Context Sensitivity for Points-to Analysis Using Annotated Inclusion Constraints

Ana Milanova[1] and Barbara G. Ryder[2]

[1] Dept. Computer Science, Rensselaer Polytechnic Institute, Troy NY 12180, USA
[2] Dept. Computer Science, Rutgers University, Piscataway NJ 08854, USA

Abstract. We present *annotated inclusion constraints* - a new general framework that enables the modeling of different dimensions of flow analysis precision using inclusion constraint systems. The annotations are a powerful formalism that can be used to model a large variety of precision dimensions, such as field sensitivity, virtual dispatch, k-CFA context sensitivity and object sensitivity, which is a new form of context sensitivity suitable for object-oriented languages.
We perform extensive experiments on a large set of Java programs with various points-to analyses that are instantiations of the framework, and identify object-sensitive analyses that run in comparable time and space to a context-insensitive analysis, while improving precision significantly. These empirical results, and the results for an earlier framework instantiation [33], indicate that the annotated constraints are effective and can be used as a unified implementation framework for a variety of relatively precise flow analyses.

1 Introduction

The two major dimensions in the design spectrum of program flow analysis are *flow sensitivity* and *context sensitivity.* Intuitively, flow-sensitive analyses take into account the flow of control between program points and compute separate solutions for these points. Flow-insensitive analyses ignore the flow of control between program points and therefore can be less precise and more efficient. Context-sensitive analyses distinguish between the different contexts under which the method is invoked and analyze the method separately for each context. Context-insensitive analyses do not separate the different invocation contexts for a method which may improve efficiency at the expense of some possible precision loss. Other dimensions include *field sensitivity* (i.e., the analysis distinguishes flow of values through distinct object fields), and the ability of the analysis to interleave call graph construction with the computation of flow information (e.g., points-to and type information) [36].

Set inclusion constraints are a powerful formalism for expressing various relatively precise flow analyses. Many existing analyses based on inclusion constraints are very efficient, but they do not model context sensitivity. Context sensitivity is of crucial importance for the analysis of object-oriented languages, because context insensitivity inherently compromises analysis precision due to fundamental object-oriented features and programming idioms (Section 3 presents

examples that illustrate how context insensitivity compromises analysis precision due to basic object-oriented features such as encapsulation, inheritance, polymorphism, and the use of containers and maps).

1.1 Annotated Inclusion Constraints

Inclusion-constraint-based analyses examine the program and construct a system of constraints of the form $X \subseteq Y$ where X and Y are sets that represent some program elements and the inclusion relation represents flow from the element corresponding to X to the element corresponding to Y. During resolution, new transitively implied constraints are added to the system; for example, constraints $X \subseteq Y \subseteq Z$ imply $X \subseteq Z$. The solution of the constraint system provides information about the flow of values in the program. Work on techniques for constraint representation and constraint resolution has led to major advances making flow analyses based on inclusion constraints very efficient [15, 39, 22]. However, these analyses are context-insensitive and field-insensitive. Therefore, directly reusing existing technology for the analysis of object-oriented languages may result in *imprecise* analysis.

Because of the importance of analysis precision as well as its efficiency, it is important to investigate approaches for incorporating precision dimensions in inclusion constraints, while in the same time retaining efficiency as much as possible.

This paper presents *annotated inclusion constraints* - a general framework that enables the modeling of dimensions of analysis precision such as context sensitivity, in inclusion constraint systems. It extends the constraint system in [15, 39] by introducing a *parameterization* based on *language of constraint annotations* and *annotation operations.* The annotated constraints have the form $X \subseteq_a Y$ and the role of the annotations is to restrict infeasible flow in the system, i.e., a new transitive constraint $X \subseteq_c Z$ is inferred from $X \subseteq_a Y \subseteq_b Z$ if only if a and b "match" according to a specific binary predicate *match*. The analysis designer can instantiate the framework with appropriate annotations and annotation operations to model an inclusion-based flow analysis that incorporates some dimension of precision. One instance of this framework is the context-insensitive *points-to* analysis for Java [33], a popular form of flow analysis; in this analysis, field annotations are used to model precisely flow of values through object fields, and method annotations are used to model virtual dispatch. This paper describes how the framework can be instantiated to express various context-sensitive points-to analyses precisely and efficiently.

1.2 Context-Sensitive Points-to Analysis for Java

The goal of points-to analysis for Java is to find out the set of objects that a given reference variable or a reference object field may point to. These points-to sets are typically abstracted by *points-to graphs* (An example points-to graph is shown in Figure 2, which is discussed in Section 3.1).

Because of the wide variety of applications of points-to analysis in optimizing Java compilers and software tools, it is important to investigate techniques for precise and efficient computation of points-to information. Since points-to anal-

ysis precision is inherently compromised by context insensitivity due to object-oriented features such as inheritance, encapsulation and polymorphism, it is necessary to develop techniques for practical context-sensitive points-to analysis of object-oriented programs.

The framework for annotated constraints can be instantiated to model several distinct forms of context sensitivity for points-to analysis such as *object sensitivity*, an instance of the functional approach to context sensitivity and *k-CFA context sensitivity*, an instance of the call-string approach [37]. Object sensitivity is a new approach to context sensitivity for flow-insensitive points-to analysis that uses the *receiver object name* to distinguish context [26, 25]. The degree of precision in the object naming scheme is determined by analysis parameter k (Section 4.1 describes the object naming scheme in detail). *k-CFA* is a popular mechanism for context sensitivity that distinguishes context by a sequence of k call sites. Object-sensitive analysis of depth k and *k-CFA* analysis can be modeled by defining appropriate annotation language and operations.

In addition, we propose a parameterization mechanism that can be used to define a large spectrum of points-to analyses that range from context-insensitive analysis to object-sensitive analysis of depth k and k-CFA analysis. It allows *targeted* context sensitivity because the analysis designer can select parts of the program for which keeping more precise information is likely to improve analysis precision. The parameterization can be modeled easily in the framework for annotated constraints by using appropriate annotations and operations.

We present experiments on a set of 23 Java programs for three object-sensitive analyses of depth one (i.e., context is distinguished per object allocation site); we compare these analyses with a popular context-insensitive points-to analysis for Java [33] based on Andersen's analysis for C [5]. Using the framework with appropriate annotations, we explore the spectrum between Andersen's context-insensitive analysis and the most precise object-sensitive analysis of depth one, and identify analyses that have cost comparable to Andersen's context-insensitive analysis and substantially better precision, comparable to the most precise object-sensitive analysis.

These results and the results in [33] indicate that the framework for annotated constraints is a powerful and effective formalism for expressing inclusion-based flow analyses that incorporate dimensions of precision such as context sensitivity, field sensitivity and virtual dispatch. The practicality of relatively precise analyses implemented with annotated constraints, makes them realistic candidates for use in software productivity tools and optimizing compilers.

Contributions The contributions presented in the paper are the following:

- We develop annotated inclusion constraints - a general framework that enables the modeling of various dimensions of flow analysis precision using inclusion constraints. We show that this is a powerful formalism by expressing object-sensitive points-to analysis of depth k, k-CFA points-to analysis and a spectrum of related analyses as instances of the framework.
- Experiments on a large set of Java programs show that the annotated constraints are effective and can be used as an implementation framework for a

variety of relatively precise flow analyses. Object-sensitive points-to analyses implemented in the framework have cost comparable to a context-insensitive analysis for most programs, while improving precision significantly.

Outline The rest of this paper is organized as follows. Section 2 describes the general structure of the annotated inclusion constraints. Section 3 describes a popular flow- and context-insensitive points-to analysis for Java, and demonstrates the imprecisions that arise due to context insensitivity. Section 4 outlines the semantics of the object-sensitive analysis of depth k, and the semantics of the k-CFA context-sensitive points-to analyses. Section 5 shows how these analyses can be modeled using annotated inclusion constraints. Section 6 describes the parameterization mechanism. The empirical results are presented in Section 7. Section 8 discusses related work and Section 9 presents conclusions.

