Tobias Gutzmann

Towards a Gold Standard for Points-to Analysis

Licentiate Thesis

Computer Science

2010

Linnaeus University
A thesis for the Degree of Licentiate of Philosophy in Computer Science.

Towards a Gold Standard for Points-to Analysis
Tobias Gutzmann

Linnaeus University
School of Computer Science, Physics and Mathematics
SE-35195 Växjö, Sweden
http://www.lnu.se/
Abstract

Points-to analysis is a static program analysis that computes reference information for a given input program. It serves as input to many client applications in optimizing compilers and software engineering tools. Unfortunately, the Gold Standard – i.e., the exact reference information for a given program – is impossible to compute automatically for all but trivial cases, and thus, little can been said about the accuracy of points-to analysis.

This thesis aims at paving the way towards a Gold Standard for points-to analysis. For this, we discuss theoretical implications and practical challenges that occur when comparing results obtained by different points-to analyses. We also show ways to improve points-to analysis by different means, e.g., combining different analysis implementations, and a novel approach to path-sensitivity.

We support our theories with a number of experiments.

Key-words: Points-to Analysis, Dataflow Analysis, Static Analysis, Dynamic Analysis, Gold Standard
This thesis is based on the following refereed publications:


Acknowledgments

First, I would like to thank my supervisors, Welf Löwe and Jonas Lundberg. Without them, this thesis would not exist.

A great amount of gratitude goes to my parents and my siblings, who have supported me all these years through my studies. Further, thanks to all my friends who I so often needed to put off in favor of work. Good that we are still friends!

Last but not least, I would like to thank Therése, who has also supported me a lot during the past years, and who has (almost) never complained about long working nights.
### Contents

6 Evaluation 58  
6.1 Experimental Setup .......................... 58  
6.2 Comparing, Combining, and Improving Analyses .................. 61  
6.3 Evaluating the Points-to SSA Front-end .................. 71  
6.4 Summary .................................... 72  

7 Conclusion 74  
7.1 Review of the Goals and Goal Criteria .................. 76  

8 Future Work 78  
8.1 Improving Points-to Analysis Precision .................. 78  
8.2 Extensions of the Comparison Platform .................. 79  
8.3 Tools for Supporting Computing the Gold Standard ....... 79  

A Points-to SSA Node Types 82  
A.1 Memory-related operations .................. 82  
A.2 Exception handling .................. 84  
A.3 Other intra-procedural node types .................. 84  
A.4 Inter-procedural node types .................. 85  

B The Exchange Format Schema 89  

C Evaluation: The Complete Metrics Data 94
Chapter 1

Introduction

Points-to analysis is a static program analysis that extracts reference information from a given input program, e.g., possible targets of a call and possible objects referenced by a field. Such information is essential input to many client applications in optimizing compilers and software engineering tools.

However, little can be said about the absolute precision of points-to analysis, as there is no Gold Standard for it. Unfortunately, such a Gold Standard seems to be impossible to compute automatically, and, if determined by human insight, very time consuming to create, and likely only possible for small programs, i.e., not for real-life programs.

Additionally, even comparing the relative precision of two different points-to analyses is difficult, as there are no standardized means to perform such a comparison; in current research, different authors use different metrics and benchmark programs for evaluating their approaches to points-to analysis.

1.1 Goals

This thesis aims at preparing the way towards such a Gold Standard. Besides requiring very precision points-to analysis as an approximation of the Gold Standard, this also demands standardized means to compare different points-to analysis implementations; otherwise, it would not be possible to compare an analysis with the (so far hypothetical) Gold Standard either.

The goals for this thesis are thus as follows:

1. Develop a framework that allows to create (and incrementally improve) very precise points-to analyses.

2. Develop a framework that allows to compare different points-to analysis instantiations.

1.2 Restrictions

We choose to base our framework on our Points-to SSA based implementation. This is because it already provides fast, precise, flow- and context-
Chapter 1. Introduction

sensitive points-to analyses, and thus provides a good starting point for our work.

This limits the analyzed target programming language to Java. We expect that our findings can be applied to other object-oriented programming languages as well, given that enough resources are put into the effort to adapt and implement them. For example, it would be required to find a proper handling for delegates in C#.

1.3 Goal Criteria

The criteria for fulfilling our first goal are:

1. Points-to SSA is to be extended with additional node types that can improve analysis precision. Means to further enhance Points-to SSA in the future shall also be provided. The underlying analysis must be adaptable without too much effort.

2. Analysis speed is a subordinated goal criteria. The analysis should run in adequate time, i.e., not days, but does not need to be usable in an edit-compile-cycle.

The criteria for fulfilling our second goal are:

1. The theoretical implications when comparing and combining two analyses have to be evaluated.

2. Comparison of metrics commonly used for evaluating points-to analysis, regardless of the analysis technique, shall be made possible. We see this criterion fulfilled if we can compare at least two different implementations - our own and a widely accepted third-party implementation - as well as results obtained from dynamic analysis.

3. Tools are to be provided in order to automatically compare the results of these analyses. Additionally, the results obtained from different points-to analysis implementations shall be combined automatically.

1.4 Tasks

For our first goal, creating a flexible points-to analysis infrastructure that can be extended over time, the following tasks need to be performed:

1. A front-end able to generate Points-to SSA of different abstraction levels has to be created. That is, the front-end has to be flexible, so it can be increasingly improved with further extensions that allow to capture
more of a program’s original semantics, thus allowing analyses of higher precision.

2. A number of example extensions are to be implemented into the front-end, and backed by the actual points-to analysis.

For our second goal, it is required to analyze the requirements for a framework for comparing different points-to analysis implementations and variants. The following tasks need to be performed:

1. Define an exchange format that is suitable to capture results obtained by different points-to analysis implementations. This requires to find a common denominator of the different implementations, i.e., data that can be obtained from different implementations.

2. Implement the tools necessary to compare and combine points-to analysis information based on this exchange format.

3. Connect different points-to analysis implementations to this exchange format, in order to show its applicability.

1.5 Motivation

Reference information computed by points-to analysis is an essential input to many types of client applications in optimizing compilers and software engineering tools. Examples of such client applications are: metrics analyses computing coupling and cohesion between objects [8, 16] and architectural recovery by class clustering proposing groupings of classes, either based on coupling and cohesion or directly on reference information [49, 43]. Source code browsers compute forward and backward slices [19] of a program point which, in turn, requires reference information. In software testing, class dependencies determine the test order [6, 50, 32]. Reverse engineering of UML interaction diagrams requires very precise reference information in order to be useful [51]. Finally, static design pattern detection needs to identify the interaction among participating classes and object instances in order to exclude false positives [44].

The more precise the underlying points-to analysis is, the better becomes the client application. A Gold Standard for points-to analysis would thus also get the best out of these client applications, showing their full potential in their application areas.

The only attempt known to us towards creating a Gold Standard for an analysis related to points-to analysis – computing exact call chains for a given project – was performed by Rountev et al. [41, 40]. Here, the authors took a lower bound for the call chains as obtained by a dynamic analysis, and an
upper bound as obtained from a static (conservative) analysis. Then, the authors either created input for the dynamic analysis which creates a call chain that is present in the static analysis, or they tried to prove that a given call chain is infeasible. Outgoing from an under- and over-approximation obtained by dynamic and static analysis, respectively, each element in the difference between the over- and under-approximation was inspected manually. The authors did not have any specialized tool for supporting this task at hand.

The amount of time required for such an approach is obviously dependent on the quality of both the dynamic and the static analysis: The closer both the under- and the over-approximations are together, the less work is left to do afterwards. For the dynamic analysis, this must be achieved by better test-case input. The static analysis results improve with higher-precision algorithms. Both approaches can, in general, never define the Gold Standard by themselves: A program cannot be fed with all possible input (there is infinitely much), and a static analysis needs to perform certain abstractions, which leads to imprecision (again, otherwise it would compute the exact result for every possible input).

Therefore, in order to create the Gold Standard, tool support for performing proofs for the absence of certain points-to information is desirable. We expect that specialized tools can increase the efficiency and accuracy of a human performing the given task. A fundamental prerequisite is, however, to have an as precise as possible points-to analysis as a basis, which reduces the (semi-)manual work that has to be done in later steps.

Both goals of this thesis work together on the effort to create a result for points-to analysis “as precise as possible”, thus working towards the final goal to create a Gold Standard, which is required to accurately benchmark points-to analysis. The contribution of Goal 1 - in whatever way it is achieved - is obvious. The contribution of Goal 2 is not as obvious, but not less valuable: By comparing two points-to analyses qualitatively, i.e., not only the sizes of result sets, but, e.g., the concrete nodes and edges in a call graph, they can also be combined to a more precise points-to analysis; this is of value when two analyses are not strictly ordered in terms of precision.

1.6 Thesis Outline

The remainder of this thesis is as follows. In Chapter 2, we discuss concepts of program analysis, with a strong focus on points-to analysis. We also discuss literature that compares dynamic and static analysis. Concrete points-to analysis implementations, as well as a tool that captures points-to information dynamically, are presented in Chapter 3. The theoretical implications when comparing, combining, and improving dataflow analysis are discussed
in Chapter 4. In Chapter 5, we present tools and methods to perform these task, as well as solutions to challenges that arose when evaluating these tools and methods. We also present a new front-end to Points-to SSA which enables us to create different versions of Points-to SSA that yield higher analysis precision. The results of experiments which aim at showing the applicability of our tools and methods are presented in Chapter 6. We evaluate these experiments with respect to the theories from Chapter 4. Chapter 7 concludes this thesis, and in Chapter 8 we discuss future steps which are required to eventually create a Gold Standard for points-to analysis.
Chapter 2

Background

In this chapter, we discuss approaches to program analysis in general and points-to analysis in special.

First, we present general program analysis concepts in Section 2.1, followed by an overview of program representations targeted at dataflow analysis in Section 2.2. Then, we conclude this chapter with an overview of approaches to and concepts of points-to analysis (Section 2.3). These are then referenced in the next chapter, where we describe concrete points-to analysis implementations.

2.1 Program Analysis

Fundamental concepts

There are two ways to formulate a program analysis question: (1) What facts must hold for a given program (must-analysis), and (2) what facts may hold for a given program (may-analysis). Then, there are two fundamental approaches to solve such a program analysis question: (a) Conservative and (b) optimistic analysis. Intuitively, a conservative analysis will be careful when making statements about a program in case of uncertainties, while an optimistic analysis assumes to know the whole truth (even if it does not). In the following, we shortly discuss all four possible combinations of these two concepts.

1a A conservative must-analysis makes statements about a given program only if these statements are 100% guaranteed; thus, a correct (yet not useful) answer to a given analysis question by this kind of analysis is not to say anything at all.

1b An optimistic must-analysis, on the other hand, reports facts about a program that it cannot prove wrong (but, on the other hand, cannot prove right either).

2a A conservative may-analysis aims at excluding facts about a program that it can prove as not being possible. A conservative (yet, again, not useful) answer to a “may”-question is that everything is possible in the
program. In this case, we also speak of *over-approximation* of the exact answer.

**2b** An optimistic may-analysis, then, answers a given analysis question by collecting a facts that are definitely true. However, other facts that are not found by the analysis may also hold. In this case, we speak of *under-approximation* of the exact answer.

Since points-to analysis is formulated as a “may”-analysis, we omit “must”-analyses in the following.

As an example, consider the following program analysis question: What values may a given integer variable \( v \) assume during an execution of a given input program? This question can, in general, not be answered, as otherwise the halting problem would be solvable as well\(^1\). An optimistic analysis could now construct cases in which \( v \) assumes values from 5..10, while a conservative analysis could prove that \( v \) can not assume values bigger than 20 or smaller than 3. The exact answer must then lie somewhere in between.

### Static vs. Dynamic Analysis

There are two fundamental approaches to program analysis: *static* and *dynamic* analysis.

The former analyzes a program without actually executing it, i.e., independent of a given input\(^2\). The latter monitors concrete executions of the program under given inputs in order to collect information.

Consider our problem from above – what values may a given variable assume during any program run. Then, a static analysis might analyze which statements can influence this variable, find constraints to these statements (e.g., what values other variables influencing the variable in question may be assigned), and finally approximate the solution to the problem. Static analysis usually, but not necessarily, results in an over-approximation of the result and is thus usually considered to be conservative.

On the other hand, the results of a dynamic analysis are valid for the analyzed runs in question, but cannot be generalized. For example, a dynamic analysis solving the same analysis question could simply record the values that the analysis is being assigned; however, since the program can be monitored under all possible input only in trivial cases, this will generally lead to an under-approximation of the analysis problem. Dynamic analysis is thus always an optimistic analysis.

---

\(^1\) Assign a global variable the value 0 at program start, and before any point the program can terminate put a statement that assigns the variable the value 1. Otherwise, the variable is not used. If an analysis proves that the variable may assume the value 1, then the halting problem is solved.

\(^2\)Input refers here to, for example, files read from the hard drive, arguments given on the command line, and also input given by a user via mouse or keyboard.
Approaches to Static Analysis

There are different approaches to static program analysis, e.g., constraint-based approaches \cite{38, 3} and dataflow analysis. We describe only the latter, as the concrete points-to analysis implementations that we present in the next chapter are dataflow analysis based approaches.

The basis for many dataflow analyses is the theory of monotone dataflow frameworks (MDF) \cite{31, 38}. An MDF is defined by a value lattice $L_V = \{V, \sqcup, \sqcap, \top, \bot\}$ and a set $F$ of transfer functions $f : L_V \mapsto L_V$.

It is then required that the transfer functions are monotone, i.e., it holds

$$\forall v, w \in V, \forall f \in F : v \sqsubseteq w \Rightarrow f(v) \sqsubseteq f(w),$$

and that the value lattice satisfies the ascending chain condition, i.e., for every infinite ascending chain $v_0 \sqsubseteq v_1 \sqsubseteq \ldots \sqsubseteq v_i \sqsubseteq \ldots$ in $V$, there is an element $v_i$ such that $j > i \Rightarrow v_i = v_j$. This implies that termination of the analysis is guaranteed: Since the intermediate analysis results only get bigger, and the algorithm terminates as soon as applying the transfer functions to all of the nodes does not change their values, i.e., the analysis reaches a fixed point.

A very simple approach to applying the transfer functions to the program graph is to apply the transfer functions to all nodes until no more changes occur. Specialized strategies, e.g., interval analysis \cite{2, 36} and loop tree analysis \cite{52}, stabilize inner loops before outer loops. They require less iterations to achieve the fixed point, which yields better performance.

2.2 Program Representations

Program representations can capture either the full semantics of a program, or they can focus on parts of a program that are sufficient for a given task. Commonly known full program representations are stack-machine code (e.g., Java bytecode) and triple form (e.g., Jimple \cite{53}).

For some analysis tasks, not all information of the complete program are required. For example, a points-to analysis may abstract from variables and operations concerning primitive types or certain operations concerning control flow. Thus, they can be removed from the program representation, yielding a thinned or sparse program representation. This program representation is then more compact, which allows for a more efficient analysis. Both points-to analyses presented in the next chapter make use of sparse program representations.

In the following, we discuss two full program representations, Static Single Assignment form – in short, SSA – and its derivate Memory SSA, as the latter is used later on in this thesis.
Static Single Assignment Form (SSA)

Static Single Assignment form is an intermediate representation technique first developed by Cytron et al. [9]. Every variable is assigned a value exactly once. For each definition in original form, a new version of that variable is created during SSA construction. To decide what version of a variable is valid after meets in the control flow, \( \varphi \) nodes are introduced: \( \varphi \) nodes are artificial operations that take the possible versions of a variable as arguments and decide, depending on control flow, which of these operands is the currently valid definition. SSA provides many benefits for program analysis, for instance, use-def relations become explicit.

Memory SSA and FIRM

Memory SSA [52] is a graph-based extension to the traditional SSA. In Memory SSA, the traditional ordering of operations within a basic block structure is replaced by a directed graph structure. Local variables are resolved to dataflow edges connecting operations (nodes), which has the effect that def-use relations become explicit additionally to use-def relations. Dependencies on accessing the memory are modeled by memory edges, putting memory on the same level as data, including the use of \( \varphi \) nodes at control flow confluence points. These memory edges dictate a correct order in which memory accesses must be executed for a given program.

FIRM [26] is a concrete specification of Memory SSA. A concrete implementation is described in [25]. Figure 2.1 shows a very simple method and its corresponding FIRM graph. We describe some of the core features of FIRM by looking at this simple example.

Operations in FIRM are represented as nodes in the graph. Each node has zero or more ordered predecessor nodes, its input nodes, produces an output tuple of values, from which the different results (e.g., memory dependency, result value of a load operation) can be selected by using \( \text{Proj} \) nodes. In our example, we see \( \text{Proj} \) nodes for selecting the method argument (\( \text{Proj}($1) \)), memory dependencies (\( \text{Proj}(\text{mem}) \)), and the return value of a call (\( \text{Proj}(\text{val}) \)).

The boxes surrounding multiple nodes depict the basic blocks of the method. There is always one block with no predecessor and one with no successor, the start and end block, respectively. Each block except the end-block contains a control flow operation node, e.g., \( \text{Jmp} \), \( \text{Proj}(\text{exe}) \), or \( \text{Return} \), which denotes the control flow successor(s) of a given basic block.

Note that even \( \text{Load} \) nodes, which do not change the memory, define a memory value; this models anti-dependencies (write-after-read dependencies).

FIRM aims at using atomic operations. For instance, a \( \text{Call} \)-node has not only its memory dependency and target address as input, but also a node that
Chapter 2. Background

Figure 2.1: Source code and corresponding FIRM graph

describes which method is actually called. In our example, this node is an EntitySelect-node with a textual representation of the called method. This may look strange at first sight, but is important from the point of view of a compiler: Eventually, the textual description of selecting the method must be resolved to address arithmetic, i.e., the EntitySelect-node is replaced by a number of operation nodes that perform this task.\footnote{This process is called \textit{lowering}. The address arithmetic can be quite complex, for example, when a polymorphic call is resolved, a \textit{vtable} lookup must be performed.}

In short, exception handling is performed by special control flow operation nodes. We have \texttt{ExcJmp} and \texttt{ExcReturn} nodes in our example, which specify the intra-procedural exception handling, and a special field named \texttt{exc\_obj}, which contains a possibly thrown exception from the \texttt{Call} operation. Any operation that may cause an exception can implicitly write a value to this field as well (e.g., when a division by zero occurs in an integer division). The \texttt{ExcJmp} is executed if this field contains a non-null value, or when any operation in the same basic block triggers an exception. Otherwise, the regular \texttt{Jmp} node is followed.

In summary, Memory SSA is a full program representation where all local dependencies are explicitly modeled via edges in the graph. FIRM is a concrete specification of Memory SSA. It aims at specifying atomic operations, which allows for the flexibility a compiler requires from an intermediate rep-
2.3 Points-to Analysis

The task of points-to analysis as well as its use has already been presented in the introduction of this thesis.

Now we discuss different aspects that affect the precision and cost of points-to analysis. We keep close to the categorization of Ryder [42].

Naming schemes

A program analysis needs to abstract from the values which expressions may take during a real application run in some way, as it is impossible to model the exact program state at any time of any possible run of a program. For objects, such an abstraction is called a naming scheme. For a given program and naming scheme, there is then a set $N$ of names for all abstract objects. Each abstract object $n \in N$ represents a set of concrete runtime objects $o(n)$.

For this must hold:

$$\forall n_1, n_2 \in N : n_1 \neq n_2 \Rightarrow o(n_1) \cap o(n_2) = \emptyset$$

Thus, an abstract object may denote an arbitrary number of runtime objects, but each runtime object must be represented by exactly one abstract object.

Two well-known naming schemes are the class naming scheme and the creation point naming scheme. For the former, one abstract object per class is used; for the latter, objects created at the same syntactical location are grouped together. While the former requires less resources (for instance, fewer abstract objects can be represented by data structures that require less memory) and is sufficient for, e.g., call-graph construction, the latter should be preferred for more sophisticated analyses [42].

More precise naming schemes are also possible; for example, objects can be – additionally to their creation site – categorized by their calling context, confer the discussion on context sensitivity below. Such approaches have been used by, e.g., Liang et al. [24] and Lhoták and Hendren [23].

Flow sensitivity

Flow-sensitivity is a concept that is frequently used, but there is no consensus as to its precise definition [30]. Informally, an analysis is flow-sensitive if it takes control flow information into account [17]. Many people also require the use of so-called strong (or killing) updates as a criteria for flow-sensitivity [42].

Strong updates occur when an assignment supersedes (or kills the results of) an earlier assignment. The problem with strong updates is that they are only
permitted if the ordering of the reads and writes of a given variable is sure, and if the variable identifies a unique memory location. For local variables, these cases can be detected using a def-use analysis, i.e., an analysis that computes for every definition of a variable all uses of that variable along a definition free control flow path. One way to achieve this is to base dataflow analysis on an SSA-based representation, which implies local flow-sensitivity as demonstrated by Hasti and Horwitz [14].

**Context sensitivity**

In a context-insensitive program analysis, analysis values of different call sites are propagated to the same method and get mixed there. The analysis value is then the merger of all calls targeting that method. Thus, results from two distinct calls to the same method are merged, which induces imprecision to the analysis result of each of these calls. A context-sensitive analysis addresses this source of imprecision by distinguishing between different calling contexts of a method. It analyzes a method separately for each calling context [42]. Context sensitivity will therefore, in general, give a more precise analysis. The drawbacks are the increased memory cost that comes with maintaining a larger number of contexts and their analysis values, and the increased analysis time that may be required to reach a fixed point.

Context-sensitive approaches use a finite abstraction of the call stack possibly occurring at each call site in order to separate different calling contexts. The two traditional approaches to define a context are referred to as the call string approach and the functional approach [47]. The call string approach defines a context by the top \(k\) callers, i.e., return addresses on the call stack top [48], referred to as the family of \(k\)-CFA (Control Flow Analysis). The functional approach uses some abstractions of the call site’s actual parameters to distinguish different contexts [47, 12]. Both the call string approach and the functional approach were evaluated and put into a common framework by Grove et al. [12].

A functional approach designed for object-oriented languages is referred to as object-sensitivity [33, 34]. It distinguishes contexts by separately analyzing the targeted method for each abstract object in the implicit this-parameter. Similarly to \(k\)-CFA, a family of \(k\)-object-sensitive algorithms distinguishing contexts by the top \(k\) abstract target objects on the call stack can be defined. The authors evaluated a simplified version of 1-object-sensitivity. Here, only method parameters and return values are treated context-sensitively. Compared to 1-CFA, increased precision of side-effect analysis and, to a lesser degree, call graph construction, was reported. Both approaches show similar costs in time and memory. These results generalize to variants where \(k > 1\), which, however, are very costly in terms of memory and provide only a small increase in precision [23]. A variation of object-sensitivity, this-sensitivity, has
been presented by Lundberg et al. [28, 27]. In contrast to object-sensitivity, which analyzes a method separately for each abstract object reaching the implicit this-variable, this-sensitivity analyzes a method separately for each set of abstract objects reaching the implicit this-parameter.

Milanova et al. [34] as well as Lhoták and Hendren [23] have compared object-sensitivity with the call string approach. Their findings are that, in theory, none of the two is more precise than the other, but experiments show that 1-object-sensitivity yields better analysis results than 1-CFA. Additionally, Lundberg et al. have shown that, in practice, 1-this-sensitivity is almost as precise as 1-object-sensitivity but an order of magnitude faster. Again, none of the two is more precise than the other in theory [28, 27].

