Benchmarking Points-to Analysis
Abstract
Points-to analysis is a static program analysis that, simply put, computes which objects created at certain points of a given program might show up at which other points of the same program. In particular, it computes 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.

Comparing experimental results with respect to accuracy and performance is required in order to distinguish the promising from the less promising approaches to points-to analysis. Unfortunately, comparing the accuracy of two different points-to analysis implementations is difficult, as there are many pitfalls in the details. In particular, there are no standardized means to perform such a comparison, i.e., no benchmark suite – a set of programs with well-defined rules of how to compare different points-to analysis results – exists. Therefore, different researchers use their own means to evaluate their approaches to points-to analysis. To complicate matters, even the same researchers do not stick to the same evaluation methods, which often makes it impossible to take two research publications and reliably tell which one describes the more accurate points-to analysis.

In this thesis, we define a methodology on how to benchmark points-to analysis. We create a benchmark suite, compare three different points-to analysis implementations with each other based on this methodology, and explain differences in analysis accuracy.

We also argue for the need of a Gold Standard, i.e., a set of benchmark programs with exact analysis results. Such a Gold Standard is often required to compare points-to analysis results, and it also allows to assess the exact accuracy of points-to analysis results. Since such a Gold Standard cannot be computed automatically, it needs to be created semi-automatically by the research community. We propose a process for creating a Gold Standard based on under-approximating it through optimistic (dynamic) analysis and over-approximating it through conservative (static) analysis. With the help of improved static and dynamic points-to analysis and expert knowledge about benchmark programs, we present a first attempt towards a Gold Standard.

We also provide a Web-based benchmarking platform, through which researchers can compare their own experimental results with those of other researchers, and can contribute towards the creation of a Gold Standard.
Abstract

Points-to analysis is a static program analysis that, simply put, computes which objects created at certain points of a given program might show up at which other points of the same program. In particular, it computes 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.

Comparing experimental results with respect to accuracy and performance is required in order to distinguish the promising from the less promising approaches to points-to analysis. Unfortunately, comparing the accuracy of two different points-to analysis implementations is difficult, as there are many pitfalls in the details. In particular, there are no standardized means to perform such a comparison, i.e., no benchmark suite – a set of programs with well-defined rules of how to compare different points-to analysis results – exists. Therefore, different researchers use their own means to evaluate their approaches to points-to analysis. To complicate matters, even the same researchers do not stick to the same evaluation methods, which often makes it impossible to take two research publications and reliably tell which one describes the more accurate points-to analysis.

In this thesis, we define a methodology on how to benchmark points-to analysis. We create a benchmark suite, compare three different points-to analysis implementations with each other based on this methodology, and explain differences in analysis accuracy.

We also argue for the need of a Gold Standard, i.e., a set of benchmark programs with exact analysis results. Such a Gold Standard is often required to compare points-to analysis results, and it also allows to assess the exact accuracy of points-to analysis results. Since such a Gold Standard cannot be computed automatically, it needs to be created semi-automatically by the research community. We propose a process for creating a Gold Standard based on under-approximating it through optimistic (dynamic) analysis and over-approximating it through conservative (static) analysis. With the help of improved static and dynamic points-to analysis and expert knowledge about benchmark programs, we present a first attempt towards a Gold Standard.

We also provide a Web-based benchmarking platform, through which researchers can compare their own experimental results with those of other researchers, and can contribute towards the creation of a Gold Standard.
This thesis is based on the following refereed publications:


Tobias Gutzmann, Welf Löwe: Reducing the Performance Overhead of Dynamic Analysis through Custom-made Agents. 5th International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2010).


This thesis is also a direct extension of:

This thesis is based on the following refereed publications:


Tobias Gutzmann, Welf Löwe: Reducing the Performance Overhead of Dynamic Analysis through Custom-made Agents. 5th International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2010).


This thesis is also a direct extension of:

Acknowledgments

First, my thanks go to my supervisor, Welf Löwe, without whom this thesis would not have become reality, and to all my colleagues at Växjö respectively Linnaeus University, for the pleasant working environment and fruitful discussions.

Second, to my mother, who certainly would have liked me staying closer to home, but who has nevertheless supported me all these years. To my dad, who sadly cannot witness the completion of this thesis. I am sure he would have been proud of me.

Finally, my thanks go to my love, Therése, for all her support and patience these last years.
Acknowledgments

First, my thanks go to my supervisor, Welf Löwe, without whom this thesis would not have become reality, and to all my colleagues at Växjö respectively Linnaeus University, for the pleasant working environment and fruitful discussions.

Second, to my mother, who certainly would have liked me staying closer to home, but who has nevertheless supported me all these years. To my dad, who sadly cannot witness the completion of this thesis. I am sure he would have been proud of me.

Finally, my thanks go to my love, Therése, for all her support and patience these last years.
# Contents

Abstract ................................................................. i

Acknowledgments ......................................................... v

1 Introduction ...................................................................... 1
  1.1 Goals ........................................................................ 2
  1.2 Restrictions ............................................................ 2
  1.3 Goal Criteria ........................................................... 2
  1.4 Tasks ........................................................................ 3
  1.5 Motivation ............................................................... 4
  1.6 Thesis Outline ......................................................... 6
  1.7 Disclaimer ................................................................... 7

2 Points-to Analysis .......................................................... 8
  2.1 Program Analysis ....................................................... 8
  2.2 Points-to Analysis Overview ....................................... 10
  2.3 Naming Schemes ....................................................... 11
  2.4 Flow Sensitivity ....................................................... 12
  2.5 Context Sensitivity ................................................. 12
  2.6 Abstract Heap Modeling .......................................... 15
  2.7 Path Sensitivity ........................................................ 15
  2.8 Open Problems ........................................................ 16
  2.9 Concrete Implementations ......................................... 17
  2.10 Conclusion ............................................................. 29

3 Related Work .................................................................... 30
  3.1 Comparing Dynamic and Static Analyses .................... 30
  3.2 Gold Standards ....................................................... 32
  3.3 Fast Dynamic Analysis with help of Static Analysis ........ 33
  3.4 Evaluation Methods for Points-to Analysis .................. 33
  3.5 Conclusion ............................................................. 44

4 Comparing “May” Dataflow Analyses ................................. 47
  4.1 Comparing Analyses .................................................. 48
  4.2 Improving General Analyses ...................................... 56
  4.3 Conclusion ............................................................. 57
Introduction

Points-to analysis is a static program analysis that, simply put, computes which objects created at certain points of a given program might show up at which other points of the same program. In particular, it computes 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.

Comparing research results with respect to accuracy and performance is required in order to distinguish the promising from the less promising approaches to points-to analysis.

Unfortunately, comparing merely the accuracy of two different points-to analysis implementations is difficult, as there are no standardized means to perform such a comparison. In current research, different authors use different accuracy assessment methods and different benchmark programs for evaluating their approaches to points-to analysis. Researchers commonly use their "own" points-to analysis implementation and compare variations thereof, i.e., they compare analysis X with X'. However, comparison of analysis X to another research group's implementation Y (and Y' etc.) rarely happens. This makes it difficult to take two research publications and reliably tell which one describes the more accurate points-to analysis.

Further, little can be said about the absolute accuracy of points-to analysis, as there is no Gold Standard, i.e., a specific set of benchmark programs with an accepted set of correct analysis results, for it. Such a Gold Standard seems impossible to be computed automatically, and thus creating it requires manual steps by researchers. This task has not yet been tackled for points-to analysis.

Finally, points-to analyses are usually assumed to be conservative, i.e., compute super-sets of the exact analysis results. However, points-to analyses quite often turn out to be, at best, conservative only for subsets of programs. Otherwise, they are general analyses, as they lack support for certain language features or native methods. As we shall see, assessing the accuracy of general points-to analysis also requires a Gold Standard.
Chapter 1

Introduction

Points-to analysis is a static program analysis that, simply put, computes which objects created at certain points of a given program might show up at which other points of the same program. In particular, it computes 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.

Comparing research results with respect to accuracy and performance is required in order to distinguish the promising from the less promising approaches to points-to analysis.

Unfortunately, comparing merely the accuracy of two different points-to analysis implementations is difficult, as there are no standardized means to perform such a comparison. In current research, different authors use different accuracy assessment methods and different benchmark programs for evaluating their approaches to points-to analysis. Researchers commonly use their “own” points-to analysis implementation and compare variations thereof, i.e., they compare analysis X with X’. However, comparison of analysis X to another research group’s implementation Y (and Y’ etc.) rarely happens. This makes it difficult to take two research publications and reliably tell which one describes the more accurate points-to analysis.

Further, little can be said about the absolute accuracy of points-to analysis, as there is no Gold Standard, i.e., a specific set of benchmark programs with an accepted set of correct analysis results, for it. Such a Gold Standard seems impossible to be computed automatically, and thus creating it requires manual steps by researchers. This task has not yet been tackled for points-to analysis.

Finally, points-to analyses are usually assumed to be conservative, i.e., compute super-sets of the exact analysis results. However, points-to analyses quite often turn out to be, at best, conservative only for subsets of programs. Otherwise, they are general analyses, as they lack support for certain language features or native methods. As we shall see, assessing the accuracy of general points-to analysis also requires a Gold Standard.
Chapter 1. Introduction

1.1 Goals

The goals of this thesis are the following:

1. Make points-to analyses comparable in terms of accuracy.

2. Show how to create a Gold Standard for points-to analysis in order to assess the absolute accuracy of points-to analyses.

1.2 Restrictions

We limit our work to the programming language Java, but we expect that many of our findings can be applied to other programming languages as well. We look only at comparison of analysis accuracy, and omit comparison of performance.

1.3 Goal Criteria

The criteria for fulfilling our first goal are:

1. Theoretical foundation: We define a theoretically-founded benchmarking methodology for assessing the accuracy of points-to analysis.

2. Efficiency: We provide tool support for applying this benchmarking methodology.

3. Practicability: We show the practicability of the benchmarking methodology by presenting an experimental comparison of at least two existing, fundamentally different points-to analysis implementations.

The criteria for fulfilling our second goal are:

4. Distribution: We define a converging process that allows researchers to collaborate on working towards a Gold Standard for points-to analysis.

5. Efficiency: We provide tool support for applying this distributed process.

6. Feasibility: We show the feasibility of the process by presenting a first attempt towards a Gold Standard.
1.4 Tasks

For our first goal, we perform the following tasks:

- Make a survey of evaluation methods for points-to analysis in literature, and point out commonalities and differences among them.

- Define a theoretical framework for assessing the accuracy of may-dataflow analysis results in general, and points-to analysis in particular. We have published work on this in [32].

- Propose a benchmarking methodology for assessing accuracy of points-to analysis.

- Provide a benchmarking platform where researchers can submit results from their points-to analysis implementations, and compare them to other submitted results (by themselves or other researchers).

- Connect at least two different points-to analysis implementations to the benchmarking platform and experimentally compare them with help of the platform. We have performed such comparisons for the first time in [32] and described an initial tool set for comparing different points-to analysis implementations in [31].

For our second goal, we perform the following tasks:

- Propose a converging process that researchers should follow for a joint effort on creating a Gold Standard for points-to analysis.

- Extend the benchmarking platform so that it supports the single steps of this process.

- Describe what and how (semi-)manual efforts must aid the creation of a Gold Standard. What such manual efforts might look like has been described in [35].

- Find and evaluate new ways for improving the accuracy of points-to analysis. We have published several papers on this [34, 52, 35, 36].

- Improve dynamic coverage of benchmark programs with respect to points-to analysis. This has been partly addressed in [35].

- Decrease the performance overhead of dynamic analysis in order to allow efficient collection of dynamic points-to information. We have presented an approach to this in [33].
Chapter 1. Introduction

1.5 Motivation

Points-to analysis is an important research area as it is widely used in many types of client applications in optimizing compilers and software engineering tools. Examples for the application in optimizing compilers are: virtual call resolution (e.g., [43, 61]), thread-escape analysis and type refinement [92].

Examples of software engineering tool applications are: metrics analyses computing coupling and cohesion between objects [26, 38] and architectural recovery by class clustering proposing groupings of classes, either based on coupling and cohesion or directly on reference information [84, 75]. Source code browsers compute forward and backward slices [41] of a program point which, in turn, requires reference information. In software testing, class dependencies determine the test order [13, 88, 59]. Reverse engineering of UML interaction diagrams requires very accurate reference information in order to be useful [89]. Finally, static design pattern detection needs to identify the interaction among participating classes and object instances in order to exclude false positives [76].

Sim et al. observe that the development of benchmarks in computer science disciplines is often accompanied with technical progress and community building [82]. The lack of such benchmarks, in turn, makes it difficult to further develop a field by adopting the successful and avoiding the less promising approaches.

A benchmark for points-to analysis would specify what programs to analyze, and how to actually measure accuracy. The latter requires a set of well-defined client analyses, i.e., concrete applications of points-to information, as points-to information in itself has little value and is, in general, not comparable experimentally among different implementations.

No such commonly accepted benchmark exists for points-to analysis. Quite the contrary: For points-to analysis, even qualitative comparability of results from different research groups is not straight-forward because different authors often do not use the same set of client analyses for evaluation. Further, even if authors use the same client analyses, they often use different interpretations of them. To exemplify this matter, consider the commonly used client analysis “call graph construction,” which computes which methods can possibly call what other methods during any program run. Authors often make different assumptions on whether or not to include static initializers, and whether or not (or to what extent) calls to and from library methods are included in the call graph. Additionally, applying the same client analyses to different versions of the same programs makes the results incomparable again, as different versions of the same program must be considered as different programs from a points-to analysis’ perspective. Unfortunately, it happens frequently that versions of analyzed programs are not specified in research publications.
Moreover, common points-to analysis implementations often support only a subset of the programming language that they are written for. For instance, in Java, dynamic class loading (or reflection in general) and native methods cause problems. The supported subsets of different implementations are often not equal, which again hampers comparability of analysis results – the lack of support for a given programming language feature means that only a subset $P'$ of a given program $P$ is actually analyzed. If $P'$ differs between two points-to analysis implementations, then comparability is not possible even for results for the same program in the same version.

Susan Horwitz is cited in [39]:

Improvements proposed by researchers seem promising, but seldom are claims independently verified, and often promising leads are abandoned. It seems that duplicating others’ results is considered very important in the physical sciences, but gets short shrift in computer science. Should we/can we change that attitude?

To our knowledge, only a few fundamental approaches to points-to analysis have been individually verified, because not all researchers make their implementations publicly available – there may be good reasons for that, like licensing issues or simply shortage of time for making an experimental implementation “fit for public use” – and re-implementing another’s ideas is tedious work and has little chances of success in becoming a publication, which is the researcher’s daily bread.

The criteria for our first goal tackle all of the above problems: By defining a benchmark methodology and providing tool support in the form of a platform where researchers can submit their results for a specified set of programs and according to given rules, results become comparable. Further, results can be easily made available for assessment by third parties, and the need to disclose source code for third-party verification is diminished.