2 Annotated Inclusion Constraints

This section defines the general structure of the annotations and the annotated inclusion constraints. Previous constraint-based flow analyses employ non-annotated inclusion constraints [15, 39]. We develop a new general constraint-based approach that extends previous work [15, 39] by introducing constraint annotations for the purposes of modeling dimensions of analysis precision.

2.1 Annotation Language and Operations

The set of annotations is a finite set of strings $\mathcal{S}$. One element of the set is designated as the empty annotation. The following operations on $\mathcal{S}$ are necessary for the integration of the annotations in the constraint solution procedure.

$$\begin{array}{ll} \mathit{match}: & \mathcal{S} \times \mathcal{S} \rightarrow \text{boolean} \\ \mathit{concat}: & \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S} \\ \mathit{transpose}: & \mathcal{S} \rightarrow \mathcal{S} \end{array}$$

Operation *concat* requires the binary predicate *match* to hold. The set of strings $\mathcal{S}$ and the operations acting on the elements of $\mathcal{S}$ can be instantiated to model a specific precision dimension. For example, if the annotations are used to model field sensitivity, $\mathcal{S}$ includes the set of field identifiers; an inclusion relation annotated with $f \in \mathcal{S}$ represents flow of values through object field f.

2.2 Constraint Language

Flow analyses that incorporate dimensions of precision can be modeled using annotated inclusion constraints of the form $L \subseteq_a R$, where L and R correspond to some program elements and $a \in \mathcal{S}$ is chosen from the set of annotations described in Section 2.1. We use $L \subseteq R$ to denote constraints labeled with the empty annotation. L and R are set expressions, defined by the following grammar:

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

Here v and v_i are set variables, $c(\ldots)$ are constructed terms and $\mathit{proj}(\ldots)$ are projection terms. Each *constructed term* is built from an n-ary constructor c. A

constructor is either *covariant* or *contravariant* in each of its arguments; the role of this variance in constraint resolution will be explained shortly. Constructed terms may appear on both sides of inclusion relations. 0 and 1 represent the empty set and the universal set; they are treated as nullary constructors. *Projections* of the form $proj(c, i, v)$ are terms used to select the i-th argument of any constructed term $c(.., v_i,..)$. Projection terms may appear only on the right-hand side of an inclusion. Intuitively, set variables, constructed terms and projection terms represent specific program elements and the inclusion relation represents flow of values between these elements.

2.3 Annotated Constraint Graphs

Systems of constraints can be represented as directed multi-graphs. Constraint $L \subseteq_a R$ is represented by an edge from the node for L to the node for R; the edge is labeled with the annotation a. There can be multiple edges between the same pair of nodes, each with a different annotation.

The nodes in the graph can be classified as variables, sources, and sinks. *Sources* are constructed terms that occur on the left-hand side of inclusions. *Sinks* are constructed terms or projections that occur on the right-hand side of inclusions. The graph only contains edges that represent *atomic constraints* of the following forms: $Source \subseteq_a Var$, $Var \subseteq_a Var$, or $Var \subseteq_a Sink$. If the constraint system contains a non-atomic constraint, the resolution rules from Figure 1 are used to generate new atomic constraints, as described in Section 2.4.

$$c(v_1, ..., v_n) \subseteq_a c(v'_1, ..., v'_n) \Rightarrow \begin{cases} v_i \subseteq_a v'_i & \text{if } c \text{ is covariant in } i \text{ for } i\text{=}1...n \\ v'_i \subseteq_{transpose(a)} v_i & \text{if } c \text{ is contravariant in } i \text{ for } i\text{=}1...n \end{cases}$$

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

Fig. 1. Resolution rules for non-atomic constraints.

We use annotated constraint graphs based on the *inductive form* representation [4]. Inductive form is an efficient sparse representation that does not explicitly represent the transitive closure of the constraint graph. The graphs are represented with adjacency lists $pred(n)$ and $succ(n)$ stored at each node n. Edge (n_1, n_2, a), where a is an annotation, is represented either as a predecessor edge by having $\langle n_1, a\rangle \in pred(n_2)$, or as a successor edge by having $\langle n_2, a\rangle \in succ(n_1)$, but not both. $Source \subseteq_a Var$ is always a predecessor edge and $Var \subseteq_a Sink$ is always a successor edge. $Var \subseteq_a Var$ is either a predecessor or a successor edge, based on a fixed total order $\tau : Vars \rightarrow \mathcal{N}$. Edge (v_1, v_2, a) is a predecessor edge if and only if $\tau(v_1) < \tau(v_2)$.

2.4 Solving Systems of Annotated Inclusion Constraints

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

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

The closure rule can be applied locally, by examining $\mathit{pred}(v)$ and $\mathit{succ}(v)$. The new transitive constraint is created *only if* the annotations of the two existing constraints "match"—that is, only if $\mathit{match}(a, b)$ holds, where *match* is the specific binary predicate on the set of annotations as described in Section 2.1. Intuitively, the annotations define a coloring of the edges in the constraint graph and the TRANS rule uses the coloring to filter out some flow of values in the constraint graph. The annotation of the new constraint is produced by applying the concatenation operation *concat* to annotations a and b.

If the new constraint generated by the TRANS rule is atomic, a new edge is added to the graph. Otherwise, the resolution rules from Figure 1 are used to transform the constraint into several atomic constraints and their corresponding edges are added to the graph. For covariant arguments, the direction of flow is preserved and the annotation is also preserved. For contravariant arguments the direction of flow is reversed and a new annotation is produced by applying the specific *transpose* operation.

The closure of a constraint graph under the TRANS rule is the *solved inductive form* of the corresponding constraint system. The least solution of the system is not explicit in the solved inductive form [4], but is easy to compute by examining all predecessors of each variable.

For an annotated constraint graph, the least solution is computed by applying transitive acyclic traversal analogously to [4], but the annotations are used as in rule TRANS:

$$\begin{array}{l} \mathit{LS}(v) = \{\langle c(\ldots), a\rangle \mid \langle c(\ldots), a\rangle \in \mathit{pred}(v)\} \cup \\ \{\langle c(\ldots), \mathit{concat}(x, y)\rangle \mid \langle u, y\rangle \in \mathit{pred}(v) \wedge \langle c(\ldots), x\rangle \in \mathit{LS}(u) \wedge\ \mathit{match}(x, y)\} \end{array}$$

In general, the least solution depends on the variable order function τ, but if the concatenation operation is *associative*, it does not. The framework can be easily generalized to handle multiple sets of annotations (for different dimensions of precision) by performing component-wise *match*, *concat*, and *transpose* [25].

3 Context-Insensitive Points-to Analysis for Java

This section discusses a points-to analysis for Java [33] which is derived from Andersen's points-to analysis for C [5]. Section 3.1 defines the semantics of the analysis, which is flow-insensitive, context-insensitive and field-sensitive. Section 3.2 illustrates how context insensitivity inherently compromises analysis precision for object-oriented languages.

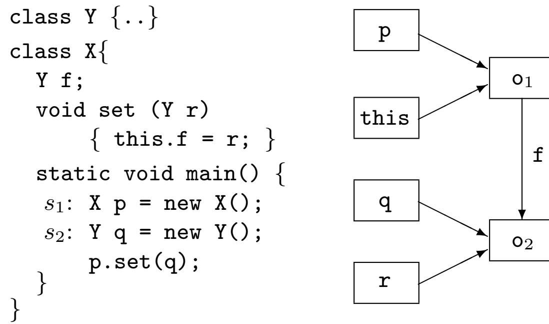

Fig. 2. Sample program and its points-to graph.

3.1 Analysis Semantics

Andersen's analysis for Java is defined in terms of three sets. Set R contains all reference variables in the analyzed program (including static variables). Set O contains names for all objects created at object allocation sites; for each allocation site s_i, there is a separate object name $o_i \in O$. Set F contains all instance fields in program classes. The analysis constructs *points-to graphs* containing two kinds of edges. Edge $(r, o_i) \in R \times O$ shows that reference variable r points to object o_i. Edge $(\langle o_i, f\rangle, o_j) \in (O \times F) \times O$ shows that field f of object o_i points to object o_j. A sample program and its points-to graph are shown in Figure 2.

For brevity, we only discuss the kinds of statements listed below. Other kinds of statements (e.g. calls to static methods) are handled in a similar fashion.