Context Definitions  A context definition is a rule that associates a call with a set of contexts under which the target method should be analyzed. Actually, ObjSens is the only context definition (in this selection below) that may associate a call with more than one context. Each context is in turn defined by a tuple; the tuple elements, its number and content, depend on what context definition we are using. In this thesis, we will use the following context definitions for a given call from a call site cs_i: a.m(v_1, . . . , v_n) where Pt(a) = \{o_1, . . . , o_p\}.

**ConIns**: cs_i \mapsto \{(m)\}

All calls targeting method m are mapped to the same context. This is the context-insensitive baseline approach.

**CallString**: cs_i \mapsto \{(m, cs_i)\}

Calls from the same call site cs_i are mapped to the same context.

**ObjSens**: cs_i \mapsto \{(m)\} if m.isStatic, \{(m, o_1), . . . , (m, o_p)\} otherwise.

Calls targeting the same receiving abstract object o_i \in Pt(a) are mapped to the same context. Static calls are handled context-insensitively.

**ThisSens**: cs_i \mapsto \{(m)\} if m.isStatic, \{(m, Pt(a))\} otherwise.

Calls targeting the same points-to set Pt(a) are mapped to the same context. Static calls are handled context-insensitively.

For example, given a (non-static) call a.m(v_1) with Pt(a) = \{o_1, o_2\}, ThisSens would map it to the single context (m, \{o_1, o_2\}), whereas ObjSens would map it to the two contexts (m, \{o_1\}) and (m, \{o_2\}).
Chapter 2. Background

```java
class A {
    static A f;
    void foo(A p) {
        p.bar();
    }
    void bar() {
        if (f != null) {
            f.foobar();
        }
        f = new A();
    }
    void foobar() {
    }
}
```

Figure 2.2: Java Source Code Leading to Negative Synergy Effect between Object Sensitivity and Flow Sensitivity.

**Contra-productive Synergy Effects** In theory, flow sensitivity and context sensitivity are orthogonal approaches. However, in rare cases, they can have a negative influence on each other: Consider the example Java code in Figure 2.2. Assume a flow-sensitive, context-insensitive analysis, $P_t(p) = \{o_1, o_2\}$ in line 3, and $P_t([A, f]) = \emptyset$. When analyzing the statement $p.bar()$ in line 4, method $bar()$ is analyzed under a single context. Then, when analyzing $bar()$ under the first context, the field $A.f$ is assigned an abstract object; when $bar()$ is then analyzed again, the call $f.foobar()$ in line 8 is not performed, as the points-to set of the static field $A.f$ is empty; only afterwards is $A.f$ assigned an abstract object. Thus, $foobar()$ is not reachable in this scenario.

However, when analyzing the source code with an object-sensitive approach, then the call $p.bar()$ in line 4 will be analyzed under two contexts: Once for the abstract object $o_1$, and once for the abstract object $o_2$. When analyzing $bar()$ under the first context, the field $A.f$ is assigned an abstract object; when $bar()$ is then analyzed again, the call $f.foobar()$ becomes suddenly reachable, because the points-to set of field $A.f$ is not empty any longer; thus, an object-sensitive approach will find that $foobar()$ is reachable, and thus be less accurate than context-insensitive analysis in this scenario.

**Field sensitivity**

An analysis is **field-sensitive** if it distinguishes the analysis results for instance fields of different abstract objects of the same type. Otherwise, it merges the contents of all fields of objects of a given class and is thus field-insensitive. Thus, for an expression $o.f$, a field-sensitive analysis will take both $o$ and $f$ to determine the memory location of the referenced fields; non-field-sensitive approaches would use only either $o$ (field-independent) or $f$ (field-based) instead. Whaley and Lam showed that field sensitivity is essential for points-
to analysis for strictly types languages such as Java, not only for the precision but even for the performance of the analysis [54].

Path sensitivity

An analysis is path-sensitive if it takes the feasibility of different execution paths into account. Feasibility is determined by evaluating the expressions in control flow statements.

In the so-called Gated SSA formalism [15], \( \varphi \) nodes are extended to \( \gamma \) nodes which are annotated with the corresponding branching conditions. These may then be used to make statements about taken paths after control flow confluences.

Many approaches deal with the meet over all paths (MOP) dataflow problem, e.g., [7]. Since the number of paths is, in general, unbounded, approaches narrow down the set of paths, e.g., by finding correlations between branch conditions [10]. Xie et al. [55] use path-sensitive analysis in their array access checker ARCHER. Their approach to path-sensitivity selects a set of execution paths – both a super- and subset of legal paths – and eliminate infeasible paths based on branching conditions. A different approach limits the number of paths to investigate by selecting interesting paths based on dynamic analysis, e.g., [4].

Open Problems

Some features of modern programming languages cause common problems to the implementation of points-to analysis. These include dynamic class loading, reflection, and native methods. The exact semantics of methods dealing with these features are usually not known, or commonly valid abstractions are too imprecise to be feasible. For instance, a native method can change the entire memory.

Thus, often only subsets of a given programming language can be analyzed statically. For programs making use of these features, often no conservative analysis is possible.

An approach to dealing with such features is described by Hirzel et al. [18]. They perform a regular points-to analysis and use the results for program optimizations. Then, they monitored the program execution, and perform analysis updates online, i.e., at runtime. Each time a language feature that is not handled by their static analysis is invoked during runtime, the execution of the program is interrupted and the points-to sets are updated. The authors also describe how clients that consume these points-to sets, e.g., for program optimization, have to deal with such changes. The authors ensure that their implementation is correct by comparing dynamic points-to sets with the static results at garbage collection time. However, a disadvantage is that their
analysis has to be very fast; they therefore use a rather imprecise points-to analysis.

An offline-approach, named *internal analysis*, is presented by Nguyen and Xue [37]. They describe an algorithm that computes which points-to sets are definitely not affected by features like dynamic class loading, and which can therefore be safely used for, e.g., program optimizations. The authors show the applicability of their approach by using it for partial redundancy elimination and field propagation.

## 2.4 Comparing Dynamic and Static Analyses

A number of researchers have dealt with investigating the precision (or rather imprecision) of points-to analysis, or static analysis in general. Mock *et al.* [35] compare dynamic pointer-behavior with statically computed points-to sets for C. They come to the conclusion that static points-to information is often very imprecise, by factor ten to hundred bigger than pointer-behavior during runtime. They assume the dynamic analysis as the reference but, in fact, it is not obvious whether the static analysis contains too much garbage or the dynamic analysis has too many misses. Moreover, the authors used a rather imprecise static algorithm; the results of current points-to analysis approaches are likely to be better.

Ribeiro and Cintra [39] investigate how precise a points-to analysis for C can actually become and therefore use a state-of-the-art flow- and context-sensitive points-to analysis. For assessing accuracy, they also investigate the pointer-dereferences which differ in dynamic and static analysis and give explanations for why static points-to analysis fails at these dereferences. The authors perform their studies in the context of compiler-optimizations. Although a better static analysis is used, they also cannot decide whether the dynamic or the static analysis has the higher accuracy.

Liang *et al.* [24] investigate the precision of naming schemes for Java: how precise does a given naming scheme determine instances of objects, i.e., is a name for an object shared by multiple instances in a given program run? They perform two studies, measuring precisions at call sites. The first study investigates how precise a naming schema could actually be; here, an imprecision occurs when a call site is called on two different runtime object instances which would be mapped to the same abstract object. The authors use context-aware naming schemes and compare them. A call site where an abstract object would identify exactly one runtime object is named “empirically precise”. The authors find that creation site naming schemes are very precise at a high percentage of call sites for many test programs. In their second study, the authors compare points-to analysis results with dynamic analysis. They set the amount of abstract objects that reach a given call
2.4. Comparing Dynamic and Static Analyses

Comparing Dynamic and Static Analyses

The authors conclude that the creation site naming scheme is precise enough in many cases, but more precise algorithms that can model complex runtime data structures must be developed in other cases.

Rountev et al. discuss the imprecision of static analysis [40] in general. They also define upper and lower bounds of the exact solutions and propose, for the better understanding of imprecision of static analysis, to manually investigate the results of static analysis compared to dynamic analysis, i.e., to create a Gold Standard. They base their thoughts on the assumptions that “static analysis is intrinsically conservative”, which is not quite true. The same authors perform an empirical study for Java programs, where they compare static, dynamic, and – where those analysis results differ – manual investigation of feasible call chains [41].

Lhoták presents a tool for finding differences in call graphs [21]. It allows for easily finding differences between call graphs, as well as identifying root causes for differences, e.g., between dynamic and static analysis. He too states that the exact call graph of a program has a lower bound given by dynamic analysis, and an upper bound from static analysis. To guarantee the latter, he enriches his points-to analysis implementation with expert knowledge about the input programs, so that the open questions to points-to analysis discussed in Section 2.3 – e.g., dynamic class loading – are avoided.
Chapter 3

Points-to Analysis - Concrete Implementations

In this chapter, we present two concrete implementations of points-to analysis: Spark and Points-to SSA based Simulated Execution. These points-to analysis implementations will be used for the evaluation in Chapter 6.

Spark (Section 3.1) is a well-known and widely used context- and flow-insensitive points-to analysis that is part of the Soot framework\(^1\). Points-to SSA is a sparse, Memory SSA based program representation targeted at points-to analysis. It is interpreted using Simulated Execution, a semi-flow-sensitive analysis technique. We describe Points-to SSA and Simulated Execution in detail in Section 3.2, as we will present an extension to it in Section 5.4. Finally, in Section 3.3, we present the agent that we use to capture dynamic points-to sets.

A tabularly comparison of Spark and Points-to SSA based Simulated Execution with respect to the concepts discussed in Section 2.3 is given in Table 3.1.

### 3.1 Spark

Spark, the Soot Pointer Analysis Research Kit [22], is a static points-to analysis framework taken from the Soot 2.3.0 framework. It is configurable in its precision; we describe here its most precise instantiation which is field-sensitive, context- and flow-insensitive, and uses a creation site naming scheme.

Spark constructs a Pointer Assignment Graph (PAG) as its representation for the program to be analyzed. Nodes and (directed) edges in the graph correspond to program statements. The PAG construction is done by associating each relevant program statement (statements involving abstract object transport) with the construction of different PAG entities. Table 3.2 shows the different PAG entity types, and what kind of program statements cause the creation of these. Note that the set of all allocation nodes in the PAG

\(^{1}\)http://www.sable.mcgill.ca/soot
3.1. Spark

<table>
<thead>
<tr>
<th>Table 3.1: Spark vs. Points-to SSA based Simulated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAG entity</strong></td>
</tr>
<tr>
<td>Allocation node</td>
</tr>
<tr>
<td>Variable node</td>
</tr>
<tr>
<td>Field reference node</td>
</tr>
<tr>
<td>Allocation edge</td>
</tr>
<tr>
<td>Assignment edge</td>
</tr>
<tr>
<td>Store edge</td>
</tr>
<tr>
<td>Load edge</td>
</tr>
</tbody>
</table>

form the set of abstract object $O$. **Variable and field reference nodes** represent any access to these. Load and store operations are distinguished by the kind of edges (store, load edges) that connect these nodes with other nodes.

In the handling of monomorphic calls $l = a.m(v)$, edges are added that correspond to assignments of address $a$, argument $v$, and return values $ret$, to the implicit variable $this$, formal parameter $p$, and receiving variable $l$.

A simple “Linked List” implementation and its corresponding PAG is shown in Figure 3.1. Here, ellipses are variable nodes, rectangles are field references, and stars are allocation nodes. The thin lines depict the intra-procedural allocation-, assignment-, store- and load-edges, whereas the thick lines show the inter-procedural call relations (recursive calls in in the case of method `append`).

After PAG construction, dataflow analysis is performed by propagating points-to sets along the PAG edges. For this, each node $n$ is associated with a points-to set $Pt(n) \subseteq O$ which, when the analysis is completed, is interpreted as: $Pt(n)$ is the set of abstract objects that may be referenced by $n$. Initially, the points-to set of each allocation node contains exactly itself. In our example, this points-to set would be propagated to the $this$ parameter of $L.<init>$, further on to the $this$ parameter of $Object.<init>$, and to the field reference node $this.next$ in method $L.append$, etc...

Note that field sensitivity is achieved by adding nodes of another type,
public class L {
    V value = null;
    L next = null;

    public L (V v) {
        value = v;
    }

    public void append(V v) {
        if (next == null)
            next = new L(v);
        else
            next.append(v);
    }

    public void putAt(int n, V v) {
        int count = 0;
        L l = this;
        while (count < n) {
            l = l.next;
            count++;
        }
        l.value = v;
    }
}

Figure 3.1: Source code fragment and corresponding PAG.

concrete field nodes, to the PAG during propagation. Each such node is attributed with a single abstract object and the field it represents.

Spark has support for specifying classes which could be loaded dynamically during runtime (via the -dynamic-class option). However, this approach requires either expert-knowledge provided by the user, or support by dynamic tools. We thus do not use this option in this thesis. When the input string to the dynamic class loading mechanism is a constant, then Spark automatically resolves the class loading.

Spark’s naming scheme merges, by default, all StringBuffer and StringBuilder objects to one abstract object. However, the creation site naming scheme can be enforced even for these types.

3.2 Points-to SSA based Simulated Execution

Points-to SSA is our sparse, graph-based program representation based on Memory SSA. On top of it, we perform Simulated Execution. These two concepts are presented in the following, after we presented the analysis value abstraction that we use.
3.2. Points-to SSA based Simulated Execution

Analysis Values

Our points-to analysis needs to represent references to abstract objects and an abstraction of the heap-memory.

In the analysis, reference variables will in general hold references to more than one abstract object. Hence, we assume that each points-to value \( v \) in the analysis of a program is an element in the points-to value lattice \( L_V = \{ V, \sqcup, \sqcap, \top, \bot \} \) where \( V = 2^O \) is the power set of \( O \), \( \top = O \), \( \bot = \emptyset \), and \( \sqcup, \sqcap \) are the set operations \( \cup \) (union) and \( \cap \) (intersection). The height of the points-to value lattice is \( h_o = |O| \). We use the notation \( Pt(a) \) to refer to the points-to value that is referenced by the expression \( a \).

Each abstract object \( o \in O \) has a unique set of object fields \([o, f] \in OF\), where \( f \in F \) is a unique identifier for a field (capturing references). Each object field \([o, f]\) is in turn associated with a memory slot \(( [o, f], v) \), where \( v \) is a points-to value. A memory slot represents the abstract object references stored in object field \([o, f]\).

The abstraction of the heap-memory associated with an analyzed program, referred to as abstract memory \( Mem \), is defined as the set of all memory slots \(( [o, f], v) \). In our approach, we use a single global memory configuration. Our reason for introducing an abstract memory is not only to mimic the runtime behavior; it is a necessary construct to handle field store and load operations and the transport of abstract objects from one method to another that follows as a result of these operations. We think of the abstract memory as a mapping from object fields to points-to values. The memory is therefore equipped with two operations

\[
Mem.get(OF) \to V \quad \text{and} \quad Mem.addTo(OF, V)
\]

with the interpretation of reading the points-to value stored in an object field \([o, f] \in OF\), and merging the points-to value \( v \in V \) with the points-to value already stored in an object field \([o, f] \in OF\), respectively. Note that we never override previously stored object field values in memory store operations, i.e., we never execute strong updates. Instead, we merge the new value with the old one using the points-to value lattice’s join operation, i.e., we perform weak updates.

The abstract memory is updated as a side effect of the analysis. In order to quickly determine the fixed point, we use memory sizes indicating whether or not the memory has changed. In what follows, we refer to the size of the abstract memory as a memory size \( x \in X = [0, h_m] \), where \( h_m \) is the maximum memory size. It corresponds to the case where all object fields contain all abstract objects, hence, \( h_m = |OF| \cdot |O| \).

In order to apply the theory of monotone dataflow frameworks to memory size values as well, we introduce a lattice \( L_X \) referred to as the memory size lattice. The memory size lattice \( L_X \) is a single ascending chain of integers,
i.e., $L_X = \{X, \sqcup, \sqcap, \top, \bot\}$, where $X = \{0, 1, 2, \ldots, h_m\}$, $\top = h_m$, $\bot = 0$, $x_1 \sqcup x_2 = \max(x_1, x_2)$, and $x_1 \sqcap x_2 = \min(x_1, x_2)$. The height of $L_X$ is $h_m$.

**Points-to SSA**

Points-to SSA is highly inspired by Memory SSA. Features of Memory SSA, e.g., local variables are represented by dataflow edges between operations (nodes), are also present in Points-to SSA. In fact, Points-to SSA can be considered a sparse Memory SSA representation.

Figure 3.2 shows the simple “Linked List” implementation that we already used as an example above, when describing Spark’s PAG, but this time with the corresponding Points-to SSA graphs. Note that the graphs are much more compact than FIRM graphs, which was presented in the previous chapter. Each method is represented by a graph and each node in the graph represents an operation in the method. We have for example Entry and Exit nodes representing method entry/exit points, and Store and Load nodes representing field write/read operations. The so-called ports at the top of a node represent operation input values (e.g., memory size $x$, the values $v$ to store in the Store nodes, and target address values $a$ as a special of values), and the ports at the bottom represent operation results (e.g., a new memory size $x$ in the Store nodes). Edges connecting node ports represent the flow of values from defining nodes (operation results) to using nodes (operation input values). More details regarding these notations will be presented later on.

Notice that the constructor $L.<init>$ starts by calling its super constructor $Object.<init>$ and that object creation, in $L.append$, is done in two steps: we first allocate an object of class $L$ and then call the constructor $L.<init>$. $\varphi$ nodes are used in $L.append$ to merge the memory size values from the two selective branches, and in $L.putAt$ as the loop head of the iteration.

A Points-to SSA method graph can be seen as an abstraction of a method’s semantics, an SSA graph representation specially designed for points-to analysis. It is an abstraction since we have removed all operations not directly related to reference computations, e.g., operations related to primitive types.

Another feature of Points-to SSA that is inspired by Memory SSA is the use of memory edges to explicitly model dependencies between different memory accessing operations. An operation that may change the memory defines a new memory size value, and operations that may access this updated memory use the new memory size value. Thus, memory sizes are considered as data, and memory size edges have the same semantics – including the use of $\varphi$ nodes at join points – as def-use edges for other types of data. The introduction of memory size edges in Points-to SSA is important since they also imply a correct order in which the memory accessing operations are analyzed, which ensures that the analysis is a intra-procedural flow-sensitive
public class L {
    V value = null;
    L next = null;

    public L (V v) {
        value = v;
    }

    public void append (V v) {
        if (next == null)
            next = new L(v);
        else
            next.append(v);
    }

    public void putAt (int n, V v) {
        int count = 0;
        L l = this;
        while (count < n) {
            l = l.next;
            count++;
        }
        l.value = v;
    }
}

Figure 3.2: Source code fragment and corresponding Points-to SSA graphs.
A Points-to SSA method graph is now defined as a directed and ordered multi-graph $G = \{N, E, Entry, Exit\}$, where $N$ is a set of Points-to SSA nodes, $E$ is a set of Points-to SSA edges, $Entry$ is a graph entry node satisfying $|\text{pred}(Entry)| = 0$, and $Exit$ is a graph exit node satisfying $|\text{succ}(Exit)| = 0$.

The reference-related semantics of different language constructs (e.g., calls and field accesses) are described by a set of operation node types. Each node $n$ in a Points-to SSA graph is of exactly one such type. It has further a number of in-ports $\text{in}(n) = [\text{in}_1(n), \ldots, \text{in}_k(n)]$, and a number of out-ports $\text{out}(n) = [\text{out}_1(n), \ldots, \text{out}_l(n)]$. The in-ports represent input values to the operation in question, whereas the out-ports represent the results produced by the operation. All ports have a fixed type ($V$ or $X$) and a current analysis value of that type ($v \in L_V$ or $x \in L_X$). Note that nodes of the same type may have a different number of in- and out-ports; for instance, the number of in-ports of a node representing a method call depends on the number of arguments of the called method.

An edge $e = \text{out}_i(src) \rightarrow \text{in}_j(tgt)$ connects an out-port of a node $src$ with an in-port of a node $tgt$. An edge may only connect out- and in-ports of the same type. An out-port $\text{out}_i(n)$ may be connected to one or more outgoing edges. An in-port $\text{in}_j(n)$ is always connected to a single incoming edge. The last property reflects our underlying SSA approach – each value has one, and only one, definition.

Certain node types have attributes that refer to node-specific, static information. For example, each $\text{Alloc}^C$ node is decorated with a class identifier $C$ that identifies the class of the object to be created.

Finally, each type of node is associated with a unique analysis semantics (or transfer function) which can be seen as a mapping from in-ports to out-ports that may have a side-effect on the memory. As an example, Algorithm 1 shows the analysis semantics for the $\text{Store}^f$ node, which abstracts the actual semantics of a field write statement $a.f = v$. For each abstract object $o$ in

\begin{algorithm}
\caption{\textit{Store}^f : [x_{in}, a, v] \mapsto x_{out}}
\begin{align*}
x_{out} &= x_{in} \\
\text{for each } o \in \text{Pt}(a) \text{ do} \\
&\quad \text{prev} = \text{Mem.get}([o, f]) \\
&\quad \text{if } v \not\subseteq \text{prev} \text{ then} \\
&\quad \quad \text{Mem.addTo}([o, f], v) \\
&\quad \quad x_{out} = \text{Mem.getSize}() \\
&\quad \text{end if} \\
\text{end for} \\
\text{return } x_{out}
\end{align*}
\end{algorithm}
3.2. Points-to SSA based Simulated Execution

the address reference $a$, we look up the points-to value previously stored in object field $[o, f]$. If the new value to be stored changes the memory (i.e., if $v \not\subseteq \text{prev}$), we merge $v$ with the previous value and save the result. Notice also that we compute a new memory out-port value (a new memory size) if the memory has been changed during this operation.

The transfer functions of all node types currently in use in Points-to SSA are listed in Appendix A.

Context-Sensitive Simulated Execution

Our dataflow analysis technique, called Simulated Execution, is an abstract interpretation of the program based on the abstract analysis and program representation discussed in the previous section. It simulates the actual execution of a program where the analysis of a method is interrupted when a call occurs, and later resumed when the analysis of the called method was completed.

The Simulated Execution approach can be seen as a recursive interaction between the analysis of an individual Points-to SSA method graph and the transfer function associated with monomorphic calls, which handle the transition from one method to another. Polymorphic calls are handled as selections over possible target methods $m_i$, which are then processed as a sequence of monomorphic calls targeting $m_i$.

This approach implies global (inter-procedural) flow-sensitivity, as a memory accessing operation (call or field access) $a_1.x$ will never be affected by another memory access $a_2.x$ that is executed after $a_1.x$ in all runs of a program.

The context-sensitive approach described here is a modification of the context-insensitive approach described by Lundberg and Löwe [29]. The most noticeable difference is that we associate each monomorphic call targeting a method $m$ with a number of contexts, and process $m$ separately for each such context.

Method Graph Processing

For each method graph, we have a pre-computed node order that is determined by the data and memory dependencies between the nodes. We compute a topological sorting for forward edges. To order the nodes in loops, we use a so-called loop tree analysis [52] where we identify inner and outer loops and their loop heads, which are always $\varphi$ nodes.