Barbara Ryder is cited in the very same paper:

We can all write an unbounded number of papers that compare different pointer analysis\textsuperscript{1} approximations in the abstract.

Our first goal aims at easing this issue, as comparing different points-to analyses experimentally is not a straightforward task. However, this citation is taken a bit out of context. Ryder continues:

However, this does not accomplish the key goal, which is to design and engineer pointer analyses that are useful for solving real software problems for realistic programs.

\textsuperscript{1}Note that pointer analysis and points-to analysis are equivalent terms.
Chapter 1. Introduction

The more accurate the underlying points-to analysis becomes, the better becomes the client application (solution to the “software problem,” as Ryder calls it). Using a Gold Standard for points-to analysis as input to a client application would show its full potential (or lack thereof) in its respective application area. Using this, researchers could determine if there even is a points-to analysis that is accurate enough for a given software problem.

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 program – was performed by Rountev et al. [73, 72]. 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 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. Beginning with 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. A platform that allows researchers to work together on these tasks would distribute, and thus ease, the effort for everybody.

The amount of time required for such an approach is obviously dependent on the quality of both the dynamic analysis 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 more and diverse input to running the programs. The static analysis results improve through more accurate algorithms. The amount of time required for such an approach is obviously dependent on the quality of both the dynamic analysis 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 more and diverse input to running the programs. The static analysis results improve through more accurate algorithms. Both approaches can, in general, never define the Gold Standard by themselves as the general problem is not decidable: 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 inaccuracies.

1.6 Thesis Outline

This thesis is organized as follows: In Chapter 2, we describe concepts of points-to analysis and also present three concrete implementations. In Chapter 3, we discuss related work with respect to this thesis. Then, in Chapter 4, we discuss theoretical considerations when comparing experimental results of two may-dataflow analyses in general. As we shall see, this is possible only in
special cases or with the help of a Gold Standard. We present a benchmarking methodology for points-to analysis in Chapter 5 that is based on making these special cases appear more often in practice. In Chapter 6, we describe a process for creating a Gold Standard, which is based on improving both dynamic and conservative static analysis. We also present new improvements to both static and dynamic analysis. In Chapter 7, we describe a benchmarking platform that supports the application of our benchmarking methodology and that also aids our proposed process for creating a Gold Standard. In Chapter 8, we evaluate the different contributions of this thesis. Finally, in Chapter 9, we conclude this thesis and discuss future work.

1.7 Disclaimer

This thesis is based on previous publications, and it is especially an extension of the author’s licentiate thesis [31]. Therefore, some text sections are similar, yet revised and/or extended, and in some cases almost identical to text found in those publications. In particular, the following parts of this thesis are closely based on previous publications: Chapter 2 is, with some adaptions, taken from [31]. Chapter 4 is based on [32, 31]; Section 6.3 is based on [34, 32, 36] and Section 6.4 is based on [33].
Chapter 2

Points-to Analysis

In this chapter, we describe the technical aspects of points-to analysis. We describe the general idea of how points-to analyses work, name different variation points that affect its accuracy and performance, point out open problems with respect to soundness, and describe three concrete implementations.

This chapter is organized as follows: First, we present general concepts of program analysis in Section 2.1 on which points-to analysis is founded. Then we give an overview of how points-to analysis works internally in Section 2.2. We then discuss different variation points of points-to analysis that influence its accuracy: In Section 2.3, we discuss naming schemes, that is, how points-to analysis deals with the potentially infinitely many runtime objects. Then we discuss flow sensitivity in Section 2.4, context sensitivity in Section 2.5, modeling of the abstract heap in Section 2.6, and path sensitivity in Section 2.7. We then discuss open problems in Section 2.8 and present three concrete points-to analysis implementations: Spark, Paddle, and Points-to SSA-based Simulated Execution in Section 2.9. We conclude this chapter in Section 2.10.

2.1 Program Analysis

In this section, we first describe some more general concepts of program analysis on which points-to analysis is based.

2.1.1 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 briefly 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 for each execution of the program; thus, a correct (yet not useful) answer to a given analysis question by this kind of analysis is to say nothing at all.

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

2a A conservative may-analysis aims to exclude 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.

2b An optimistic may-analysis answers a given analysis question by collecting facts that are true for at least one execution of the program. However, other facts that are not found by the analysis may also hold.

Conservative must-analysis and optimistic may-analysis compute under-approximations of the exact answer, whereas optimistic must-analysis and conservative may-analysis compute over-approximations of the exact answer.

As an example for the two may-analyses, consider the following program analysis question: What values may a given integer variable $v$ of a given program assume during an execution of the program? An optimistic analysis could now observe cases in which $v$ assumes values from 5 to 10, while a conservative analysis could prove that $v$ cannot assume values bigger than 20 or smaller than 3. The exact answer must then lie somewhere in between.

A similar must-analysis question could be formulated: Must $v > 0$ always hold? Now an optimistic analysis would attempt to disprove it, and if it fails to do so, it would answer the question with “yes.” A conservative must-analysis would attempt to prove that this is the case, and it would answer “no” unless it succeeds.

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

2.1.2 Static vs. Dynamic Analysis

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

The former analyzes a program without actually executing it, i.e., independent of a given input\(^1\). 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

\(^1\)Input can be, for example, files read from the hard drive, arguments given on the command line, and also user input via mouse or keyboard.
Chapter 2. Points-to Analysis

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 over-approximates the result and is thus often considered to be conservative; however, lack of support for a certain part of the programming language or native methods may yield general analysis results.

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 variable is being assigned; however, since the program can be monitored under all possible input only in trivial cases, this will lead to an under-approximation of the exact analysis results. Dynamic analysis is thus an optimistic analysis.

2.1.3 Approaches to Static Analysis

There are different approaches to static program analysis, e.g., constraint-based approaches [64, 8] and dataflow analysis. We describe only the latter, as the concrete points-to analysis implementations that we present later in this chapter are dataflow analysis-based approaches.

The basis for dataflow analysis is the theory of monotone dataflow frameworks (MDF) [57, 64]. An MDF is defined by a bounded value lattice, i.e., a partially ordered set with top and bottom elements, \( \mathcal{L}_V = \{ \bot, \cap, \sqcup, \top \} \) and a set \( \mathcal{F} \) of monotone transfer functions \( f : \mathcal{L}_V \rightarrow \mathcal{L}_V \). Transfer functions are derived from concrete semantics of programming language constructs, whereas value lattices abstract answers to the analysis question, e.g., from values a variable may take during a program run.

It is required that the transfer functions are monotone, i.e., it holds
\[
\forall v, w \in V, \forall f \in \mathcal{F} : v \sqsubseteq w \Rightarrow f(v) \sqsubseteq f(w),
\]
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, the algorithm terminates as soon as applying the transfer functions to all intermediate analysis results does not change their values, i.e., the analysis reaches a fixed point. There may, in general, be more than one fixed point for a given analysis problem.

2.2 Points-to Analysis Overview

The task of points-to analysis, as well as its use in client applications, has already been presented in the introduction of this thesis. In this and the
following sections, we describe points-to analysis as an instantiation of the MDF and discuss different aspects that affect its analysis accuracy and cost.

Many points-to analyses work as follows: A program is represented by a program graph where nodes correspond to program points, and edges correspond to control and data dependencies among them. Outgoing from a set of entry points, the program graph is created by computing what code is potentially reachable through them. These entry points are methods that are called through system initialization; in Java, these are at least the program’s main() method as well as System.\_initialize\_System\_Class(). Analysis values for each node are computed iteratively by merging values from predecessor nodes and by applying transfer functions that represent the abstract program behavior at these nodes. For instance, Allocation-nodes create abstract objects; Load- and Store-nodes read from and write to a common abstract heap. At control flow confluence points, analysis values are merged, as the points-to analysis must assume that every possible control flow path might be taken.

Points-to analysis needs to abstract from the values which expressions may take during a real application run, as it is impossible to statically model the exact program state at any time of any possible run of a program. In particular, the possibly infinitely many runtime objects need to be mapped to a finite set $O$ of abstract objects. An abstraction that maps concrete objects to abstract objects is called a naming scheme. Naming schemes are further discussed in the next section.

In a points-to analysis, reference variables will in general hold references to more than one abstract object. Hence, each points-to value $v$ in the analysis of a program is an element in the points-to value lattice $L_V = \{V, \cup, \cap, \top, \bot\}$ where $V = 2^O$ is the power set of $O$, $\top = O$, $\bot = \emptyset$, and $\cup, \cap$ 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 variable or expression $a$.

The analysis stops once a fixed point is reached, which is guaranteed to happen with the above abstractions and if the transfer functions are monotone, e.g., no strong updates (no analysis values are ever overwritten but only added and merged) are performed. Having no strong updates is sufficient but not mandatory for the MDF criterion of monotony, cf. Section 2.1.3.

### 2.3 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 finite set $O$ of abstract objects. Each abstract object $o \in O$ represents a set of concrete runtime objects $r(o)$. For this, the following must hold:

$$\forall o_1, o_2 \in O : o_1 \neq o_2 \Rightarrow r(o_1) \cap r(o_2) = \emptyset$$

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 site 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 is more accurate and should be preferred for more sophisticated analyses [74].

Even more accurate 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. [48] as well as Lhoták and Hendren [44].

### 2.4 Flow Sensitivity

Flow sensitivity is a concept that is frequently used, but there is no consensus as to its exact definition [56]. Informally, an analysis is flow-sensitive if it takes control-flow information into account [39]. Many people also require the use of so-called strong (or killing) updates as a criteria for flow sensitivity [74]. 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 [37]. In such a case, the strong updates do not violate the monotony criteria of an underlying MDF.

### 2.5 Context Sensitivity

In a context-insensitive analysis, analysis values of different call sites are propagated to the same method and get mixed there. For example, a context-insensitive analysis does not distinguish between arguments passed to (and
2.5. Context Sensitivity

```java
1: class T {
2:     T self() {
3:         return this;
4:     }
5:     void foo() {
6:         // assume pt(this) = { o1 }
7:         T t1 = self();
8:     }
9:     void bar() {
10:        // assume pt(this) = { o2 }
11:        T t2 = self();
12:    }
```

Figure 2.1: Example for benefits of context-sensitive analysis

values returned by) a method through calls at program points $S_1$ and $S_2$. 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 inaccuracies to the analysis result of each of these calls. A context-sensitive analysis addresses this source of inaccuracy by distinguishing between different calling contexts of a method. It analyzes a method separately for each calling context [74].

Context sensitivity will therefore, in general, yield a more accurate analysis. The drawbacks are the increased memory cost that comes with maintaining a larger number of contexts and their analysis values, along with the increased analysis time that may be required to reach a fixed point.

For an example of how context sensitivity can improve analysis accuracy, look at Figure 2.1. Assume that the method `foo()` is analyzed in a context with points-to set $pt(this) = \{o_1\}$, and that `bar()` is analyzed in a context with points-to set $pt(this) = \{o_2\}$. In the case of context-insensitive analysis, the points-to set for the implicit `this`-parameter of method `self()` is, once the fixed point of the analysis is reached, $pt(this) = \{o_1, o_2\}$, which is also the analysis result of the method. Therefore, each of the two variables `t1` and `t2` at lines 7 and 11, respectively, points to both abstract objects. A context-sensitive analysis could analyze `self()` in separate contexts for each call: one calling context (self, $pt(\{o_1\})$) for the call at line 7, and one calling context (self, $pt(\{o_2\})$) for the call at line 11. Then, `t1` at line 7 would point to $o_1$ only, and `t2` at line 11 would point to $o_2$ only, which is a more accurate analysis result.

Context-sensitive approaches use a finite abstraction of the 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 [80]. The call string approach defines a context by the top $k$ callers, i.e., return addresses on the
call stack top \[81\], 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 \[80, 30\]. Both the call string approach and the functional approach were evaluated and put into a common framework by Grove et al. \[30\].

A well-known functional approach designed for object-oriented languages is called object sensitivity \[60, 61\]. 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 distinguishes contexts by the top \(k\) abstract target objects on the call stack. 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 accuracy of side-effect analysis and, to a lesser degree, call graph construction, was reported. Both approaches, 1-CFA and 1-object-sensitivity, 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 accuracy \[44\]. A variation of object sensitivity, this sensitivity, has been presented by Lundberg et al. \[53, 52\]. 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. In practice, 1-this-sensitivity is almost as accurate as 1-object-sensitivity but an order of magnitude faster.

### 2.5.1 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. Each context is defined by a tuple; the tuple elements (its number and content) depend on what context definition is being used. In this thesis, we will use the following context definitions for a given call from a call site \(cs_i: a.m(v_1, \ldots, v_n)\) where \(pt(a) = \{o_1, \ldots, o_p\}\).

**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 \(m.isStatic\), \(\{(m, o_1), \ldots, (m, o_p)\}\) 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.
2.6 Abstract Heap Modeling

ThisSens: \( cs_i \rightarrow \{(m)\} \) if \( m.\text{isStatic} \), \( \{(m, pt(a))\} \) 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 \( (m, \{o_2\}) \).

Actually, ObjSens is the only context definition in this selection that may associate a call with more than one context, but multiple calls \( cs_i \) are, in general, associated with the same context.

There exist many more context definitions, e.g., the extensions of ObjSens and ThisSens to all method parameters, not just the implicit this parameter, or the combination of CallString with one of the functional approaches.

2.6 Abstract Heap Modeling

An important design decision of points-to analysis is how to model the abstract heap.

An analysis is field-sensitive if it, for an expression \( o.f \), takes both \( o \) and \( f \) to determine the abstract 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 typed languages such as Java, not only for the accuracy but even for the performance of the analysis [91].

Often, there is one global abstract heap, but Trapp mentions the possibility of partitioning the heap [90]. Prabhu and Shankar present an approach to model the heap in a field flow sensitive way, which allows maintaining different heap states for different analysis paths [69]. This could also be achieved by using so-called \( \chi \)-terms as presented by Trapp [90].

2.7 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.

Many path-sensitive approaches deal with the meet over all paths (MOP) dataflow problem, e.g., [17]. Since the number of paths is, in general, unbounded, approaches narrow down the set of paths, e.g., by finding correlations between branch conditions [28, 90]. Xie et al. [94] use path-sensitive analysis in their array access checker. Their approach to path sensitivity selects a set of execution paths – both a super- and subset of legal paths – and
eliminates infeasible paths based on branching conditions. A different approach limits the number of investigated paths by selecting interesting paths based on dynamic analysis, e.g., [9].

We have presented an approach to path sensitivity to points-to analysis that utilizes control-flow expressions to exclude infeasible dataflow on certain paths [34]. It will be presented in Section 2.9.3.8.

### 2.8 Open Problems

Some features of modern programming languages cause common problems for 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 inaccurate to be feasible. For instance, a native method could theoretically change the entire memory.