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

At a virtual call, name `m` uniquely identifies a method in the program. This method is the *compile-time* target of the call, and is determined based on the declared type of $\texttt{r}_0$ [20, Section 15.11.3]. At run-time, the invoked method is determined by examining the class of the receiver object and all of its superclasses, and finding the first method that matches the signature and the return type of `m` [20, Section 15.11.4].

Analysis semantics is defined in terms of *transfer functions* that add new edges to points-to graphs. Each transfer function represents the semantics of a program statement. The functions for different statements are shown in Figure 3 in the format $f(G, s) = G'$, where s is a statement, G is an input points-to graph, and G' is the resulting points-to graph. $Pt(G, x)$ denotes the points-to set (i.e., the set of all successors) of x in graph G. The solution computed by the analysis is a points-to graph that is the closure of the empty graph under the application of all transfer functions for program statements.

For most statements, the effects on the points-to graph are straightforward; for example, statement $l = r$ creates new points-to edges from l to all objects

$$f(G, s_i : l = new\ C) = G \cup \{(l, o_i)\}$$

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

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

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

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

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

Fig. 3. Points-to effects of program statements for Andersen's analysis.

pointed to by r. For virtual calls, resolution is performed for every receiver object pointed to by r_0. Function *dispatch* uses the class of the receiver object and the compile-time target of the call to determine the actual method m_j invoked at run-time. Variables $p_0, ..., p_n$ are the formal parameters of m_j; variable p_0 corresponds to the implicit parameter `this`. Variable ret_j contains the return values of m_j (we assume that each method has a unique variable that is assigned all values returned by the method; this can be achieved by auxiliary assignments).

3.2 The Imprecision of Context-Insensitive Analysis

Context-insensitive analysis is imprecise for many fundamental object-oriented features and programming idioms such as encapsulation, inheritance, and containers and maps. For example, if field f is written through a method invocation, during context-insensitive analysis field f of *each* receiver will point to *all* objects passed as arguments to the method which writes that value of f. As a result, context-insensitive analysis can incur significant imprecision.

The detailed example in Figure 4 illustrates basic object-oriented features (inheritance and encapsulation) for which context-insensitive analysis produces imprecise results. At line 2, `B.this` points to o_3 and `B.xb` points to o_1. After the analysis processes the call to the superclass constructor, `A.this` and `A.xa` point to o_3 and o_1, respectively. Because of the call at line 3, `A.this` will point to o_4 and `A.xa` will point to o_2. Thus, at statement `this.f=xa` at line 1, spurious edges $(\langle o_3, f\rangle, o_2)$ and $(\langle o_4, f\rangle, o_1)$ are added to the graph. This imprecision usually propagates further and affects both the points-to analysis and its clients.

In inheritance hierarchies, instance fields are often located in superclasses and are written through invocations of superclass constructors or methods. During context-insensitive analysis, fields of subclass instances are perceived to point to objects intended for instances of other subclasses; thus, context insensitivity can lead to substantial imprecision. A similar imprecision occurs in the presence of containers and maps; due to context insensitivity all objects added to a container

```
  class X { void n() {...} }
  class Y extends X { void n() {...} }
  class Z extends X { void n() {...} }

  class A {
      X f;
1     A(X xa) { this.f = xa; } }

  class B extends A {
2     B(X xb) { super(xb); ... }
      ... }

  class C extends A {
3     C(X xc) { super(xc); ... }
      ... }

4  s1: Y y = new Y();
5  s2: Z z = new Z();
6  s3: B b = new B(y);
7  s4: C c = new C(z);
```

Fig. 4. Field assignment through a superclass.

appear to be "shared" among all containers of that class (or its subclasses).

4 Context-Sensitive Analysis

This section defines the semantics for two categories of context-sensitive analysis. The first one is *object-sensitive analysis*, a new form of context-sensitive analysis for object-oriented languages [26, 25]. With object sensitivity, instance methods (i.e., non-static methods) and constructors are analyzed separately for objects on which such methods may be invoked. This approach is an instance of the functional approach to context sensitivity, as defined by Sharir and Pnueli [37]. The second one is k-CFA context sensitivity, which is a popular form of context sensitivity that distinguishes context by a string of k enclosing call sites. This is an instance of the call-string approach to context sensitivity [37]. Section 5 shows how these analyses can be modeled using annotated inclusion constraints.

4.1 Semantics of Object-Sensitive Analysis

The semantics of object-sensitive analysis is defined in terms of a set of contexts $\mathcal{C}$, a set of replicated object names O', a set of replicated reference variables R', and a set of transfer functions.

Object contexts. Let S be the set of all allocation sites in the program, and k be an analysis parameter. Set $\mathcal{C} \subseteq \{S \cup S^2 \ldots \cup S^k\} \cup \{\epsilon\}$ represents the space of all possible *object contexts*. If allocation site s_i is enclosed in an instance method or constructor, a particular object context $(s_i, s_j, \ldots, s_p, s_q)$ represents all run-time objects that were created at s_i when the enclosing method was invoked on an object created by $(s_j, \ldots, s_p, s_q)$. For example, for k=2, context (s_i, s_j) denotes all objects created at allocation site s_i, when the enclosing

method of s_i was invoked on an object created at allocation site s_j. For the rest of the paper we will use $o_{ij...pq}$ to denote the sequence of allocation sites $(s_i, s_j, \ldots, s_p, s_q)$. An instance method or constructor m is separately analyzed for context $(s_i, s_j, \ldots, s_p, s_q) \in \mathcal{C}$, if m is invoked on an object represented by name $o_{ij...pq}$. A static method (e.g., `main`) is analyzed in the special context ϵ.

Set $O' \subseteq S \cup S^2 \ldots \cup S^{k+1}$ represents the set of replicated object names. If allocation site s_i is enclosed by a static method, there is a single object name o_i that represents the objects created at that site. If s_i appears in an instance method or constructor, there is a separate object name $o_{ij...pq} \in O'$ for each context $(s_j, \ldots, s_p, s_q) \in \mathcal{C}$ reaching the enclosing method of s_i. The separate analysis for reference variables is achieved by maintaining *replicas* of reference variables for each possible context. The set of replicated reference variables R' is defined by a function $map : R \times \mathcal{C} \rightarrow R'$. If $r \in R$ is a static variable or a local in a static method, r is mapped to itself. If r is a local variable in an instance method or constructor, r is mapped to a "fresh" variable r^c for every context $c \in \mathcal{C}$ that reaches m during the analysis.

Transfer Functions. The analysis constructs points-to graphs where the nodes are elements of R' and O'. Analysis semantics is defined by transfer functions that add new edges to these points-to graphs. Let $\mathcal{C}_m$ denote the set of reaching contexts for method m. The effects of the new transfer functions $F(G, s)$, are essentially equivalent to applying the corresponding $f(G, s)$ for each $c \in \mathcal{C}_m$, where m is the enclosing method of s. For example, the function for object creation becomes

$$F(G, s_i\colon l = \mathit{new}\ C) = G \cup \bigcup_{o_{j...pq} \in \mathcal{C}_m} \{(l^{o_{j...pq}}, o_{ij...pq})\}$$

Similarly, the function for virtual calls becomes

$$F(G, l = r_0.n(r_1, \ldots, r_n)) = G \cup$$
$$\bigcup_{c \in \mathcal{C}_m} \{\mathit{resolve}(G, n, o_{i...pq}, r_1^c, \ldots, r_n^c, l^c) \mid o_{i...pq} \in Pt(G, r_0^c)\}$$

$$\begin{aligned}
&\mathit{resolve}(G, n, o_{i...pq}, r_1^c, \ldots, r_n^c, l^c) = \\
&\quad \texttt{let}\ \ c' = \text{if } |i \ldots pq| \leq k \text{ then } o_{i...pq} \text{ else } o_{i...p} \\
&\qquad\quad n_j(p_0, p_1, \ldots, p_n, ret_j) = \mathit{dispatch}(o_{i...pq}, n)\ \ \texttt{in} \\
&\qquad \mathcal{C}_{n_j} = \{c'\} \cup \mathcal{C}_{n_j} \\
&\qquad \{(p_0^{c'}, o_{i...pq})\} \cup f(G, p_1^{c'} = r_1^c) \cup \ldots \cup f(G, l^c = ret_j^{c'})
\end{aligned}$$

The transfer function resolves the call separately for each enclosing context. If the length of the name of the receiver is $k+1$, the allocation site q is dropped from the sequence in order to accommodate the restriction for context length $\leq k$. During the resolution, the analysis first updates the set of reaching contexts $\mathcal{C}_{n_j}$ with the new context c'. It maps the formal parameters and return variable of n_j to the context of c' and performs appropriate updates.