The method processing starts in the method entry node, follows the node ordering, and iterates over loops until a fixed point is reached. Inner loops are stabilized before their outer loops. Consequences of this approach are: (1) All nodes in a method graph $g_m$ are analyzed at least once every time method $m$ is analyzed. (2) All nodes, except the loop head $\varphi$ nodes, have
all their predecessor nodes updated before they are analyzed themselves. (3) The order in which the nodes are analyzed respects all control and data dependencies and is therefore an abstraction of the control flow of an actual execution. The final point is a crucial step to assure flow-sensitivity in the Points-to SSA based Simulated Execution technique.

The above properties of analyzing single method graphs is taken into consideration by `processMethod` as given in Algorithm 2. It should only be considered as a rough outline of the approach actually implemented. The idea is simple: We start by initializing the method entry node with the method input to be used in this particular method activation. We then analyze the method nodes repeatedly until we reach the method exit node. Therefore, we compute a node’s transfer function given by the node type, update the successor in-ports, and determine the next node to analyze to get its values stable. The transition from one method to another is embedded

```
Algorithm 2 processMethod : (m, [x_in, a, v_1, . . . , v_n]) ↦ [x_out, r]
n = m.entryNode
in(n) = [x_in, a, v_1, . . . , v_n]
do
  n.computeTransferFunction()
  n.updateSuccs()
  n = n.next()
while n ≠ m.exitNode
return in(n)
```

in the statement `n.computeTransferFunction()` if `n` is of a monomorphic call type (MCall\(^m,csi\)). Note that the processing of a call in turn may lead to the analysis of the call target method `m` as defined in `processMethod`.

**Call Processing**

Our approach to analyzing individual calls (see Algorithm 3) describes the handling of a call to method `m` in a context `ctx^m`. For the understanding of our call processing, it is safe to assume that all calls to `m` are associated with only one context `ctx^m`, i.e., that we perform a context-insensitive analysis. This is generalized to more contexts later on in this section.

The processing of (recursive) method calls must guarantee that the analysis terminates and that the analysis values reach a global fixed point.

The crucial step to ensure termination is that each context `ctx^m` is associated with two attributes `prev_args` and `prev_return` where we store previous input and return values of the calls to `m` in that context `ctx^m`. The former of these attributes is used to decide whether we have seen a more general call targeting `m` in the same context `ctx^m` before, i.e., if `[x_in, a, v_1, . . . , v_n] ⊆`
3.2. Points-to SSA based Simulated Execution

prev\_args, in which case we interrupt the call processing and reuse the previous result from prev\_return. The alternative, a call targeting \( m \) in \( ctx^m \)

Algorithm 3 \textit{processCall}(\( ctx^m, [x_{in}, a, v_1, \ldots, v_n] \)) \( \mapsto [x_{out}, r] \)

- - if \( ctx^m \) was already analyzed with larger parameters before
if \( [x_{in}, a, v_1, \ldots, v_n] \sqsubseteq ctx^m\.prev\_args \) then
  return \( ctx^m\.prev\_return \)
end if
\( ctx^m\.prev\_args = ctx^m\.prev\_args \sqcup [x_{in}, a, v_1, \ldots, v_n] \)
- - if \( ctx^m \) is on the analysis stack
if \( ctx^m\.is\_active \) then
  \( ctx^m\.is\_recursive = true \)
  return \( ctx^m\.prev\_return \)
end if
\( ctx^m\.is\_active = true \)
\( [x_{out}, r] = processMethod(m, ctx^m\.prev\_args) \)
- - if \( ctx^m \) was not recursively called within \textit{processMethod}
if \( \neg ctx^m\.is\_recursive \) then
  \( ctx^m\.prev\_return = [x_{out}, r] \)
  \( ctx^m\.is\_active = false \)
  return \( [x_{out}, r] \)
end if
- - while \( ctx^m \)'s recursive call results have not reached their fixed point
while \( ctx^m\.prev\_return \sqsubseteq [x_{out}, r] \) do
  \( ctx^m\.prev\_return = [x_{out}, r] \)
  \( [x_{out}, r] = processMethod(m, ctx^m\.prev\_args) \)
end while
\( ctx^m\.is\_recursive = false \)
\( ctx^m\.is\_active = false \)
return \( [x_{out}, r] \)

with new arguments, leads to a new method activation where we process the target method \( m \) by invoking \textit{processMethod} using the merged input \( prev\_args \sqcup [x_{in}, a, v_1, \ldots, v_n] \). We also update the two attributes \( prev\_args \) and \( prev\_return \) in preparation for the next call targeting \( m \) in \( ctx^m \).

Termination of our analysis is ensured since we incrementally merge our arguments \( prev\_args \sqcup [x_{in}, a, v_1, \ldots, v_n] \) before we start processing a method \( m \). Thus, the sequence of arguments \( args_i \) used for a given context \( ctx^m \) forms an ascending chain satisfying

\[
args_0 \sqsubseteq args_1 \sqsubseteq \ldots \sqsubseteq args_n.
\]

Each such chain must have finite length since our value lattices have finite heights (both \( L_X \) and \( L_V \) are finite). Thus, each method can only be processed a finite number of times, and analysis termination is guaranteed. This
argument also holds for calls involving recursion; terminations is guaranteed for these programs as well.

In order to guarantee that the fixed point is reached, especially in loops induced by recursive method calls, we need a few more attributes associated with each context: \textit{is\_active} is used to check if we are processing a call in a context that is currently being analyzed, i.e., if \( m \) is called recursively in \( ctx^m \). In this case, we directly return \textit{prev\_return} for the recursive call and undefined \([0, \bot]\) if we have no previous results, respectively. Also we set \textit{is\_recursive} = \textit{true} which indicates, upon return from \textit{processMethod}, that we have seen a recursive call during \textit{processMethod}. In this case, we need to stabilize the results by iteratively re-invoking \textit{processMethod} until the fixed point is reached.

Transfer Function of Context-Sensitive Calls

The transfer function of monomorphic call nodes \( MCall^{m,cs_i} \) is given in Algorithm 4. It completes the definition of the analysis semantics of a call from a call site \( cs_i : \text{r} = a.m(v_1, \ldots, v_n) \). The algorithm first selects a set of contexts for each call: \textit{selectContextsFor:} \( [m, cs_i, a] \mapsto [ctx_1, \ldots, ctx_q] \). It creates new contexts if and only if they have not been created before when processing similar calls. The implementation of this method depends on the chosen context sensitivity, as discussed in Section 2.3. Note that a call can be associated with more than one context, in particular for object-sensitivity.

We assume that each context \( ctx^m \) is aware of the corresponding points-to value for the implicit variable \textit{this}. In short, it is a singleton abstract object set \( \{o_i\}, o_i \in Pt(a) \) for the 1-object-sensitive analysis, and the whole set \( Pt(a) \) for context-insensitive, 1-this-sensitive and 1-CFA. This information is embodied in the assignment \textit{this} = \( ctx^m\).\textit{getThis()} that we are using to simplify the notations.

\begin{algorithm}
\begin{footnotesize}
\begin{align*}
\textbf{Algorithm 4} & \quad MCall^{m,cs_i} : [x_{in}, a, v_1, \ldots, v_n] \mapsto [x_{out}, r] \\
\text{Context}[] & \quad ctxs = \text{selectContextsFor}(m, cs_i, a) \\
[\text{x}_{out}, r] & \quad = [0, \bot] \\
\text{for each} & \quad ctx^m \in ctxs \text{ do} \\
& \quad \text{this} = ctx^m\text{.getThis()} \\
& \quad \text{args} = [x_{in}, \text{this}, v_1, \ldots, v_n] \\
& \quad [x_{out}, r] = \text{processCall}(ctx^m, args) \cup [x_{out}, r] \\
\text{end for} \\
\text{return} & \quad [x_{out}, r]
\end{align*}
\end{footnotesize}
\end{algorithm}

Algorithm 4 lists the transfer function for MCall nodes. It computes all contexts \( ctxs \) and merges the analysis results of the individual calls to the target method \( m \) in each context \( ctx^m \) as returned from \textit{processCall}.
3.3 Capturing Dynamic Points-to Information

Monitoring running program is possible in two ways: First, the program code can be enriched with additional code that captures information about the program execution. Second, the operating environment can support hooking, i.e., code can register to be informed about certain events through callbacks. Hooking is provided by many operating systems (e.g., for supporting residual virus scanners) and virtual machines.

Java provides the Java Virtual Machine Tool Interface (JVMTI)\(^2\) for the purpose of monitoring and controlling applications. So-called agents can be loaded with the start of the virtual machine and interact with it – or even modify its state – via JVMTI. JVMTI allows an agent to register for numerous events, e.g., like class loading, exception generation, field reads and writes, method entry and exit, and more. When a class is about to being loaded, it is possible to alter its bytecode; this is called instrumentation.

When capturing runtime information related to points-to analysis, a requirement is to be able to identify runtime objects by their according abstract objects. In the following, we describe how mapping a runtime object to its abstract object is performed.

In Java bytecode, object allocation is divided into two steps: allocating the heap space for the new object, and calling its constructor. For the purpose of mapping a runtime object to its abstract object, we instrument the java.lang.Object constructor, since each constructor must eventually call it. This follows a suggestion from the JVMTI reference\(^3\) for object allocation tracking\(^4\). Once the constructor for the Object-class is reached, we traverse the stack in order to locate the stack frame that corresponds to the constructor invocation of this newly created object. The instruction that is the return address of this stack frame is then the *syntactical creation point* of this runtime object; for it, we can derive line number and the class that encloses the instruction. The stack-traversal is described in Algorithm 5, which works as follows: First, the type of the newly created object is retrieved. Then, the stack is traversed frame by frame, until the current frame denotes the invocation of a constructor (indicated by the name “\(<\text{init}>\)” of the searched type (as retrieved by getInstanceType). If such a frame is found, then it has to be checked if the next frame is not also a matching constructor with the

\(^2\)http://java.sun.com/javase/6/docs/technotes/guides/jvmti/

\(^3\)http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html

\(^4\)Our first approach to map runtime objects to abstract objects was to use bytecode instrumentation to insert a method call with this newly allocated object as a parameter to a method that would set the tag, even before the corresponding constructor is called. This, however, is objected by the bytecode verifier during runtime. We then decided to disable bytecode verification (via the -Xverify:none option), which worked at first. Unfortunately, for some – but not all – benchmark programs, this caused reproducible crashes of the Java Virtual Machine.
same conditions: This happens when a constructor calls another of the same class (via the this-keyword) instead of a constructor of its super-class. If this is not the case, then the stack frame following this frame denotes the call site of the object creation.

**Algorithm 5** Mapping Runtime Objects to Abstract Objects.

```java
instanceType = getInstanceType(getFrame(0))
i = 1
found = false
while ¬found do
    frame = getFrame(i)
    if getMethodName(frame) = "<init>" ∧ getInstanceType(frame) = instanceType then
        nextFrame = getFrame(i + 1)
        if getMethodName(nextFrame) ≠ "<init>" ∨ getInstanceType(nextFrame) ≠ instanceType then
            found = true
        end if
    end if
    i = i + 1
end while
return nextFrame
```

After the syntactical creation site of the runtime object has been determined, we need to annotate the runtime object with this information in some way. For this, we use a feature of the JVMTI that allows to tag runtime objects with arbitrary, 64 bit values. Of these 64 bit, we use 20 bits for each line number, type of the object, and class that encloses the allocation instruction. This allows $2^{20}$ different types and files of up to $2^{20}$ lines long, which is more than sufficient for current programs.

At this point, the agent needs to be extended in that it needs to monitor concrete events which are of interest. The infrastructure described here allows to easily capture dynamic points-to information; the required mapping to abstract object is available at any point in the program that is being monitored.
Chapter 4

Comparing, Combining, and Improving Analyses - The Theory

In this chapter, we discuss the theory behind comparing, combining, and improving dataflow analyses. When we talk about the result of an analysis $A$ here, we mean a set of facts about a given input program (also referred to as benchmark program). To such sets can then the usual mathematical set-operations like union $\cup$ and intersection $\cap$, as well as comparators like subset $\subset$ be applied. We implicitly assume that, when comparing two analyses $A_1$ and $A_2$, that those analyses solve the same analysis question, e.g., points-to analysis. However, the claims from this chapter are not necessarily restricted to points-to analysis. Instead, they should hold for most dataflow analyses that compute result sets.

We discuss in Section 4.1 how comparing the results obtained by two analyses can be done with and without a Gold Standard at hand. Further, we show how and when two analyses of the same kind can be combined to a better, more accurate analysis in Section 4.2. We then present a new idea on how to iteratively improve analysis precision in Section 4.3, where we also point out pitfalls that must be observed when evaluating such improvements on non-conservative baseline analyses.

In the next chapter, we will then present a set of tools that allow us to apply the findings from this chapter to points-to analysis. Thereafter, in Chapter 6, we will present the actual experimental results that investigate the claims from this chapter.

4.1 Comparing Analyses

To compare two analyses, we have to evaluate the garbage (false positives) and misses (false negatives) that they contain, and trade off garbage against misses. Let the Gold Standard $G$ be the exact set of analysis results and $A$ the set of results found with an analysis algorithm (implementation). Precision $P$ and Recall $R$ are then defined as

$$P = \frac{|A \cap G|}{|A|} \quad \text{and} \quad R = \frac{|A \cap G|}{|G|},$$
respectively. $R$ assesses the amount of misses and $P$ the amount of garbage of an analysis, with scores between 0 (worst) and 1 (best).

The recall can often be increased at the cost of reducing the precision and vice versa. Therefore, these two measures are combined into a single measure called $F$-score (harmonic mean of precision and recall) balancing between $R$ and $P$:

$$F = \frac{2 \times P \times R}{P + R}.$$ 

A general $F$-score can even favor either precision or recall, depending on their relevance for an analysis question. For $\omega \in \mathbb{R}$, the weighted F-Score is defined as

$$F_\omega = \frac{(1 + \omega^2) \times P \times R}{\omega^2 \times P + R}.$$ 

The problem is then to decide what “relevant” means, especially if this is determined by a client application of the analysis. Therefore, we stick here to the unbiased definition above.

For any non-trivial analysis question, automated analysis algorithms can only approximate the exact analysis result. In special cases, the approximations may be conservative, $G \subseteq A$, or optimistic, $A \subseteq G$, cf. Section 2.1. Consequently, $R = 1$ for conservative and $P = 1$ for optimistic analyses. For the precision of a conservative analysis and the recall of an optimistic analysis, it then holds

$$P_{cons} = \frac{|G|}{|A|} \text{ and } R_{opt} = \frac{|A|}{|G|}.$$ 

Remember from Chapter 2 that static analysis, which is generally said to compute a conservative solution to an analysis question, quite often cannot handle all programming language features, e.g., for points-to analysis. In this case, we speak of a general analysis.

For comparing two analyses regarding their accuracy, one could measure $P$ and $R$ in some benchmark programs with the Gold Standard known and compare the resulting $F$-scores. This approach is impracticable if no such Gold Standards exist, like for points-to analysis.

**Direct Comparison**

Our first attempt to compare analyses even without a Gold Standard at hand exploits that one of the two analyses or both might be special, i.e., conservative or optimistic, and there are five situations that may occur when comparing different analyses with special cases involved. They are shown in Table 4.1 together with the implications on the result of comparing their accuracy.
### 4.1. Comparing Analyses

Special case of Comparison of $A_1, A_2$ with respect to Analyses $A_1, A_2$

| (1) $A_1, A_2$ conservative | (a) $P_1 \geq P_2 \Leftrightarrow |A_1| \leq |A_2|$ | (b) $R_1 = R_2 = 1$ | (c) $F_1 \geq F_2 \Leftrightarrow |A_1| \leq |A_2|$ |
|-------------------------------|---------------------------------|-----------------|---------------------------------|
| (2) $A_1, A_2$ optimistic     | $P_1 = P_2 = 1$                 | $R_1 \geq R_2 \Leftrightarrow |A_1| \geq |A_2|$ | $F_1 \geq F_2 \Leftrightarrow |A_1| \geq |A_2|$ |
| (3) $A_1$ conservative, $A_2$ optimistic | $P_1 \leq P_2 = 1$ | $R_1 = 1 \geq R_2$ | $F_1 \geq F_2 \Leftrightarrow \frac{|G|}{|A_1|} \geq \frac{|A_2|}{|G|}$ |
| (4) $A_1$ conservative, $A_2$ general | $P_1 \geq P_2 \Leftrightarrow |A_1| \leq |A_2|$ | $R_1 = 1 \geq R_2$ | $F_1 \geq F_2 \Leftrightarrow |A_1| \leq |A_2|$ |
| (5) $A_1$ optimistic, $A_2$ general | $P_1 = 1 \geq P_2$ | $R_1 \geq R_2 \Leftrightarrow |A_1| \geq |A_2|$ | $F_1 \geq F_2 \Leftrightarrow |A_1| \geq |A_2|$ |

Table 4.1: Comparing conservative, optimistic, and general analysis. $G$ denotes the Gold Standard.
As we can see, comparisons based on the analyses results, without knowing the Gold Standard, are (trivially) possible for the first two cases only, when both analyses are conservative or optimistic. Is either of the two analyses a general one, only semi-decisions can be made based on the analysis results alone.

Also, in the special case of comparing a conservative with an optimistic analysis, we need the Gold Standard to get an exact comparison. Note that the semi-decisions for the cases with analysis $A_1$ conservative (respectively optimistic) and $A_2$ general still apply when $A_2$ is optimistic (respectively conservative). Together, they only lead to the trivial observation that $F_1 = F_2 \iff |A_1| = |A_2|$. The proofs for all claims in Table 4.1 are shown in the following:

**Proof 1a.** With $P_{cons} = \frac{|G|}{|A|}$, we get

$$P_1 \geq P_2 \iff \frac{|G|}{|A_1|} \geq \frac{|G|}{|A_2|} \iff |G| \times |A_2| \geq |G| \times |A_1| \iff |A_2| \geq |A_1|$$

**Proof 1b.** If $A$ conservative, then, by definition, $R = 1$

**Proof 1c.** With $R_1 = R_2 = 1$, we get

$$F_1 \geq F_2 \iff \frac{2 \times P_1 \times R_1}{P_1 + R_1} \geq \frac{2 \times P_2 \times R_2}{P_2 + R_2} \iff \frac{2 \times P_1}{P_1 + 1} \geq \frac{2 \times P_2}{P_2 + 1}$$

$$\iff 2 \times P_1 \times (P_2 + 1) \geq 2 \times P_2 \times (P_1 + 1)$$

$$\iff 2 \times P_1 \times P_2 + 2 \times P_1 \geq 2 \times P_1 \times P_2 + 2 \times P_2$$

$$\iff 2 \times P_1 \geq 2 \times P_2 \iff P_1 \geq P_2 \iff \text{(proof 1a)} \ |A_1| \leq |A_2|$$

**Proof 2a.** If $A$ optimistic, then, by definition, $P = 1$

**Proof 2b.** With $R_{opt} = \frac{|A|}{|G|}$, we get

$$R_1 \geq R_2 \iff \frac{|A_1|}{R_1} \geq \frac{A_2}{R_2} \iff |G| \times |A_1| \geq |G| \times |A_2| \iff |A_1| \geq |A_2|$$

**Proof 2c.** With $R_1 = R_2 = 1$, we get

$$F_1 \geq F_2 \iff \frac{2 \times P_1 \times R_1}{P_1 + R_1} \geq \frac{2 \times P_2 \times R_2}{P_2 + R_2} \iff \frac{2 \times R_1}{1 + R_1} \geq \frac{2 \times R_2}{1 + R_2}$$

$$\iff 2 \times R_1 \times (1 + R_2) \geq 2 \times R_2 \times (1 + R_1)$$

$$\iff 2 \times R_1 \times R_2 + 2 \times R_1 \geq 2 \times R_1 \times R_2 + 2 \times R_2$$

$$\iff 2 \times R_1 \geq 2 \times R_2 \iff R_1 \geq R_2 \iff \text{(proof 2a)} \ |A_1| \geq |A_2|$$
4.1. Comparing Analyses

Proof 3a. By definition, if $A_1$ conservative $\Rightarrow P_1 \leq 1$ and $A_2$ optimistic $\Rightarrow P_2 = 1$

Proof 3b. By definition, if $A_1$ conservative $\Rightarrow R_1 = 1$ and $A_2$ optimistic $\Rightarrow R_2 \leq 1$

Proof 3c. With $R_1 = 1, P_2 = 1, P_1 = \frac{|G|}{|A_1|}, R_2 = \frac{|A_2|}{|G|}$, we get

$$F_1 \geq F_2 \Leftrightarrow \frac{2 \times P_1 \times R_1}{P_1 + R_1} \geq \frac{2 \times P_2 \times R_2}{P_2 + R_2} \Leftrightarrow \frac{2 \times P_1}{P_1 + 1} \geq \frac{2 \times R_2}{1 + R_2} \Leftrightarrow \frac{P_1}{1 + P_1} \geq \frac{R_2}{1 + R_2} \Leftrightarrow P_1 \times (1 + R_2) \geq R_2 \times (P_1 + 1) \Leftrightarrow P_1 + P_1 \times R_2 \geq P_1 \times R_2 + R_2 \Leftrightarrow P_1 \geq R_2 \Leftrightarrow \frac{|G|}{|A_1|} \geq \frac{|A_2|}{|G|}$$

Proof 4a. With $|A_1| = \frac{|G|}{P_1}, |A_2| = \frac{|A_2 \cap G|}{P_2}$, we get

$$|A_1| \leq |A_2| \Leftrightarrow \frac{|G|}{P_1} \leq \frac{|A_2 \cap G|}{P_2} \Leftrightarrow \frac{|G|}{|A_2 \cap G|} \times P_2 \leq P_1 \Rightarrow \frac{|G|}{|A_2 \cap G|} \geq 1 \Rightarrow P_2 \leq P_1$$

Proof 4b. By definition, if $A_1$ conservative $\Rightarrow R_1 = 1$ and $A_2$ general $\Rightarrow R_2 \leq 1$

Proof 4c. With $|A_1| \leq |A_2| \Rightarrow \text{proof 4a} \Rightarrow P_1 \geq P_2$ and $R_1 = 1 \geq R_2$ are both precision and recall equal to or better for $A_1$ than for $A_2$, so $F_1 \geq F_2$

Proof 5a. By definition, if $A_1$ optimistic $\Rightarrow P_1 = 1$ and $A_2$ general $\Rightarrow P_2 \leq 1$

Proof 5b. $R_1 \geq R_2 \Leftrightarrow \frac{|A_1 \cap G|}{|A|} \geq \frac{|A_2 \cap G|}{|G|} \Leftrightarrow |A_1| \geq |A_2 \cap G|$. Then,

$$|A_1| \geq |A_2| \Rightarrow |A_1| \geq |A_2 \cap G| \Rightarrow R_1 \geq R_2$$

The reverse does not hold: Let $A_1 \leq A_2, A_2 \cap G = \emptyset$, then $R_1 \geq R_2$.