Thus, often only subsets of a given programming language or software system are analyzed by static analysis. For programs making use of these features, often no conservative analysis is feasible.

An approach to dealing with such features is described by Hirzel et al. [40]. They perform a regular points-to analysis and use the results for program optimizations. Then, they monitor 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; therefore, they use a rather inaccurate baseline points-to analysis.

An offline-approach, named *internal analysis*, is presented by Nguyen and Xue [63]. 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.

Bodden et al. present a hybrid approach [16]: An analyzed program is first analyzed with dynamic analysis. Then, the collected information regarding reflective calls is used as additional input to static analysis.
2.9 Concrete Implementations

Knowing how points-to analysis may be performed is a requirement for the understanding of this thesis, so we present three concrete implementations: Spark, Paddle, and Points-to SSA-based Simulated Execution (we will refer to the latter simply as P2SSA).

**Spark** (Section 2.9.1) is a well-known and widely used context- and flow-insensitive points-to analysis that is part of the Soot framework [5]. **Paddle** is also based on the Soot framework and supports a variety of naming schemes, context sensitivities and, most noteworthy, uses Binary Decision Diagrams (BDDs) [22, 23] for memory-efficiently storing points-to sets. **Points-to SSA** is a sparse graph-based program representation-based on static single assignment form (SSA) that is targeted at points-to analysis. It is interpreted using Simulated Execution, a semi-flow-sensitive analysis technique. We describe all three in the following; we will explain P2SSA in great detail, as it is also the implementation that we will use most for experimentation, and provide an overview of Spark’s functionality. Paddle’s general idea works similar to Spark, so we will describe it only very briefly.

A tabular comparison of Spark, Paddle, and P2SSA with respect to the concepts discussed in Sections 2.3 to 2.7 is given in Table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>flow-sensitive</th>
<th>naming scheme</th>
<th>context-sensitive</th>
<th>field-sensitive</th>
<th>path-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark</td>
<td>no</td>
<td>creation site</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Paddle</td>
<td>no</td>
<td>creation site</td>
<td>ConIns, CallString, ObjSens</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Points-to SSA-based Simulated Execution (P2SSA)</td>
<td>intra-procedural, inter-procedural</td>
<td>creation site</td>
<td>ConIns, CallString, ObjSens, ThisSens</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of Spark, Paddle, and Points-to SSA-based Simulated Execution (P2SSA).

**2.9 Concrete Implementations**

Spark, the Soot Pointer Analysis Research Kit [43], is a static points-to analysis framework taken from the Soot 2.5.0 framework. It is configurable in its accuracy; we describe here its most accurate instantiation, which is field-sensitive, context- and flow-insensitive, and uses a creation site naming scheme.
Chapter 2. Points-to Analysis

<table>
<thead>
<tr>
<th>PAG entity</th>
<th>corresponding source code pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation node</td>
<td>new T()</td>
</tr>
<tr>
<td>Variable node</td>
<td>t.t;</td>
</tr>
<tr>
<td>Field reference node</td>
<td>v = new T();</td>
</tr>
<tr>
<td>Allocation edge</td>
<td>v = w;</td>
</tr>
<tr>
<td>Assignment edge</td>
<td>v.f = w;</td>
</tr>
<tr>
<td>Store edge</td>
<td>v = w.f;</td>
</tr>
</tbody>
</table>

Table 2.2: Spark PAG: Entities and Construction

Spark constructs a *Pointer Assignment Graph (PAG)* as its representation for the program to be analyzed. Nodes model source code entities and the directed edges between them points-to constraints. The PAG construction is done by associating each relevant program statement (relevant are statements involving abstract object transport) with the construction of different PAG entities. Table 2.2 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 form the set of abstract objects $O$. *Variable* and *field reference nodes* represent accesses 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 2.2. 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 (including a recursive call in the case of method `L.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 $L.<init>$, then to the $this$ parameter of $Object.<init>$, and then to the field reference node $this.next$ in method $L.append$, etc.

Note that field sensitivity is achieved by adding nodes of another type, *concrete field nodes*, to the PAG during propagation. Each such node is attributed with a single abstract object and the field it represents.
2.9. Concrete Implementations

```java
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 2.2: Source code fragment and corresponding PAG.

2.9.2 Paddle

Paddle is a context-sensitive, flow-insensitive, and field-sensitive points-to analysis that is, like Spark, based on the Soot framework. Its basic functionality is similar to Spark in the sense that it processes the same types of constraints, so we keep the presentation of Paddle here short. Paddle’s central feature is that constraints are solved by inference rules implemented in Binary Decision Diagrams (BDDs) [22, 23]. This allows for memory-efficient storing points-to sets which, in turn, allows for the use of expensive context-sensitive approaches as well as context-sensitive naming schemes for real-life programs.

2.9.3 Points-to SSA-based Simulated Execution

Points-to SSA is a sparse, graph-based program representation. On top of it, Simulated Execution is performed. These two concepts are presented in the following, after a short introduction to SSA and Memory SSA (on which Points-to SSA is based) and a presentation on how the abstract memory is handled.

2.9.3.1 SSA and Memory SSA

Static Single Assignment Form (in short SSA) is an intermediate program representation first developed by
Cytron et al. [27]. Every variable has exactly one static assignment. In order to represent multiple assignments in the original program representation, a new version of that variable is created during SSA construction. To decide what version of a variable is valid after confluences in the control flow, \( \varphi \) functions are introduced: \( \varphi \) functions are pseudo-operations that take the possible versions of a variable as arguments and choose, depending on control flow, which of the operands is the currently valid value. SSA provides many benefits for program analysis; for instance, use-def relations become explicit.

Memory SSA [90] 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-accessing operations must be executed for a given program.

### 2.9.3.2 Heap Modeling in P2SSA

Points-to SSA-based Simulated Execution is field-sensitive: 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 P2SSA, a single global abstract memory is used. An abstract memory is not only used 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. The abstract memory can be seen as a mapping from object fields to points-to values. The memory is, therefore, equipped with two operations

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

with the interpretation of reading the points-to value stored in an object field \( [o, f] \in OF \) and then 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 previously stored object field values in memory store operations are never overridden, i.e., no strong updates are performed. Instead, the new value is merged with the old one using the points-to value lattice’s join operation.

The abstract memory is updated as a side effect of the analysis. In order to quickly determine the fixed point, memory sizes indicate 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, a memory size lattice $L_X$ is introduced. 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 $h_m$.

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

Figure 2.3 shows the simple “Linked List” implementation that has already been used as an example above, when describing Spark’s PAG, but this time with the corresponding Points-to SSA graphs. Each method is represented by a graph and each node in the graph represents an operation in the method. For example, Entry and Exit nodes represent method entry/exit points, and Store and Load nodes represent 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 $\mathbf{a}$ as a special of values), while 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).

Note that the constructor $\text{L.<init>}$ starts by calling its super constructor $\text{Object.<init>}$ and that object creation, in $\text{L.append}$, is done in two steps: first, an object of class $\text{L}$ is allocated, and then the constructor $\text{L.<init>}$ is called. $\varphi$ nodes are used in $\text{L.append}$ to merge the memory size values from the two selective branches and, in $\text{L.putAt}$, 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 especially designed for points-to analysis. It is an abstraction since all operations not directly related to reference computations are removed, 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 all 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
2.9. Concrete Implementations

They also imply a correct order in which the memory-accessing operations are analyzed, which ensures that the analysis is an intra-procedural flow-sensitive abstraction of the semantics of the program.

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

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 \( (n) = [\text{in}_1(n), ..., \text{in}_k(n)] \), and a number of out-ports \( \text{out}(n) = [\text{out}_1(n), ..., \text{out}_l(n)] \). 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 \( (V \text{ or } X) \) and a current analysis value \( (v \in L_V \text{ or } x \in L_X) \). 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 \( e = \text{out}_i(\text{src}) \rightarrow \text{in}_j(\text{tgt}) \) connects an out-port of a node \( \text{src} \) with an in-port of a node \( \text{tgt} \). 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 \( \text{AllocC} \) 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 \( \text{Storef} \) node, which abstracts the actual semantics of a field write statement \( a.f = v \). For each abstract object \( o \) in the address reference \( a \), the points-to value previously stored in object field \( [o, f] \) is looked up. If the new value to be stored changes the memory (i.e., if \( v \not\subseteq \text{prev} \)), \( v \) is merged with the previous value and the result is saved. Notice also that a new memory out-port value (a new memory size) is computed if the memory has been changed during this operation.

The transfer functions of all node types currently in use in Points-to SSA can be found in Appendix A of [31].

2.9.3.4 Context-Sensitive Simulated Execution

The dataflow analysis technique called Simulated Execution is an abstract interpretation of the...
2.9. Concrete Implementations

they also imply a correct order in which the memory-accessing operations are analyzed, which ensures that the analysis is an intra-procedural flow-sensitive abstraction of the semantics of the program.

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

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 $e = \text{out}_i(\text{src}) \rightarrow \text{in}_j(\text{tgt})$ connects an out-port of a node $\text{src}$ with an in-port of a node $\text{tgt}$. 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 the address reference $a$, the points-to value previously stored in object field $[o, f]$ is looked up. If the new value to be stored changes the memory (i.e., if $v \not\subseteq \text{prev}$), $v$ is merged with the previous value and the result is saved. Notice also that a new memory out-port value (a new memory size) is computed if the memory has been changed during this operation.

The transfer functions of all node types currently in use in Points-to SSA can be found in Appendix A of [31].

2.9.3.4 Context-Sensitive Simulated Execution

The dataflow analysis technique called Simulated Execution is an abstract interpretation of the
Chapter 2. Points-to Analysis

Algorithm 1 \( \text{Store}^f : [x_{in}, a, v] \mapsto x_{out} \)

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

\[
\text{for each } o \in pt(a) \text{ do}
\]
\[
\text{prev} = \text{Mem.get}([o, f])
\]
\[
\text{if } v \nsubseteq \text{prev} \text{ then}
\]
\[
\text{Mem.addTo}([o, f], v)
\]
\[
x_{out} = \text{Mem.getSize}()
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{return } x_{out}
\]

program based on the abstract analysis and program representation discussed above. It simulates the actual execution of a program where the analysis of a method is interrupted when a call occurs and is 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 \( a_2.x \) which is executed after \( a_1.x \) in all runs of a program.

2.9.3.5 Method Graph Processing For each method graph, there is a pre-computed node order that is determined by the data and memory dependencies between the nodes. To get this order, a topological sorting for forward edges is computed. To order the nodes in loops, a so-called loop tree analysis [90] is used, where inner and outer loops and their loop heads (always \( \varphi \) nodes) are identified.

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 are taken into consideration by processMethod as given in Algorithm 2. The idea is simple: First, the method entry node is initialized with the method input to be used in this particular method activation. Then the method nodes are repeatedly analyzed until the method exit node is reached. Therefore, a node’s transfer function given by the node type is computed, the successor in-ports are updated, and the next node is determined until the method’s analysis values stabilize. The transition from one method to another is embedded in the statement n.computeTransferFunction() if n is of a monomorphic call type (MCall\textsuperscript{m,cs\textsubscript{i}}). Note that the processing of a call in turn may lead to the analysis of the call target method m, as defined in processMethod.

### 2.9.3.6 Call Processing

The approach to analyzing individual calls (see Algorithm 3) describes the handling of a call to method m in a context ctx\textsuperscript{m}. For the understanding of call processing, it is safe to assume that all calls to m are associated with only one context ctx\textsuperscript{m}, i.e., that a context-insensitive analysis is performed. 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\textsuperscript{m} is associated with two attributes, prev\_args and prev\_return, where previous input and return values of the calls to m in that context ctx\textsuperscript{m} are stored. The former of these attributes is used to decide whether a more general call targeting m in the same context ctx\textsuperscript{m} has been seen before, i.e., if [x\textsubscript{in}, a, v\textsubscript{1}, \ldots, v\textsubscript{n}] ⋐ prev\_args, in which case the previous result from prev\_return are reused. The alternative, a call targeting m in ctx\textsuperscript{m} with new arguments, leads to a new method activation where the target method m is processed by invoking processMethod using the merged input prev\_args ⊔ [x\textsubscript{in}, a, v\textsubscript{1}, \ldots, v\textsubscript{n}]. The two attributes prev\_args and prev\_return are updated in preparation for the next call targeting m in ctx\textsuperscript{m}.

Termination of the analysis is ensured since method arguments

---

**Algorithm 2** processMethod : (m, [x\textsubscript{in}, a, v\textsubscript{1}, \ldots, v\textsubscript{n}]) ↦ [x\textsubscript{out}, r]

\[
n = m.\text{entryNode}
\]

\[
in(n) = [x\textsubscript{in}, a, v\textsubscript{1}, \ldots, v\textsubscript{n}]
\]

\[\text{do}\]

\[n.\text{computeTransferFunction()}
\]

\[n.\text{updateSuccs()}
\]

\[n = n.\text{next()}
\]

\[\text{while } n \neq m.\text{exitNode}
\]

\[\text{return } in(n)
\]
Chapter 2. Points-to Analysis

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

- - if ctx^m was already analyzed with larger parameters before
  if \([x_{in}, a, v_1, \ldots, v_n] \subseteq ctx^m.prev\_args\) 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 \texttt{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]
prev \_ args \sqcup [x_m, a, v_1, \ldots, v_n] \text{ are incrementally merged before the processing of a method } m \text{ is started. Thus, the sequence of arguments } args_i \text{ used for a given context } ctx^m \text{ forms an ascending chain satisfying }

\text{args}_0 \sqsubseteq \text{args}_1 \sqsubseteq \ldots \sqsubseteq \text{args}_n.

Each such chain must have finite length since the value lattices have finite heights (both } L_X \text{ and } L_V \text{ 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; termination 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, a few more attributes associated with each context are needed: \text{is\_active} \text{ is used to check if processing a call in a context that is currently being analyzed, i.e., if } m \text{ is called recursively in } ctx^m. \text{ In this case, } prev\_return \text{ is directly returned for the recursive call and undefined [0, ⊥] if there are no previous results, respectively. Also, } is\_recursive = \text{true} \text{ is set, which indicates, upon return from } processMethod, \text{ that a recursive call during } processMethod \text{ has been seen. In this case, the results need to be stabilized by iteratively re-invoking } processMethod \text{ until the fixed point is reached.}

2.9.3.7 Transfer Function of Context-Sensitive Calls The transfer function of monomorphic call nodes MCall^m.cs_i \text{ 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). \text{ The algorithm first selects a set of contexts for each call as described in Section 2.5.1: selectContextsFor: } [m, cs_i, a] \mapsto [ctx_1, \ldots, ctx_q]. \text{ 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 2.5. Note that a call can be associated with more than one context, in particular for object sensitivity.}

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

Algorithm 4 lists the transfer function for MCall nodes. It computes all contexts } ctxs \text{ and merges the analysis results of the individual calls to the target method } m \text{ in each context } ctx^m \text{ as returned from } processCall.

2.9.3.8 Path Sensitivity in Points-to SSA Points-to SSA supports a novel approach to path sensitivity for points-to analysis (first presented in [34]) that is based on the observation that dataflow analyses quite often
2.10 Conclusion

In this chapter, we have discussed different variation points of points-to analysis, such as flow sensitivity and context sensitivity. While such features yield, in theory, more accurate results, their real values have to be shown experimentally in praxis. We have also presented three different points-to analysis implementations: Spark, Paddle, and P2SSA. These differ in their supported sets of variation points, as well as their underlying algorithms. In particular, P2SSA uses a very different approach (Simulated Execution, which induces flow-sensitivity) compared to Spark and Paddle, whereas Paddle mainly aims at memory efficiency. Additionally, we have pointed out that there are open problems for points-to analysis, such as reflection and native methods, which cannot always be dealt with in a sound manner.

In summary, there is a necessity of experimental evaluations of points-to analysis, e.g., to assess the effects on accuracy and analysis costs of features such as flow sensitivity and context sensitivity, as well as the effects of supporting (or not supporting) language features such as reflection and native methods. We shall investigate how such experiments are conducted by different researchers in Section 3.4 of the next chapter.

Algorithm 4 $M\text{Call}^{m,cs_i} : [x_{in}, a, v_1, \ldots, v_n] \mapsto [x_{out}, r]$

| Context[] | $ctxs = \text{selectContextsFor}(m, cs_i, a)$ |
| [x_{out}, r] = [0, \perp] |
| for each $ctx^m \in ctxs$ do |
| this = $ctx^m$.getThis() |
| args = $[x_{in}, \text{this}, v_1, \ldots, v_n]$ |
| $[x_{out}, r] = \text{processCall}(ctx^m, args) \sqcup [x_{out}, r]$ |
| end for |
| return $[x_{out}, r]$ |

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 relationship 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 $a$. Thus, an assignment to variable $a$ can be logically introduced where the points-to set of $a$ is restricted at the beginning of the following branch, as in the following example:

```java
if (a instanceof T) {
    // then-block
    a = \text{filter\_type\_restrict}(a, T);
} else {
    // else-block
    a = \text{filter\_type\_exclude}(a, T);
}
```

In [34], five such filter operations are described: For the equality operator (inequality is the same, just with reversed then/else blocks), `instanceof` as in the examples above, comparisons with class literals (expressions such as `x.getClass() == T.class`), calls to `equals()`, and comparison with null-literals. P2SSA currently supports `instanceof` filters.


2.10 Conclusion

In this chapter, we have discussed different variation points of points-to analysis, such as flow sensitivity and context sensitivity. While such features yield, in theory, more accurate results, their real values have to be shown experimentally in praxis. We have also presented three different points-to analysis implementations: Spark, Paddle, and P2SSA. These differ in their supported sets of variation points, as well as their underlying algorithms. In particular, P2SSA uses a very different approach (Simulated Execution, which induces flow-sensitivity) compared to Spark and Paddle, whereas Paddle mainly aims at memory efficiency. Additionally, we have pointed out that there are open problems for points-to analysis, such as reflection and native methods, which cannot always be dealt with in a sound manner.

In summary, there is a necessity of experimental evaluations of points-to analysis, e.g., to assess the effects on accuracy and analysis costs of features such as flow sensitivity and context sensitivity, as well as the effects of supporting (or not supporting) language features such as reflection and native methods. We shall investigate how such experiments are conducted by different researchers in Section 3.4 of the next chapter.
Chapter 3

Related Work

In this chapter, we present work related to different aspects of this thesis. In Section 3.1, we discuss work that compares dynamic with static analysis results. In this thesis, we discuss such comparisons for points-to analysis from a theoretical perspective and also perform such comparisons experimentally. In Section 3.2, we present the use of Gold Standards in computer science. We will use Gold Standards as a theoretical construct in Chapter 4 and work towards the creation of a Gold Standard for points-to analysis in Chapter 6 of this thesis. In Section 3.3, we present work that deals with speeding up dynamic analysis based on static analysis results. We will present our own approach for fast dynamic points-to analysis in Section 6.4. Finally, in Section 3.4, we will summarize and compare the different methods for evaluating points-to analysis in literature. It will become clear that those methods are roughly comparable at best, so this section serves as additional motivation for the present thesis. We conclude this chapter in Section 3.5.

3.1 Comparing Dynamic and Static Analyses

In this section, we look at related work where static analysis results are compared to dynamic analysis results.

A number of researchers have dealt with investigating the accuracy (or rather inaccuracy) of points-to analysis, or static analysis in general. Mock et al. [62] 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 inaccurate, 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 inaccurate static algorithm; the results of current points-to analysis approaches are likely to be better.

Ribeiro and Cintra [70] investigate how accurate 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...
3.1. Comparing Dynamic and Static Analyses

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. [48] investigate the accuracy due to different naming schemes for Java: how well 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 that measure accuracy at call sites. The first study investigates how accurate a naming schema could actually be; here, an inaccuracy occurs when a call site is called on two different runtime object instances that 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”\(^1\). The authors find that creation site naming schemes are very accurate 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 site into relation to the number of concrete runtime objects – mapped to the same naming scheme as used in the points-to analysis – that reach this call site in dynamic runs of the analysis. If this value is close to 1, then they can conclude that no other points-to analysis algorithm could be significantly more accurate, as a value of 1 would be an exact solution to the problem. The authors conclude that the creation site naming scheme is accurate enough in many cases, but more accurate algorithms that can model complex runtime data structures must be developed in other cases.

The same authors also have worked towards creating the best possible input for programs to increase dynamic coverage [47]. Unfortunately, it does not seem that they have made their efforts publicly available.

Lhoták presents a tool for finding differences in call graphs [42]. 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 also 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 2.2 – e.g., dynamic class loading – are avoided.

ProBe [4] 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. To-

\(^1\)Note that, in literature, “precise” is often used as a synonym for “accurate.” We will use the term “accurate” for now except in citations, and will provide definitions of both precision and accuracy in Chapter 4.
3.3 Fast Dynamic Analysis with help of Static Analysis

The results of using the micro-benchmarks were consistent with the researchers’ previous experiences with their own tools. Sim et al. also observe that the development of benchmarks in computer science disciplines is often accompanied with technical progress and community building [82]. The lack of such benchmarks, in turn, makes it difficult to further develop a field by adopting the successful and avoiding the less promising approaches.

3.3 Fast Dynamic Analysis with help of Static Analysis

We now look at research dealing with reducing performance overhead of dynamic analysis based on static analysis results. We will present our own approach to fast dynamic points-to analysis in Section 6.4.

Yong and Horwitz [95] reduce the overhead of a runtime type checker for C programs by using static analysis. Their approach is to find code points where instrumentation can be removed, not how storing observed information can be made more efficient. Despite reducing the overhead by 40%, their instrumented code still runs 23 times slower than the original code.

Ostrand et al. use static analysis to find the places in a program that are most likely to produce errors, and on which then dynamic testing efforts should be focused [66]. Their idea is where to put instrumentation code in the first place, not how to decrease the performance overhead caused by the instrumentation code.

Different approaches for efficiently collecting dynamic data have been presented in literature. For example, Binder et al. [15] collect profiling information (e.g., number of invocations and execution times for each method) using so-called method call trees. Their approach shows slowdown factors of less than 10 for a number of benchmark programs, with an average slowdown factor of less than 5. Pothier et al. [68] collect complete execution traces, including field writes, method invocations, method return values, etc. They report a performance overhead of 10 and 100 for a realistic and a worst-case scenario, respectively.

3.4 Evaluation Methods for Points-to Analysis

In this section, we discuss the evaluation methods of points-to analysis for Java, taking work published between 2002 and 2013 into consideration. We focus on the following aspects:

- Benchmark programs, including their versions and where they stem together with the specification, the dynamic and static results for a number of benchmark programs are published.

### 3.2 Gold Standards

A Gold Standard is, in general, a data set with annotations where the annotations are considered “correct.” However, since Gold Standards can, in general, not be computed automatically, they rely on human annotations and are thus often only a “best effort” and not perfect. Such imperfection can be dealt with, for example, by accepting a (hopefully very) low error rate or by continuously improving on it. We shall see examples for the former in linguistics and machine learning, whereas we will present a process for the latter in Chapter 6.

Gold Standards can be found in several computer science areas:

- Machine learning has a high demand for highly qualitative annotated data. Many data sets exist for different areas; a repository that currently contains 235 such sets is available at [http://archive.ics.uci.edu/ml/](http://archive.ics.uci.edu/ml/).

- In linguistics, Gold Standards are used to benchmark, for instance, language parsers. In this area, there are even competitions arranged². Two examples for so-called tree banks (corresponding to Gold Standards) used in linguistics are the PENN treebank [55] and the TIGER treebank [18], both containing annotated newspaper text. The process for creating treebanks is usually like this: two or more human experts independently annotate, with tool support, the same text sentence by sentence. The results of the experts are then compared, and discrepancies are discussed among the annotators.

- Pettersson et al. published an initial attempt towards a Gold Standard for design patterns [67]. In their future work, the authors point out that the Gold Standard needs to evolve through a moderated process by the research community.

- The only attempt known to us to create a Gold Standard related to points-to analysis is performed by Rountev et al. [73]. They compare static and dynamic analysis results for call-chains in Java and, where those analysis results differ, manually attempt to create proper input or provide a proof that the call chain is not feasible. The authors also applied the four-eyes principle.

- Sim et al. [83] present a benchmark for C++ fact extractors. The benchmark consists of a set of micro-benchmarks where each micro-benchmark poses a different challenge to fact-extractors. They confirmed the usefulness of their benchmark by inviting researchers to apply their own fact extractors to the suite and report on the results at a workshop. It showed that the

²Information on one out of many can be found here: [http://www.cs.toronto.edu/acl08parsinggerman/sharedtask.html](http://www.cs.toronto.edu/acl08parsinggerman/sharedtask.html)
results of using the micro-benchmarks were consistent with the researchers’ previous experiences with their own tools.

Sim et al. also observe that the development of benchmarks in computer science disciplines is often accompanied with technical progress and community building [82]. The lack of such benchmarks, in turn, makes it difficult to further develop a field by adopting the successful and avoiding the less promising approaches.

3.3 Fast Dynamic Analysis with help of Static Analysis

We now look at research dealing with reducing performance overhead of dynamic analysis based on static analysis results. We will present our own approach to fast dynamic points-to analysis in Section 6.4.

Yong and Horwitz [95] reduce the overhead of a runtime type checker for C programs by using static analysis. Their approach is to find code points where instrumentation can be removed, not how storing observed information can be made more efficient. Despite reducing the overhead by 40%, their instrumented code still runs 23 times slower than the original code.

Ostrand et al. use static analysis to find the places in a program that are most likely to produce errors, and on which then dynamic testing efforts should be focused [66]. Their idea is where to put instrumentation code in the first place, not how to decrease the performance overhead caused by the instrumentation code.

Different approaches for efficiently collecting dynamic data have been presented in literature. For example, Binder et al. [15] collect profiling information (e.g., number of invocations and execution times for each method) using so-called method call trees. Their approach shows slowdown factors of less than 10 for a number of benchmark programs, with an average slowdown factor of less than 5. Pothier et al. [68] collect complete execution traces, including field writes, method invocations, method return values, etc. They report a performance overhead of 10 and 100 for a realistic and a worst-case scenario, respectively.

3.4 Evaluation Methods for Points-to Analysis

In this section, we discuss the evaluation methods of points-to analysis for Java, taking work published between 2002 and 2013 into consideration. We focus on the following aspects:

- Benchmark programs, including their versions and where they stem from
Chapter 3. Related Work

- Version of the Java classes that they have been analyzed with
- Support for native methods
- Support for dynamic class loading and reflection
- Whether or not results were verified by comparing to results obtained from dynamic analysis
- Used client analyses for measuring accuracy
- What the authors compare their approach with
- Other aspects of interest, e.g., unclear aspects of evaluation methods and comparison to other papers.

Whaley and Lam, SAS 2002 [91]: They use all the benchmarks from the SPEC JVM98 benchmark suite, version 1.03\textsuperscript{3}[6], components taken from the J2EE 1.3 reference implementation, “joeq,”\textsuperscript{4} their own virtual machine on which their implementation is based, the database server “Cloudscape” shipped with J2EE, and text editor “jEdit”. Note that, for the latter three, no versions are given. They use Java 1.3.1\_01 for analysis; note that Berndl et al. (see below) state that Whaley and Lam “[...] made optimistic, potentially unsafe, assumptions about what part of the library needs to be analyzed”[12]. Whaley and Lam claim to handle native calls and reflection properly, but do not explain any details. They do not mention verifying their results by comparing them to dynamic analysis. For evaluation, they use “average number of targets per call site”. Note that their experimental setup does not explicitly specify if only virtual calls or even static or “special” (e.g., constructor and super) calls are included in their client computation. They compare their approach against a rapid type analysis [11].

Lhoták and Hendren, CC 2003 [43]: They use all the benchmarks from the SPEC JVM98 benchmark suite, sablecc and soot from the Ashes Suite Collection[1], as well as jEdit. Ashes has been unchanged since 2000, so the two benchmark programs taken from there have a fixed version, but, again, no version is provided for jEdit. They use Java 1.3.1\_01 for analysis, and additionally analyze javac from SPEC JVM98 with the Java 1.1.8 classes “for comparison with other studies”. An interesting aspect of this is that the number of potentially reachable methods, obtained with class hierarchy analysis (CHA), increases from 4602 to 16307 by switching the Java standard library classes. For handling native methods, they have created stubs and included them in the Soot[5] framework. They claim (and to our knowledge

\textsuperscript{3}Note: The changes in the final 1.04 version affect the benchmark harness only, so we omit version number of the SPEC JVM98 benchmark suite in the following.

\textsuperscript{4}http://joeq.sourceforge.net/
3.4. Evaluation Methods for Points-to Analysis

this is the case) that all native methods in the Java 1.3.1 standard library are supported. The authors compiled by hand a list of possible reflective method invocations. They do not mention verifying their results by comparing them to dynamic analysis. The used clients are “dereferencing field accesses”, i.e., how many potential abstract objects can reach a given field access (results are given grouped: 0 possible targets meaning “not reachable”, 1 possible target, 2, 3 to 10, 11 to 100, 101 to 1000, 1001 or more possible targets) as well as possible target methods of virtual calls (invokevirtual, invokespecial). Again, the results given were grouped (0, 1, 2, 3 or more). They list their results once including, once excluding Java standard library classes. The authors compare different instantiations of their own framework with each other.

Berndl et al., PLDI 2003 [12]: They use some, but not all, of the programs from the SPEC JVM98 benchmark suite, the “SPECjbb 2000” benchmark [7], sablecc and soot from the Ashes suite collection, as well as jEdit (again no version given). Note that Lhoták and Hendren, the authors of [43], are co-authors of this paper. They use Java 1.3.1_01 for analysis. Native methods are supported as their analysis is implemented on top of Spark; reflection is not separately mentioned, but it is probably the same setup as in the previously discussed paper. They do not mention verifying their results by comparing them to dynamic analysis. No client analyses are used as the work focuses on performance and memory footprint. For this, they compare their BDD-based approach with Spark.

Whaley and Lam, PLDI 2004 [92]: They use “21 of the most popular Java projects on Sourceforge as of November 2003”. It does not become clear from their description what Java standard library classes they use. Some native methods and reflective calls are handled explicitly (which ones is not specified further), while others are treated as returning unknown objects. They do not mention verifying their results by comparing them to dynamic analysis. A large part of the paper deals with concrete applications for points-to analysis; they use two of the so-called queries for their evaluation: Thread escape analysis and type refinement. They compare different instantiations of their framework with each other. Note: Many of the programs they analyze make heavy use of GUI, leading to program entry points through the underlying operating system. The authors have not claimed to handle those, though.

Milanova and Rountev, TSE 2005 [61]: They use all the programs from the SPEC JVM98 benchmark suite, “other benchmarks used in previous work on analysis for Java”, and programs from www.jars.com. For most, yet not all, non-SPECJVM programs, version numbers are given. They use Java 1.1.8 for analysis. Native methods are handled through stubs, and dynamic class loading and reflective calls are resolved manually. They do not mention verifying their results by comparing them to dynamic analysis. The used client analyses for evaluation are side-effect analysis, reachable methods, and virtual call resolution. They compare different instantiations of their framework
Chapter 3. Related Work

with each other.

Liang et al., PASTE 2005 [47]: They use twelve different benchmark programs, but give neither where they obtained them from nor version numbers. Nor do they state which Java standard library classes they use for analysis. Handling of native methods is not mentioned, but reflective calls are handled through user-provided input. They compare their results with results obtained through dynamic analysis. As client analysis, they use “reachable methods”. They compare different instantiations of their framework with each other. The authors do not count library classes in their evaluation.

Sridharan et al., OOPSLA 2005 [87]: They use most of the programs from the SPEC JVM98 benchmark suite, soot and sablecc from the Ashes Suite Collection, and jEdit (no version given). They use Java 1.3.1 (no subversion specified) for analysis. Native methods are supported as their analysis is based on Spark, which comes with Soot 2.2.1. Otherwise unsupported methods, as well as unsupported reflection, are handled through telling the client analysis that a given query cannot be answered, and thus optimizations should not be performed at this program point. They do not mention verifying their results by comparing them to dynamic analysis. As client analysis, they use “virtual call resolution”. They compare different instantiations of their framework with each other. The authors report different numbers of reachable methods through CHA compared to [43] for the same input programs. They explain it with “improvements in the handling of reflection in Soot 2.2.1”.

Sridharan and Bodík, PLDI 2006 [86]: They use most of the programs from the SPEC JVM98 benchmark suite, soot and sablecc from the Ashes Suite Collection, polyglot [65], and several programs from the DaCapo benchmark suite, version beta050224 [2]. Note that they name the same benchmark programs differently than in their own previous work presented above. They use Java 1.3.1 (no subversion specified) for analysis. Native methods are supported as their analysis is based on Spark that comes with Soot 2.2.1. Otherwise unsupported methods, as well as unsupported reflection, are handled through telling the client analysis that a given query cannot be answered, and thus optimizations should not be performed at this program point. They do not mention verifying their results by comparing them to dynamic analysis. As client analysis, they use “virtual call resolution”. They compare different instantiations of their framework with each other. The authors report different numbers of reachable methods compared to [43] for the same input programs. They explain it with “improvements in the handling of reflection in Soot 2.2.1”.