4.2 Semantics of k-CFA Context-Sensitive Analysis

The *call string* approach is one of the most popular mechanisms for context sensitivity. It distinguishes calling context by a string of k enclosing call sites. For example, if k=2, a method m is analyzed separately for each pair of call sites (c_i, c_j), such that c_i invokes m, and c_j invokes the method containing c_i.

The semantics of k-CFA analysis can be defined in terms of a set of contexts $\mathcal{C}$, a set of replicated object names O', a set of reference variables R', and transfer functions. Let C be the set of all call sites in the program. Set $\mathcal{C} \subseteq \{C \cup C^2 \cup \ldots C^k\} \cup \{\epsilon\}$ represents the set of all *call-string* contexts; ϵ is used to denote the context for start-up methods (e.g., `main`, static initializers `<clinit>`). Similarly to the object-sensitive analysis, the analysis maintains replicas of object names and reference variables for every reaching context.

The transfer functions $F(G, s)$ are essentially equivalent to the transfer functions for the object-sensitive analysis, except for virtual calls.

$$F(G, c_i : l = r_0.n(r_1, \ldots, r_n)) = G \cup$$
$$\bigcup_{c \in \mathcal{C}_m} \{resolve(G, n, o, r_1^c, \ldots, r_n^c, l^c) \mid o \in Pt(G, r_0^c)\}$$

$$\begin{aligned}
&resolve(G, n, o, r_1^c, \ldots, r_n^c, l^c) = \\
&\qquad \texttt{let}\ j \ldots pq = c \\
&\qquad\quad c' = \text{if } |j \ldots pq| < k \text{ then } ij \ldots pq \text{ else } ij \ldots p \\
&\qquad\quad n_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o, n)\ \texttt{in} \\
&\qquad\ \mathcal{C}_{n_j} = \{c'\} \cup \mathcal{C}_{n_j} \\
&\qquad\ \{(p_0^{c'}, o)\} \cup f(G, p_1^{c'} = r_1^c) \cup \ldots \cup f(G, l^c = ret_j^{c'})
\end{aligned}$$

5 Context-Sensitive Points-to Analysis Using Annotated Inclusion Constraints

In order to model context-sensitive points-to analysis we use the points-to representation from [15, 39]. In this model, the set of objects pointed to by location x is represented by a set variable v_x and each object is represented by a constructed term of the form $ref(o, v_o, \overline{v_o})$. The last two arguments to the *ref* constructor are the same variable but with different variance. [3] Intuitively, the second argument is used to read the fields of object o, while the third argument is used to write the fields of o. Constraint $ref(o, v_o, \overline{v_o}) \subseteq v_r$ represents the fact that r points to object o and constraint $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$ represents the fact that field f of object o_1 points to object o_2.

Our analysis examines the program *once*, and generates a system of annotated inclusion constraints. It uses the resolution rules which incorporate operations *match*, *concat*, and *transpose* to compute the least solution of the system. The least solution contains information about the points-to relationships in the program. Potentially costly replication of set variables and terms is avoided by using constraint annotations to model flow of values through different contexts

[3] Analogously to [15, 39] the overline bar denotes contravariant arguments.

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

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

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

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

Fig. 5. Constraints for assignment statements.

(there is at most one set variable corresponding to a reference variable in the program, and at most one *ref* term corresponding to an object allocation site).

Section 5.1 describes how object-sensitive analysis of depth k can be modeled as an instance of the framework from Section 2, by using *object annotations*. Similarly, Section 5.2 describes how k-CFA analysis can be modeled as an instance of the framework, by using *call-string annotations*.

5.1 Object-Sensitive Analysis

Constraints for Assignment Statements. For intraprocedural assignment statements, the analysis generates constraints with empty object annotations as shown in Figure 5. The role of the empty annotation is to represent all possible contexts $c \in \mathcal{C}_m$ for m, the enclosing method of the statement. Intuitively, the points-to set and the object annotations reaching the left-hand side of the constraint (e.g., v_r of $v_r \subseteq v_l$ in Figure 5) need to be propagated to its right-hand side. Suppose that for every context $o_{k\ldots rs}$ and object $o_{ij\ldots pq}$, such that $r^{o_{k\ldots rs}}$ points to $o_{ij\ldots pq}$, the system contains a constraint $ref(o_i, v_i, \overline{v_i}) \subseteq^{[X\times Y]} v_r$, where $[X \times Y]$ is an object annotation, such that strings X and Y represent contexts $o_{j\ldots pq}$ and $o_{k\ldots rs}$ respectively. In other words, annotated constraint $ref(o_i, v_i, \overline{v_i}) \subseteq^{[X\times Y]} v_r$ implies points-to edge $(r^{o_{k\ldots rs}}, o_{ij\ldots pq})$. Combining this constraint with the one generated for `l=r` will result in a constraint with an annotation that implies that $l^{o_{k\ldots rs}}$ points to $o_{ij\ldots pq}$ (intuitively, an empty annotation matches and preserves any other annotation). This reflects the semantics of the object-sensitive analysis described in Section 4. [4]

Constraints for Virtual Calls. For every virtual call the analysis generates a constraint according to the rule:

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

The *lam* (lambda) constructed term encapsulates the actual arguments and the left-hand side at the call site. Method annotation m corresponds to the unique *compile-time* target of the call. In order to model the semantics of the object-sensitive *resolve*, closure rule VIRTUAL from [33] is modified to handle

[4] A brief discussion on the handling of static variables appears later in this section. A more detailed discussion appears in [25].

object annotations. The new rule, denoted by OS-VIRTUAL is shown below. If object annotations s_1 and s_2 "match" in

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{s_1} v \qquad v \subseteq^{s_2}_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

the analysis consults a lookup table and based on the class of object o_i and the compile-time target m finds a lambda term $lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret})$ corresponding to the definition of the run-time target method. The analysis computes a new object annotation $[X \times Y]$ by concatenation of s_1 and s_2 (i.e., $[X \times Y] = concat(s_1, s_2)$). The analysis introduces the following constraints: [5]

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{[X \times i \oplus_k X]} v_{p0}$$
$$lam(\overline{v_{p0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) \subseteq^{[i \oplus_k X \times Y]} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

Intuitively, string X represents certain object contexts $o_{j...pq} \in \mathcal{C}$ for o_i (i.e., the contexts of invocation of the enclosing method of o_i). The callee should be analyzed in context $o_{ij...pq} \in \mathcal{C}$; this is modeled by adding site i in front of X.

Object Annotations. We now formally define the annotation language for object-sensitive analysis of depth k. The object annotations are tuples of strings of allocation site indices of the form $[i_1 i_2 ... i_m \times j_1 j_2 ... j_n]$, where $0 \leq m, n \leq k$, and if $m, n < k$, allocation sites s_{i_m} and s_{j_n} appear in the *same* method. Intuitively, the string on the left describes the object context of the term on the left of a constraint, and the string on the right describes the context of the term on the right. An annotation may represent *multiple* tuples of object contexts.

Consider constraint $v \subseteq^{[i_1 i_2 ... i_m \times j_1 j_2 ... j_n]} w$. The annotation may represent multiple tuples of object contexts as follows: if $m, n < k$, there is flow from $v^{o_{i_1 i_2 \ldots i_m \oplus_k xy \ldots z}}$ to $w^{o_{j_1 j_2 \ldots j_n \oplus_k xy \ldots z}}$ for every context $o_{xy...z} \in \mathcal{C}_m$, where m is the enclosing method of s_{i_m} and s_{j_n}. The annotations model sets of contexts that may be unknown at the time of constraint generation. For example, when the analysis processes statement 1 in Figure 4 the possible contexts of invocation of the statement are unknown and depend on the possible receivers of `A.A`. The analysis creates constraints $v_{A.this} \subseteq proj(ref, 3, u)$ and $v_{A.xa} \subseteq_f u$, where the empty annotation, denoted also by $[\ \times\ ]$, is used to represent flow for all possible contexts of `A.A`; this context information is refined during constraint resolution.