Proof 5c. With $|A_1| \geq |A_2| \Rightarrow \text{proof 5b} \Rightarrow R_1 \geq R_2$ and $P_1 = 1 \geq P_2$ are both precision and recall equal to or better for $A_1$ than for $A_2$, so $F_1 \geq F_2$
Chapter 4. Comparing, Combining, and Improving Analyses -
The Theory

Approximating Accuracy

Approximating the accuracy is a value in itself and also our second attempt
to compare analyses without a Gold Standard. We assume that we can over-
or under-approximate the Gold Standard using an analysis. Such approxima-
tions can be calculated either from a conservative analysis leading to an over-
approximation, or from an optimistic one leading to an under-approximation.
Define $G^+ \supseteq G$ an over-approximation of the Gold Standard $G$. Analogously,
define $G^- \subseteq G$ an under-approximation of the Gold Standard $G$. Let $F^+$ be
an upper bound of the actual $F$-score, and $F^-$ a lower bound. The idea is
the following: if we could get upper and lower bounds of the exact $F$-scores
$F_1$ and $F_2$ of two analyses $A_1$ and $A_2$, we could (semi-) compare the analyses
by $F_1^- \geq F_2^+ \Rightarrow F_1 \geq F_2$.

We look at the cases when approximating the accuracy of conservative,
optimistic, and general analyses separately.

**Theorem 1.** It is possible to compute upper and lower bounds of the $F$-score
of a conservative analysis based on over- and under-approximations of the
Gold Standard, respectively.

**Proof.** With $A$ conservative, we get

$$F = \frac{2 \times P}{P + 1} = \frac{2 \times |G|}{|A| + 1} = \frac{2 \times |G|}{|G| + |A|}$$

Applying the $F$-score definition accordingly to over- and under-approximations
of $G$ yields

$$\hat{F} = \frac{2 \times |G^+ \cap A|}{|G^+ \cap A| + |A|}, \quad \tilde{F} = \frac{2 \times |G^- \cap A|}{|G^- \cap A| + |A|}.$$

From $G^+ \subseteq A$ follows $G^+ \cap A = G^+$, and from $G^- \subseteq A$ follows $G^- \cap A = G^-$. Thus, $\hat{F} = \frac{2 \times |G^+|}{|G^+| + |A|}$ and $\tilde{F} = \frac{2 \times |G^-|}{|G^-| + |A|}$. Together with $|G^+| \geq |G|$ and
$|G^-| \leq |G|$, it then follows that $\hat{F} \geq F \geq \tilde{F}$.

$G^+ \subseteq A$ does not necessarily hold for every over-approximation of $G$. Assume an over-approximation $G^{++}$ with $G^{++} \setminus A \neq \emptyset$, then we can construct an even better over-approximation $G^+ = G^{++} \cap A$ for which $G^+ \subseteq A$ holds.

Hence, we can always derive lower and upper bounds for the $F$-scores of
conservative analyses: $F^- = \hat{F}$, $F^+ = \tilde{F}$.

Note that this has no effect on the comparability of conservative analyses;
we can compare them directly.

**Theorem 2.** It is possible to compute upper and lower bounds of the $F$-score
of an optimistic analysis based on under- and over-approximations of the
Gold Standard, respectively.

36
4.1. Comparing Analyses

Proof. With $A$ optimistic, we get

$$F = \frac{2 \times R}{1 + R} = \frac{2 \times \frac{|A|}{|G|}}{1 + \frac{|A|}{|G|}} = 2 \times \frac{|A|}{|G| + |A|}.$$  

Applying the $F$-score definition accordingly to over- and under-approximations of $G$ yields

$$\hat{F} = \frac{2 \times |G^+ \cap A|}{|G^+ \cap A| + |G^+|}, \quad \bar{F} = \frac{2 \times |G^- \cap A|}{|G^- \cap A| + |G^-|}.$$  

From $G^+ \supseteq A$ follows $G^+ \cap A = A$, and from $G^- \supseteq A$ follows $G^- \cap A = A$. Thus, $\hat{F} = \frac{2 \times |A|}{|A| + |G^+|}$ and $\bar{F} = \frac{2 \times |A|}{|A| + |G^-|}$. Together with $|G^+| \geq |G|$ and $|G^-| \leq |G|$, it then follows that $\hat{F} \leq F \leq \bar{F}$.

Again, $G^- \supseteq A$ does not hold for all under-approximations of $G$, but can be enforced by setting $G^- = G^- \cup A$ for an arbitrary under-approximation $G^{--}$ if necessary.

In this case $F^- = \hat{F}, F^+ = \bar{F}$. \hfill $\square$

Again, this has no effect on the comparability of optimistic analyses as we can compare them directly as well.

Theorem 3. It is not possible to over- or under-approximate the $F$-score of a general analysis based on approximations of the Gold Standard.

Proof. We construct counter examples from extreme cases:

Case (i): $G^- \subset G \subset A \Rightarrow F > \hat{F}$, since

$$R = \frac{|A \cap G|}{|G|} > \frac{|A \cap G^-|}{|G^-|} = \bar{R} \quad \text{and}$$

$$P = \frac{|A \cap G|}{|A|} > \frac{|A \cap G^-|}{|A|} = \bar{P},$$

Case (ii): $G^- = A \cap G \Rightarrow F < \hat{F}$, since

$$R = \frac{|A \cap G|}{|G|} = \frac{|A \cap G^-|}{|G^-|} = \hat{R} \quad \text{and}$$

$$P = \frac{|A \cap G|}{|A|} = \frac{|A \cap G^-|}{|A|} = \hat{P},$$

Case (iii): $G^+ \supset G \supset A \Rightarrow F > \bar{F}$, since

$$R = \frac{|A \cap G|}{|G|} > \frac{|A \cap G^+|}{|G^+|} \quad \text{and}$$
Chapter 4. Comparing, Combining, and Improving Analyses -

The Theory

\[ P = \frac{|A \cap G|}{|A|} = \frac{|A \cap G^+|}{|A|}, \]

Case (iv): \( G^+ = A \cup G, A \cap G = \emptyset \Rightarrow F < \hat{F}, \) since

\[ P = 0 < \hat{P} = \frac{|A \cap G^+|}{|A|} \quad \text{and} \]

\[ R = 0 < \hat{R} = \frac{|A \cap G^+|}{|G^+|}. \]

This leads to the conclusion that, even with over- or under-approximations of the Gold Standard, we still cannot get any closer to comparing general analyses with one another.

4.2 Combining Analyses

However, to get better analysis results, it is possible to combine analyses. This is interesting if the combined analysis is better than the original input analyses. Based on the results above, we can decide this only for special (conservative or optimistic) analyses. Additionally, it is interesting if the combined analysis is a special case – i.e., conservative or optimistic – too. Table 4.2 summarizes our findings.

The first two cases, combining two conservative or two optimistic analyses, respectively, are quite obvious based on the conclusions from the previous section. The combined analysis is always more accurate than either of the two input analyses. For conservative analyses, this is simply because \( A_1 \cap A_2 \subseteq A_1 \) and because the combined analysis is also conservative. Analogously, for optimistic analyses, \( A_1 \cup A_2 \supseteq A_{1,2} \), and the resulting analysis is optimistic.

When intersecting a conservative analysis \( A_1 \) with a general analysis \( A_2 \) to a combined, general analysis \( A \), the accuracy of this general analysis \( A \) is better than the one of \( A_2 \), as we take away results from \( A_2 \) that are not contained in the Gold Standard only (\( A_1 \supseteq G \)). On the other hand, we cannot say if the conservative analysis \( A_1 \) gets improved by this: The fact that we compare the conservative analysis \( A_1 \) with the combined, general analysis \( A \) allows us to apply the semi-decider \( F_1 \geq F \iff |A_1| \leq |A| \). However, as \( A = A_1 \cap A_2, \ A \subseteq A_1 \) and, hence, \( |A| \leq |A_1| \). Therefore, we can only guarantee \( F_1 \geq F \) in the trivial case of \( |A| = |A_1| \), i.e., when we have not cut off anything from the conservative analysis \( A_1 \).

Likewise, when combining an optimistic analysis \( A_1 \) with a general analysis \( A_2 \) to a combined, general analysis \( A \), the accuracy of the combined,
### Table 4.2: Combining analyses $A_1, A_2$ to analysis $A$.

<table>
<thead>
<tr>
<th>Special case of analyses $A_1, A_2$</th>
<th>Combined analysis $A$</th>
<th>Comparison $A_1, A_2$ with $A$</th>
<th>Combined analysis $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1, A_2$ conservative</td>
<td>$A = A_1 \cap A_2$</td>
<td>$F_1 \leq F, F_2 \leq F$</td>
<td>$A$ conservative</td>
</tr>
<tr>
<td>$A_1, A_2$ optimistic</td>
<td>$A = A_1 \cup A_2$</td>
<td>$F_1 \leq F, F_2 \leq F$</td>
<td>$A$ optimistic</td>
</tr>
<tr>
<td>$A_1$ conservative, $A_2$ general</td>
<td>$A = A_1 \cap A_2$</td>
<td>$F_2 \leq F$</td>
<td>$A$ general</td>
</tr>
<tr>
<td>$A_1$ optimistic, $A_2$ general</td>
<td>$A = A_1 \cup A_2$</td>
<td>$F_2 \leq F$</td>
<td>$A$ general</td>
</tr>
</tbody>
</table>
general analysis \( A \) is better than the accuracy of \( A_2 \), as we only add results that are contained in the Gold Standard (because \( A_1 \subseteq G \)).

However, we again cannot say if the optimistic input analysis \( A_1 \) gets improved: Since we compare the optimistic analysis \( A_1 \) with the combined, general analysis \( A \) allows us to apply the semi-decider \( F_1 \geq F \iff |A_1| \geq |A| \). \( A \) is the union of \( A_1 \) and \( A_2 \), so \( A \supseteq A_1 \), and, hence, \( |A| \geq |A_1| \). Therefore, we can only guarantee \( F_1 \geq F \) in the trivial case of \( |A| = |A_1| \), i.e., when we have not added anything to the optimistic analysis.

The reasoning above continues to hold in the special cases when the general analysis \( A_2 \) is an optimistic or conservative one, respectively. Then we would combine a conservative analysis with an optimistic one, in which case the optimistic analysis is always a subset of the conservative analysis. Then, the intersection or union of the two trivially always leads to one of the input analyses. These cases are therefore omitted in Table 4.2.

### 4.3 Iteratively Improving Analysis Precision

Dataflow analyses can often be iteratively improved by exploiting its intermediate analysis results. For example, when using functional approaches to context sensitivity, cf. Section 2.3, the intermediate points-to sets at call sites are used to determine under which contexts a given call shall be analyzed.

In the following, we present a new approach to path-sensitivity based on filter operations, that also exploits intermediate analysis results to iteratively improve the overall analysis. Then, we discuss pitfalls that must be taken into consideration when evaluating the impact of these filter operations.

**Filter Operations**

Dataflow analyses quite often drop control flow information, e.g., conditions of if-statements. On first sight, this is the obvious approach, since such conditions usually do not contain any assignments and thus do not influence dataflow.

However, in some cases there can be a relation established between the conditions of control flow statements and dataflow. Consider the following piece of Java source code, where the control flow depends on the runtime-type of the variable \( a \):

```java
if (a instanceof T) {
    // then-block
} else {
    // else-block
}
```

In the then-block, the variable \( a \) is guaranteed to be of type \( T \), thus, all abstract objects not of type \( T \) can be excluded from the points-to set of \( a \).
within this block. The opposite goes for the else block: here, all abstract objects that are of type \( T \) can be excluded from the points-to set of \( a \). We can thus logically introduce an assignment to variable \( a \) where we restrict the points-to set of \( a \) at the beginning of the following branch, as in the following example:

```java
if (a instanceof T) {
    // then-block
    a = filter-type-restrict(a, T);
} else {
    // else-block
    a = filter-type-exclude(a, T);
}
```

So far, we have identified five such filter operations applicable to points-to analysis, which are summarized in Table 4.3. The table shows the condition of the control flow, and what filter operations can be introduced in the corresponding branches.

For the first filter operation listed, reference equality, the influence on the true-branch is just the intersection of the points-to sets of both variables. For the false-branch, however, we cannot narrow the possible values of the involved variables, as an abstract object may denote several concrete runtime objects, cf. Section 2.3; an exception is a comparison with the null-constant, our second filter operation. If an equality check for null is performed, then it can be guaranteed on the false branch that the checked variable has a non-null value. An extension to this would be to identify abstract objects that are guaranteed to identify only one concrete runtime object. These could then be treated like null-checks as well. A good candidate for this would be to identify singletons.

Another variation of this kind of filter is comparing the runtime type of an object with a given type, which is the third filter operation shown. Logically, this filtering has been widely used: In [52], for example, the author transforms polymorphic calls in his Memory SSA based program representation into a number of monomorphic calls and inserts type-based filters on the this-values. Figure 4.1 shows code for a polymorphic call together with an outline of the dispatch logic of \( \text{foo()} \), which illustrates this process on source code level: The call \( a.\text{foo()} \) is transformed into comparisons for the runtime type of the expression \( a \), and then monomorphic calls (indicated through static method calls with Java semantics) are performed depending on this runtime type.

The fourth proposed filter operation targets a method invocation to \( \text{Object.equals()} \). Here, \( x1.\text{class}^* == \text{Object.class} \) means that the polymorphic call resolution for an object \( a \) targets the implementation of \( \text{Object.equals()} \); i.e., the class of \( a \) does not overwrite that method. This filter is similar to reference equality, with the exception that it must be limited to those abstract objects whose types do not overwrite \( \text{Object.equals()} \). For other ob-
### Table 4.3: Filter Operations

<table>
<thead>
<tr>
<th>Condition</th>
<th>True Branch</th>
<th>False Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1 \equiv x_2)</td>
<td>(pt(x_1) = pt(x_2))</td>
<td>(pt(x_1) \neq pt(x_2))</td>
</tr>
<tr>
<td>(x \equiv \text{null})</td>
<td>(pt(x) = \text{null})</td>
<td>(pt(x) \neq \text{null})</td>
</tr>
<tr>
<td>(x \text{ instanceof } T)</td>
<td>({a \in pt(x)</td>
<td>a \text{ instanceof } T})</td>
</tr>
<tr>
<td>(x \text{ equals}(x))</td>
<td>({a \in pt(x)</td>
<td>a \text{ equals}(a)})</td>
</tr>
</tbody>
</table>

---
4.3. Iteratively Improving Analysis Precision

```java
class A {
    void foo () { . . . }
    void bar (A a) {
        A. foo ( );
    }
}
class B extends A {
    void foo () { }
}
static dispatch_foo (A a) {
    if (a.getClass () == B. class)
        B. foo (a);
    else if (a.getClass () == A. class)
        A. foo (a);
    // . . .
}
```

Figure 4.1: Polymorphic Call and Dispatch Logic

jects, equals() may return true even if the abstract objects differ, e.g., when comparing two strings.

Our fifth and last proposed filter is the instanceof operation, which has been explained in the introductory example above. Note that the most common use of the instanceof operator is to check an object for a given type and then to cast the object to that type. This may, on first sight, reduce the efficiency of the filter operation in the true-branch, but we gain two advantages here: First, we can assure that the cast itself can be regarded as never failing. Then, a cast in Java will not fail if the value of the casted expression is null; an instanceof-operation will, however, return false in that very case, so that a benefit in filtering is still achieved.

We consider the filter operations that we presented above as a new approach to path-sensitivity for dataflow analysis, as they target the dataflow on different control flow paths. We have presented other approaches to path sensitivity in Section 2.3; compared to those, it is noticeable that our approach does not actually require distinct paths to be analyzed separately, but that it blends in into the baseline dataflow analysis.

**Pitfalls When Improving Analyses**

We have shown that a conservative analysis can be improved iteratively by exploiting intermediate analysis results for proving some dependencies between program entities as non-essential or even some parts of the program as unreachable, e.g., through filter operations. A conservative, improved analysis $A^f_{cons}$ is always smaller and, hence, more accurate than its corresponding baseline-analysis $A^{u}_{cons}$, i.e., it is more precise as $P_f \geq P_{uf}$ and has the same recall $R^f_{cons} = R^{u}_{cons} = 1$.

This does not continue to hold for general analyses, as it might miss some actual results. A general improved analysis $A^f$ is still smaller but not necessarily more accurate than its baseline-analysis $A^{u}_{f}$. As shown with two extreme case, the $F$-scores can theoretically both improve (i) and deteriorate (ii).
Chapter 4. Comparing, Combining, and Improving Analyses -
The Theory

Case (i): \((A^f = A^{uf} \cap G) \Rightarrow \)

\[
p^f = \frac{|A^f \cap G|}{|A^f|} = \frac{|A^{uf} \cap G|}{|A^{uf}|} \geq \frac{|A^{uf} \cap G|}{|A^{uf}|} = p^{uf}
\]
\[
R^f = \frac{|A^f \cap G|}{|G|} = \frac{|A^{uf} \cap G|}{|G|} = R^{uf}.
\]

Case (ii): \((A^f = A^{uf} \setminus G) \Rightarrow \)

\[
p^f = \frac{|A^f \cap G|}{|A^f|} = \frac{0}{|A^{uf} \setminus G|} \leq \frac{|A^{uf} \cap G|}{|A^{uf}|} = p^{uf}
\]
\[
R^f = \frac{|A^f \cap G|}{|G|} = \frac{0}{|G|} \leq \frac{|A^{uf} \cap G|}{|G|} = R^{uf}.
\]

The same considerations are also valid for (new) context-sensitive approaches. This observation has consequences for all researchers evaluating any improvement to dataflow analysis by comparing its result \(A^f\) with the baseline analysis \(A^{uf}\). If the baseline analysis is non-conservative, e.g., due to some unexpected dynamic class loading, we cannot in general deduce that the improved analysis is more precise, although the measurements say that \(A^{uf} > A^f\).

We demonstrate this with an example. Consider the following piece of Java code:

```java
1: void foo(Object p1, Object p2) {
2:    if (p1 == p2) {
3:        p1.toString();
4:    }
5: }
```

Assume that the correct points-to sets of \(p1\) and \(p2\) are \(Pt(p1) = \{o\}\) and \(Pt(p2) = \{o\}\), respectively. Now assume that a general analysis falsely computes that \(o\) is not part of the points-to set of \(p2\). Then, the filtered points-to set of \(p1\) in line 3 is the empty set, and, thus, the call to \(toString()\) is falsely analyzed as unreachable. This would not be the case when not using the filter operations.

4.4 Conclusion

We can compare analyses with respect to accuracy only in special cases, i.e., when we compare either conservative or optimistic analyses. For all other kinds of comparison, we would need a Gold Standard, which currently does not exist for points-to analysis.
Combining two analyses is possible when both are either conservative or optimistic. The intersection of two conservative analyses is at least as accurate as the better of the two, and the same holds for the union of two optimistic analyses. In Section 6.2, we will evaluate if combining two conservative analyses shows benefits in practice.

Additionally, a general analysis can be improved by either adding the results from an optimistic analysis, or intersecting it with the results from a conservative analysis. Any other combination of analyses yields, in the absence of a Gold Standard, an unspecified analysis with respect to accuracy.

We have also presented a new approach to path-sensitivity for points-to analysis through the means of special filter operations that are derived from control flow statements. With the help of these filter operations, control flow can be converted to dataflow in certain cases, which then allows it to be seamlessly integrated into existing dataflow analysis implementations. However, we have also shown that we must be careful when evaluating such improvements: if the baseline analysis is not conservative, then we cannot tell if the improvement does more harm than it has use.
Chapter 5

Comparing, Combining, and Improving Points-to Analyses - The Praxis

In the previous chapter, we have discussed how different dataflow analyses can be compared and combined, and what has to be taken into consideration when improving a given analysis. In this chapter, we present a set of tools and specifications that allows us to perform these tasks for points-to analysis.

Points-to analysis has no value in itself, but serves as input to a numerous amount of client analyses, such as call-graph construction, alias-analysis, escape analysis, etc. We therefore describe a number of such client analyses in Section 5.1. For the purpose of this thesis, we limit ourselves to simple clients that can be easily derived from points-to analysis (e.g., call-graph construction), and omit more complex analyses like escape analysis. These metrics will then be used in the evaluation part of this thesis, Chapter 6.

Different points-to analysis implementations use different underlying internal data-structures. We therefore need to define a common interface to our tools, which we then can connect the implementations described in Chapter 3 to. We decided to define an XML specification, presented in Section 5.2, that captures the data required for the clients described in Section 5.1. Even our dynamic agent generates data for this XML specification. As the specification abstracts from the implementation details of the different points-to analysis implementations, it can also be used to make points-to information accessible to the research community.

The general process of comparing and combining points-to analyses is depicted in Figure 5.1: Dynamic and different static points-to analyses are performed on a given input program. Then, client analyses are computed with the results from these points-to analyses; these client analyses need to be implemented once for each points-to analysis implementation. Their results are then stored in a common exchange format. The actual comparison and combination of the different points-to analysis results is performed by a small tool, which simply reads the results which are present in XML-documents, and computes the approximated precision, recall, and combinations of differ-
5.1 Client analyses

When we compared the results obtained from the static and the dynamic analyses, we found a number of practical problems. We discuss these in Section 5.3.

In Section 5.4, we present a new front-end for Points-to SSA that is capable of generating the filter operations presented in Section 4.3.

5.1 Client analyses

The results of points-to analysis are usually very low-level, e.g., for each local variable access, a points-to set of abstract objects possibly reaching this variable access is computed. Such information has little value in itself. Further, flow-insensitive analysis does not distinguish different accesses to the same local variable, and in Memory SSA, local variables are resolved to dataflow edges and a mapping back to the local variables may not be possible. This means that a direct comparison is not only of questionable use, but also infeasible in practice.

Therefore, in order to compare points-to analysis results, we need to use client analyses that can be derived from different points-to analysis implementations. Well-known client analyses are, for example, call-graph construction and escape analysis. We stick to rather simple and easy-to-derive client analyses. Some of these are still very low-level, i.e., they are close to the underlying points-to information. We thus have realistic client analyses that should still be capable of distinguishing small differences in analysis precision. In the following, we list the client analyses that we use for the evaluation in the next chapter and that the support tools and the XML exchange format presented in this chapter support.
Definition For a given program $P$, let $M$ be the set of methods in $P$ and $O$ be the set of all object creation sites in $P$. Let then the **exact object call graph** consist of all the edges $[o_i, m_j] \rightarrow [o_k, m_l] \in O \times M \times O \times M$ where there exists a concrete execution of $P$ so that (1) $m_j$ is called on an instance object $i_v$, (2) during the execution of $i_v.m_j$ occurs a call to $m_l$ on instance object $i_w$, and (3) $i_v, i_w$ were created at the object creation sites $o_i$ and $o_k$, respectively.

Definition For a given program $P$, the **exact call graph** is the projection of the exact object call graph so that edges $[o_i, m_j] \rightarrow [o_k, m_l]$ are projected to edges $m_j \rightarrow m_l$.

Definition For a given program $P$, let $F$ be the set of all fields in $P$ and $O$ be the set of all object creation sites in $P$. Let then the exact abstract heap consist of all the relations $[o_i, f_s] \leftarrow o_j \in O \times F \times O$ where there exists a concrete execution of $P$ so that (1) an instance object $i_w$ is stored into the field $f_s$ of an instance object $i_v$, and (2) $i_v, i_w$ were created at the object creation sites $o_i$ and $o_j$, respectively.

The exact object call graph, exact call graph, and exact abstract heap can be approximated by both dynamic and static analysis, which yields either under- or over-approximations of these. We then speak of the client analyses **object call graph**, **call graph**, and **abstract heap**, respectively.