Milanova, PASTE 2007 [58]: The author uses soot and sablecc from the Ashes benchmark collection, polyglot, and several benchmarks from the DaCapo benchmark suite, version beta051009. The author also states that earlier work on points-to analysis has used programs from earlier versions of
the same benchmark suite. The author uses Java 1.4.1 (no subversion specified) for analysis. Native methods are supported through Soot, although Soot seemingly does not support native methods of Java classes newer than 1.3.1, for example java.lang.reflect.Array.newInstance(). Targets of reflective calls have been specified manually. They do not mention verifying their results by comparing them to dynamic analysis. As client analyses, the author uses “polymorphic call resolution” as well as call graph edge (though names both differently). The presented object-oriented approach is compared with a context-insensitive baseline approach.

Gutzmann et al., SCAM 2007 [34]: The authors use some of the benchmarks from the DaCapo benchmark suite, but do not specify from which version, as well as sablecc and four other projects. They give the versions only for some of them. They do not state which Java standard library classes they use for analysis. They say that they have “stubs for only few native methods”, but do not specify which, and do not state if reflection is supported. The authors also say that no exception handling is supported in the experimental implementation of their analysis. They do not mention verifying their results by comparing them to dynamic analysis. As client analyses, the authors use “object call graph edges”, with object call graphs being an extension on object-level of call graphs with edges \([o_i, f] \rightarrow [o_k, m_l]\), “heap” meaning relations of the form \([o_i, f] \leftarrow o_j\), “cast safety”, “dead code”, and “null pointer safety”. They restrict their client analyses to application entities, e.g., for object call graph edges, at least caller or callee must be an application method. Application entities are identified through their fully qualified names. Different instantiations of their framework are compared with each other.

Prabhu and Shankar, SAS 2008 [69]: The authors use some of the benchmarks from the SPEC JVM98 benchmark suite, as well as “some of the popular Java programs from SourceForge”. Version numbers are not given, and some of the programs appear to be neither part of the SPEC JVM98 benchmark suite nor sourceforge projects. They do not state which Java standard library classes they use for analysis. They neither state the support of their analysis for native methods and reflection nor mention verifying their results by comparing them to dynamic analysis. For evaluation, they use the size of points-to sets at load/store operations, escape analysis, as well as “caller-captured” objects, meaning objects that do not escape the caller context. Different instantiations of their framework are compared with each other.

Lhoták and Hendren, TSEM 2008 [45], an extended version of their CC 2006 paper [44]: The authors use all of the benchmark programs from the SPEC JVM98 benchmark suite, the JOlden benchmark suite [25]⁵, soot-c, sablecc, polyglot, as well as some of the benchmarks from the DaCapo

⁵A more recent version of this benchmark suite can be found at [3].
Chapter 3. Related Work

benchmark suite, version beta050224. They use Java 1.3.1_01 for analysis. The authors do not state their support for native methods. However, their implementation, Paddle, is based on the Soot framework and thus the same support as for Spark can be expected. The authors do not mention how they handle reflective calls, but assumedly they have at least configured dynamic class loading as in their previous work. They do not mention verifying their results by comparing them to dynamic analysis. For evaluation, they use a number of performance metrics like “contexts per method”, which serve the purpose of comparing the efficiency of different context-sensitive variations. As client analyses for measuring accuracy, they use “reachable methods” (with and without Java library methods), “Call edges originating from a Benchmark Method”, “polymorphic call resolution”, and “unsafe casts”. Different instantiations of their framework are compared with each other.

Bravenboer and Smaragdakis, ISSTA 2009 [19]: The authors use most of the benchmarks from the DaCapo benchmark suite, version 2006-10-MR2. They use Java 1.4.2_18 for analysis. In this paper, the authors have implemented their points-to analysis to mimic the behavior of Paddle down to support of native methods and reflection (except for exception handling), so the same support as in the paper presented previously can be expected. They do not mention verifying their results by comparing them to dynamic analysis. For evaluation, they use call graph construction, size of points-to sets on variables, and “Throw Points-to”, i.e., what exception can possibly be throws by each methods. Different instantiations of their framework (DOOP) are compared with each other, but verified their results by comparing them with Spark and Paddle.

Gutzmann et al., SCAM 2009 [32]: The authors use one of the benchmarks that is also part of the SPEC JVM98 benchmark suite, some of the benchmarks that are also part of the DaCapo benchmark suite, Soot, sablecc, and four other benchmark programs. They specify the versions of each benchmark program, which do not necessarily correspond to the versions in the benchmark suites. They use different versions of the Java standard library for analysis, depending on each benchmark program, and specify the version as either 1.6.0, 1.4.2, or 1.3.1. They qualitatively compare the results as obtained from static analysis with results obtained from dynamic analysis. For evaluation, they use call graph construction, object call graph construction, and heap. They limit their “metrics to those cases where the abstract objects and the members are declared and/or created in application code”. They compare results obtained by their own points-to analysis implementation with results obtained by Spark.

Lundberg et al., JIST 2009 [52], an extended version of their SCAM 2008 paper [53]: The authors use one of the programs from the SPEC JVM98 benchmark suite, some from the DaCapo benchmark suite (but do not specify which version), sablecc and soot-c from the Ashes Suite Collection, and
five other programs taken from the Internet, for which they specify the exact version. They use Java 1.4.2 for analysis. Support for native methods is incomplete, only array-manipulating methods are completely supported. Reflection is not handled correctly. They do not mention verifying their results by comparing them to dynamic analysis. For evaluation, they use “call graph”, “potentially polymorphic call sites”, “potentially failing casts”, “object call graph”, “heap”, and “enter” (“The number of abstract objects entering an application method”). They additionally perform escape analysis. They restrict their client analyses to application entities, e.g., for call graph edges, at least caller or callee must be an application method. Different instantiations of their framework are compared with each other.

Bravenboer and Smaragdakis, OOPSLA 2009 [20]: The authors use the same benchmark programs and Java version as in their paper presented above. They do not mention verifying their results by comparing them to dynamic analysis. For evaluation, they use “call graph nodes and edges”, “var points-to” and “field points-to” metrics. They compare different instantiations of their own framework, DOOP, with each other as well as with different instantiations of Paddle. However, besides their “Paddle-Compatibility” mode, they also describe how they handle native methods and reflection and classify “DOOP-supported analyses as much more precise and full-featured than previous declarative pointer analyses in the literature”. Regarding the work by Whaley and Lam, they write [91, 92] that it “lacks in support for many Java features, such as native code, reflection, finalization, etc.”

Smaragdakis et al., POPL 2011 [85]: The authors use the experimental setup as in their previous work (the paper presented above), except that they exclude the benchmark programs jython and hsqldb, which show performance problems. They do not mention verifying their results by comparing them to dynamic analysis. The authors use the client analyses “reachable methods”, “polymorphic call sites”, “casts that may fail”, “average var-points-to”, “call graph edges”, and – for some of those – results restricted to application methods.

Gutzmann et al., SAC 2011 [35]: The authors use two of the benchmarks that are also part of the SPEC JVM98 benchmark suite, bloat, which is also part of the DaCapo benchmark suite, sablecc, and two other benchmark programs. They specify the versions of each benchmark program, which do not necessarily correspond to the versions in the benchmark suites. They do not state which Java standard library classes they use for analysis. The benchmark programs they use are picked so that their analysis is (claimed) conservative. They compare their results with results obtained by dynamic analysis. The authors use “call graph construction,” “object call graph construction,” and “heap” for their evaluation. Different instantiations of their framework are compared with each other.
Xiao and Zhang, ISSTA 2011 [93]: The authors use soot and sablecc from the Ashes Suite Collection, some of the benchmarks from the DaCapo benchmark suite, version beta20050224, as well as other programs taken from other sources, without giving version information. They use Java 1.3.1_20 for analysis. Native methods are supported as in Soot 2.4.0, on which their implementation is built upon, but it is not mentioned how they handle reflection. They do not mention verifying their results by comparing them to dynamic analysis. As client analyses, they use “virtual call resolution” and “alias analysis”. They compare their results with Spark as well as an instantiation of Paddle.

Edvinsson et al., HiPEAC 2011 [29]: The authors use javac from the SPEC JVM98 benchmark suite, two from the DaCapo benchmark suite, soot-c from the Ashes Suite Collection, and four other programs taken from the Internet, for which they specify the exact version. Note that the authors also specify the versions of the programs taken from benchmark suites, which do not correspond to the versions found there. The authors do not mention what Java version they have used for analysis, what support for native methods their analysis provides, or how they handle reflection. They also do not use any accuracy metrics, as their work is solely focused on parallelization of points-to analysis for performance reasons.

Shang et al., CGO 2012 [79]: The authors use some of the programs from the DaCapo benchmark suite, but it does not become clear what version; based on the selection of benchmark programs, it must be from the 2009 release branch. They also used “jack” and “javac” from the SPEC JVM98 benchmark suite. They use Java 1.6.0_16 for analysis. Native methods and reflection are supported through the use of Tamiflex [16]. They do not mention verifying their results by comparing them to dynamic analysis. As client analyses, they use “safe casts”, “null dereferences”, and “factory methods”. However, they do not give any accuracy results as the focus of their paper is performance (on-demand analysis).

Gutzmann et al., SCAM 2012 [36]: The authors use two of the benchmarks that are also part of the SPEC JVM98 benchmark suite, some of which are also part of the DaCapo benchmark suite, sablecc, and two other benchmark programs. They specify the versions of each benchmark program, which do not necessarily correspond to the versions in the benchmark suites. They use Java 1.6.0_22 for analysis. The authors state that they have used only benchmark programs that make use of native methods that their analysis supports, and that they specified the results of reflective calls manually. The authors say that they ran dynamic analysis and compared the results qualitatively, with the outcome that the dynamic analysis results are a subset of the static analysis results, for validation purposes. The authors use “call graph construction,” “object call graph construction,” and “heap” for their evaluation. They restrict their client analyses to application entities, e.g., for
3.4. Evaluation Methods for Points-to Analysis

call graph edges, both caller or callee must be an application method. Different instantiations of their framework are compared with each other, and they also perform experiments with Paddle and Spark.

Lundberg and Löwe, JUCS 2013 [54]: The authors use one of the programs from the SPEC JVM98 benchmark suite, some from the DaCapo benchmark suite (but do not specify which version), sablecc and soot-c from the Ashes Suite Collection, and four other programs taken from the Internet, for which they do not specify the version. They use Java 1.4.2 for analysis. Support for native methods is incomplete, only array-manipulating methods are completely supported. reflection is not handled correctly. They do not mention verifying their results by comparing them to dynamic analysis. For evaluation, they use “call graph”, “object call graph”, “heap”, and “enter” (“The number of abstract objects entering an application method”). They also introduce a more fine-grained context graph, which can be roughly described as a call graph on the context level. They restrict their client analyses to application entities. Different instantiations of their framework are compared with each other.

3.4.1 Discussion

Table 3.1 summarizes which authors have used which benchmark programs for their evaluation. During recent years, the programs from the SPEC JVM98 benchmark suite, as well as formerly often-used programs “soot” and “sablecc,” seem to have fallen a little out of fashion, whereas programs from the DaCapo benchmark suite have been used much more frequently. Some authors use neither but use their own set of benchmark programs instead, while others use a mix of both. Quite often program versions are not provided, making comparison of experimental results difficult. Even when it comes to programs taken from benchmark suites, the versions of the individual programs depend on the version of the benchmark suites. Additionally, many different versions of Java standard library classes are used, which also effectively makes results incomparable. This becomes clear from the experimental results by Lhoták and Hendren: switching from Java 1.1.8 to 1.3.1_01 increased the number of potentially reachable methods from 4602 to 16307 for a benchmark program that is otherwise the same. Also noteworthy is that authors are often not consistent in their selections of benchmark programs from experiment to experiment.

Table 3.2 summarizes the client analyses that are used in different papers (note that those papers that do not have any client analysis listed focus on performance instead of accuracy). Frequently used client analyses are “reachable methods” and “call graph construction” in some form. Virtual call resolution and cast safety are somewhat less often used. Escape analysis is used in two papers, whereas all other client analyses are not used by more
<table>
<thead>
<tr>
<th>Page</th>
<th>Year</th>
<th>Author(s)</th>
<th>Specification</th>
<th>Subject</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>2002</td>
<td>Whaley, Lam</td>
<td>JDK 1.4.2</td>
<td>SPEC JVM98, JEdit, JEdit</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2003</td>
<td>Lhoták, Hendren</td>
<td>X</td>
<td>Dereference sites</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2003</td>
<td>Berndl et al.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2004</td>
<td>Whaley, Lam</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2005</td>
<td>Milanova, Rountev</td>
<td>X</td>
<td>Side-effect analysis</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2005</td>
<td>Liang et al.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2005</td>
<td>Sridharan et al.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2006</td>
<td>Sridharan, Bodík</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2007</td>
<td>Milanova</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2007</td>
<td>Gutzmann et al.</td>
<td>X</td>
<td>Object call graph edges, Heap, null dereferences, dead code</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2008</td>
<td>Prabhu, Shankar</td>
<td>X</td>
<td>Escape analysis, caller-captured objects, points-to sets at load/store</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2008</td>
<td>Lhoták, Hendren</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2009</td>
<td>Bravenboer, Smaragdakis</td>
<td>X</td>
<td>Exception analysis, points-to sets of variables</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2009</td>
<td>Gutzmann et al.</td>
<td>X</td>
<td>Object call graph, Heap</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2009</td>
<td>Lundberg et al.</td>
<td>X</td>
<td>Object call graph, heap, enter, escape, analysis</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2009</td>
<td>Bravenboer, Smaragdakis</td>
<td>X</td>
<td>Points-to sets of variables</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2011</td>
<td>Smaragdakis et al.</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2011</td>
<td>Gutzmann et al.</td>
<td>X</td>
<td>Object call graph, Heap</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2011</td>
<td>Xiao, Zhang</td>
<td>X</td>
<td>Alias analysis</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2011</td>
<td>Edvinsson et al.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2012</td>
<td>Shang et al.</td>
<td>X</td>
<td>Null dereferences, factory methods</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2012</td>
<td>Gutzmann et al.</td>
<td>X</td>
<td>Object call graph, Heap</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>2013</td>
<td>Lundberg, Löwe</td>
<td>X</td>
<td>Object call graph, Heap, Context-graph</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Summary of Used Client Analysis: RM = Reachable Methods, CGE = Call Graph Edges, Virtual Call Resolution = Virtual Call Resolution, SC = Safe Casts
### Evaluation Methods for Points-to Analysis

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Year</th>
<th>RM</th>
<th>CGE</th>
<th>VCR</th>
<th>SC</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whaley, Lam [91]</td>
<td>2002</td>
<td>(x)</td>
<td></td>
<td></td>
<td></td>
<td>Derereference sites</td>
</tr>
<tr>
<td>Lhoták, Hendren [43]</td>
<td>2003</td>
<td>(x)</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Berndl et al. [12]</td>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Whaley, Lam [92]</td>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Milanova, Routnev [61]</td>
<td>2005</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>Side-effect analysis</td>
</tr>
<tr>
<td>Liang et al. [47]</td>
<td>2005</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sridharan et al. [87]</td>
<td>2005</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sridharan, Bodík [86]</td>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Milanova [58]</td>
<td>2007</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Gutzmann et al. [34]</td>
<td>2007</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>object call graph edges, Heap, null dereferences, dead code</td>
</tr>
<tr>
<td>Prabhu, Shankar [69]</td>
<td>2008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lhoták, Hendren [45]</td>
<td>2008</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>exception analysis, points-to sets at load/store</td>
</tr>
<tr>
<td>Bravenboer, Smaragdakis [19]</td>
<td>2009</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>exception analysis, points-to sets of variables</td>
</tr>
<tr>
<td>Gutzmann et al. [32]</td>
<td>2009</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>Object call graph, Heap</td>
</tr>
<tr>
<td>Lundberg et al. [52]</td>
<td>2009</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>object call graph, heap, enter, escape analysis</td>
</tr>
<tr>
<td>Bravenboer, Smaragdakis [20]</td>
<td>2009</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>points-to sets of variables</td>
</tr>
<tr>
<td>Smaragdakis et al. [85]</td>
<td>2011</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>points-to sets of variables</td>
</tr>
<tr>
<td>Gutzmann et al. [35]</td>
<td>2011</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>Object call graph, Heap</td>
</tr>
<tr>
<td>Xiao, Zhang [93]</td>
<td>2011</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>Alias analysis</td>
</tr>
<tr>
<td>Edvinsson et al. [29]</td>
<td>2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shang et al. [79]</td>
<td>2012</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>null dereferences, factory methods</td>
</tr>
<tr>
<td>Gutzmann et al. [36]</td>
<td>2012</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>Object call graph, Heap</td>
</tr>
<tr>