Consider the case when $k = 1$ (i.e., the object context is distinguished by allocation site); thus, $r \subseteq^{[i \times j]} l$ represents the fact that the points-to set of r^{o_i} is a subset of the points-to set of l^{o_j}. Similarly, $r \subseteq^{[\ \times j]} l$ represents the fact that for every object context o_i of the enclosing method of r the points-to set of r^{o_i} is a subset of the points-to set of l^{o_j}.

Operations on the Annotations. It remains to define $match(s_1, s_2)$, *concat*(s_1, s_2), and *transpose*(s).

$$match([i_1 \ldots i_m \times j_1 \ldots j_n], [k_1 ... k_p \times l_1 ... l_q]) \Rightarrow$$

[5] The operator $\oplus_k$ concatenates two strings and truncates the second one, such that the length of the resulting string does not exceed k (e.g., 12$\oplus_3$456=124).

$$\begin{cases} \text{true if } j_1...j_{min(n,p)} = k_1...k_{min(n,p)} \\ \text{false otherwise} \end{cases}$$

Consider the case $n < p$. Clearly, if $j_1...j_n$ is not a prefix of $k_1...k_p$, the constraint on the left and the constraint on the right represent respectively flow into, and flow from different context copies of the intermediate variable; thus, the two constraints cannot be combined to represent valid flow and *match* returns false. If $j_1...j_n$ is a prefix of $k_1...k_p$, the combination of constraints represents valid flow and *match* returns true. If *match* holds, the analysis performs concatenation:

$$concat([i_1...i_m \times j_1...j_n],[k_1...k_p \times l_1...l_q]) \Rightarrow$$
$$\begin{cases} [i_1..i_m \oplus_k k_{n+1}..k_p \times l_1..l_q] & \text{if } n < p \\ [i_1..i_m \times l_1..l_q \oplus_k j_{p+1}..j_n] & \text{if } n > p \\ [i_1..i_m \times l_1..l_q] & \text{if } n = p \end{cases}$$

Consider the case when $n < p$. In this case, $j_1...j_n$ is the more general context string, and $k_1...k_p$ is the more specific one. Therefore, the flow implied by the two constraints is valid only for context $k_{n+1}...k_p$ for the enclosing method of o_{j_n} and o_{i_m} and the context string $i_1...i_m$ is refined appropriately if $m < k$. To illustrate the idea of refinement, consider the case when $k = 2$ and the set of constraints $u \subseteq^{[i \times jk]} v \subseteq^{[j \times l]} w$. The meaning of the first constraint is that for every object context o_{xy} of the method enclosing s_i, $u^{o_{ix}}$ flows to $v^{o_{jk}}$. Similarly, the meaning of the second constraint is that for every context o_{yz} of the method enclosing s_j and s_l, $v^{o_{jy}}$ flows to $w^{o_{ly}}$; in particular this constraint implies that $v^{o_{jk}}$ flows to $w^{o_{lk}}$. Thus, for every enclosing context o_{xy} of s_i, we $u^{o_{ix}}$ flows to $w^{o_{lk}}$ which is reflected in the resulting constraint $u \subseteq^{[i \times lk]} w$. It remains to define *transpose*: $transpose([i_1...i_m \times j_1...j_n]) = [j_1...j_n \times i_1...i_m]$.

The analysis uses *wildcard* annotations of the form $[i_1 i_2...i_m * \times j_1 j_2...j_n *]$ to handle flow through non-replicated variables (e.g., static variables). The wildcard denotes the fact that the two context strings can be expanded arbitrarily for each context enclosing s_{i_m} and for each context enclosing s_{j_n}. Operations *match* and *concat* can be extended to handle wildcard annotations; this is achieved by restricting context refinement and propagating the wildcard.

Example. This example shows object-sensitive analysis of depth $k = 1$ on the set of statements in Figure 4. At line 6, the analysis creates constraints

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq v_b \subseteq lam_B(\overline{0}, \overline{v_y}).$$

Applying OS-Virtual results in the following constraints:

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[\ \times o_3]} v_{B.this} \qquad lam_B(\overline{v_{B.this}}, \overline{v_{B.xb}}) \subseteq^{[o_3 \times\ ]} lam_B(\overline{0}, \overline{v_y})$$

After applying the resolution rule for structural constraints in Figure 1 we have

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[\ \times o_3]} v_{B.this} \qquad v_y \subseteq^{[\ \times o_3]} v_{B.xb}$$

and after line 2 we have

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[\ \times o_3]} v_{B.this} \subseteq lam_A(\overline{0}, \overline{v_{B.xb}})$$

The analysis applies *concat* on $s_1 = [\ \times o_3]$ and $s_2 = [\ \times\ ]$; after appropriate context refinement of $[\ \times\ ]$ it infers that object o_3 flows to the invocation site denoted by the lambda term only when the invocation occurs in the context of o_3. Applying OS-VIRTUAL followed by the rules in Figure 1 results in the two constraints shown below:

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[\ \times o_3]} v_{A.this} \quad (1) \quad v_{B.xb} \subseteq^{[o_3 \times o_3]} v_{A.xa} \quad (2)$$

Applying the analogous sequence of rules on object o_4 at lines 7 and 3 results in

$$ref(o_4, v_{o_4}, \overline{v_{o_4}}) \subseteq^{[\ \times o_4]} v_{C.this} \quad v_z \subseteq^{[\ \times o_4]} v_{C.xc}$$

and subsequently in

$$ref(o_4, v_{o_4}, \overline{v_{o_4}}) \subseteq^{[\ \times o_4]} v_{A.this} \quad (3) \quad v_{C.xc} \subseteq^{[o_4 \times o_4]} v_{A.xa} \quad (4)$$

At line 1 the analysis generates the following constraints:

$$v_{A.this} \subseteq proj(ref, 3, u) \quad (5) \quad v_{A.xa} \subseteq_f u \quad (6)$$

Applying TRANS followed by the resolution rules for structural constraints to (1) and (5) results in $u \subseteq^{[o_3 \times\ ]} o_3$. Similarly, from (3) and (5) we have $u \subseteq^{[o_4 \times\ ]} o_4$. Clearly, the annotations are used to distinguish between flow from variable u in two separate object contexts. In the first case, the constraint represents flow from the context copy of u associated with object context o_3 and in the second case it represents flow from the context copy associated with o_4. Thus,

$$\begin{array}{l} v_y \subseteq^{[\ \times o_3]} v_{B.xb} \subseteq^{[o_3 \times o_3]} v_{A.xa} \subseteq_f u \subseteq^{[o_3 \times\ ]} v_{o_3} \\ v_z \subseteq^{[\ \times o_4]} v_{C.xc} \subseteq^{[o_4 \times o_4]} v_{A.xa} \subseteq_f u \subseteq^{[o_4 \times\ ]} v_{o_4} \end{array}$$

Clearly, $[o_4 \times o_4]$ and $[o_3 \times\ ]$ do not "match" and the analysis avoids inferring constraints $v_{C.xc} \subseteq_f v_{o_3}$ and subsequently $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_3}$. The last constraint erroneously implies a field edge between o_3 and o_2. Similarly, because $[o_3 \times o_3]$ and $[o_4 \times\ ]$ do not match, the constraint that implies a spurious field edge between o_4 and o_1 is avoided as well.

5.2 *k-CFA* Context-Sensitive Analysis

The *k-CFA* context-sensitive analysis is modeled similarly to the object-sensitive analysis of depth k. The *call-string annotations* are tuples of strings, where the strings are sequences of indices of *call sites* instead of sequences of indices of object allocation sites. Operations *match*, *concat*, and *transpose* are defined analogously to the operations on object annotations described in Section 5.1.