**Metrics**

Based on these clients, we define the metrics *reachable methods* (in short, $N$ for nodes), *call graph edges* ($E$), *object call graph nodes* ($ON$), *object call graph edges* ($OE$), and *heap size* ($H$). These are defined as the number of nodes and edges, respectively, in the (object) call graph clients, and the number of relations in the heap client.

**Application entities**

In order to avoid taking into account results due to the same set of Java library and Java Virtual Machine start-up classes over and over again, we decided to limit the client analyses as follows:

We select a subset of all types in each benchmark program and denote them **application types**. A simple name filter on the fully qualified type names does this job. Members defined in these types are denoted application members, and abstract objects corresponding to allocations of application classes are denoted application objects.
5.2 XML exchange format

The only specification for exchanging results obtained from points-to information that we are aware is ProBe [1], which describes data formats for recording program behavior. ProBe specifies data formats for different client analyses, namely call graph, polymorphic call sites, cast safety, side effect information, and escape analysis. It also specifies a format for capturing complete points-to information. However, these data formats are not fit for our purposes because of the following reasons: (1) The clients object call graph and heap are not specified. We feel that these two metrics capture a very precise picture of the precision of points-to analysis and should thus be supported by the Gold Standard. (2) These metrics could of course be computed by the format for exchanging complete points-to information. However, this format is tailored to Java bytecode, i.e., points-to information is saved for each bytecode instruction. However, in Points-to SSA, local variables are no longer represented as nodes, and we did not see a straight-forward approach to map back dataflow-edges to the operations that correspond to stack-related bytecode instructions. (3) Storing points-to information for every instruction in the dynamic agent appears infeasible to us. We thus need an own data format for the dynamic agent anyway.

We have thus designed an XML schema definition (XSD) that allows to describe the results for the client analyses presented in the previous section, as well as some additional information. The current version of the XSD, together with an example usage of it, can be found in Appendix B. In the schema, abstract objects are identified by their type, and the line number and class at which they are created in source code, i.e., the creation site naming scheme is used. Such information must be available from any connected points-to analysis implementation, otherwise the exchange format cannot be used.

A valid XML-file with regards to this schema consists of the following:

1. The name and version of the analyzed project.

2. The analysis setup, i.e., how the results contained in the XML file have been obtained, as a textual description.

3. The name filter for identifying application entities, as described in Section 5.1.

4. The nature of the analysis: optimistic, conservative, general, or manual proof.

5. A number of boolean flags that indicate the exact content of the XML file, e.g., what clients have been computed.
5.3 Problems when comparing results from different implementations

Even if two analyses use the same ideas for abstraction, there may still be difficult-to-compare differences in analysis details. In this section, we discuss problems that arise when comparing results from different implementations as well as results obtained from static and dynamic analyses, and present solutions (or workarounds) for them where possible.

The difficulties that we discovered while comparing analysis results are the following: (1) Array handling, (2) class initializer (<clinit>) handling, (3) constructor handling, (4) field handling, and (5) Mapping to abstract objects. The former two are related to different implementation choices in the static analyses, while the latter three are imposed by the implementation of the dynamic agent. We discuss each of these issues in more detail in the following.

Array handling

Arrays in the sense of Java are objects (almost) just like any other object. However, points-to analyses often treat them specially, for example, multi-dimensional arrays are “flattened” to one-dimensional arrays – this is ok as the shape of the array usually cannot be analyzed anyway – or they could even...
be treated as variables of their base-type. This simplifies and speeds up the analysis, but points-to information for the array objects are not computed. For example, such an abstraction does not capture the fact that creating a multi-dimensional array creates many objects on the heap at runtime. Since such an approach is done at least in Points-to SSA, we decided not to take array objects into consideration in our client analyses.

Class initializer handling

Each Java class may have a class initializer (denoted <clinit>), which is executed once a class is being loaded, i.e., referenced for the first time. Points-to analysis implementations need to take them into account. However, no such class initializer needs to be present, in which case some implementations may choose to create an artificial, empty one. These then show up in the (object) call graphs and alter the results, that is why we choose to not count (object) call graph edges to class initializers. We do count edges from class initializers, as they require the class initializers to be present in the bytecode.

Constructor handling

A problem with the mapping from runtime to abstract objects in the dynamic agent is that this mapping is not computed before the Object constructor is reached. That means that, in the call graph obtained by dynamic analysis, the edges from constructors to their superclass-constructors are missing. Therefore, we choose not to count these edges when comparing dynamic and static analyses, to get a better approximation of the precision of static analysis. Usually, this is not even a bigger problem at all, as the first statement in a constructor must always be an invocation to either another constructor of the same class, or of a superclass constructor (if there is none, then it is implicitly added by the compiler, cf. § 8.8.7 in [11]). Unfortunately, such a special constructor invocation may contain method calls, which are effectively placed before the mapping to the runtime object to the abstract object, so that our dynamic agent misses some (object) call graph edges – and possibly even field assignments. Thus, our current dynamic agent implementation captures less information than could actually be observed. However, at least for the benchmark programs we use, this does not happen often. Note that this does not invalidate the results from the dynamic agent, as the optimistic analysis only gets “more optimistic”.

Field handling

The definition of our heap client says that abstract objects of application types stored in application fields belong to the heap client. Due to performance reasons, our dynamic agent monitors only those fields whose static
Chapter 5. Comparing, Combining, and Improving Points-to Analyses -
The Praxis

type is of an application type; however, an abstract object of application
type may be stored into a field of, e.g., static type java.lang.Object. In order
to make the results from the static and the dynamic analyses better com-
parable, we wrote a small tool that creates a list of fields that should be
considered when comparing the different heap results. On this list are the
fully-qualified names of all fields declared in application classes that are also
of an application type.

Mapping to abstract objects

The points-to analyses used in this thesis use the creation site naming scheme
(i.e., the newinstance instruction in Java bytecode) for mapping possible runtime objects to abstract objects\(^1\). Once the Object-constructor has been
called, our dynamic agent inspects the stack to identify the corresponding
constructor invocation (i.e., the invokespecial instruction in Java bytecode),
and uses this instruction for determining the abstract object, cf. Section 3.3.
Unfortunately, the line numbers for these two instructions differ when the
syntactical new and the closing parenthesis of the constructor call are not on
the same source code line. Since this would cause errors in the evaluation,
we decided to manipulate the source code prior to analysis. For this, we
have written a small source code analysis tool that identifies positions in
the source code where this problem occurs. We then manually patched the
corresponding source code lines\(^2\).

Refined Process

Due to the issues described above, the process described in Figure 5.1 is
refined, cf. Figure 5.2. New are the source code modification, which is
required for handling the mapping to abstract objects, and the field list,
which is used to identify the fields that should be evaluated for the heap
client.

5.4 The Points-to SSA Front-end

We have a running front-end implementation for Points-to SSA based on
Shimple\(^3\), which is part of the soot framework. From Shimple, this front-end
generates Points-to SSA. However, the process of the mapping Shimple to
Points-to SSA is not very flexible, so that introducing the filter operations,

\(^1\)As discussed in Section 3.1, Spark merges abstract objects of the same type created in
the same method. However, this is not relevant for this discussion.
\(^2\)Performing a format source code with a properly configured formatter solves most of
these, so that hardly any work is involved.
\(^3\)http://www.sable.mcgill.ca/soot/tutorial/shimple/
5.4. The Points-to SSA Front-end

as described in Section 4.3, appears too difficult. We thus decided to implement a new front-end based on the Java implementation of FIRM, which was kindly provided by Florian Liekweg. This front-end is based on our initial implementation described before [13].

With FIRM, we have a complete program representation in Memory SSA at hand, from which we can create Points-to SSA as well as extensions to it. The steps involved for generating a Points-to SSA graph from a FIRM graph are the following: (1) We map FIRM to a Points-to SSA like sparse representation, which is still suited for graph-transformations. (2) Transformations are applied. This includes introducing the filter operations described in Section 4.3. (3) Another mapping, this time from our sparse intermediate representation to Points-to SSA, is performed.

**Step 1: Mapping to a sparse representation**

In this first step, we map the full intermediate representation FIRM to a sparse, compact representation that is more similar to Points-to SSA. For instance, calls, with consist of at least two nodes in FIRM (cf. Section 2.2), are now represented by a single node. Additionally, Proj nodes are removed; their semantics are captured by ports, cf. Section 3.2.

Besides compacting the representation, there are two options that steer the level of detail of the sparse representation. These options are: (a) whether to remove the basic block structure including control flow nodes, e.g., Jmp nodes; (b) whether to remove operations regarding primitive types. If control flow operations are kept, then this will not affect operations that will later on be transformed into filter operations.

When both option (a) and (b) are enabled, then the result will be a variation of Points-to SSA as described in Section 3.2 and Appendix A. Option
Step 2: Transformations

We currently perform two transformations: (a) Removing redundant \textit{Load} nodes, (b) enable the filter operations. While removing redundant \textit{Load} nodes is actually done first, its motivation comes from introducing the filter operations; we thus discuss it last.

\textbf{Introducing Filter Operations}  
The nodes required for the filter operations could be retrieved by extending the original source code (or any other full program representation) and add statements as in the example in Section 4.3. However, this would require changing the Memory SSA construction algorithm as well as all involved intermediate program representations.

Instead of introducing additional nodes within Memory SSA, the existing nodes – for example, the \textit{instanceof} and \textit{EQ} nodes – take over the computations of the filtered points-to sets. We thus define the nodes’ transfer functions accordingly: Instead of computing boolean values, they now compute points-to sets.

Uses of the operands in the according branches – computed by dominator relation – are replaced by the filtered values computed by these operations.

Figure 5.3 shows an example of this transformation. Note that, for simplicity, the basic block structure, which is required for the computation of the dominator relations, as well operations related to intra-procedural control flow are not included in the graph. The left of the two graphs shows a regular Points-to SSA graph with the relevant equality operation retained. It does, at this point, not have any out-port because the normal semantics of such a node is to compute a \textit{boolean} value, which is not represented in Points-to SSA. After the transformation, the equality operation is turned into a filter operation and, since it clearly dominates the store operation to \textit{A.y}, that operation’s input value is replaced by the filtered value, as can be seen in the right graph. Note that this filter-operation computes three values: The intersection of the two input values for the \textit{then}-branch, and filtered values for each of the variables for the \textit{else}-branch, which are however not used in this example.

\textbf{Removing Redundant Load nodes}  
Depending on programming style, multiple accesses to the same field may occur in sequence. In Figure 5.4(a), we have rewritten the previous example such that we do not create a local variable to which we copy the contents of the field \textit{x}. Now, an additional \textit{Load} node is inserted. The filter operation is now performed for the first


5.4. The Points-to SSA Front-end

```java
class A {
    Object x, y, z;
    void foo(A a) {
        Object m = y;
        if (a.x == m) {
            z = m;
        }
    }
}
```

Figure 5.3: Transformation on Points-to SSA Graphs: Source code, and Points-to SSA graph before and after transformation.

load operation in line 4, while the assignment in line 5 will use the second load operation, which is not affected by the filter operation.

Such redundancies could be removed by SSA construction algorithms, but the implementation that we use does not do this. In order to keep our filter operations effective in such cases, we implemented an algorithm that removes such redundant operations. The algorithm is outlined in Algorithm 6 and works as follows: It searches the memory dependencies of read accesses upwards and checks if it discovers an equal (same field, same address) `Load` node. If so, the two nodes are merged. It stops the upward traversal if it encounters an operation that potentially changes the memory with respect to the referenced field. Such operations include store operations to the same field, method invocations, and memory-ϕ nodes.

**Step 3: Mapping to Points-to SSA**

Once the transformations from step 2 have been performed, we can map the representation to Points-to SSA as it is expected by the analysis back-end. For instance, the basic block structure as well as `Jmp` nodes can be discarded.
Chapter 5. Comparing, Combining, and Improving Points-to Analyses - The Praxis

```java
class A {
    Object x, y, z;
    void foo(A a) {
        if (a.x == y) {
            z = a.x;
        }
    }
}
```

Figure 5.4: Sequential Accesses to Same Field

**Algorithm 6** RemoveDuplicateReads

```java
for each n ∈ graph.nodes do
    if n instanceof Load then
        in = getMemoryInNode(n)
        while doesNotChangeMemFor(n, in) do
            if in instanceof Read
                ∧ accessSameField(n, in)
                ∧ useSameAddr(n, in) then
                    n.replaceWith(in)
                break
            else
                in = in.getMemoryInNode(n)
            end if
        end while
    end if
end for
```
Adaptation of the actual analysis

Since we now have two additional node types, $EQ$ and $instanceof$, in our Points-to SSA graphs, the analysis back-end also needs to be updated. This is simply done by defining and implementing the transfer functions for these two node types.

Algorithm 7 shows the transfer function of $EQ$ nodes. Three values are computed: The first one is the set-intersection of the two input values, which is used in the then-block of a corresponding branch. The other two are for filtered values of $v_{in1}$ and $v_{in2}$ in the else block. Here, only null values can be filtered under certain conditions; this corresponds to filter 2 from Table 4.3.

Algorithm 8 shows the transfer function of $instanceof$ nodes. Here, the computation is done according to the filter-definition from Table 4.3.

---

Algorithm 7 $EQ$: $[v_{in1}, v_{in2}] \mapsto [v_{out1}, v_{out2}, v_{out3}]$

\[
\begin{align*}
    v_{out1} &= v_{in1} \cap v_{in2} \\
    v_{out2} &= v_{in1} \\
    v_{out3} &= v_{in2} \\
    \text{if } v_{in1} &= \{\text{null}\} \text{ then} \\
    &\quad v_{out3} = v_{out3} \setminus \{\text{null}\} \\
    \text{end if} \\
    \text{if } v_{in2} &= \{\text{null}\} \text{ then} \\
    &\quad v_{out2} = v_{out2} \setminus \{\text{null}\} \\
    \text{end if} \\
    \text{return } [v_{out1}, v_{out2}, v_{out3}]
\end{align*}
\]

---

Algorithm 8 $InstanceOf^T$: $v \mapsto [v_T, v_{\neg T}]$

\[
\begin{align*}
    v_T &= \{o \in Pt(v) \mid o \text{ instanceof } T\} \\
    v_{\neg T} &= \{o \in Pt(v) \mid \neg(o \text{ instanceof } T)\} \\
    \text{return } [v_T, v_{\neg T}]
\end{align*}
\]
Chapter 6

Evaluation

In Chapter 3, we have presented two static points-to analysis implementations and an agent that captures dynamic points-to information. Then, in Chapter 4, we have discussed how results from different dataflow (including points-to) analyses can be compared, combined, and we have pointed out pitfalls that must be observed when evaluating an improvement to dataflow analysis if the baseline analysis is not sound. We have also discussed a new way to improve (conservative) points-to analysis, filter-operations. In the previous chapter, we have presented an implementation of these filter operations, and we also discussed practical problems when comparing and combining points-to analyses, as well as the necessary tools to perform the actual tasks. Now, we present the results that we obtained from our experiments, which we performed with the following ideas in mind:

1. We want to compare different points-to analyses.
2. We want to evaluate the synergy effects of combining different points-to analyses.
3. We want to investigate the effects of improving analyses as discussed in Section 4.3.

This chapter is organized as follows: In Section 6.1, we present the benchmark programs we use and describe the overall setup of our experiments. We also give a description of what input data we fed into the dynamic runs.

In Section 6.2, we compare the results of the different points-to analyses. Since investigating the effects when applying improvements to analyses or combining different analyses can only be done by comparing, we do this in the same section. Then, in Section 6.3, we evaluate the new Points-to SSA front-end with respect to its flexibility.

6.1 Experimental Setup

We performed our experiments with the concrete points-to analysis implementations presented in Chapter 3. Our Points-to SSA based Simulated Execution comes in four variants: context-insensitive (we refer to it as ConIns),
1-this-sensitive (*ThisSens*), 1-object-sensitive (*ObjSens*), and 1-CFA (*CallString*). Additionally, we run *Spark*. In order to get the performance of the different implementations as comparable as possible, we configured Spark so that it uses a creation site naming scheme even for *StringBuffer* and *StringBuilder* objects\(^1\), cf. the description of Spark, Section 3.1. This way, all variants use an identical naming scheme; however, note that differences in support for native methods still cause that different parts of the same benchmark program are considered by each implementation.

We performed our experiments on twelve different benchmark programs, which are listed in Table 6.1. We list the name and version of the program, its homepage, a short description of what it does, and with what input we run the data collection runs for the dynamic analysis. For the first four programs, all our static analyses are conservative. For the fifth program, Spark is not conservative, as it lacks support for a native method used by the program\(^2\). For the other programs, all the static analyses are general. We discuss how we decided whether or not an analysis is sound for a given benchmark program further down.

Note that we inspected and changed the source code of the benchmark programs as described in Section 5.3.

All experimental data presented in this thesis is the median value of three runs on the same computer (Dell PowerEdge 1850, 6GB RAM, Quad Intel Xeon 3.2GHz under Linux x86-64, kernel 2.6.22.1). We run Java 1.6.0 _11, 64 bit, with the options `-Xmx5500M -Xss100M`.

### Conservative Points-to Analyses?

We shortly discuss how we decided if a points-to analysis implementation is conservative for a given benchmark program. Of course, this is never the case, as, for example at program start-up, lots of dynamic class loading is performed, and lots of native code is executed. However, this should not influence our client analyses, as we defined them so that they focus solely on application entities.

Therefore, we made sure that native methods – this includes dynamic class loading and reflection – that are directly or indirectly referenced from the program are supported by the respective analysis. We performed this by simply searching for references to external libraries, and checking if the referenced code appears to make use of native methods. This was done by manual inspection, so that the results are, naturally, unverified. However, the results obtained from dynamic analysis did not prove us wrong so far.

---

\(^1\) We set the options `merge-stringbuffer` to *false*.

\(^2\) `java.lang.reflect.Array.newInstance(Class)`
<table>
<thead>
<tr>
<th>Program</th>
<th>Homepage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat 1.0</td>
<td>javabloat.sf.net</td>
<td>EDU A Java bytecode optimizer</td>
</tr>
<tr>
<td>javac 1.3.1</td>
<td>com.sun.tools, sun.tools</td>
<td>The Java compiler that comes with JDK 1.3.1</td>
</tr>
<tr>
<td>javac and javadoc 1.3.1</td>
<td>javadoc that comes with javac</td>
<td>same as javac</td>
</tr>
<tr>
<td>javacc 4.2</td>
<td>javacc.dev.java.net</td>
<td>org.javacc An LL(k) parser generator</td>
</tr>
<tr>
<td>sablecc 3.2</td>
<td>sablecc.org</td>
<td>org.sablecc An LALR(1) based parser generator</td>
</tr>
<tr>
<td>recoder 0.93</td>
<td>recoder.sf.net</td>
<td>recoder A program analysis and metaprogramming framework for Java</td>
</tr>
<tr>
<td>antlr 2.7.7</td>
<td><a href="http://www.antlr.org">www.antlr.org</a></td>
<td>antlr All grammars that come with the distribution</td>
</tr>
<tr>
<td>emma 2.0.5312</td>
<td>emma.sf.net</td>
<td>com.vladium A Java code coverage tool</td>
</tr>
<tr>
<td>javadoc 1.3.1</td>
<td>com.sun.tools, sun.tools</td>
<td>same as javac</td>
</tr>
<tr>
<td>jfreechart 1.0.13</td>
<td>jfree.org/jfreechart</td>
<td>org.jfree A Java library for generating and displaying charts</td>
</tr>
<tr>
<td>jython 2.2.1</td>
<td><a href="http://www.jython.org">www.jython.org</a></td>
<td>org.python Java-based Python implementation</td>
</tr>
<tr>
<td>lucene 2.4.1</td>
<td>lucene.apache.org</td>
<td>org.apache A search engine</td>
</tr>
<tr>
<td>pmd 4.2.5</td>
<td>pmd.sf.net</td>
<td>net.sourceforge.pmd A code style / linter check-er</td>
</tr>
</tbody>
</table>

Table 6.1: Benchmark Programs
6.2 Comparing, Combining, and Improving Analyses

We will first take a general overview of the analysis results based on average precision and recall for the different analyses. Then, we look at the performance, before checking if the special cases listed in Table 4.1 can be observed in practice. After that, we check what happens when improving general analyses, and then give an evaluation of the filter operations from Section 4.3.

Our approach in this section is to use the results obtained from dynamic analysis as an under-approximation of the Gold Standard, and compare these with the results of the different static analyses. We can thus compute $\bar{P}$, $\bar{R}$, and $\bar{F}$. Remember from Chapter 4 that, in case of conservative analyses, this yields lower bounds $P^{-}$ and $F^{-}$ for the static analyses ($R = 1$ holds for the recall). For general analyses, this approach has theoretically no value, as it is unclear if $\bar{P}$, $\bar{R}$, and $\bar{F}$ are upper or lower bounds of the actual values. However, we believe that, in practice, the results can still be used as an indicator for comparing analyses.

For the sake of readability, we give $\bar{P}$, $\bar{R}$, and $\bar{F}$ in percentage, where 0% is the worst and 100% is the best possible value. The complete metrics data can be found in Appendix C. In this chapter, we use diagrams summarizing the data and cite numbers from the tables in Appendix C where we find this fit.

Overview

Figures 6.1 to 6.5 show the average, approximated $\bar{P}$, $\bar{R}$ and $\bar{F}$ for the for the metrics heap, call graph nodes, call graph edges, object call graph nodes, and object call graph edges, respectively. Shown are the results for the different Points-to SSA based Simulated Execution setups, Spark, as well the results from the combination of the context-sensitive variants. Note that the benchmark programs jfreechart and jython did not run through for the object-sensitive approach. The analyses either run out of memory, or we stopped the analyses after running them for three straight days. Thus, we did not count these two benchmark programs into the average results.

Looking at the average $\bar{P}$, we see that ConIns seems more precise than Spark; this comes at no surprise, since the former is, unlike Spark, flow-sensitive. The context-sensitive variants of Points-to SSA based Simulated Execution have naturally a higher $\bar{P}$ than ConIns; among them, ThisSens and ObjSens are more precise than CallString for the more low-level metrics (heap and object call graph edges); ObjSens seems most precise for object call

---

3The complete metrics data per benchmark program are listed in Tables C.2 to C.6.
Figure 6.1: The heap metric. Average, approximated precision $\hat{P}$, recall $\hat{R}$, and $F$-score $\hat{F}$ for the different analyses. "*-F", e.g., "ConIns-F", indicates path-sensitivity.

The average $\hat{R}$ is lowest for the object call graph edges metric (76-78%) and object call graph nodes metric (79-81%), better for call graph nodes (80-83%) and edges (83-85%), and best for heap (86-88%). Spark shows an overall better $\hat{R}$, which can be explained by its better handling of dynamic class loading. Note, for Points-to SSA based Simulated Execution, $\hat{R}$ does not seem to deteriorate when going from the baseline analysis (ConIns) to the improved analyses; this holds for both context sensitivity as well as the filter-operations.

When looking at the average $\hat{F}$, it appears that the better $\hat{P}$ of Points-to SSA based Simulated Execution outweighs the better $\hat{R}$ of Spark.
6.2. Comparing, Combining, and Improving Analyses

Figure 6.2: The call graph nodes metric. Average, approximated precision $\hat{P}$, recall $\hat{R}$, and $F$-score $\hat{F}$ for the different analyses. “-F” indicates with path-sensitivity.