<td>Lundberg, Löwe [54]</td>
<td>2013</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>Object call graph, Heap, Context-graph</td>
</tr>
</tbody>
</table>

Table 3.2: Summary of Used Client Analysis: RM = Reachable Methods, CGE = Call Graph Edges, Virtual Call Resolution = Virtual Call Resolution, SC = Safe Casts
Chapter 3. Related Work

than one research group.

An interesting observation is that some authors use “safe casts”, while others use “potentially failing casts”, which is effectively the same, just “the other way around”. We will come back to what we believe is the better definition in Section 5.4. Similar is the case of “polymorphic calls” vs. “monomorphic calls”.

Table 3.3 summarizes in which papers static analysis results are compared to results obtained from dynamic analysis, in which papers third-party implementations are used for comparison in evaluations, and which authors restrict their client analyses to application entities.

Static points-to analysis results are seldom verified by comparing them to results obtained from dynamic analysis. However, we will show in the next chapter that, if non-conservative baseline analyses are used, no experimental evaluation is theoretically valid, and we will present experimental proof for this in Chapter 8. We, therefore, think that validation of static analysis results through comparison with dynamic analysis results must become common practice; however, we readily admit that this is not something that we have always done ourselves in the past.

Comparison between different points-to analysis implementations is also not often performed; for example, Bravenboer and Smaragdakis have put a lot of effort to bring two fundamentally different analyses (their own and Paddle) to the exact same level of language support.

Most authors do not exclude library code when calculating client analysis results. However, we will argue in Section 5.1 in favor of use of application entities.

3.5 Conclusion

In this chapter, we have discussed work related to different aspects of the present thesis. There has been previous work on comparing dynamic with static analysis results with the intention to assess the accuracy of static dataflow analysis, or to find source of inaccuracy of static analysis. However, very little work has been done in the field of dataflow analysis to actually create Gold Standards, which are rather common in linguistics and machine learning.

Points-to analysis itself has been subject to many experimental evaluations. However, it is noteworthy that there is no commonly accepted method to assess its accuracy; in particular, there is no benchmark suite dedicated to points-to analysis, which makes it hard to compare experimental results of different researcher papers. In particular, we found the following obstacles for comparing points-to analysis results from different research papers:

1. Different client analyses as well as definitions and interpretations thereof
### Table 3.3: Summary of Evaluation Methods

<table>
<thead>
<tr>
<th>Year</th>
<th>Comparison with</th>
<th>Dynamic Analysis</th>
<th>Comparison with third-party P2A implementations</th>
<th>Use of third-party P2A application entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Whaley, Lam [91]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Lhoták, Hendren [43]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Berndt et al. [12]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Whaley, Lam [92]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>Milanova, Romoutev [61]</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Liang et al. [47]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>Sridharan, Bodik [86]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>Milanova [58]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>Gutzmann et al. [34]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>Prabh, Shankar [69]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>Lhoták, Hendren [45]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>Bravenboer, Smaragdakis [19]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Gutzmann et al. [32]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Bravenboer, Smaragdakis [20]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>Lundberg et al. [52]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>Liang et al. [58]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>Gutzmann et al. [35]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>Xiao, Zhang [93]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>Gutzmann et al. [36]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>Edvinsson et al. [29]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>Shang et al. [79]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>Smaragdakis et al. [85]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>Gutzmann et al. [36]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4. Comparing "May" Dataflow Analyses

In this chapter, we discuss the theory behind comparing results from different approaches to the same "may"-dataflow problem. Recall from Section 2.1 that a "may"-dataflow analysis is one that answers questions of the form "what fact may be true?" or "what may happen at runtime (but does not have to)?"

When we talk about the result of an analysis $A$ here, we mean a set of facts that are possibly true about a given benchmark program. The usual mathematical set-operations, like union $\cup$ and intersection $\cap$, as well as comparators like subset $\subset$, can be applied to such analysis results, e.g., $A_1 \cap A_2$ yields the set of facts that are computed by both $A_1$ and $A_2$. An example in the context of points-to analysis would be the intersection of two points-to values $\text{pt}(a_1) \cap \text{pt}(a_2)$ at a given variable or expression $a$.

As a concretion of the general description of a Gold Standard as given in Section 3.2, we define the term Gold Standard in the context of "may"-dataflow analysis.

Definition

For a given program $P$ and a "may"-dataflow analysis, the Gold Standard $G$ is defined as: for each fact $f \in G$, there exists at least one concrete execution of $P$ such that $f$ is observed.

Note that, in this chapter, we assume the Gold Standard to be correct.

As mentioned in Section 3.2, Gold Standards are, in practice, often only a "best effort" and may be inaccurate, e.g., due to human errors.

The elements in the result space of an analysis $A$ applied to a program $P$ can now be categorized with respect to the Gold Standard $G$ as follows:

- **true positives:** $\text{tp}(A) = \{f | f \in A \land f \in G\}$
- **true negatives:** $\text{tn}(A) = \{f | f \notin A \land f \in G\}$
- **false positives:** $\text{fp}(A) = \{f | f \in A \land f \notin G\}$
- **false negatives:** $\text{fn}(A) = \{f | f \notin A \land f \in G\}$

Based on the Gold Standard, we also define three kinds of "may"-dataflow analysis that we will investigate in this chapter.

Definitions:

We have pointed out cases where researchers are not consistent even within their own research groups when it comes to evaluation methodology. We would like to note that this does not question the validity of any individual paper. However, comparability amongst different approaches to points-to analysis is made difficult. In Chapter 5, we will propose a guideline for benchmarking points-to analysis that improves on this matter.
Chapter 4
Comparing “May” Dataflow Analyses

In this chapter, we discuss the theory behind comparing results from different approaches to the same “may”-dataflow problem. Recall from Section 2.1 that a “may”-dataflow analysis is one that answers questions of the form “what fact may be true?” or “what may happen at runtime (but does not have to)?”.

When we talk about the result of an analysis \( A \) here, we mean a set of facts that are possibly true about a given benchmark program. The usual mathematical set-operations, like union \( \cup \) and intersection \( \cap \), as well as comparators like subset \( \subseteq \), can be applied to such analysis results, e.g., \( A_1 \cap A_2 \) yields the set of facts that are computed by both \( A_1 \) and \( A_2 \). An example in the context of points-to analysis would be the intersection of two points-to values \( pt(a_1) \cap pt(a_1) \) at a given variable or expression \( a \).

As a concretion of the general description of a Gold Standard as given in Section 3.2, we define the term Gold Standard in the context of “may”-dataflow analysis.

Definition For a given program \( P \) and a “may”-dataflow analysis, the Gold Standard \( G \) is defined as: for each fact \( f \in G \), there exists at least one concrete execution of \( P \) such that \( f \) is observed.

Note that, in this chapter, we assume the Gold Standard to be correct. As mentioned in Section 3.2, Gold Standards are, in practice, often only a “best effort” and may be inaccurate, e.g., due to human errors.

The elements in the result space of an analysis \( A \) applied to a program \( P \) can now be categorized with respect to the Gold Standard \( G \) as follows:

true positives: \( tp(A) = \{ f | f \in A \land f \in G \} \)

true negatives: \( tn(A) = \{ f | f \notin A \land f \notin G \} \)

false positives: \( fp(A) = \{ f | f \in A \land f \notin G \} \)

false negatives: \( fn(A) = \{ f | f \notin A \land f \in G \} \)

Based on the Gold Standard, we also define three kinds of “may”-dataflow analysis that we will investigate in this chapter. Definitions:
Chapter 4. Comparing “May” Dataflow Analyses

Figure 4.1: Analysis result types with respect to the Gold Standard (G): conservative (Cons), general (Gen), and dynamic (Dyn).

general A general “may”-dataflow analysis applied to a program $P$ computes any result in the analysis’ result space.

conservative A conservative “may”-dataflow analysis applied to a program $P$ computes a superset of the Gold Standard for $P$, i.e., $fn(A) = \emptyset$.

optimistic An optimistic “may”-dataflow analysis applied to a program $P$ computes a subset of the Gold Standard for $P$, i.e., $fp(A) = \emptyset$.

Conservative and optimistic analysis are special cases of general analysis. Remember from Section 2.1.2 that conservative analysis is usually obtained by (sound) static analysis, and optimistic by dynamic analysis, i.e., monitoring concrete program executions. General analysis is usually obtained by performing unsound static analysis, e.g., by not supporting dynamic class loading in Java. Note that an implementation of a static analysis can be conservative for some programs, e.g., those that do not make use of dynamic class loading, and general (in the wider sense) for others, e.g., those that do make use of dynamic class loading. The three result types are illustrated in Figure 4.1.

In the remainder of this chapter, we first discuss how comparing the results obtained by two analyses can be done with and without a Gold Standard at hand (Section 4.1). Then, in Section 4.2, we show that experimental results of an improvement of a given dataflow analysis based on a general baseline analysis must be treated carefully. We conclude our findings from this chapter in Section 4.3.

4.1 Comparing Analyses

In this section, we describe how two analyses for the same dataflow problem can be compared with each other.

To compare two analyses, we have to evaluate the garbage (false positives) and misses (false negatives) that they contain, and then trade off garbage
4.1. Comparing Analyses

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 accuracy as measured by the so-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 and non-trivial input programs, automated analysis algorithms can only approximate the exact analysis result. In special cases, the approximations may be conservative, $G \subseteq A$, or optimistic, $A \subseteq G$. Consequently, $R = 1$ holds 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

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

Remember from above 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 and it holds, in general, $P_{\text{general}} \leq 1$ and $R_{\text{general}} \leq 1$.

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 impractical if no such Gold Standards exists, like for points-to analysis.
Chapter 4. Comparing “May” Dataflow Analyses

We examine if and how two analyses for the same dataflow problem can be compared with each other (Section 4.1.1), and then take a look at how accuracy of points-to analysis can be approximated in the absence of a Gold Standard in Section 4.1.2.

4.1.1 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.

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. If either of the two analyses is 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 $P_{\text{cons}} = \frac{|G|}{|A_1|},$ 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, $P = 1.$
4.1. Comparing Analyses

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

<table>
<thead>
<tr>
<th>$k$</th>
<th>Special case of Analyses $A_1, A_2$</th>
<th>(a) $P$</th>
<th>(b) $R$</th>
<th>(c) $F$-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>$A_1, A_2$ conservative</td>
<td>$P_1 \geq P_2 \iff</td>
<td>A_1</td>
<td>\leq</td>
</tr>
<tr>
<td>(2)</td>
<td>$A_1, A_2$ optimistic</td>
<td>$P_1 = P_2 = 1$</td>
<td>$R_1 \geq R_2 \iff</td>
<td>A_1</td>
</tr>
<tr>
<td>(3)</td>
<td>$A_1$ conservative, $A_2$ optimistic</td>
<td>$P_1 \leq P_2 = 1$</td>
<td>$R_1 = 1 \geq R_2$</td>
<td>$F_1 \geq F_2 \iff \frac{</td>
</tr>
<tr>
<td>(4)</td>
<td>$A_1$ conservative, $A_2$ general</td>
<td>$P_1 \geq P_2 \iff</td>
<td>A_1</td>
<td>\leq</td>
</tr>
<tr>
<td>(5)</td>
<td>$A_1$ optimistic, $A_2$ general</td>
<td>$P_1 = 1 \geq P_2$</td>
<td>$R_1 \geq R_2 \iff</td>
<td>A_1</td>
</tr>
</tbody>
</table>

Table 4.1: Comparing conservative, optimistic, and general analysis. $A_{1,2}$ denote the respective result sets of analysis 1 and 2, and $G$ denotes the Gold Standard. $P, R$ and $F$ denote precision, recall, and harmonic F-score.
Chapter 4. Comparing “May” Dataflow Analyses

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

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

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

$F_1 \geq F_2 \iff \frac{2 \times P_1 \times R_1}{P_1 + R_1} \geq \frac{2 \times P_2 \times R_2}{P_2 + R_2} \iff 2 \times R_1 \geq 2 \times R_2 \\
\iff 2 \times R_1 \times (1 + R_2) \geq 2 \times R_2 \times (1 + R_1) \\
\iff 2 \times R_1 \times R_2 + 2 \times R_1 \geq 2 \times R_1 \times R_2 + 2 \times R_2 \\
\iff 2 \times R_1 \geq 2 \times R_2 \iff R_1 \geq R_2 \iff (\text{proof 2a}) |A_1| \geq |A_2|$. □

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 $R_1 = 1, P_2 = 1, P_1 = \frac{|G|}{|A|}, R_2 = \frac{|A_2|}{|G|}$, we get

$F_1 \geq F_2 \iff \frac{2 \times P_1 \times R_1}{P_1 + R_1} \geq \frac{2 \times P_2 \times R_2}{P_2 + R_2} \iff 2 \times \frac{P_1}{P_1 + 1} \geq \frac{R_2}{1 + R_2} \\
\iff 2 \times P_1 \times (1 + R_2) \geq R_2 \times (P_1 + 1) \\
\iff P_1 + P_1 \times R_2 \geq P_1 \times R_2 + R_2 \iff P_1 \geq R_2 \iff \frac{|G|}{|A|} \geq \frac{|A_2|}{|G|}$. □

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

$|A_1| \leq |A_2| \iff \frac{|G|}{P_1} \leq \frac{|A_2 \cap G|}{P_2} \\
\iff \frac{|G|}{|A_2 \cap G|} \times P_2 \leq P_1 \Rightarrow \frac{|G|}{|A_2 \cap G|} \geq 1, P_2 \leq P_1$. □

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

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

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

Proof 5b. $R_1 \geq R_2 \iff \frac{|A_1 \cap G|}{|G|} \geq \frac{|A_2 \cap G|}{|G|} \iff |A_1| \geq |A_2 \cap G|$. 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 $|A_1| \geq |A_2| \Rightarrow (\text{proof 5b}) R_1 \geq R_2$ and $P_1 = 1 \geq P_2$ are both precision and recall equal to or better for $A_1$ than for $A_2$, so $F_1 \geq F_2$.

4.1.2 Approximating Accuracy

Approximating the accuracy of an analysis 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 approximations can be calculated either from a conservative analysis leading to an over-approximation, or from an optimistic one leading to an under-approximation. Define $G^+ \supseteq G$ as 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 the 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 $\hat{F} \geq F \geq \tilde{F}$. 

53
Chapter 4. Comparing “May” Dataflow Analyses

$G^+ \subseteq A$ 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: $F^- = \hat{F}, F^+ = \hat{F}$.

Note that this has no effect on the comparability of conservative analyses as they can be compared 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.

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

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

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

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

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

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 $F^- = \hat{F}, F^+ = \hat{F}$.

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 counterexamples from extreme cases:

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

$$R = \tilde{R} = 1 \quad (G \subset A \text{ implies that } A \text{ is conservative})$$ and

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

54
4.1. Comparing Analyses

Case (ii): \( G^- = A \cap G \Rightarrow F < \hat{F}, \) since
\[
R = \frac{|A \cap G|}{|G|} < \frac{|A \cap G^-|}{|G^-|} = \hat{R} \quad \text{and} \quad P = \frac{|A \cap G|}{|A|} = \frac{|A \cap G^-|}{|A|} = \hat{P},
\]

Case (iii): \( G^+ \supset G \supset A \Rightarrow F > \hat{F}, \) since
\[
R = \frac{|A \cap G|}{|G|} < \frac{|A \cap G^+|}{|G^+|} \quad \text{and} \quad P = 1 \quad (G \supset A \implies \text{A is optimistic})
\]

Case (iv): \( G^+ = A \cup G, A \cap G = \emptyset \Rightarrow F < \hat{F}, \) since
\[
P = 0 < \hat{P} = \frac{|A \cap G^+|}{|A|} \quad \text{and} \quad R = 0 < \hat{R} = \frac{|A \cap G^+|}{|G^+|}.
\]

Cases (i) and (ii) deal with using an under-approximation of the Gold Standard: In case (i), the analysis is actually conservative (a special case of general analysis) and thus the actual precision (and therefore F-Score) is better than the approximated precision (F-Score). In case (ii), the analysis result is the under-approximation of the Gold Standard with additional garbage. Due to the misses of the analysis with respect to the Gold Standard, the actual recall (and thus the F-Score) is worse than the approximated recall (F-Score). However, it is unknown which of the two cases holds by comparing \( A \) with \( G^- \), so it is not known how the approximated F-Score relates to the actual F-Score.

Similar results hold for cases (iii) and (iv), which deal with using an over-approximation of the Gold Standard: In case (iii), the analysis is actually optimistic (again a special case of general analysis) and thus the actual recall (and therefore F-Score) is better than the approximated recall (F-Score). In case (iv), the analysis result is disjoint from the Gold Standard (but not its approximation) and thus the actual precision, recall, and F-Score are 0, whereas the approximated ones are not. Again, it is not known which of the two cases holds by simply comparing \( A \) with \( G^+ \); so, again, it is not known how the approximated F-Score relates to the actual F-Score.

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 Improving General Analyses

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 context sensitivity. A conservative, improved analysis $A^i_{cons}$ is always smaller and, hence, more accurate than its corresponding baseline analysis $A^b_{cons}$, i.e., it is more precise as $P_i \geq P_b$ and has the same recall $R^i_{cons} = R^b_{cons} = 1$.

This does not continue to hold for general analyses, as it might miss some actual results. A general improved analysis $A^i$ is still smaller but not necessarily more accurate than its baseline analysis $A^b$. As shown with two extreme cases, the $F$-scores can theoretically both improve (case (i)) and deteriorate (case (ii)).

Case (i):

$A^i = A^b \cap G$ \Rightarrow

\[ P^i = \frac{|A^i \cap G|}{|A^i|} = \frac{|A^b \cap G|}{|A^b \cap G|} \geq \frac{|A^b \cap G|}{|A^b|} = P^b \]

\[ R^i = \frac{|A^i \cap G|}{|G|} = \frac{|A^b \cap G|}{|G|} = R^b. \]

Case (ii):

$A^i = A^b \setminus G$ \Rightarrow

\[ P^i = \frac{|A^i \cap G|}{|A^i|} = \frac{0}{|A^b \setminus G|} \leq \frac{|A^b \cap G|}{|A^b|} = P^b \]

\[ R^i = \frac{|A^i \cap G|}{|G|} = \frac{0}{|G|} \leq \frac{|A^b \cap G|}{|G|} = R^b. \]

In case (i), the subset of the initial analysis result that is not part of the Gold Standard is optimized away, whereas in case (ii), exactly the subset of the initial analysis result that is also part of the Gold Standard is optimized away.

This observation has consequences for all researchers evaluating any improvement to dataflow analysis by comparing its result $A^i$ with the baseline analysis $A^b$. If the baseline analysis is non-conservative, e.g., due to some unexpected dynamic class loading or unsupported native methods, we cannot in general deduce that the improved analysis is more accurate, although the measurements say that $A^b > A^i$. 

Chapter 4. Comparing “May” Dataflow Analyses

56
4.3 Conclusion

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