The *k-CFA* analysis defines a new closure rule denoted by CFA-VIRTUAL. If call-string annotations s_1 and s_2 "match" in

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{s_1} v \qquad v \subseteq^{s_2}_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

the analysis consults a lookup table and based on the class of object o_i and the compile time target m finds a new lambda term $lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret})$, which corresponds to the definition of the run-time target method. Let c_i be the

call site corresponding to lambda term $lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$ and let $[X \times Y] = concat(s_1, s_2)$. The following constraints are introduced.

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{[X \times i \oplus_k Y]} v_{p_0}$$
$$lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) \subseteq^{[i \oplus_k Y \times Y]} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

Intuitively, the annotation Y represents certain call chains $j...pq \in \mathcal{C}$ for the enclosing method of the call c_i. The callee should be analyzed in each context $ij...pq$; this is achieved by adding the call site index i to Y.

Example. This example shows the application of 1-CFA analysis on the set of statements in Figure 4. At line 6, the analysis creates constraints:

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq v_b \subseteq lam_B(\overline{0}, \overline{v_y}).$$

Applying CFA-Virtual results in the following constraints:

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[\ \times c_6]} v_{B.this} \quad lam_B(\overline{v_{B.this}}, \overline{v_{B.xb}}) \subseteq^{[c_6 \times\ ]} lam_B(\overline{0}, \overline{v_y})$$

After applying the resolution rule for structural constraints in Figure 1 we have

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[\ \times c_6]} v_{B.this} \quad v_y \subseteq^{[\ \times c_6]} v_{B.xb}$$

and after line 2 we have

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[\ \times c_6]} v_{B.this} \subseteq lam_A(\overline{0}, \overline{v_{B.xb}})$$

Applying CFA-Virtual followed by the rules in Figure 1 results in

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[\ \times c_2]} v_{A.this} \ (1') \quad v_{B.xb} \subseteq^{[c_6 \times c_2]} v_{A.xa} \ (2')$$

At line 1 the analysis generates the following constraints:

$$v_{A.this} \subseteq proj(ref, 3, u) \ (3') \quad v_{A.xa} \subseteq_f u \ (4')$$

Applying Trans followed by the resolution rules for structural constraints to $(1')$ and $(3')$ results in $u \subseteq^{[c_2 \times\ ]} o_3$. Subsequently, from $(2')$ and $(4')$, the analysis infers a constraint that implies a points-to edge $(\langle o_3, f \rangle, o_1)$ and similarly, it infers $(\langle o_4, f \rangle, o_2)$. It avoids infeasible field edges $(\langle o_4, f \rangle, o_1)$ and $(\langle o_3, f \rangle, o_2)$.

6 Parameterized Context Sensitivity

In this section we define a parameterization mechanism for context-sensitive analysis. It encompasses a family of analyses that range from the least precise and least costly context-insensitive Andersen's analysis to the most precise and costly analyses in Section 4.

The parameterization is in four dimensions. First, the analysis designer can select the degree of precision in the naming scheme for *object names*. This is done by specifying a set of object allocation sites for which the context-sensitive analysis maintains multiple names. For the rest of the sites, it maintains a single object name. Second, the analysis designer can specify the set of *reference variables* for which multiple points-to sets should be maintained. The analysis

replicates only these selected variables. Third, the analysis designer can specify the degree of precision in the *context names*. In addition to using parameter k to control the degree of precision in the contexts naming scheme, the designer can define a transformation function σ that maps the set of indices $\mathcal{I}$ used to form the context strings, into a smaller set $\mathcal{I}'$. One useful transformation for the object-sensitive analysis is to map certain allocation site indices into indices corresponding to the instantiated class (i.e., making the analysis *class-sensitive*). Fourth, the analysis designer can select a set of call sites for which *context-insensitive propagation* is applied (i.e., information is propagated to the formal parameters of the callee from *every* context of the caller, and information is propagated from the return variable of the callee to every context of the caller).

The goal of the parameterization is to enhance the flexibility of the context-sensitive analysis. By varying the parameters, the analysis designer can control directly the cost of the analysis. Furthermore, it allows *targeted* context sensitivity. Instead of using the global non-discriminatory context sensitivity presented in Section 4, the analysis designer can choose objects, variables, contexts and call sites for which keeping more precise information is likely to improve the points-to solution.

The parameterizations can be easily modeled by choosing appropriate constraint annotations. The first two dimensions can be achieved by using wildcard annotations analogously to the way the analysis models flow into and from static variables. The last two dimensions can be modeled by modifying the closure rule for virtual calls appropriately. In order to handle the parameterization for context names, the context of the callee becomes $\sigma(i) \oplus_k X$ instead of $i \oplus_k X$. To model context-insensitive propagation, the analysis uses wildcard annotations. Thus, if the call site modeled by term $lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$ appears in the set of insensitively handled calls, the analysis generates constraints

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{[X \times \sigma(i) \oplus_k X]} v_{p_0}$$
$$lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) \subseteq^{[\sigma(i) \oplus_k X * \times *]} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

7 Empirical Results

We performed experiments on 23 realistic publicly available Java programs. [6] Table 1 shows some characteristics of the data programs. The first two columns show the number of user (i.e., non-library) classes and their bytecode size. The next three columns show the size of the program, including library classes, after using class hierarchy analysis (CHA) [11] to filter out irrelevant classes and methods. The number of methods is essentially the number of nodes in the call graph computed by CHA. The last column shows the number of statements in the intermediate representation.

The constraint-based implementation is based on the Bane analysis toolkit (`bane.cs.berkeley.edu`) for program analysis using non-annotated inclusion constraints [3]. We modified the constraint engine to handle parameterizations

[6] We used the same set of data in our earlier work [33, 26].

Program	User Class	Size (Kb)	Whole-program Class	Method	Stmt
proxy	18	56.6	565	3283	58837
compress	22	76.7	568	3316	60010
db	14	70.7	565	3339	60747
jb-6.1	21	55.6	574	3393	60898
echo	17	66.7	577	3544	62646
raytrace	35	115.9	582	3451	62755
mtrt	35	115.9	582	3451	62760
jtar-1.21	64	185.2	618	3583	65112
jlex-1.2.5	25	95.1	578	3381	65437
javacup-0.10	33	127.3	581	3564	66463
rabbit-2	52	157.4	615	3770	68277
jack	67	191.5	613	3573	69249
jflex-1.2.2	54	198.2	608	3692	71198
jess	160	454.2	715	3973	71207
mpegaudio	62	176.8	608	3531	71712
jjtree-1.0	72	272.0	620	4078	79587
sablecc-2.9	312	532.4	864	5151	82418
javac	182	614.7	730	4470	82947
creature	65	259.7	626	3881	83454
mindterm1.1.5	120	461.1	686	4420	90451
soot-1.beta.4	677	1070.4	1214	5669	92521
muffin-0.9.2	245	655.2	824	5253	94030
javacc-1.0	63	502.6	615	4198	102986

Table 1. Characteristics of the programs. First two columns show the number and bytecode size of user classes. Last three columns include library classes.

by possibly multiple sets of annotations with corresponding operations. We process Java bytecode and build a typed intermediate representation using Soot (`www.sable.mcgill.ca`) [42].

We implemented various context-sensitive analyses using the framework for annotated constraints and chose to present three object-sensitive analyses. The first one is essentially the object-sensitive analysis of depth one. The second and third analyses explore the design space by employing class sensitivity in the context naming scheme, and context-insensitive propagation respectively. These instantiations denoted by *ObjSens*, *CsObjSens* and *CiObjSens* respectively, were compared with Andersen's context-insensitive analysis (denoted by *And*).

CsObjSens and *CiObjSens* particularly target large applications with wide and deep inheritance hierarchies (typically compilers and rule engines such as `javac`, `soot` and `jess`). These applications make use of complex composite objects that are interpreted in different ways (i.e., they employ the Composite, Interpreter and Visitor design patterns [19]). The purpose of the wide and deep hierarchies and the patterns is to allow the creation and interpretation of arbitrarily complex objects; therefore, analyzing methods that create and inter-

pret composite structures object-sensitively, is likely to increase analysis cost at no precision gain. *CsObjSens* targets large programs [7] that are likely to make use of composite objects by applying the following heuristic: if the program contains more than fifty allocation sites of a class X, all methods with receivers o_i of type X are analyzed class-sensitively per X rather than per o_i (i.e., we have $\sigma(i){=}x$, where x is a unique identifier corresponding to X).

