incorrect fields to refine for these casts. Proving certain casts safe requires flow- or path-sensitivity, *e.g.*, for casts dominated by an `instanceof` check that ensures their safety; many such casts can be proven safe by an extra intraprocedural analysis [26, 42]. Sometimes, context-sensitive call graph construction consumes the bulk of analysis time for DemRef, but is unnecessary for a precise result; automatically determining which virtual call sites require precise handling is future work.

Precision for factory methods. Our analysis proved the contents of many factory-allocated objects disjoint (see [36] for the full data). Excluding benchmarks with fewer than 5 factory methods, the analysis proved disjointness for an average of 42.8% of the methods in each benchmark, ranging from 21.4% for `jython` to 91.7% for `jess`. This result shows that precision greater than that provided by the 1H algorithm is required for realistic clients besides downcast checking, and that our analysis can provide that precision.

Scalability of refinement approach. Due to our demand-driven approach, the memory requirements of our analysis are significantly less than those of previous approaches. In the experiments, our analysis never consumed more than 5MB of memory for any query, and our implementation does no caching between queries. The memory required to store the results from the context-insensitive analysis pre-pass is less than 30MB using BDDs [5], yielding a maximum memory requirement of 35MB for these benchmarks. In comparison, we could not run the object-sensitive algorithm of [21] on the `chart` benchmark within 2GB of RAM, and a precise equality-based analysis requires 1GB of RAM on large benchmarks [38].

Table 2 shows that the refinement algorithm scaled well for the cast checking client, taking under 13 minutes for each benchmark. The factory method client took under 4 minutes per benchmark, as it raised few queries. The longest running time for the client querying all application variables for which the 1H algorithm yielded greater precision than context-insensitive analysis was 94 minutes for `pmd`, with an average query time of 0.68 seconds (full data in [36]). Our current implementation computes each query result from scratch, and we believe that for large numbers of queries, performance could be significantly improved through more caching.

7. Related Work

We limit our discussion of related work to several closely related areas: context-sensitive points-to analysis, refinement-based analysis, demand-driven points-to analysis, CFL-reachability, and cast verification. See [11, 15, 34] for comprehensive discussions and comparisons of various points-to analyses.

Context-Sensitive Points-To Analysis. Our points-to analysis is distinguished from previous work by its ability to scalably compute a context-sensitive heap abstraction and call graph while requiring far less memory than existing approaches. Table 3 gives key properties of several other context-sensitive points-to analysis algorithms; here we summarize some of the approaches taken by these analyses.

While effective for clients like escape analysis, summary-based analyses with subset constraints [44, 45] have only been shown to

[3] Two levels of object-sensitivity (including the heap abstraction) would suffice for this case, but that analysis does not yet scale in the Paddle framework [18].

Algorithm	Eq / Sub	CS CG	CS Heap	Shown to Scale
Zhu / Whaley [43, 46]	Sub			X
Whaley [44]	Sub			
Choi [7]	Sub			
Fähndrich [9]	Eq	X		X
Rehof [27]	Sub	X		
Wilson [45]	Sub	X		
Cherem [6]	Eq		X	
Ruf [33]	Eq		X	1.1 lib
Liang [22]	Eq		X	
Guyer [12]	Sub		X	
Lattner [17]	Eq	X	X	
O'Callahan [26]	Eq	X	X	
Steensgaard (CS) [38]	Eq	X	X	X
Wang [42]	Sub	X	X	1.1 lib
Object-sensitive [21, 23]	Sub	X	X	1-limited
Naik [24]	Sub	X	X	3-limited
Current paper	Sub	X	X	X

Table 3. A comparison of key properties of previous analyses. Algorithms are named by first author unless they have been referred to differently in this paper; note that Steensgaard (CS) [38] is different than his original analysis [39]. The "Eq/Sub" column indicates whether assignments are modeled with equality or subset constraints, and the "CS CG" and "CS Heap" columns respectively indicate the use of a context-sensitive call graph and heap abstraction. Finally, the "Shown to Scale" column indicates whether the algorithm has been shown to scale to large Java benchmarks; "1.1 lib" means the smaller Java 1.1 libraries were analyzed, and *k-limiting* [35] is indicated where used.

scale to medium-sized programs. Summary-based analyses that use equality constraints have typically been more scalable [9, 17, 26, 38]. The algorithms of O'Callahan [26] and Lattner and Adve [17] scale well with a context-sensitive heap abstraction, but are less scalable when computing a context-sensitive call graph. A similar analysis for C# scales with a context-sensitive call graph [38], but still requires more than 1GB of memory on its largest benchmark.

Binary decision diagrams (BDDs) have been used in several recent systems to greatly improve the scalability of context-sensitive analysis. The Zhu and Calman [46] and Whaley and Lam [43] algorithm, while quite scalable, uses a context-insensitive heap abstraction and call graph, leading to precision loss [21]. We compare extensively with the BDD-based 1-limited object-sensitive analysis of [21] in Section 6; object-sensitive analysis [23] analyzes methods separately based on the receiver object instead of using call strings, exploiting typical object-oriented code structure for greater precision and scalability.

Naik *et. al.* present a static race detection tool based on a scalable 3-limited object-sensitive analysis [24]. It is difficult to compare our analysis with theirs directly, as their race detection client raises object-sensitive queries, which are in general incomparable with context-sensitive queries [23]. As future work we plan to design an object-sensitive version of our analysis, allowing for a better comparison.