```java
4.3. Conclusion

1: class C {
2:   void foo(C[] fooArr) { fooArr[0].bar(); }
3:   void bar(C c) { . . . }
4:   static void bar(C c1, C c2, C[] a1, C[] a2, int n) {
5:     System.arraycopy(a1, 0, a2, 0, n); // native method
6:     c1.foo(a1);
7:     c2.foo(a2);
8:   }
9: }

From a points-to analysis perspective, System.arraycopy() copies all elements from a1 to a2 in this example. Assume also that the following (conservative) results are computed for a given call to bar():
- pt(c1) = \{o1\}, pt(c2) = \{o2\}, pt(a1) = \{arr1\}, pt(arr1) = \{o3\}, pt(a2) = \{arr2\}, pt(arr2) = \{}.

It is obvious that a correct analysis should compute calls between foo() and bar() where pt(foo.this) = \{o2\} and pt(bar.this) = \{o3\}, as o3 is copied from a1 to a2. If System.arraycopy() is not supported by the analysis, a context-insensitive analysis will still compute this, as the invocation of foo() is reduced to the one and only context (pt(this) = \{o1, o2\}, pt(fooArr) = \{arr1, arr2\}). However, a context-sensitive analysis may analyze foo() under two contexts:
- (pt(this) = \{o1\}, pt(fooArr) = \{arr1\}) and (pt(this) = \{o2\}, pt(fooArr) = \{arr2\}), and when analyzing the latter context, pt(arr2) = \{} thus leading to a true negative. In this example, context-insensitivity therefore cascades that System.arraycopy() is unsupported. We will see in Section 8.3 that this happens in practice.

4.3 Conclusion

We can compare analyses with respect to accuracy only in special cases, i.e., when we compare either two conservative analyses with each other or two optimistic analyses with each other. For all other kinds of comparison, we need a Gold Standard, which currently does not exist for many dataflow analyses, e.g., for points-to analysis.

We have also shown that evaluating improvements to a given baseline analysis must be done carefully: if the baseline analysis is not conservative in the first place, then it is not clear if the improvement does more harm than good, and results are theoretically of no use.

However, as we have seen in the previous chapter, improvements to points-to analysis are often evaluated on top of general analysis, or analysis that has to be assumed to be general because experimental setups are often unclear (e.g., handling of native methods and dynamic class loading is often not mentioned). As, further, conservative analysis is very difficult to provide, es-
Chapter 4. Comparing “May” Dataflow Analyses

especially for system initialization code, most static points-to analyses are likely to be general, which would mean that assessing the accuracy of improvements to points-to analysis is, theoretically, not possible without a Gold Standard. In the next chapter, we will, therefore, present a benchmarking methodology that works around this limitation by restricting accuracy assessments to parts of programs that can be analyzed conservatively. In order to assess the exact of accuracy of points-to analysis, and to be able to assess tradeoffs between precision and recall, we shall then present a process to create a Gold Standard in Chapter 6.
Chapter 5

Benchmarking Points-to Analysis

In this chapter, we will describe how to create a benchmark suite for static points-to analysis.

Benchmarking means to run a specific set of tasks under controlled circumstances and measure certain properties, which can then be compared to other benchmarking results. Historically, execution time is typically measured, but we focus on the property accuracy in this thesis. We define the benchmarking task as running points-to analysis on a given benchmark program and computing client analyses on them. A client analysis is a concrete application of points-to analysis results, e.g., call graph construction or virtual call resolution. The measured properties are metrics calculated from the client analyses. A benchmark suite defines these controlled circumstances and a set of programs that are to be run or, in our case, analyzed.

Part of creating a benchmark suite is to define how to interpret experimental static points-to analysis results. Intuitively, this simply requires one to measure the same accuracy metrics for the same analyzed programs. There are, however, pitfalls in the details, and the methodology described in this chapter attempts to work around them.

As we have seen in Section 3.4, there is no common method for experimentally evaluating accuracy of static points-to analysis in literature. Even the same authors tend to not keep to a fixed evaluation method from study to study. However, a standardized, commonly accepted and used benchmark suite has the obvious advantage of direct comparability, i.e., pure numbers from papers can be directly compared without having to re-run other researchers’ experiments.

In Section 3.5, we identified the following differences in evaluation methods that make comparison of experimental results difficult:

1. Different client analyses as well as definitions and interpretations thereof
2. Different benchmark programs or different versions of the same program
3. Different supported subsets of programming language aspects, e.g., reflection and native methods
4. Whether or not library code is used in the calculation of client analyses
Additionally, comparing general analyses experimentally without a Gold Standard is, from a theoretical perspective, not possible, as seen in Chapter 4.

We propose a benchmarking methodology that tackles these problems. Disposing of the first two differences is a matter of discipline, requiring a commonly accepted benchmark suite (containing a set of well-defined client analyses and a fixed set of benchmark programs) that is used by the research community. Such a benchmark suite must evolve from a dialogue in the research community in order to be commonly accepted. In this chapter, we show how such a benchmark suite can be created: We define a set of proper client analyses that can be used for such a benchmark suite, and attempt to keep the definitions unambiguous. We also discuss some issues on how to properly define what composes a benchmark program, something that has not always been done properly in literature.

The third difference, handling different subsets of programming languages, goes hand-in-hand with the problem of experimentally evaluating general analyses. We tackle this by defining subsets of programs that can be analyzed conservatively by many points-to analysis implementations.