For example, `javac` has a class `UnaryExpression` at the base of a large hierarchy of expressions. The constructor of `UnaryExpression` contains the following code: `UnaryExpression(Expression e) { this.right = e; }`. The semantics of `javac` is that almost any expression object can be the `right` of any other expression object and analyzing this code object-sensitively results in substantial work and no precision gain. The heuristic in *CsObjSens* reduces such redundant work by analyzing methods and constructors in the expression hierarchy class-sensitively. As a result, the `UnaryExpression` constructor is analyzed only once, as opposed to once for each kind of expression object.

CiObjSens independently targets the interpretation of complex objects by making use of context-insensitive propagation. Consider the sequence of statements `e = this.f; e.m(c);` which appears in an interpreter method `m` early in a hierarchy. In a part-whole hierarchy, typically any object o_i can be the `f` of any other object o_j, and the *same* information `c` is propagated to context copy p^{o_i} of formal p once for each o_j; it is sufficient to propagate `c` to p^{o_i} once. *CiObjSens* uses the following heuristic: if a call is not made through implicit parameter `this`, the parameter information is propagated context-insensitively (i.e., the analysis creates one constraint $v_c \subseteq^{[* \times o_i]} v_p$ instead of many $v_c \subseteq^{[o_j \times o_i]} v_p$).

Sections 7.1 and 7.2 show that using class sensitivity and context-insensitive propagation results in analysis that has cost comparable to *And* and precision comparable to *ObjSens* in practice.

7.1 Analysis Cost

Table 2 shows the running times and memory usage of *And*, *CiObjSens*, *CsObjSens* and *ObjSens*. For more than half of our programs even the most precise and costly *ObjSens* performs comparably to *And*. Not surprisingly, *ObjSens* does not perform well on compilers and rule engines (e.g., `javac,sablecc,soot` and `jess`)[8]; these programs make heavy use of deep and wide hierarchies and complex recursive object structures. *CsObjSens* applies class sensitivity to `jess`, `javac` and `soot` and is able to reduce substantially analysis cost. *CiObjSens* is able to additionally reduce analysis cost to amounts comparable to the cost of *And*, for the majority of programs. The practicality of *CiObjSens* makes it a realistic candidate for use in optimizing compilers and software tools.

7.2 Analysis Precision

Virtual Call Resolution and Call Graph Construction. One fundamental application of points-to analysis, with many uses in optimizing compilers and

---

[7] A program that has more than five hundred allocation sites of user classes.

[8] For two of the programs, *ObjSens* did not terminate in five hundred seconds which was the acceptable upper bound we set for the analysis to complete.

Program	*And*		*CiObjSens*		*CsObjSens*		*ObjSens*	
	Time (sec)	Mem (Mb)	Time (sec)	Mem (Mb)	Time (sec)	Mem (Mb)	Time (sec)	Mem (Mb)
proxy	4.8	35.1	5.3	34.8	4.1	35.1	4.1	35.1
compress	8.3	39.6	10.1	40.1	12.6	43.7	12.6	43.7
db	9.2	40.6	10.6	42.5	14.3	42.6	14.3	42.6
jb	6.0	36.7	5.8	36.9	6.5	37.1	6.5	37.1
echo	18.7	49.2	44.9	66.2	51.8	69.4	51.8	69.4
raytrace	7.8	42.2	10.8	46.1	12.7	46.8	12.7	46.8
mtrt	9.4	42.1	11.3	46.2	12.8	46.8	12.8	46.8
jtar	16.8	50.3	24.4	58.9	28.1	57.7	28.1	57.7
jlex	6.7	39.8	7.3	40.6	7.5	40.6	7.5	40.6
javacup	23.2	55.8	21.2	58.5	21.5	58.8	21.5	58.8
rabbit	9.1	46.2	11.7	45.6	12.8	47.9	12.8	47.9
jack	28.7	54.8	24.9	56.7	62.9	65.4	62.9	65.4
jflex	28.5	63.5	30.3	66.4	29.9	67.4	29.9	67.4
jess	35.8	59.4	87.5	79.6	-	-	-	-
mpegaudio	11.6	44.0	10.4	48.4	13.7	50.1	13.7	50.1
jjtree	8.6	46.8	32.1	64.4	46.2	65.8	46.2	65.8
sablecc	34.5	78.5	51.2	75.3	404.1	102.6	404.1	102.6
javac	100.5	110.0	168.5	129.0	180.5	133.4	-	-
creature	64.3	94.3	105.5	124.8	109.2	126.5	109.2	126.5
mindterm	37.2	78.5	51.5	90.5	61.6	93.9	61.6	93.9
soot	139.4	117.8	115.9	117.9	244.6	144.0	356.6	170.0
muffin	120.7	133.9	115.1	149.7	170.1	165.1	170.1	165.1
javacc	99.6	96.6	93.4	101.9	101.8	102.0	101.8	102.0

Table 2. Running time and memory usage of the analyses.

software tools is virtual call resolution and call graph construction.

To determine the improvements from object-sensitive analysis in the number of resolved calls we considered the set V of CHA-unresolved calls which appear in methods reachable by *ObjSens*. We computed the number of sites from V that were resolved to a single target, according to *And*, *CiObjSens*, *CsObjSens* and *ObjSens*. The improvement in the number of resolved call sites for *CiObjSens*, *CsObjSens* and *ObjSens* over *And* is shown in the second, fourth, and sixth columns of Table 3 respectively. On average, all three object-sensitive analyses resolve 26% more sites than *And*. This increased precision allows better removal of redundant run-time virtual dispatch and enables additional method inlining.

We also computed the sum (over all sites in V) of the number of target methods according to *And*, as well as the corresponding sum according to *CiObjSens*, *CsObjSens* and *ObjSens*. The reduction in the total number of target methods (i.e., call edges removed at call sites) is shown in the third, fifth and seventh columns of Table 3 respectively. On average, all three object-sensitive analyses remove 21% of the target methods determined by *And*. This improved precision (i.e., fewer call edges) is beneficial for reducing the cost and improving the

Program	*CiObjSens*		*CsObjSens*		*ObjSens*	
	Resolved calls	Removed targets	Resolved calls	Removed targets	Resolved calls	Removed targets
proxy	10%	3%	10%	3%	10%	3%
compress	23%	17%	23%	17%	23%	17%
db	19%	18%	19%	18%	19%	18%
jb	68%	10%	68%	10%	68%	10%
echo	12%	16%	13%	16%	13%	16%
raytrace	21%	17%	21%	17%	21%	17%
mtrt	21%	17%	21%	17%	21%	17%
jtar	43%	8%	43%	8%	43%	8%
jlex	53%	6%	53%	6%	53%	6%
javacup	32%	7%	32%	7%	32%	7%
rabbit	35%	12%	35%	12%	35%	12%
jack	5%	16%	5%	16%	5%	16%
jflex	15%	4%	15%	5%	15%	5%
jess	20%	21%	-	-	-	-
mpegaudio	24%	21%	24%	21%	24%	21%
jjtree	63%	7%	63%	7%	63%	7%
sablecc	52%	221%	52%	221%	52%	221%
javac	7%	11%	7%	11%	-	-
creature	19%	5%	19%	5%	19%	5%
mindterm	6%	10%	8%	10%	8%	10%
soot	10%	9%	11%	7%	11%	7%
muffin	4%	14%	4%	14%	4%	14%
javacc	16%	6%	16%	6%	16%	6%
Average	26%	21%	26%	21%	25%	21%

Table 3. Improvements over context-insensitive analysis: Resolved calls shows the increase in the number of resolved call sites. Removed targets shows the reduction in the number of target methods.

precision of subsequent interprocedural analyses.

Proving Downcast Safety. Consider a downcast expression `(T)y`. Points-to information can be used to determine that `y` may point only to objects of classes that are subclasses of `T`; in this case, the downcast is safe [27].