Figure 6.3: The call graph edges metric. Average, approximated precision $\hat{P}$, recall $\hat{R}$, and $F$-score $\hat{F}$ for the different analyses. “-F” indicates with path-sensitivity.
Chapter 6. Evaluation

Figure 6.4: The *object call graph nodes* metric. Average, approximated precision $\hat{P}$, recall $\hat{R}$, and $F$-score $\hat{F}$ for the different analyses. “-F” indicates with path-sensitivity.

Figure 6.5: The *object call graph edges* metric. Average, approximated precision $\hat{P}$, recall $\hat{R}$, and $F$-score $\hat{F}$ for the different analyses. “-F” indicates with path-sensitivity.
6.2. Comparing, Combining, and Improving Analyses

Performance

Figure 6.6 shows the total time in seconds that is required to run all the analyses\(^4\). Note that we omitted graph construction, analysis setup, and metrics computation. Since SSA construction is naturally more complex than constructing Spark’s pointer assignment graph, the constant, analysis-independent overhead should be higher for Points-to SSA based Simulated Execution\(^5\). However, in our context, time is of not so much importance, and when the points-to analysis is used in an edit-compile cycle, method graphs of unchanged methods can be reused.

ObjSens is by far the slowest, while ThisSens is almost as fast as ConIns. Spark is just a little bit faster than ConIns. The filter operations slow ConIns and CallString down, while ObjSens becomes a little bit faster with them, probably because the fixed point is reached faster on average.

If we look at the absolute execution times, we get that \textit{ConIns} varies between 9.6 seconds (lucene) and 668 seconds (emma) for the ten input programs considered, while \textit{Spark} varies between 41 seconds (javadoc) and 173 seconds (recoder). Thus, Spark has much more “stable” execution times, while ConIns can be considerably faster for some input programs, and much slower for others.

However, performance is only a subordinated goal with this thesis; we thus only conclude that the filter operations do not considerably slow down the analysis, and that \textit{Spark} and \textit{ConIns} are also in the same range of speed on average.

Five Special Cases

Remember Table 4.1 from Section 4.1; there, five cases in which we can compare two different analyses are listed. We now investigate if we can observe the non-trivial cases in practice. Cases 1 and 2 are omitted, as comparing two conservative or two optimistic analyses with each other is trivial.

Case 3: Remember that the implication for the \(F\)-score requires the Gold Standard \(G\) at hand: \(F_1 \geq F_2 \iff \frac{|A_1|}{|G|} \leq \frac{|G|}{|A_2|}\). Since we do not have \(G\) for any of the benchmark programs, this case is impossible for us to observe.

Case 4: If \(A_1\) conservative, \(A_2\) general, then \(P_1 \geq P_2 \iff |A_1| \leq |A_2|\) (likewise holds for the \(F\)-score). We actually observe this case for all the

\(^4\)The analysis times for the different benchmark programs can be found in Table C.1.
\(^5\)The worst-case that we observed is for project \textit{jfreechart}. Here, Spark requires 33 seconds to build its PAG, while constructing Points-to SSA (including FIRM construction) requires 158 seconds.
Figure 6.6: Performance: Sum of analysis execution times. Points-to SSA construction and PAG construction, respectively, as well as setup and IO are omitted.

metrics in project *recoder*, where the general analysis (Spark) shows bigger result sets than the conservative Points-to SSA based analyses (including the context-insensitive one).

Case 5: If $A_1$ optimistic, $A_2$ general, then $R_1 \geq R_2 \iff |A_1| \geq |A_2|$ (likewise holds for the $F$-score). This case is observed for the call graph metrics of project *antlr* and for all static analyses, which have smaller result sets than the corresponding dynamic analysis.

**Improving General Analyses**

We observe drops in the approximated recall $\hat{R}$ only in a few cases:

- for *jython*, the filter operations cause one additional call graph edge miss as well as 16 and 20 additional misses in the object call graph nodes and edges, respectively. The approximated recalls drop accordingly by 0.016%, 0.28%, and 0.2%. Also, when going to CallString, an additional (object) call graph edge, which is not missed otherwise, is missed when using the filter operations. Then, ThisSens causes 37 additional misses for heap, which corresponds a drop of 0.86%.

- for *lucene*, when going from ConIns to ObjSens and ThisSens, there are two more misses in the object call graph edges, which corresponds to a drop of 0.28% in $\hat{R}$. 

66
• for pmd, ThisSens and ObjSens have four more object call graph edges misses than ConIns and CallString, which corresponds to a drop of 0.035% in $\hat{R}$.

In conclusion, our theory from Section 4.3 is confirmed in that applying an improvement to a conservative analysis may reduce the recall of general analysis. Thus, evaluating a (new) improvement to dataflow analysis must not be done for benchmark programs where the analysis is no longer conservative. However, in practice, the probable gain in precision should outweigh the loss of recall, but remember that we may have, theoretically, just removed true positives from the static analysis that are false negatives in our dynamic analysis.

### Evaluating the Filter Operations

We now take a closer look at the benefits of adding path-sensitivity based on filter-operations to the analysis.

We have seen that the filter operations, presented in Section 4.3, have little impact on the average analysis precision. However, when looking at the benchmark programs separately, the picture changes a little.

First, for the benchmark programs javac, javacc, sablecc, antlr, javadoc, jfreechart, and lucene, many of the metrics do not show any improvements, and the others reduce the sizes of the result sets by less than one percent (many by less than .1%).

Better improvements we get for bloat, recoder, emma, and pmd. In some cases, additional unreachable methods are found through the filter operations: 4 for recoder, and 1 respectively 3 for bloat, depending on the context sensitivity. The metrics that benefit most, though, are heap and object call graph edges; this is not surprising, as these are the most sensitive metrics to changes in the points-to sets. Depending on context sensitivity and benchmark program, object call graph edges decreases between 1.2% and 7.2%. Interesting is that the different context-sensitivities benefit very differently: For bloat, ObjSens has only 1.3% less such edges compared to its unfiltered counterpart, while for the same benchmark program, this is 7.2% for ThisSens. ConIns and CallString benefit by 3.4% and 3.7%, respectively. The picture is different for emma, where ObjSens benefits most (7.1%) compared to 3.6% for ConIns, 5.9% for CallString, and 6.4% for ThisSens.

For the metric heap, recoder benefits the most (-10.2%) for ConIns, while the context-sensitive variants benefit much less (2.6% - 3.0%).

The only of these benchmark program where the improvements are quite stable is pmd, where the filtered versions are between 3.3% and 4.1% (heap) and 1.2% - 1.9% (object call graph edges) better than their counterpart.

---

*A tabular overview of the results can be found in Tables C.7 to C.10.*
Chapter 6. Evaluation

```python
1  if (im_func instanceof PyFunction) {
2     return new PyMethod(
3         container, (PyFunction) im_func, im_class);
4  } else if (im_func instanceof PyReflectedFunction) {
5     return new PyMethod(
6         container, (PyReflectedFunction) im_func, im_class);
7  } else {
8     return new PyMethod(
9         container, im_func, im_class);
10  }
```

Figure 6.7: Code that yields rather good improvements when using filter operations. Taken from `org.python.core.PyMethod`.

The benchmark program that benefits most from the filter operations is `jython`. The metrics `heap`, `call graph edges`, and `object call graph nodes` benefit between 3.1% and 7.9%. This is quite a lot already; however, the object call graph edges metric shows an improvement between 12.5% - 18.0%, depending on the context sensitivity. Note that this has hardly any influence on the average approximated precision $\tilde{P}$, as the $\tilde{P}$ for this metric only increases from 0.14% to 0.16%. However, it is much more than context sensitivity does for `jython`. This is quite a bit, so we looked into the reason for this. Figure 6.7 shows the code that we identified as responsible for a large amount of the improvements.

The first two `instanceof` operations in lines 1 and 3 on the local variable `im_func` do not affect the `new`-expressions in lines 2 and 4, respectively, with respect to the points-to sets of `im_func`, as the `instanceof` operation is followed by a corresponding cast. However, both `instanceof` operations act as filters for the `new`-expression in line 6, which greatly reduces the input set of the second argument. More such constructs where specialized methods are called based on the type of arguments to a given method can be found in the source code of `jython`.

We conclude that the filter operations that we have implemented so far do not show a major boost in analysis precision on average; however, depending on the analyzed program, they can have quite an impact (up to -18% object call graph edges for `jython`), and they do not cost much performance in our experiments, so that including them in the analysis is worth it.

Combining Analyses

We can safely combine our analyses `Spark`, `ConIns`, `ThisSens`, `ObjSens`, and `CallString` for the conservative cases by intersecting the analysis results. The combined analysis must still be conservative and improves the two original analyses.
6.2. Comparing, Combining, and Improving Analyses

We check the combinations of analyses with the following two setups: (1) We combine Spark and ConIns to check if two totally different implementations have synergy effects, and (2) we combine ThisSens, ObjSens and CallString to check if different context sensitivities – remember that none of them is more precise than the other in theory – have synergy effects.

We also decided to also combine the context-sensitive variants for the cases where the analyses are not conservative; as we have discussed, improvements show only little effect on the approximated recall of a general analysis. Again, these results must nevertheless be regarded with care.

**Spark and ConIns** The flow-sensitive ConIns should be more precise than the flow-insensitive Spark. Exceptions from this could occur if the handling of native methods that ConIns supports, but Spark does not, affects the precision. We did not observe this case, thus, for our benchmark programs, ConIns is strictly more precise than Spark, so that the results from the combined analyses equal the results obtained from ConIns.

**ThisSens, ObjSens, CallString** When combining the context-sensitive Points-to SSA based Simulated Execution variants, we get a different picture\(^7\).

For the benchmark programs bloat, javacc, antlr, emma, and javadoc the results of ObjSens are equal to the results of the combined analysis. For jython and jfreechart, the results of the combined analyses equal the results of ThisSens; remember that ObjSens did not run through in these cases, so that we only combined ThisSens and CallString.

The project lucene gains a very small improvement – four object-call graph edges – while pmd gets a very little bit better in all metrics except heap. We did not observe any additional misses in the combined analyses.

When combining the results of javac, we found that the combined analysis becomes better in the client heap, object call graph nodes, and object call graph edges. This is rather surprising, since ObjSens is already more precise than the other context-sensitive variants in all three metrics (13% less heap objects, marginally less call graph nodes, 25.4% less object call graph edges than the respective better of the other two analyses). Still, the combined analysis has 4.3% less heap objects, marginally less object call graph nodes, and 1.9% less object call graph edges.

For the benchmark program sablecc, we have very different results depending on the metric. For heap, the combined analysis equals the two functional approaches, while CallString is much worse. For reachable methods and object call graph nodes, the results equal CallString, which is better than the functional approaches here. Then, for call graph edges, CallString is better.

---

\(^7\)A tabular overview of the results presented here can be found in Table C.12.
than the two functional approaches, but the combined analysis is even better (-0.55%). For the client object call graph edges, ObjSens is the best of the three context-sensitive analyses, but, again, the combined result is even better (-1.3%).

In project recoder, the number of reachable methods is the same for all context-sensitive approaches, but the combined analysis has one less. Additionally, CallString is best in the call graph metric, while ObjSens is best in the heap metric. For the other two clients, ObjSens is again best, but the combination is just a tiny bit better.

As a concrete example why there is no strict ordering in the precision of the context-sensitive variants, we explain the differences in the set of reachable methods for project recoder. Here, the two functional approaches identify recoder.bytecode.ClassFile.getTypeName() as reachable, while CallString does not. The other way around, it is the method recoder.service.UnresolvedBytecodeReferenceException.toString().

The functional approaches are better

The class java.lang.Throwable, which is a superclass to any exception (and thus recoder.service.UnresolvedBytecodeReferenceException), has a field named cause, which is initially initialized with itself. This cause can be obtained by the method getCause(). For the CallString approach, all causes are then mixed through this method, so that getCause() will return any abstract object of (sub-)type Throwable. getCause() is invoked in context with toString(), so that this method becomes reachable in the eyes of a CallString-based approach. On the other hand, this mixing does not occur for the functional approaches, so that those approaches are better in this case.

CallString is better

The only call site for the method recoder.bytecode.ClassFile.getTypeName() is in the method recoder.service.DefaultByteCodeInfo.getType(...). The relevant part of the code of this method is listed in Figure 6.8.

Type is a supertype of ClassFile, which contains the method in question, getTypeName(). Thus, the instanceof operation in line 3 guarantees that, in line 6, the points-to set of pme does not contain any abstract object of type ClassFile. The called method, getByteCodeElement(), however, simply returns the passed argument if the argument itself is of type ByteCodeElement, or null otherwise (and thus works like the as operator in C#). When analyzing the program with the CallString approach, then it is even guaranteed that the points-to set of variable bci does not contain any abstract object of type ClassFile, so that the method in question is identified as not reachable. The functional approaches, on the other hand, suffer from the fact that getByteCodeElement() is called from other places as well, so that abstract objects of type ClassFile get mixed into the result set of this method.

In conclusion, we do see a little advantage of combining the results of the
6.3 Evaluating the Points-to SSA Front-end

```java
public Type getType(ProgramModelElement pme) {
    Type result = null;
    if (pme instanceof Type) {
        result = (Type) pme;
    } else {
        ByteCodeElement bci = getByteCodeElement(pme);
        if (bci == null) {
            result = pme.getProgramModelInfo().getType(pme);
        } else {
            result = getType(bci);
        }
    }
    return result;
}
```

Figure 6.8: Code responsible for CallString being more precise than the functional approaches. Excerpt from recoder.service.DefaultByteCodeInfo.

different context-sensitive approaches that we evaluated in this chapter if and only if time is a subordinated goal. For our work towards a Gold Standard, this surely is the case: Up to 4.3% less heap objects for project javac justify the additionally required analysis time.

6.3 Evaluating the Points-to SSA Front-end

When implementing the new, FIRM based Points-to SSA front-end, we first provided the node types that were present in the Shimple based front-end which we had at hand. We then made sure that the analyses results obtained with the SSA graphs constructed with the FIRM based front-end yielded the same results as when using the Shimple based front-end. This way, we gained confidence in our implementation\(^8\).

We now exemplify the flexibility of the new front-end. For this, we look at the amount of code that was required to add the filter-operation `instanceof`.

First, the algorithm which maps FIRM to our Points-to SSA like representation needs to be informed to retain `instanceof` nodes. This was done by adding only three lines of code to the mapping algorithm. Then, this node type needed to be added to this Points-to SSA like representation, which required writing a new class which only contains few, short methods (e.g., methods reporting the number and type of the ports). Third, the transformation algorithm that converts boolean-operation nodes to filtering nodes was extended so it knows how to handle this new node type; this required eight lines of code. Finally, the mapping from the Points-to SSA like representation

\(^8\)We actually found a few bugs in the Shimple based front-end as well as a bug in Shimple itself: https://svn.sable.mcgill.ca/bugzilla/show_bug.cgi?id=82.
to Points-to SSA itself needed to be done. This required introducing a new node type in Points-to SSA (two lines of code), as well adapting the mapping algorithm (three lines of code). Overall, we have added only some dozens of lines of code to add a new node type to Points-to SSA.

Adapting the underlying analysis algorithm required then to add a number of visitor-methods to the Simulated Execution implementation. Most of these methods could be left as stubs (i.e., empty methods), only the transfer function contains ten lines of code.

In summary, adding additional node types to the FIRM based front-end does not require much code to be added. Thus, the front-end seems to fulfill our requirement of flexibility.

### 6.4 Summary

In this chapter, we have compared the results obtained from dynamic analysis with two points-to analysis implementations: Spark, and our Points-to SSA based Simulated Execution approach, which comes in different context-insensitive (ConIns) and context-sensitive (ThisSens, ObjSens, CallString) variants. We compared the results based on an under-approximation of a Gold Standard, which be obtained from dynamic analysis. Since we are having only general analyses in many cases, this implies that the results need to be interpreted with much care; we argue, however, that, in practice, the results are still comparable.

At no surprise, the flow-sensitive ConIns seems to be more precise than Spark, which is flow-insensitive. The context-sensitive approaches seem, again, more precise than ConIns, but there is no strict ordering in their approximated precision. On the other hand, Spark has a higher $R$ on average. However, it does not outweigh the lower $\hat{P}$ when it comes to comparing the approximated $F$-scores, $\hat{F}$.

Results obtained when applying an improvement for a conservative analysis to a non-conservative baseline analysis must be taken with care. Theoretically, just true positives may be “optimized” away, thus doing more harm than showing benefits. We investigated this by comparing the approximated recalls $\hat{R}$ of different analysis variants. We found that, in some cases, $\hat{R}$ is indeed reduced when going from context-insensitive to context-sensitive or from path-insensitive to path-sensitive analysis. However, these drops in $\hat{R}$ are very small, so it is likely that applying such improvements is useful even when the baseline analysis is not conservative.

When combining Spark and our context-insensitive baseline approach, we found that ConIns is, at least in the given benchmark programs, strictly more precise than Spark. This is to be expected, as our approach is flow-sensitive, in contrast to Spark, which is not.
6.4. Summary

When combining the different context-sensitive approaches, we found that sometimes the combined analysis is better than the best of the original analyses. Quite often, this is shown only in marginal improvements in our more fine-grained clients. However, the combination yields 25.4% less object call graph nodes for the benchmark program javac. As an example for how such synergy effects occur, we looked at project recoder, where the functional approaches identify one method as unreachable that the CallString approach does not, and vice versa. We explained the reasons for this case by looking at the source code of recoder.

Finally, we investigated the flexibility of the new Points-to SSA front-end by looking at how much effort was needed to implement the filter-operation instanceof. Only a handful of files needed to be extended, and most of them only by a few lines of straight-forward code.
Chapter 7

Conclusion

In this chapter, we summarize the findings from this thesis. Then, we discuss these with respect to the goals and goal criteria presented in Chapter 1.

In Chapter 2, we have presented concepts of program analysis with a focus on points-to analysis. Then, in Chapter 3, we have presented two concrete points-to analysis implementations, Points-to SSA based Simulated Execution and Spark, which comes in different variants, as well as Spark.

Theoretical considerations regarding how the results of dataflow-analyses can be compared and combined have been presented in Chapter 4. We found that analyses can be compared with respect to accuracy only in special cases, namely when we compare either two conservative, or two optimistic analyses. For all other kinds of comparison, a Gold Standard is required, which currently does not exist for points-to analysis.

Combining two analyses is possible when both are either conservative or optimistic. The intersection of two conservative analyses is at least as accurate as the better of the two, and the same holds for the union of two optimistic analyses. Additionally, a general analysis can always improved by combining it by either adding the results from an optimistic analysis, or intersecting it with the results from a conservative analysis. Any other combination of analyses yields, in the absence of a Gold Standard, to an unspecified analysis with respect to accuracy.

We have also presented a new approach to path-sensitivity for points-to analysis by the means of filter operations. With the help of these filter operations, control flow can be converted to dataflow in certain cases, which then allows it to be seamlessly integrated into existing points-to analysis implementations. However, we also show that we must be careful when evaluating such improvements: if the baseline analysis is not conservative, then we cannot tell if the improvement does more harm than it has use.

In Chapter 5, we presented tools and specifications that allow us to compare and combine different points-to analysis implementations. We argued that the raw points-to information can most likely not be compared over implementation borders, and thus decided to use client analyses that should be derivable from almost every points-to analyses. Some of these client analyses are still very low-level, so that even small differences in the results of different
points-to analyses can be discovered. Then, we presented an XML exchange format that captures results from these client analyses in an implementation-independent format.

Practical problems that we stumbled across when comparing the results from our dynamic and static analyses as well as our solutions and workarounds to them, are also discuss at this point.

Then, we present a new front-end for the generation of Points-to SSA. For this front-end, we describe how the generation of the filter operations from Section 4.3 can be implemented. Further, the front-end is based on FIRM, a full program representation based on Memory SSA. Thus, the front-end is flexible for future extensions of Points-to SSA.

In Chapter 6, we have compared the results obtained from dynamic analysis with two points-to analysis implementations: Spark, and our Points-to SSA based Simulated Execution approach. The latter comes in different context-insensitive (ConIns) and context-sensitive (ThisSens, ObjSens, Call-String) variants. We compared the results based on an under-approximation of a Gold Standard, which was obtained from dynamic analysis. As the used points-to analyses are general analyses in many cases, the results need to be interpreted with care; we argue, though, that the results are still comparable from a practical point of view. The flow-sensitive ConIns appear more precise than the flow-insensitive Spark. The context-sensitive approaches appear, again, more precise than ConIns, but there is no strict ordering in their approximated precision. On the other hand, Spark has a higher approximated recall on average. However, this does not outweigh the lower approximated precision when it comes to comparing F-scores.

Results obtained when applying an improvement for a conservative analysis to a non-conservative baseline analysis must be taken with care. Theoretically, just true positives may be “optimized”. Our experiments show that this happens in practice, but only in some cases and with very low impact on the approximated recall. While this confirms our theoretical considerations, we believe that applying such improvements are justified even for general analyses.

When combining Spark and our context-insensitive baseline approach, we found that ConIns is, at least in the given benchmark programs, strictly more precise than Spark. When combining the context-sensitive variants of Points-to SSA based Simulated Execution, this is also the case for many of the analyzed benchmark program. However, in some cases the improvements are considerably high: for javac, we get 25.4% less object call graph nodes. Other benchmark programs show slight improvements in our client analyses as well.

Finally, we evaluated the new Points-to SSA front-end with respect to its flexibility. We did this by looking at how much code needed to be added for introducing the filter-operation instanceof. This required only small additions.
in a few number of files, so that we argue that the front-end is indeed flexible for further extensions.

### 7.1 Review of the Goals and Goal Criteria

We now review the results of this thesis with respect to the goals we have set in Chapter 1, which are:

1. Develop a framework that allows to create (and incrementally improve) very precise points-to analyses.
2. Develop a framework that allows to compare different points-to analysis instantiations.

**Goal 1**

The goal criteria for our first goal are as following:

1. Points-to SSA is to be extended with additional node types that can improve analysis precision. Means to further enhance Points-to SSA in the future shall also be provided. The underlying analysis must be adaptable without too much effort.

2. Analysis speed is a subordinated goal criteria. The analysis should run in adequate time, i.e., not days, but does not need to be usable in an edit-compile-cycle.

Criterion 1 is fulfilled: The new Points-to SSA front-end, described in Section 5.4, is capable of creating Points-to SSA graphs enriched with filter operations, which were described in Section 4.3. We have discussed the flexibility of the front-end as well as the benefits of these filter operations in Chapter 6.

Criterion 2 is fulfilled: The original Points-to SSA based Simulated Execution is not considerably slowed down by adding these new nodes.

**Goal 2**

The goal criteria for our second goal are as following:

1. The theoretical implications when comparing and combining two analyses have to be evaluated.

2. Comparison of metrics commonly used for evaluating points-to analysis, regardless of the analysis technique, shall be made possible. We see this criterion fulfilled if we can compare at least two different implementations - our own and a widely accepted third-party implementation - as well as results obtained from dynamic analysis.
3. Tools are to be provided in order to *automatically* compare the results of these analyses. Additionally, the results obtained from different points-to analysis implementations shall be combined automatically.