Refinement-Based Analysis. Guyer and Lin [12] present a client-driven points-to analysis for C that detects which statements cause imprecision for a given client, and then re-analyzes the program with greater flow and context sensitivity for those statements. Their results show that they obtain much of the precision benefit of flow and context sensitivity at a small extra cost, and their work was an inspiration for ours. The key difference with our work is that their analysis adds sensitivity to all possibly polluting statements when imprecision is detected; this approach does not scale for Java, as it requires too much code to be treated precisely.

CFL-Reachability. Our use of CFL-reachability is based on the work of Reps et. al. on the framework [29, 31]. Given a CFL-reachability formulation, a demand-driven algorithm [16] for the single-source L-path problem can be obtained automatically by applying the magic-sets transformation to L [29]. Our `match` edges are related to the summary edges used by the efficient CFL-reachability algorithm for balanced parentheses languages [28, 31, 32]. Summary edges are computed bottom-up as L-paths between parentheses are found, while `match` edges are added exhaustively and then refined by checking for L-paths.

Demand-Driven Points-To Analysis. The current work builds on our own previous work on demand-driven points-to analysis [37]. The key addition of the present work is context sensitivity. In the previous algorithm, the balanced parentheses property of Java points-to analysis was used to create an analysis that traversed a small portion of the graph to compute an answer, for use with tight time budgets. Here, we use the balanced parentheses to both guide our refinement, as they indicate where more field and context sensitivity is needed, and to avoid processing irrelevant parts of the graph. We have also added precise handling of recursive fields in the current algorithm, which is more important in the context-sensitive setting for handling data structures like Java's `LinkedList`. Finally, the present works adds on-the-fly call graph building.

Cast Verification. Constraint-based analyses have been developed to convert legacy Java programs to use Java 5 generics, and they have been shown to prove many downcasts safe [8, 10]. These analyses rely on the generics annotations of Java 5 `java.util` classes to model their behavior. In contrast, our approach determines properties of library code without annotations, and hence handles application data structures as well. Furthermore, our analysis can be used for more than cast safety, as shown by our factory method client. In other work, Wang and Smith present a context-sensitive constraint-based type analysis [42] and show that it is effective at proving downcasts safe. However, they analyze the Java 1.1 libraries, which are significantly smaller than the Java 1.3 libraries used in the present work.

8. Conclusions

We have developed a refinement-based, demand-driven context-sensitive points-to analysis that is both scalable and precise. By refining sensitivity for heap accesses and method calls simultaneously, the analysis precisely handles the important code for a query, while often entirely skipping irrelevant code. For the demanding downcast checking client, our analysis proved the safety of 61% more casts than one of the most precise existing analyses, while running in under 13 minutes on large benchmarks and requiring under 35MB of memory. Our technique is a compelling alternative to existing points-to analysis approaches, as its combination of demand-driven analysis and refinement allow it to provide sufficiently precise results for demanding clients, while scaling to large programs with relatively little engineering effort.

Acknowledgments. This work is supported in part by the National Science Foundation with grants CCF-0085949, CCR-0105721, CCR-0243657, CNS-0225610, CCR-0326577, and CNS-0524815, the University of California MICRO program, an Okawa Research Grant, a Hellman Family Faculty Fund Award, a National Defense Science and Engineering Graduate Fellowship, and a Microsoft Graduate Fellowship. This work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. The views expressed herein are not necessarily those of DARPA.

We deeply thank Ondřej Lhoták, whose frequent help allowed us to compare our results with those in [21]. We thank Mooly Sagiv for many discussions on early versions of the paper. We also thank Laurie Hendren, Bill McCloskey, Bill Thies, Dave Mandelin, Saswat Anand, Brian Fields, Gilad Arnold, Robert O'Callahan, and the anonymous reviewers for their helpful comments.

References

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

[2] DaCapo Benchmark Suite. http://www-ali.cs.umass.edu/DaCapo/gcbm.html.

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

[4] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), San Jose, CA*, October 1996.

[5] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[6] S. Cherem and R. Rugina. Region analysis and transformation for java programs. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, 2004.

[7] J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[8] A. Donovan, A. Kiezun, M. S. Tschantz, and M. D. Ernst. Converting Java programs to use generic libraries. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[9] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[10] R. Fuhrer, F. Tip, J. Dolby, A. Kiezun, and M. Keller. Refactoring techniques for migrating applications to generic java container classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[11] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[12] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *International Static Analysis Symposium (SAS), San Diego, CA*, June 2003.

[13] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[14] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[15] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), Snowbird, Utah*, June 2001.

[16] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

[17] C. Lattner and V. Adve. Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis. Tech. Report UIUCDCS-R-2003-2340, UIUC, April 2003.

[18] O. Lhoták. Personal communication. 2005.

[19] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, Jan. 2006.

[20] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[21] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *International Conference on Compiler Construction (CC)*, 2006.

[22] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[23] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[24] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[25] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *CC*, pages 138–152, 2003.

[26] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, November 2000.

[27] J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2001.

[28] T. Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction (CC), Edinburgh, Scotland*, April 1994.

[29] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.

[30] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

[31] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1995.

[32] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), New Orleans, LA*, December 1994.

[33] E. Ruf. Effective synchronization removal for java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[34] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[35] O. Shivers. Control flow analysis in scheme. In *Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[36] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. Technical Report UCB/EECS-2006-31, UC Berkeley, 2006.

[37] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[38] B. Steensgaard. Personal communication on analysis for Microsoft Bartok compiler. 2005.

[39] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

[40] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *OOPSLA*, pages 13–26, 2003.

[41] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[42] T. Wang and S. F. Smith. Precise constraint-based type inference for java. In *15th European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[43] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[44] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[45] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[46] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.