In order to determine the impact of object-sensitive analysis on downcast safety, we computed the number of downcasts in methods reachable by *ObjSens*. The percentage of safe downcasts determined by *And*, *CiObjSens*, *CsObjSens* and *ObjSens*, is shown in the second, third, fourth, and fifth columns of Table 4 respectively. On average, all three object-sensitive analyses resolve nearly 50% of the downcasts while *And* proves safe only 17%. Presently, Java lacks parametric polymorphism and many downcasts occur at container objects. The object-sensitive analyses are able to distinguish information stored in different containers, while *And* merges information for all possible containers of the same class (and its subclasses). The improved precision may result in better removal of

Program	*And*	*CiObjSens*	*CsObjSens*	*ObjSens*
proxy	21%	60%	60%	60%
compress	20%	59%	59%	59%
db	17%	63%	65%	65%
jb	12%	39%	39%	39%
echo	17%	40%	40%	40%
raytrace	19%	68%	68%	68%
mtrt	19%	68%	68%	68%
jtar	16%	41%	41%	41%
jlex	19%	69%	69%	69%
javacup	9%	83%	84%	84%
rabbit	17%	52%	52%	52%
jack	14%	67%	67%	67%
jflex	4%	60%	60%	60%
jess	17%	67%	-	-
mpegaudio	20%	57%	59%	59%
jjtree	61%	75%	75%	75%
sablecc	32%	45%	45%	45%
javac	11%	34%	34%	-
creature	17%	32%	32%	32%
mindterm	23%	44%	45%	45%
soot	17%	25%	26%	26%
muffin	12%	32%	32%	32%
javacc	6%	56%	56%	56%
Average	17%	49%	48%	49%

Table 4. Percentage of safe downcasts.

run-time checks for `ClassCastException` [17] and more efficient application of standard optimizations such as code motion and partial redundancy elimination.

Due to space constraints we present empirical results only with respect to virtual call resolution, call graph construction and downcast safety. Experiments with respect to *side-effect analysis* can be found in [25]; they demonstrate that (i) *CiObjSens* is as precise in practice as *ObjSens*, and (ii) object sensitivity has substantial impact on the precision of side-effect analysis. Our precision experiments show that class sensitivity and context-insensitive propagation appropriately target sources of analysis complexity while preserving analysis precision.

8 Related Work

Constraint-based Analysis. Efficient constraint-based implementations of Andersen's points-to analysis for C are presented in [15, 39] and [22]. These analyses are based on non-annotated constraints and do not model field sensitivity and context sensitivity. Our work extends the work from [15, 39] by introducing constraint annotations. The framework for annotated constraints allows the precise and efficient modeling of dimensions of precision such as field sensitivity,

virtual dispatch, and various different forms of context sensitivity.

Foster et al. present a context-sensitive points-to analysis for C based on Andersen's points-to analysis using non-annotated inclusion constraints [18]. They conclude that (i) using an implementation based on constraint copying may result in impractical context-sensitive analysis and (ii) context sensitivity does not improve the precision of Andersen's analysis for C. Our experiments show that for object-oriented programs, field and context sensitivity are necessary to achieve useful points-to analyses because of inherent object-oriented features.

Das et al. present unification-based context-sensitive points-to analysis for C with directional assignments [10] based on the context-insensitive analysis for C from [9]. Similarly to [18], one of their conclusions is that context sensitivity does not improve the precision of the analysis from [9]. Our work shows that for object-oriented programs context sensitivity improves analysis precision and therefore it is important to investigate approaches for practical computation of context-sensitive points-to information.

Constraint indices and constraint polarities have been used to introduce context sensitivity in flow analysis for C based on unification and structural constraints, in the important work by Fähndrich et al. [14, 16]. This work uses annotations in similar way to our use of annotation for tracking flow of values through object fields and context copies of variables. However, our framework is entirely based on non-structural inclusion constraints and addresses problems in the analysis of object-oriented languages. In addition, our framework models different dimensions of precision (e.g., field sensitivity, virtual dispatch, as well as functional and call-string context sensitivity) while [14] and [16] model only summary-based functional context sensitivity.

The work most closely related to ours is O'Callahan's thesis work on context-sensitive alias analysis for Java [27]. The analysis is unification-based and similar to [14, 16]. However, it seems not to scale well. Our analyses are inclusion-based and the experiments indicate that they may be more practical.

Points-to Analysis for Java. Ruf presents flow-insensitive, context-sensitive alias analyses for Java [34, 35]. The analyses are based on the almost-linear unification-based Steensgaard's points-to analysis for C, and unlike our analyses, uses bottom-up traversal on an approximate call graph, and method summaries to model context sensitivity. Other context-sensitive points-to analyses for Java are presented in [21, 8]. In general, these analyses are more precise and significantly more costly than ours, which is due to their flow sensitivity. Flow-insensitive and context-insensitive points-to analyses for Java are described in [32, 38, 24, 33, 43, 23, 7].

Class Analysis. Different mechanisms for context sensitivity which typically abstract context by some combination of the parameter class types have been studied in the context of class analysis [28, 1, 30, 2, 21, 31]. Although *ObjSens* can be expressed in the general framework from [21], it is not identified or studied in [21]. To the best of our knowledge, the *CiObjSens* analysis is novel; it is unclear whether it can be expressed as an instance of [21]. Various practical context-insensitive class analyses are presented in [29, 13, 6, 12, 41, 40].

9 Conclusions

We have presented annotated inclusion constraints—a general framework that enables the modeling of different dimensions of flow analysis precision using inclusion constraint systems. We have shown that using the framework, a variety of relatively precise flow analyses can be expressed precisely and implemented efficiently. The practicality and the precision of analyses based on annotated inclusion constraints makes them realistic candidates for use in software tools and optimizing compilers.

References

1. O. Agesen. Constraint-based type inference and parametric polymorphism. In *SAS*, LNCS 864, pages 78–100, 1994.
2. O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.
3. A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Workshop on Types in Compilation*, pages 78–96, 1998.
4. A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *FPCA*, pages 31–41, 1993.
5. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
6. D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA*, pages 324–341, 1996.
7. M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDD's. In *PLDI*, pages 103–114, 2003.
8. R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.
9. M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.
10. M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, pages 260–278, 2001.
11. J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, 1995.
12. G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *POPL*, pages 222–236, 1998.
13. A. Diwan, J. B. Moss, and K. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *OOPSLA*, pages 292–305, 1996.
14. M. F˙ Technical report.
15. M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, 1998.
16. M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, pages 253–263, 2000.
17. R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experientce*, 30(3):199–232, 2000.
18. J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS*, LNCS 1824, pages 175–198, 2000.

19. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
20. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
21. D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *OOPSLA*, pages 108–124, 1997.
22. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI*, pages 254–264, 2001.
23. O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.
24. D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, 2001.
25. A. Milanova. *Precise and Practical Flow Analysis of Object-Oriented Software*. PhD thesis, Rutgers University, Aug. 2003.
26. A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, pages 1–12, 2002.
27. R. O'Callahan. *The Generalized Aliasing as a Basis for Software Tools*. PhD thesis, Carnegie Mellon University, 2000.
28. N. Oxhoj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *ECOOP*, pages 329–349, 1992.
29. J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, 1991.
30. J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, pages 324–340, 1994.
31. C. Probst. Modular control flow analysis for libraries. In *SAS*, pages 165–179, 2002.
32. C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, Dec. 1999.
33. A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.
34. E. Ruf. Effective synchronization removal for Java. In *PLDI*, pages 208–218, 2000.
35. E. Ruf. Improving the precision of equality-based dataflow analyses. In *SAS*, pages 247–262, 2002.
36. B. G. Ryder. Dimensions of precision in reference analysis of object-oriented languages. In *CC*, pages 126–137, 2003.
37. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.
38. M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an emprirical comparison. Technical report, U. Passau, Sept. 2000.
39. Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *POPL*, pages 81–95, 2000.
40. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *OOPSLA*, pages 264–280, 2000.
41. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, 2000.

42. R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, LNCS 1781, pages 18–34, 2000.
43. J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *SAS*, pages 180–195, 2002.