Criterion 1 is fulfilled: In Chapter 4, we have discussed how comparing and combining analyses is possible for different kinds of analyses. In Chapter 6, we have even experimentally shown that special cases discussed occur in practice.

Criterion 2 is fulfilled: We have defined a number of client analyses that should be computable by almost every points-to analysis. We have also defined an XML exchange format that can capture these client analyses. We have connected both our own, Points-to SSA based Simulated Execution points-to analysis, as well as *Spark*, a widely used points-to analysis implementation, to this exchange format. Additionally, we have a dynamic agent that captures information obtained by monitoring program runs into this format.

Criterion 3 is fulfilled: We have tools that read in files in our XML exchange format, compare and combine the results, and present the results to the user.

In conclusion, we see the goals that we set for this thesis as fulfilled.
Chapter 8

Future Work

In this chapter, we show possible next steps towards the ultimate goal, creating a Gold Standard for points-to analysis.

8.1 Improving Points-to Analysis Precision

We have presented five filter operations in Section 4.3, of which we have so far implemented two for our Points-to SSA based Simulated Execution implementation. Implementing and evaluating the remaining three seems a natural next step. Additionally, more filter operations could be thought of.

Another improvement that we can think of is to perform dead-code analysis. This requires that the basic block structure and control flow operations are available in the points-to analysis. Then, control flow is treated as dataflow: If it can be decided that a branch statement is, in a given context, either always true or always false, then the statements contained in basic blocks of the other branch do not need to be computed. Such a decision is possible, e.g., when the branch-operation depends on object equality or an instanceof-operation.

We have pointed out a drawback of our current approach to introduce the filter operations: if the condition is, e.g., $(x \text{ instanceof } A \lor x \text{ instanceof } B)$, then neither of the two instanceof-operations dominates any block in the branches; thus, it may be a good idea to integrate the filter-construction into the SSA construction algorithm. Then, the true-branch, for instance, will contain a $\varphi$-node of the two instanceof-operations, which will lead to an improved analysis precision in cases where such constructs occur.

In Chapter 6, we have seen that there is no exact order in analysis precision when it comes to comparing different points-to analyses. Currently, we can combine the results obtained by different (conservative) analyses on the client-analysis level. However, it is also thinkable to take the results obtained by one points-to analysis into consideration while running another points-to analysis: For starters, we can limit the set of reachable methods prior to analysis; this makes sense because we have seen that the functional approaches to context sensitivity and the call string approach are not strictly ordered when it comes to the precision of reachable methods (e.g., for project recoder). In
a next step, one could even limit the pairs [abstract object, method] and the heap-objects in the analyses.

So far, we have connected our own, Points-to SSA based analysis as well as the well-known Spark points-to analysis to our XML exchange format. More implementations should follow; a good candidate is Paddle [20].

8.2 Extensions of the Comparison Platform

We have presented an XML based exchange format in which client analyses derived from different points-to analysis implementations and approaches (static or dynamic analyses) can be stored. We also have a tool that computes the differences, combinations and approximated precision and recall for these. A next step towards a joint effort on creating a Gold Standard for some points-to analysis clients is to create a web-based platform that allows collecting these results. Then, (1) researchers can submit their own contributions, e.g., their results obtained by their points-to analyses, results obtained by dynamic analysis, or even manual proofs that a given item is not in the result set of a given client analysis. Further, (2) in case that conservative analysis are available for a given benchmark program, the best available over- and under- approximations to the Gold Standard can be computed from the collected data. These can then be made publicly available to interested people.

A precondition for people being able to contribute to the dynamic analysis is that the dynamic analysis is high-performance. Ideally, the agent monitoring the program execution slows down the program execution marginally, so that the user does not even feel this. Considering the amount of operations that need to be monitored and evaluated (every method call, object allocation and field-write), this is of course a very hard task to achieve, if at all possible. However, our current implementation uses an ad-hock approach when it comes to the data-structures and implementation, and thus leaves plenty of room for enhancement. Additionally, the problems described in Section 5.3 should be addressed as well.

8.3 Tools for Supporting Computing the Gold Standard

Currently, we have the means to over- and under-approximate the Gold Standard for points-to analysis clients, at least when we have a conservative (respectively optimistic) analysis at hand. As we have seen in Chapter 6, the difference between these two may be quite large, depending on analyzed project and client analyses. Currently, for the elements in the difference sets,
a researcher will have to either create a proper input to trigger its occurrence in the optimistic analysis, or he has to proof that the element is infeasible. We are not aware of any tools that help any of these two steps, so developing those is another task for future work. In the following, we discuss two ways tools could help a researcher, visualization and aiding in proving.

Visualization

The client analyses that we used in this thesis do interact, i.e., improving the precision of one metric improves result in other metrics to improve as well. For example, proving that a certain heap relation is not present in the result set, can trigger (object) call graph edges to not be present either. This happens, for example, when a call is executed on a field. By removing an abstract object from the points-to set of the field, object call graph edges become removed as well.

We assume that manually proving the absence of a certain heap-relation or (object) call graph edge is quite time-consuming; thus, the effort should be worth it. A visualization tool that shows dependencies between different program relations could guide the researcher at which point it would be useful to perform these proofs.

Lhoták has done work into this direction for call graphs [21]. He found that quite often eliminating a single spurious call graph edge can remove a rather big subgraph from a call graph, and he also presents an algorithm that suggests call graph edges to be looked at more closely. Lhoták suggests such tools and algorithms should be developed for points-to information as well, which we agree with.

Aiding in proving

Once a possible single point of reason for a large part of imprecision has been found, it needs to be investigated if this actually is a source of imprecision, or if it is a false negative in the dynamic input set. Thus, either a concrete input for the dynamic analysis needs to be created that causes this object to appear in the result set, or it needs to be proven otherwise. Tools should thus aim at supporting two things: Help proving the absence of certain relations, and help creating input cases, so that the result sets of dynamic analysis get bigger.

For the former, the applicability of theorem provers for our needs should be evaluated. A good starting point for the latter is dynamic test case generation (also referred to as concolic testing): Here, the idea is to execute a program on a given input, and then vary the input so that different control flow paths are taken. This new input is obtained by solving symbolic constraints on the conditions of branch statements. Concolic testing is applied in the area of bug-finding, but its idea should be adaptable to our needs as well.
8.3. Tools for Supporting Computing the Gold Standard

Dynamic test case generation tools exist for many programming languages, e.g., C [46], PHP [5], and Java [45].
Appendix A

Points-to SSA Node Types

Here, we list the transfer functions, accompanied by a short explanation, of all node types that are currently part of Points-to SSA.

A.1 Memory-related operations

All memory-related operations define a new memory-value, even if they only read from the memory. This is required to model anti-dependencies in Memory SSA and thus also a feature of Points-to SSA.

Algorithm 9 shows the transfer function for Store nodes. Each Store node has an attribute $f$ which defines the accessed field, and takes the current memory value $x_{in}$, an address $a$, and a value $v$ that is to be stored into $f$. Then, into each memory slot $[o, f]$, where $o \in Pt(a)$, the value $v$ is merged, i.e., a weak update is performed. Note that $x_{out} \neq x_{in}$ if and only if the memory has been changed.

\begin{algorithm}
\caption{Store$^f$ : $[x_{in}, a, v] \mapsto x_{out}$}
\begin{algorithmic}
\State $x_{out} = x_{in}$
\For {each $o \in Pt(a)$}
\State $prev = \text{Mem.get}([o, f])$
\If {$v \not\subseteq prev$}
\State $\text{Mem.addTo}([o, f], v)$
\State $x_{out} = \text{Mem.getSize}()$
\EndIf
\EndFor
\State \text{return $x_{out}$}
\end{algorithmic}
\end{algorithm}

Algorithm 10 shows the transfer function for Load nodes. Like for Store nodes, each such node has an attribute $f$ that defines the accessed field. Load nodes take the current memory value $x_{in}$ and an address $a$ as input values. The transfer function then computes the join of all memory slots $[o, f]$, where $o \in Pt(a)$. The memory-value is simply passed through.

Algorithm 11 shows the transfer function for object creation nodes. The function passes through the memory-value, and initializes $v$ with the abstract
A.1. Memory-related operations

Algorithm 10 \( \text{Load}^T : [x_{in}, a] \mapsto [x_{out}, v] \)

\[
x_{out} = x_{in} \\
v = \bot \\
\text{for each } o \in Pt(a) \text{ do} \\
\quad v = v \sqcup \text{Mem.get}([o, f]) \\
\text{end for} \\
\text{return } [x_{out}, v]
\]

object \( o_T^C \), where \( C \) is the syntactical position of the allocation operation and \( T \) the type of the object. This corresponds to the creation point naming scheme. This transfer function can be adapted so that it supports more fine-grained naming schemes, cf. Section 2.3.

Algorithm 11 \( \text{Alloc}^T_C : x_{in} \mapsto [x_{out}, v] \)

\[
x_{out} = x_{in} \\
v = \{o_T^C\} \\
\text{return } [x_{out}, v]
\]

Array handling

Algorithm 12 shows the transfer function for the allocation of arrays, which is similar to the transfer function for \( \text{Alloc} \) nodes. An array is attributed with the type \( T \) that is a container for.

Algorithm 12 \( \text{ArrayAlloc}^T_C : x_{in} \mapsto [x_{out}, v] \)

\[
x_{out} = x_{in} \\
v = \{o_T^C\} \\
\text{return } [x_{out}, v]
\]

Algorithms 13 and 14 show the transfer function for the array-accessing operations \( \text{ArrayLoad} \) and \( \text{ArrayStore} \), respectively. They correspond to \( \text{Load} \) and \( \text{Store} \) operations with the difference that they are not attributed with a field. Instead, each array object itself is a memory slot on the abstract heap.

ArrayCopy, Algorithm 15, is used by \texttt{System.arraycopy()} and when an array is stored into an array of higher dimension. The latter is for the aforementioned flattening to one-dimensional arrays. The algorithm takes three arguments: The obligatory current memory value \( x_{in} \), the source of the array-copy \( b_{src} \) as well as the target of the operation \( b_{tgt} \). The algorithm copies the points-to sets contained in each source-array to all those target-arrays that are matching arrays. Thus, copies from an array of type \( T_1 \) are not performed to arrays of type \( T_2 \) if \( T_2 \) is not widening \( T_1 \).
Appendix A. Points-to SSA Node Types

Algorithm 13 ArrayLoad: \([x_{\text{in}}, b] \mapsto [x_{\text{out}}, v]\)

\[
x_{\text{out}} = x_{\text{in}}
\]
\[
v = \perp
\]
for each \(o \in Pt(b)\) do
\[
v = v \sqcup \text{Mem.get}(o)
\]
end for
return \([x_{\text{out}}, v]\)

Algorithm 14 ArrayStore: \([x_{\text{in}}, b, v] \mapsto x_{\text{out}}\)

\[
x_{\text{out}} = x_{\text{in}}
\]
for each \(o \in Pt(b)\) do
\[
\text{prev} = \text{Mem.get}(o)
\]
if \(v \not\sqsubseteq \text{prev}\) then
\[
\text{Mem.addTo}(o, v)
\]
\[
x_{\text{out}} = \text{Mem.getSize}()
\]
end if
end for
return \([x_{\text{out}}, v]\)

A.2 Exception handling

Exception handling in Points-to SSA is performed as follows: Each context is associated with a points-to set of exceptions that may occur within it.

Algorithm 16 shows the transfer function for \(\text{Throw}\) nodes. Additional to the current memory value, the transfer function takes an exception address-value \(e\), which is added to the current context’s set of exceptions.

Algorithm 17 shows the transfer function for \(\text{Catch}\) nodes. A \(\text{Catch}\) node filters the set of exceptions of the current context by the type \(T\), which is an attribute to each such node.

A.3 Other intra-procedural node types

\(\varphi\) nodes (Algorithm 18) are nodes which are artificially inserted in SSA representations at control flow confluences. In Points-to SSA based Simulated Execution, they compute the join of the input values.

Algorithm 19 shows the transfer function for \(\text{Cast}\) nodes. It filters all the abstract objects from \(v_{\text{out}}\) that are not an instance of \(T\), and returns them.

Algorithm 20 shows the computation for \(\text{EQ}\) nodes, according to the filter operations described in Section 4.3. Three values are computed: The first one is the set-intersection of the two input values, which is used in the then-block of a corresponding branch. The other two are for filtered values of \(v_{\text{in1}}\) and \(v_{\text{in2}}\) in the else block. These filter out only \textit{null} values in the case of \textit{null}
Algorithm 15 ArrayCopy: \([x_{in}, b_{src}, b_{tgt}] \mapsto x_{out}\)

\[x_{out} = x_{in}\]

for each \(o_{tgt} \in Pt(b_{tgt})\) do
  for each \(o_{src} \in Pt(b_{src})\) do
    if \(o_{src}\) instanceof \(o_{tgt}\) then
      \(prev = Mem.get(o_{tgt})\)
      \(v = Mem.get(o_{src})\)
      if \(v \not
        \subseteq prev\) then
        \(Mem.addT o(o_{tgt}, v)\)
        \(x_{out} = Mem.getSize()\)
    end if
  end for
end for
return \(x_{out}\)

Algorithm 16 Throw: \([x_{in}, e] \mapsto [x_{out}]\)

\[x_{in} = x_{out}\]

\(current\_context.addExceptions(e)\)

return \(x_{out}\)

checks.

Algorithm 21 shows the transfer function for InstanceOf nodes, according to the filter operations described in Section 4.3. It is very similar to the transfer function of Cast nodes, but does not pass null-values through.

The node-types Entry and Exit are required to have unique entry- and exit-points in a Points-to SSA graph; no computation are actually performed, they simply pass through the input-values to a method invocation and return the result of the current method invocation, respectively. Their transfer functions are shown in Algorithms 22 and 23.

A.4 Inter-procedural node types

Algorithm 24 shows the processing of monomorphic calls. A number of contexts are selected for the called methods, and the method is analyzed for each of these contexts separately. The results of the analyses under different contexts are then merged. The actual processing of a single context is described in Section 3.2.

Algorithm 25 shows the processing of static calls, which is similar to that of monomorphic calls, with the difference that no address value (this) is present. Note that monomorphic calls can still be treated context-sensitively, namely for the CallString approach.

85
Appendix A. Points-to SSA Node Types

**Algorithm 17** \( \text{Catch}^T: x_{in} \mapsto [x_{out}, e] \)

\[
x_{out} = x_{in} \\
e = \{ v \in \text{current\_context.getExceptions()} \mid v \text{ instanceof T} \} \\
\text{return } [x_{out}, e]
\]

**Algorithm 18** \( \varphi: [v_1, \ldots, v_n] \mapsto v_{out} \)

\[
v_{out} = \bigsqcup_{i=1}^{n} v_i \\
\text{return } v_{out}
\]

Algorithm 26 describes the handling of polymorphic calls. The address value (\( \text{this} \)) is filtered with respect to each method in the set of possible target methods. Then, each such method is d as a monomorphic call.
A.4. Inter-procedural node types

Algorithm 19 \( \text{Cast}^T : v_{\text{in}} \mapsto v_{\text{out}} \)

\( v_T = \{ o \in \text{Pt}(v_{\text{in}}) | o = \text{null} \lor o \text{ instanceof } T \} \)

\text{return } v_{\text{out}}

Algorithm 20 \( \text{EQ}: [v_{\text{in}1}, v_{\text{in}2}] \mapsto [v_{\text{out}1}, v_{\text{out}2}, v_{\text{out}3}] \)

\( v_{\text{out}1} = v_{\text{in}1} \lor v_{\text{in}2} \)
\( v_{\text{out}2} = v_{\text{in}1} \)
\( v_{\text{out}3} = v_{\text{in}2} \)

\text{if } v_{\text{in}1} == \{ \text{null} \} \text{ then}
\( v_{\text{out}3} = v_{\text{out}3} \setminus \{ \text{null} \} \)
\text{end if}

\text{if } v_{\text{in}2} == \{ \text{null} \} \text{ then}
\( v_{\text{out}2} = v_{\text{out}2} \setminus \{ \text{null} \} \)
\text{end if}

\text{return } [v_{\text{out}1}, v_{\text{out}2}, v_{\text{out}3}]

Algorithm 21 \( \text{InstanceOf}^T : v \mapsto [v_T, v_{\neg T}] \)

\( v_T = \{ o \in \text{Pt}(v) | o \text{ instanceof } T \} \)
\( v_{\neg T} = \{ o \in \text{Pt}(v) | \neg (o \text{ instanceof } T) \} \)

\text{return } [v_T, v_{\neg T}]

Algorithm 22 \( \text{Entry}: [x_{\text{in}}, a_{\text{in}}, v_{\text{in}1} \ldots v_{\text{in}n}] \mapsto [x_{\text{out}}, a_{\text{out}}, v_{\text{out}1} \ldots v_{\text{out}n}] \)

\( x_{\text{out}} = x_{\text{in}} \)
\( a_{\text{out}} = a_{\text{in}} \)
\( v_{\text{out}i} = v_{\text{in}i}, \ i = 1..n \)

\text{return } [x_{\text{out}}, a_{\text{out}}, v_{\text{out}1} \ldots v_{\text{out}n}]

Algorithm 23 \( \text{Exit}: [x_{\text{in}}, v_{\text{in}}] \mapsto [x_{\text{out}}, v_{\text{out}}] \)

\( x_{\text{out}} = x_{\text{in}} \)
\( v_{\text{out}} = v_{\text{in}} \)

\text{return } [x_{\text{out}}, v_{\text{out}}]

Algorithm 24 \( \text{MCall}^{m,cs_i} : [x_{\text{in}}, a, v_1, \ldots, v_n] \mapsto [x_{\text{out}}, r] \)

\text{Context[ ] \ ctxs = selectContextsFor}(m, cs_i, a)
\( [x_{\text{out}}, r] = [0, \bot] \)

\text{for each } ctx^m \in \text{ctxs do}
\text{\hspace{1em} \this = ctx^m\.getThis()}
\text{\hspace{1em} \args = [x_{\text{in}}, this, v_1, \ldots, v_n]}
\text{\hspace{1em} \[x_{\text{out}}, r\] = \text{processCall}(ctx^m, \args) \cup [x_{\text{out}}, r]}\n
\text{end for}

\text{return } [x_{\text{out}}, r]
Algorithm 25 $SCall^{m,cs_i} : [x_{in}, v_1, \ldots, v_n] \mapsto [x_{out}, r]$

$\quad Context[] \ctxs = selectContextsFor(m, cs_i)$
$\quad [x_{out}, r] = [0, \bot]$ \[Context[] \]
$\quad$ for each $ctx^m \in \ctxs$ do
$\quad\quad args = [x_{in}, \bot, v_1, \ldots, v_n]$ \[Context[] \]
$\quad\quad [x_{out}, r] = processCall(ctx^m, args) \cup [x_{out}, r]$ \[Context[] \]
$\quad$ end for
$\quad$ return $[x_{out}, r]$ \[Context[] \]

Algorithm 26 $PCall^{\{tgts\},cs_i} : [x_{in}, a, v_1, \ldots, v_n] \mapsto [x_{out}, r]$

$\quad [x_{out}, r] = [0, \bot]$ \[Context[] \]
$\quad$ for each $m \in tgts$ do
$\quad\quad a_m = filterByTarget(a, m)$ \[Context[] \]
$\quad\quad [x_{out}, r] = MCall^{m,cs_i}(x_{in}, a_m, v_1, \ldots, v_n) \cup [x_{out}, r]$ \[Context[] \]
$\quad$ end for
$\quad$ return $[x_{out}, r]$ \[Context[] \]
Appendix B

The Exchange Format Schema

The schema definition:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="p2a">
    <xs:complexType>
      <xs:sequence>
        <!-- The name and version of the analyzed program. -->
        <xs:element name="analyzedProject" type="xs:string"/>
        <!-- A textual description of the analysis setup. -->
        <xs:element name="analysisSetup" type="xs:string"/>
        <!-- Applied application name filter. Zero or more. -->
        <xs:element name="applicationNameFilter" type="xs:string" minOccurs="0"/>
        <!-- The kind of the analysis: optimistic, conservative,
             general, or manual proof. -->
        <xs:element name="analysisKind" type="analysisKind"/>
        <!-- Whether or not this file contains "heap" information. Defaults to "false". -->
        <xs:element name="hasHeap" type="xs:boolean" minOccurs="0"/>
        <!-- Whether or not this file contains "object/uni2423call/uni2423graph" information. Defaults to "false". -->
        <xs:element name="hasObjCG" type="xs:boolean" minOccurs="0"/>
        <!-- Whether or not this file contains return values for object call graph edges. May be true only if "hasObjCG" is "true" as well. Defaults to "false". -->
        <xs:element name="hasReturnValues" type="xs:boolean" minOccurs="0"/>
        <!-- Whether or not this file contains method argument values (other than for "this") for object call graph edges. May be true only if "hasObjCG" is "true" as well. Defaults to "false". -->
        <xs:element name="hasArgumentValue" type="xs:boolean" minOccurs="0"/>
        <!-- The heap objects. May be present only if "hasHeap" is "true". -->
        <xs:element name="heapObj" type="heapObj"
```
Appendix B. The Exchange Format Schema

```xml
<!— The object call graph edges objects. May be present only if "hasObjCG" is "true". -->
<xs:element name="callEdge" type="callEdge"
  minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:simpleType name="analysisKind">
  <xs:restriction base="xs:string">
    <xs:enumeration value="optimistic"/>
    <xs:enumeration value="conservative"/>
    <xs:enumeration value="general"/>
    <xs:enumeration value="absenceProof"/>
  </xs:restriction>
</xs:simpleType>

<!— the complex types used above -->
<xs:complexType name="abstractObject">
  <xs:attribute name="creatorClass" type="xs:string" use="required"/>
  <xs:attribute name="lineNo" type="xs:integer" use="required"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="value">
  <xs:choice>
    <xs:element name="abstrObj" type="abstractObject"/>
    <xs:element name="nullObj"/>
  </xs:choice>
</xs:complexType>  

<xs:complexType name="nonnullValue">
  <xs:sequence>
    <xs:element name="abstrObj" type="abstractObject"/>
  </xs:sequence>
</xs:complexType>  

<xs:complexType name="heapObj">
  <xs:sequence>
    <xs:element name="instanceObj" type="nonnullValue"/>
  </xs:sequence>
</xs:complexType>
```

---

90
<xs:element name="method" type="xs:string" use="required"/>
</xs:complexType>
</xs:schema>
Sample source code. A corresponding result file is listed on the next page.

```java
definitions

class C {
    public static void main(String ... args) {
        new C().foo();
    }
}

class D {
    C obj;
    void bar(C c) {
        obj = c;
    }
}
```
A result file corresponding to the sample project from the previous page.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<p2a>
<analyzedProject>Sample program</analyzedProject>
<analysisSetup>P2SSA</analysisSetup>
<applicationNameFilter>p</applicationNameFilter>
<analysisKind>conservative</analysisKind>
<hasHeap>true</hasHeap>
<hasObjCG>true</hasObjCG>

<heapObj fieldName="p.D. obj">
<instanceObj>
<abstrObj creatorClass="p.C" lineNo="9" type="p.D"/>
</instanceObj>
/value>
</heapObj>

<callEdge>
<tgt method="p.C.main()V"/>
</callEdge>

<src method="p.C.main()V" />
<tgt method="p.C. foo ()V">
<thisObj>
<abstrObj creatorClass="p.C" lineNo="5" type="p.C"/>
</thisObj>
</tgt>
</callEdge>

<src method="p.C. foo ()V">
<thisObj>
<abstrObj creatorClass="p.C" lineNo="5" type="p.C"/>
</thisObj>
</src>
<tgt method="p.D. bar(Lp.C;)V">
<thisObj>
<abstrObj creatorClass="p.C" lineNo="9" type="p.D"/>
</thisObj>
</tgt>
</callEdge>
</p2a>
```
Appendix C

Evaluation: The Complete Metrics Data

In this appendix, we list the metrics results from the client analyses on which the evaluation part of this thesis is based on.
<table>
<thead>
<tr>
<th>Program</th>
<th>CI</th>
<th>CI-F</th>
<th>TS</th>
<th>TS-F</th>
<th>OS</th>
<th>OS-F</th>
<th>CS</th>
<th>CS-F</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>264</td>
<td>268</td>
<td>640</td>
<td>578</td>
<td>16412</td>
<td>14570</td>
<td>1046</td>
<td>900</td>
<td>154</td>
</tr>
<tr>
<td>javac</td>
<td>38.1</td>
<td>40.4</td>
<td>62.9</td>
<td>71.4</td>
<td>1367</td>
<td>1386</td>
<td>148</td>
<td>181</td>
<td>54</td>
</tr>
<tr>
<td>javacc</td>
<td>19.8</td>
<td>18.2</td>
<td>22.2</td>
<td>23.4</td>
<td>101</td>
<td>90.4</td>
<td>92.9</td>
<td>96</td>
<td>136</td>
</tr>
<tr>
<td>sablecc</td>
<td>36.2</td>
<td>34</td>
<td>33.7</td>
<td>33.9</td>
<td>210</td>
<td>219</td>
<td>32.8</td>
<td>33.6</td>
<td>137</td>
</tr>
<tr>
<td>recoder</td>
<td>330</td>
<td>392</td>
<td>714</td>
<td>785</td>
<td>3111</td>
<td>2859</td>
<td>1549</td>
<td>1662</td>
<td>173</td>
</tr>
<tr>
<td>antlr</td>
<td>43.1</td>
<td>42.8</td>
<td>76.5</td>
<td>78.8</td>
<td>1026</td>
<td>1775</td>
<td>109</td>
<td>107</td>
<td>114</td>
</tr>
<tr>
<td>emma</td>
<td>668</td>
<td>1129</td>
<td>355</td>
<td>301</td>
<td>9083</td>
<td>5844</td>
<td>2450</td>
<td>3791</td>
<td>132</td>
</tr>
<tr>
<td>javadoc</td>
<td>22.2</td>
<td>21.9</td>
<td>23.6</td>
<td>23.9</td>
<td>704</td>
<td>707</td>
<td>56</td>
<td>52.6</td>
<td>41</td>
</tr>
<tr>
<td>jfreechart</td>
<td>1083</td>
<td>1139</td>
<td>1412</td>
<td>1415</td>
<td>3652</td>
<td>3240</td>
<td>640</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>2118</td>
<td>1837</td>
<td>3312</td>
<td>3226</td>
<td>5893</td>
<td>7469</td>
<td>236</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lucene</td>
<td>9.6</td>
<td>10.9</td>
<td>11.8</td>
<td>11.7</td>
<td>32.3</td>
<td>35.9</td>
<td>20.2</td>
<td>19.9</td>
<td>112</td>
</tr>
<tr>
<td>pmd</td>
<td>40</td>
<td>38.6</td>
<td>38.7</td>
<td>44.5</td>
<td>181</td>
<td>208</td>
<td>495</td>
<td>454</td>
<td>141</td>
</tr>
</tbody>
</table>

Table C.1: Execution times in seconds for context-insensitive (CI), this-sensitive (TS) etc. analysis, without and with filters (“F”).
### Table C.2: Complete results for the heap metric. Sizes (upper row) and misses (lower row) of the result sets for context insensitive (CI), this-sensitive (TS) etc. analysis, without and with filters ("F").

<table>
<thead>
<tr>
<th></th>
<th>dyn</th>
<th>CI</th>
<th>CI-F</th>
<th>TS</th>
<th>TS-F</th>
<th>OS</th>
<th>OS-F</th>
<th>CS</th>
<th>CS-F</th>
<th>Spark</th>
<th>din</th>
<th>C#</th>
<th>CS</th>
<th>OS</th>
<th>TSL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>211</td>
<td>311</td>
<td>312</td>
<td>312</td>
<td>310</td>
<td>310</td>
<td>310</td>
<td>310</td>
<td>310</td>
<td></td>
<td>24</td>
<td>327</td>
<td>216</td>
<td>538</td>
<td>2014</td>
</tr>
<tr>
<td>63</td>
<td>309</td>
<td>302</td>
<td>300</td>
<td>24</td>
<td>309</td>
<td>309</td>
<td>24</td>
<td>309</td>
<td>24</td>
<td></td>
<td>24</td>
<td>327</td>
<td>216</td>
<td>538</td>
<td>2014</td>
</tr>
<tr>
<td>151</td>
<td>309</td>
<td>302</td>
<td>300</td>
<td>24</td>
<td>309</td>
<td>309</td>
<td>24</td>
<td>309</td>
<td>24</td>
<td></td>
<td>24</td>
<td>327</td>
<td>216</td>
<td>538</td>
<td>2014</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td></td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td></td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
</tr>
<tr>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td></td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td></td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td></td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
</tr>
<tr>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td></td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td></td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td></td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
</tr>
<tr>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td></td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td></td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td></td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
<td>463</td>
</tr>
<tr>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td></td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Program</td>
<td>dyn</td>
<td>CI</td>
<td>CI-F</td>
<td>TS</td>
<td>TS-F</td>
<td>OS</td>
<td>OS-F</td>
<td>CS</td>
<td>CS-F</td>
<td>Spark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>----</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>1242</td>
<td>2157</td>
<td>2156</td>
<td>2146</td>
<td>2143</td>
<td>2146</td>
<td>2143</td>
<td>2146</td>
<td>2145</td>
<td>2253</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javac</td>
<td>991</td>
<td>1314</td>
<td>1314</td>
<td>1314</td>
<td>1314</td>
<td>1314</td>
<td>1314</td>
<td>1314</td>
<td>1314</td>
<td>1340</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javacc</td>
<td>679</td>
<td>946</td>
<td>946</td>
<td>945</td>
<td>945</td>
<td>945</td>
<td>945</td>
<td>945</td>
<td>945</td>
<td>952</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sablecc</td>
<td>1017</td>
<td>1526</td>
<td>1526</td>
<td>1474</td>
<td>1474</td>
<td>1474</td>
<td>1474</td>
<td>1450</td>
<td>1450</td>
<td>1561</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recoder</td>
<td>3219</td>
<td>4196</td>
<td>4192</td>
<td>4178</td>
<td>4174</td>
<td>4178</td>
<td>4174</td>
<td>4174</td>
<td>4358</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>antlr</td>
<td>1129</td>
<td>609</td>
<td>609</td>
<td>609</td>
<td>609</td>
<td>609</td>
<td>609</td>
<td>609</td>
<td>786</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>emma</td>
<td>513</td>
<td>883</td>
<td>883</td>
<td>874</td>
<td>874</td>
<td>873</td>
<td>874</td>
<td>873</td>
<td>873</td>
<td>960</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javadoc</td>
<td>1082</td>
<td>628</td>
<td>628</td>
<td>628</td>
<td>628</td>
<td>628</td>
<td>628</td>
<td>628</td>
<td>628</td>
<td>680</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jfreechart</td>
<td>713</td>
<td>1061</td>
<td>1060</td>
<td>1058</td>
<td>1057</td>
<td>1058</td>
<td>1057</td>
<td>1057</td>
<td>2243</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>3001</td>
<td>2159</td>
<td>2140</td>
<td>2126</td>
<td>2137</td>
<td>2137</td>
<td>2137</td>
<td>2137</td>
<td>3410</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lucene</td>
<td>491</td>
<td>689</td>
<td>688</td>
<td>688</td>
<td>688</td>
<td>688</td>
<td>688</td>
<td>688</td>
<td>1016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>1398</td>
<td>1842</td>
<td>1838</td>
<td>1838</td>
<td>1838</td>
<td>1838</td>
<td>1838</td>
<td>1838</td>
<td>1841</td>
<td>1912</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.3: Complete results for the call graph nodes metric. Sizes (upper row) and misses (lower row) of the result sets for context insensitive (CI), this-sensitive (TS) etc. analysis, without and with filters (“F”).
Table C.4: Complete results for the call graph edges metric. Sizes (upper row) and misses (lower row) of the result sets for context insensitive (CI), this-sensitive (TS), etc. analysis, without and with filters (‘F’).

<table>
<thead>
<tr>
<th>Program</th>
<th>CI</th>
<th>CI-F</th>
<th>TS</th>
<th>TS-F</th>
<th>OS</th>
<th>OS-F</th>
<th>CS</th>
<th>CS-F</th>
<th>Spark</th>
<th>dylib</th>
</tr>
</thead>
<tbody>
<tr>
<td>jython</td>
<td>619</td>
<td>1331</td>
<td>38</td>
<td>1116</td>
<td>30</td>
<td>2461</td>
<td>94</td>
<td>1683</td>
<td>383</td>
<td>1604</td>
</tr>
<tr>
<td>javacc</td>
<td>377</td>
<td>5385</td>
<td>38</td>
<td>5366</td>
<td>30</td>
<td>5210</td>
<td>94</td>
<td>5366</td>
<td>383</td>
<td>5606</td>
</tr>
<tr>
<td>antlr</td>
<td>320</td>
<td>3742</td>
<td>38</td>
<td>3742</td>
<td>30</td>
<td>3742</td>
<td>94</td>
<td>3742</td>
<td>383</td>
<td>2625</td>
</tr>
<tr>
<td>emma</td>
<td>182</td>
<td>761</td>
<td>38</td>
<td>761</td>
<td>30</td>
<td>761</td>
<td>94</td>
<td>761</td>
<td>383</td>
<td>75</td>
</tr>
<tr>
<td>ant</td>
<td>168</td>
<td>6185</td>
<td>38</td>
<td>6185</td>
<td>30</td>
<td>6185</td>
<td>94</td>
<td>6185</td>
<td>383</td>
<td>2577</td>
</tr>
<tr>
<td>lucene</td>
<td>170</td>
<td>643</td>
<td>38</td>
<td>643</td>
<td>30</td>
<td>643</td>
<td>94</td>
<td>643</td>
<td>383</td>
<td>125</td>
</tr>
<tr>
<td>javadoc</td>
<td>244</td>
<td>761</td>
<td>38</td>
<td>761</td>
<td>30</td>
<td>761</td>
<td>94</td>
<td>761</td>
<td>383</td>
<td>75</td>
</tr>
<tr>
<td>jfreech</td>
<td>320</td>
<td>2635</td>
<td>38</td>
<td>2635</td>
<td>30</td>
<td>2635</td>
<td>94</td>
<td>2635</td>
<td>383</td>
<td>75</td>
</tr>
<tr>
<td>recoder</td>
<td>388</td>
<td>98</td>
<td>38</td>
<td>98</td>
<td>30</td>
<td>98</td>
<td>94</td>
<td>98</td>
<td>383</td>
<td>75</td>
</tr>
<tr>
<td>Program</td>
<td>dyn</td>
<td>CI</td>
<td>CI-F</td>
<td>TS</td>
<td>TS-F</td>
<td>OS</td>
<td>OS-F</td>
<td>CS</td>
<td>CS-F</td>
<td>Spark</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>bloat</td>
<td>4787</td>
<td>19454</td>
<td>19309</td>
<td>18551</td>
<td>18355</td>
<td>18551</td>
<td>18355</td>
<td>18926</td>
<td>18756</td>
<td>22094</td>
</tr>
<tr>
<td>javac</td>
<td>1507</td>
<td>4681</td>
<td>4681</td>
<td>4582</td>
<td>4582</td>
<td>4580</td>
<td>4580</td>
<td>4608</td>
<td>4608</td>
<td>5604</td>
</tr>
<tr>
<td>javacc</td>
<td>912</td>
<td>1716</td>
<td>1716</td>
<td>1715</td>
<td>1715</td>
<td>1715</td>
<td>1715</td>
<td>1715</td>
<td>1715</td>
<td>1821</td>
</tr>
<tr>
<td>sablecc</td>
<td>5646</td>
<td>12478</td>
<td>12478</td>
<td>11676</td>
<td>11676</td>
<td>11676</td>
<td>11676</td>
<td>11433</td>
<td>11433</td>
<td>14990</td>
</tr>
<tr>
<td>recoder</td>
<td>8289</td>
<td>15361</td>
<td>15304</td>
<td>15318</td>
<td>15261</td>
<td>15288</td>
<td>15224</td>
<td>15325</td>
<td>15268</td>
<td>16752</td>
</tr>
<tr>
<td>emma</td>
<td>737</td>
<td>2158</td>
<td>2106</td>
<td>2055</td>
<td>2017</td>
<td>2055</td>
<td>2017</td>
<td>2073</td>
<td>2022</td>
<td>2680</td>
</tr>
<tr>
<td>javadoc</td>
<td>2458</td>
<td>2112</td>
<td>2112</td>
<td>2092</td>
<td>2092</td>
<td>2092</td>
<td>2092</td>
<td>2092</td>
<td>2092</td>
<td>2723</td>
</tr>
<tr>
<td>jfreechart</td>
<td>1136</td>
<td>2065</td>
<td>2064</td>
<td>2476</td>
<td>2475</td>
<td>2475</td>
<td>2475</td>
<td>2518</td>
<td>2517</td>
<td>6204</td>
</tr>
<tr>
<td>jython</td>
<td>5758</td>
<td>31197</td>
<td>29834</td>
<td>30868</td>
<td>28797</td>
<td>30071</td>
<td>28807</td>
<td>27695</td>
<td>3018</td>
<td></td>
</tr>
<tr>
<td>lucene</td>
<td>557</td>
<td>932</td>
<td>925</td>
<td>929</td>
<td>922</td>
<td>929</td>
<td>922</td>
<td>930</td>
<td>923</td>
<td>1498</td>
</tr>
<tr>
<td>pmd</td>
<td>4594</td>
<td>7977</td>
<td>7883</td>
<td>7823</td>
<td>7729</td>
<td>7823</td>
<td>7729</td>
<td>7832</td>
<td>7738</td>
<td>8590</td>
</tr>
</tbody>
</table>

Table C.5: Complete results for the object call graph nodes metric. Sizes (upper row) and misses (lower row) of the result sets for context insensitive (CI), this-sensitive (TS) etc. analysis, without and with filters (“F”).
### Table C.6: Complete results for the object call graph edges metric. Sizes (upper row) and misses (lower row) of the result sets for context insensitive (CI), this-sensitive (TS) etc. analysis, without and with filters ("F").

<table>
<thead>
<tr>
<th>Program</th>
<th>CI</th>
<th>CI-F</th>
<th>TS</th>
<th>TS-F</th>
<th>OS</th>
<th>OS-F</th>
<th>CS</th>
<th>CS-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>dyn</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>javac</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>javacc</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>sablecc</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>recoder</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sablecc</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>javac</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>sablecc</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>javac</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>sablecc</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Program</td>
<td>CI vs CI-F</td>
<td>TS vs TS-F</td>
<td>OS vs OS-F</td>
<td>CS vs CS-F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>-1143</td>
<td>-855</td>
<td>-855</td>
<td>-864</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.67%</td>
<td>-0.65%</td>
<td>-0.65%</td>
<td>-0.66%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javac</td>
<td>-7</td>
<td>-7</td>
<td>-7</td>
<td>-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.04%</td>
<td>-0.05%</td>
<td>-0.06%</td>
<td>-0.05%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javacc</td>
<td>-2</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.23%</td>
<td>-0.73%</td>
<td>-0.73%</td>
<td>-0.73%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sablecc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recoder</td>
<td>-4744</td>
<td>-974</td>
<td>-1072</td>
<td>-974</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-10.19%</td>
<td>-2.57%</td>
<td>-3.03%</td>
<td>-2.53%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>antlr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>emma</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1.48%</td>
<td>-1.48%</td>
<td>-0.71%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javadoc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jfreechart</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>-3303</td>
<td>-2896</td>
<td>-2157</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-6.88%</td>
<td>-7.90%</td>
<td>-5.99%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lucene</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.31%</td>
<td>-0.32%</td>
<td>-0.32%</td>
<td>-0.32%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>-651</td>
<td>-586</td>
<td>-586</td>
<td>-692</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-3.25%</td>
<td>-3.48%</td>
<td>-3.48%</td>
<td>-4.08%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.7: Changes in precision when applying the filter operations, absolute and relative, metric heap.
Table C.8: Changes in precision when applying the filter operations, absolute and relative, metric call graph nodes.
<table>
<thead>
<tr>
<th>Program</th>
<th>CI vs CI-F</th>
<th>TS vs TS-F</th>
<th>OS vs OS-F</th>
<th>CS vs CS-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>-62</td>
<td>-67</td>
<td>-67</td>
<td>-63</td>
</tr>
<tr>
<td></td>
<td>-0.54%</td>
<td>-0.60%</td>
<td>-0.60%</td>
<td>-0.56%</td>
</tr>
<tr>
<td>javac</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javacc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sablecc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recoder</td>
<td>-295</td>
<td>-304</td>
<td>-304</td>
<td>-309</td>
</tr>
<tr>
<td></td>
<td>-1.88%</td>
<td>-1.94%</td>
<td>-1.94%</td>
<td>-1.97%</td>
</tr>
<tr>
<td>antlr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>emma</td>
<td>-2</td>
<td>-29</td>
<td>-29</td>
<td>-27</td>
</tr>
<tr>
<td></td>
<td>-0.10%</td>
<td>-1.51%</td>
<td>-1.51%</td>
<td>-1.40%</td>
</tr>
<tr>
<td>javadoc</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td></td>
<td>-0.08%</td>
<td>-0.08%</td>
<td>-0.08%</td>
<td>-0.08%</td>
</tr>
<tr>
<td>jfreechart</td>
<td>-9</td>
<td>-9</td>
<td>-9</td>
<td>-9</td>
</tr>
<tr>
<td></td>
<td>-0.47%</td>
<td>-0.47%</td>
<td>-0.47%</td>
<td>-0.47%</td>
</tr>
<tr>
<td>jython</td>
<td>-301</td>
<td>-294</td>
<td>-233</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-6.88%</td>
<td>-7.90%</td>
<td>-5.99%</td>
<td></td>
</tr>
<tr>
<td>lucene</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td>-0.27%</td>
<td>-0.27%</td>
<td>-0.27%</td>
<td>-0.27%</td>
</tr>
<tr>
<td>pmd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.9: Changes when applying the filter operations, absolute and relative, metric call graph edges. Changes in recall are observed only for project jython.
Table C.10: Changes when applying the filter operations, absolute and relative, metric object call graph nodes. Changes in recall are observed only for project jython.
<table>
<thead>
<tr>
<th>Program</th>
<th>CI vs CI-F</th>
<th>TS vs TS-F</th>
<th>OS vs OS-F</th>
<th>CS vs CS-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>-33813</td>
<td>-56165</td>
<td>-8464</td>
<td>-34491</td>
</tr>
<tr>
<td></td>
<td>-3.40%</td>
<td>-7.25%</td>
<td>-1.28%</td>
<td>-3.70%</td>
</tr>
<tr>
<td>javac</td>
<td>-7</td>
<td>-7</td>
<td>-7</td>
<td>-7</td>
</tr>
<tr>
<td></td>
<td>-0.01%</td>
<td>-0.01%</td>
<td>-0.01%</td>
<td>-0.01%</td>
</tr>
<tr>
<td>javacc</td>
<td>-13</td>
<td>-13</td>
<td>-13</td>
<td>-13</td>
</tr>
<tr>
<td></td>
<td>-0.25%</td>
<td>-0.25%</td>
<td>-0.26%</td>
<td>-0.25%</td>
</tr>
<tr>
<td>sablecc</td>
<td>-157</td>
<td>-482</td>
<td>-52</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.03%</td>
<td>-0.30%</td>
<td>-0.03%</td>
<td></td>
</tr>
<tr>
<td>recoder</td>
<td>-2254</td>
<td>-2313</td>
<td>-2319</td>
<td>-2338</td>
</tr>
<tr>
<td></td>
<td>-1.18%</td>
<td>-1.24%</td>
<td>-1.30%</td>
<td>-1.25%</td>
</tr>
<tr>
<td>anthr</td>
<td>-13</td>
<td>-13</td>
<td>-13</td>
<td>-13</td>
</tr>
<tr>
<td></td>
<td>-0.17%</td>
<td>-0.18%</td>
<td>-0.18%</td>
<td>-0.18%</td>
</tr>
<tr>
<td>emma</td>
<td>-269</td>
<td>-414</td>
<td>-422</td>
<td>-402</td>
</tr>
<tr>
<td></td>
<td>-3.60%</td>
<td>-6.39%</td>
<td>-7.06%</td>
<td>-5.95%</td>
</tr>
<tr>
<td>javadoc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jfreechart</td>
<td>-15</td>
<td>-15</td>
<td>-15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.13%</td>
<td>-0.18%</td>
<td>-0.17%</td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>-421458</td>
<td>-532707</td>
<td>-346350</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-14.00%</td>
<td>-18.00%</td>
<td>-12.47%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>20</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>lucene</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td>-0.19%</td>
<td>-0.20%</td>
<td>-0.20%</td>
<td>-0.20%</td>
</tr>
<tr>
<td>pmd</td>
<td>-2070</td>
<td>-2070</td>
<td>-1854</td>
<td>-2070</td>
</tr>
<tr>
<td></td>
<td>-1.61%</td>
<td>-1.88%</td>
<td>-1.74%</td>
<td>-1.88%</td>
</tr>
</tbody>
</table>

Table C.11: Changes when applying the filter operations, absolute and relative, metric object call graph edges. Changes in recall are observed only for project jython.
### Appendix C. Evaluation: The Complete Metrics Data

Table C.12: Combining the context-sensitive approaches. Listed are the best single results, and the improvement after combination. Empty cells mean no improvement.

<table>
<thead>
<tr>
<th>Program</th>
<th>best</th>
<th>combined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>heap</td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>best</td>
<td>TS, OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>129710</td>
</tr>
<tr>
<td>javac</td>
<td>best</td>
<td>OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11538</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-4.3%</td>
</tr>
<tr>
<td>javacc</td>
<td>best</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>678</td>
</tr>
<tr>
<td>sablecc</td>
<td>best</td>
<td>TS, OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5028</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recoder</td>
<td>best</td>
<td>OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35252</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anltlr</td>
<td>best</td>
<td>TS, OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1284</td>
</tr>
<tr>
<td>emma</td>
<td>best</td>
<td>TS, OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>332</td>
</tr>
<tr>
<td>javadoc</td>
<td>best</td>
<td>OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12857</td>
</tr>
<tr>
<td>jfreech</td>
<td>best</td>
<td>TS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>713</td>
</tr>
<tr>
<td>jython</td>
<td>best</td>
<td>TS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>33760</td>
</tr>
<tr>
<td>lucene</td>
<td>best</td>
<td>TS, OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>309</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>best</td>
<td>TS, OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16273</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


Bibliography