The remainder of this chapter is organized as follows: We first present an approach to defining sub-parts of programs that can be analyzed conservatively. For this, we argue for the use of application entities in Section 5.1 to define the sub-parts for programs and discuss how to determine if the sub-parts can be analyzed conservatively in Section 5.2. We strongly suggest to validate that an analysis is indeed conservative for those sub-parts of a benchmark program by comparing static analysis results with dynamic analysis results; in Section 5.3, we suggest how to create input for good dynamic analysis results with relatively little effort. We then present client analyses that can be used for evaluating points-to analysis in Section 5.4. In Section 5.5, we discuss some aspects of the selection of benchmark programs for a benchmark suite. Finally, in Section 5.6, we conclude this chapter.

5.1 Application Entities

In this section, we argue for the use of application entities for the client analyses presented in the following section. Roughly speaking, an application entity of a benchmark program is “part of application code” as opposed to “part of a library code”.

The main benefits of using application entities is that native methods and dynamic class loading do not have to be fully supported for system initialization by a points-to analysis implementation, but only for those methods (in)directly called from application code (cf. Section 5.2). However, there are some more benefits of using application entities: (1) The results of system initialization and common libraries are not counted over and over again; (2)
5.1. Application Entities

stubs can be used for certain library classes, which mimic a correct behavior with respect to points-to analysis yet are more easily and more precisely analyzable; points-to information within those stubs are not correct, though (confer [46, 36] and Section 6.3.1); (3) writing a dynamic monitoring tool that monitors everything beginning with system initialization is non-trivial; (4) for the practical reason of reducing the amount of manual work when it comes to work towards a Gold Standard, cf. Chapter 6.

An entity is an application entity if its fully qualified name starts with a prefix identified by a name filter. More concretely, the following rules for identifying application entities are applied:

• A type (class, interface, enum) is an application type if and only if its fully qualified name starts with one of the name filters.

• A method is an application method if and only if it is directly declared in an application type.

• A field is an application field if and only if it is directly declared in an application type.

• An object creation site is an application object creation site if and only if the runtime type of one of the objects created at this site is potentially an application type.

Note: If object creation is done with a call to Class.newInstance() (or other similar reflective calls), then this object is to be regarded as an application type as it could be an application abstract object. This is what the “potentially” is for in the definition of application object creation site. Note also that methods inherited by an application type from a non-application type are not application methods.

We demand that, if a type is an application type, then all its subtypes also have to be application types. This ensures that a (non-static) application method is always called on an application abstract object. If we had a class $T$ that is an application class and that declares an application method $m$, and a class $U$ extends $T$ that is not an application class and that does not override $m$, then a call to application method $m$ of an abstract object of non-application type $U$ could occur. We want to eliminate this special case.

Note that it is perfectly fine to define a name filter as an empty string, which effectively makes all entities to application entities, so our arguing for application entities does not lead to loss of generality but is merely based on practical advantages.

---

1. Static instrumentation would be required for not missing a lot of system initialization code; see [15] for how this could be done.

2. There may be cases where application entities must be defined by more than one application filter. In those cases, an entity must match (at least) one in order to be considered an application entity.
5.2 Determining Conservative Analysis

Conservative points-to analysis hardly exists, mainly due to dynamic class loading and native methods. However, when evaluating points-to analysis with client analyses restricted to application entities, then the client analyses are conservative with respect to a given benchmark program if the following holds:

1. The basic analysis algorithm is conservative.

2. Reachable native methods do not have impact on the points-to sets relevant to client analyses.

The first condition is obvious and not further discussed.

In order to show that the second condition holds, it first has to be determined which native methods are reachable, and from where. These native methods can be divided into two sets: Methods reachable through system initialization, which are the same for each analyzed benchmark program, and methods reachable (transitively) through the benchmark program’s main()-method. Each reachable native method must then be investigated for potential effects on application client analyses, i.e., each such method must be investigated manually for potential impact on relevant points-to sets. Note that this has to be done for every Java version that is used for any evaluation.

In order to quantify the amount of methods to look at, we have modified P2SSA to collect all reachable yet unsupported native methods, and then run system initialization for Java 1.6.0_38. The analysis found 90 such methods. By means of looking at the documentation or, in cases where no documentation is present, the name, signature and return type, we decided whether or not we believe the method must be supported in order to allow for conservative analysis with respect to application entities.

It showed that many of the methods are related to character sets, localization, file system, and security, i.e., initializing the Java Sandbox. The latter is usually used only by applets in web browsers or if the application itself defines a security manager. Therefore, it sounds reasonable to assume that security manager related methods do not have side effects for stand-alone programs, at least not during system initialization. Character sets and localization-related methods are concerned with strings and it is, therefore, also safe to assume that they do not have any relevant effects. Similar goes for file system operations, e.g., data read from disks (or networks, for that matter).

We have looked at a few other, randomly selected methods and found no evidence that they have any impact on points-to sets of application code.

\(^3\text{cf. } \text{http://docs.oracle.com/javase/tutorial/essential/environment/security.html}\)
However, from prior experience we also know that the two methods `System.setOut0()` and `System.setErr0()` must be supported: They initialize the fields `System.out` and `System.err`, and calls from application code to `System.out.println(Object)` are deemed unreachable by points-to-analysis if these fields are not initialized. These calls can then trigger callbacks into application code by calling the argument’s `toString()` method.

The above investigation must also be done for the main()-method of each analyzed benchmark program. The amount of work here depends on each used benchmark program; our experience with the benchmark programs used for our evaluation in Chapter 8 is that most calls to native methods reachable here are either reflective calls, which pose a general problem, harmless methods as above for analyzing system initialization, or one of two array-handling related methods (`System.arraycopy()` and `Arrays.newInstance()`), for which stubs must be provided.

### 5.3 Dynamic Analysis

Comparing static analysis results with dynamic analysis results serves two purposes: To validate that an analysis is indeed conservative, and to approximate $P^-$ of static analysis, as discussed in Section 4.1.2.

The better the input to dynamic analysis is with respect to coverage, the better is the approximation of $P^-$ and, more importantly, the greater is the confidence in the soundness of an analysis. Bad dynamic coverage may, for instance, not uncover that a program is not analyzed conservatively, as the part that is causing problems is not at all triggered when dynamic analysis is run.

In literature, programs from performance benchmark suites are often used for evaluating points-to-analysis. However, performance benchmark suites have bad dynamic coverage [21]. It is understandable that performance benchmark suites do not aim at good coverage but instead use common use cases, but this hampers the usefulness of the provided input to benchmark programs for using in evaluating of points-to-analysis. Therefore, researchers in points-to-analysis must put effort into increasing the dynamic coverage of their benchmark programs in order to get more meaningful approximations of $P^-$. Programs may come along with numerous examples, e.g., parser generators with a set of example grammars. For an initial good coverage, all of these should be run. In cases where no such examples are provided, diverse input must be created. Additionally, invalid input (for triggering exception handling) and different command line options should be provided to the program. We will present the effect of such measures in Section 8.2.

We present our implementation of a dynamic analysis tool in Appendix A.
5.4 Client Analyses and Metrics

We now present a set of client analyses that can be used for benchmarking points-to analysis.

Our selection of client analyses covers many of the client analyses used by researchers as listed in Section 3.4. In some cases, practically the same client analysis was performed with “reversed” ideas, e.g., some authors used the client analysis safe casts while others used possibly failing casts. We define all client analyses so that static analysis is always a “can-possibly-happen” (or “cannot disprove”) analysis, whereas dynamic analysis is always a “there-is-at-least-one-case-where-this-happens” analysis. With other words, a more precise static analysis finds a smaller set of relations in a client analysis, and a more precise dynamic analysis finds more. This is important for dynamic analysis as a dynamic analysis can never prove that a cast cannot fail, but it can show that a cast can fail.

We decided to omit two types of client analysis that other researchers have used: (1) Sizes of points-to sets at member accesses (or dereferencing) level; this is because especially flow-insensitive analyses may merge multiple accesses to the same member in the same method. However, if there are multiple accesses in the same method, then the result must be counted several times, and this information may not be available after running a given points-to analysis without adapting the points-to analysis itself, which is undesirable according to our requirements. (2) Client analyses too difficult to implement in either dynamic or static analysis: For example, escape analysis is not easily put on top of another points-to analysis, and doing dynamic escape analysis would require a very complex dynamic analysis tool or must be performed within the virtual machine itself as it would require access to the garbage collector (whether or not an object can be garbage collected cannot be queried outside the virtual machine). However, we want dynamic analyses that are easy to use so that as many researchers as possible can contribute, and for that, independence from virtual machine implementations is required.

We attempt to achieve unambiguity in the following client analysis definitions. Unambiguity is not naturally given in computer science. For instance, Lincke et al. have shown that tools implementing the same well-known software metrics computed very different results for them [49].

In order to simplify the client analysis definitions, we first define how static and non-static members are treated uniformly:

**Definition** For a given program \( P \), let \( o_s \) be a special object creation site in \( P \). Static field accesses and static method invocations are treated as if they were accessed on an object created at this special object creation site.

In order to treat array and non-array objects uniformly, we treat each array as having an artificial content-field, similar to an array’s length-field in
Java. Array accesses \texttt{a[i]} are modeled as \texttt{a.content}.

These simplify the following client analysis definitions, as arrays and static members do not have to be treated explicitly.

**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** be a directed graph that consists of all the nodes \([o_i, m_j]\) so that (1) \(m_j\) is invoked on an instance object \([i_v]\), and (2) \(i_v\) was created at the object creation site \(o_i\). Further, the graph contains 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. The **exact application object call graph** is the exact object call graph with all those nodes (and edges from/to) removed where abstract objects and methods do not correspond to application entities.

Note that neither the exact object call graph nor the exact application object call graph are, in general, connected, as special methods such as finalizers and class initializers are called by the JVM, which means that there are no caller nodes (no in-edges) in the graph.

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

**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 application abstract heap** is the subset of the exact abstract heap restricted to abstract application objects stored into application fields.

**Definition** For a given program \(P\), let \(C\) be the set of all casts to non-primitive types in \(P\). Let then the **exact set of unsafe casts** consist of all the casts to non-array types \(c \in C\) where there exists a concrete execution of \(P\) so that \(c\) causes a \texttt{ClassCastException}. The **exact set of unsafe application casts** is the subset of the exact set of unsafe casts where the static type of casts are application types.

Casts to array types are omitted for two reasons: First, since points-to analysis implementations usually disregard primitive types, they may also disregard arrays of primitive types. Requiring casts to arrays of primitive types to be supported would require implementation changes for comparabil-
Chapter 5. Benchmarking Points-to Analysis

ity reasons. Second, points-to analyses may regard arrays as pure container objects and “flatten” multi-dimensional arrays to one-dimensional arrays for performance reasons.

Casts between primitive types cannot cause ClassCastException and are thus excluded in the above definition (we learned of this through a remark in [86]). Note also that explicit exception object instantiations like throw new ClassCastException(...), as in java.lang.Class.cast(...), are not included in this client analysis either.

**Definition** For a given program \( P \), let \( A \) be the set of all accesses to non-static members (methods and fields). Let then the **exact set of unsafe accesses** consist of all the accesses \( a \in A \) where there exists a concrete execution of \( P \) so that \( a \) causes a NullPointerException. The **exact set of unsafe application accesses** is the subset of the exact set of unsafe accesses that occur within an application method.

Note that an unsafe application access does not have to be an access to an application field or method.

**Definition** For a given program \( P \), let \( M \) be the set of all methods in \( P \) and \( E \) the set of all known exception types. Let then the **exact method-throws set** consists of all the pairs \( (m, e) \in M \times E \) where there exists a concrete execution of \( P \) so that \( m \) is exited by throwing an exception of type \( e \). The **exact application method-throws set** consists of the subset of the exact method-throws set where \( m \) is an application method, and \( e \) is an application type.

Note: We do not believe that it is important what exact exception instance is thrown by a given method and we therefore restrict this client to the types of exceptions. This is more likely to allow comparability to other, non-points-to analysis-based implementations of exception analysis, e.g. [24, 71].

**Definition** For a given program \( P \), let \( M \) be the set of all methods in \( P \) and \( C \) be the set of all non-static, non-constructor call sites in \( P \). Let then the **exact set of virtual call sites** consist of all the call sites \( c \in C \) where there exist two executions of \( P \) so that targets of \( c \) resolve to different \( m \in M \). The **exact set of application virtual call sites** consists of the subset of the exact set of virtual call sites where \( c \) occurs in an application method.

Note that the exact set of virtual call sites can be derived directly from the exact call graph. However, the exact set of application virtual call sites does not follow directly from the exact application call graph as a virtual call site may target non-application methods as well.

The above can be approximated by both dynamic and static analysis. We then speak of the client analyses **application object call graph**, **application call graph**, **application abstract heap**, **set of application unsafe casts**, **application method-throws set**, and **set of application virtual...**
5.5 Selecting Benchmark Programs

5.5 Selecting Benchmark Programs

A good benchmark program shows differences in at least some of the client analyses when analyzed with different points-to analyses that differ in precision. A “hello world” program likely does not do the job, but even small real-world programs may suffice. For instance, one of the benchmark programs that we will use for evaluation in the next chapter has only 48 used classes (including anonymous classes), yet the analysis results of different points-to analysis implementations differ. On the other hand, “larger” programs should be used as well in order to show performance scalability of used points-to analysis implementations. As to the definition of what “large” is, we can only guess.

We believe that such a benchmark suite has to evolve from agreement within the research community, and, until then, ad-hoc benchmark program selections will have to suffice in praxis. However, until such a benchmark
suite for points-to analysis exists, it is important to not only specify which benchmark programs were used, but also in what version. Otherwise, the results cannot be reproduced by third parties. In cases where benchmark programs make use of third-party libraries, this also applies to the versions of the libraries. This holds even if the library classes are not included in the name filters: First, they may change the dynamically observable program behavior, i.e., by changing API contracts, and second, even if the observable program behavior is not changed, they may still impact the results of static analysis.

The JDK is a somewhat special case as new versions aim at not breaking downward compatibility, i.e., changing observable program behavior. However, new features often lead to that static analysis deems a lot more library code reachable, cf. the comment by Lhoták and Hendren that switching from Java 1.1.8 to 1.3.1 increased the number of reachable methods from 4602 to 16307 in [43] for one of their benchmark programs.

5.6 Conclusion

In this chapter, we have presented a benchmarking methodology for points-to analysis.

As purely conservative analysis is usually very difficult to achieve, especially for analyzing system initialization, we argue for the use of application entities to which the evaluation of client analyses should be limited. This way, the evaluation (but not the actual analysis) is limited to a well-defined subset of an analyzed program, which, under certain testable circumstances (certain native methods reachable through the program are supported by the analysis), can be analyzed conservatively.

We present a set of client analyses to be used when evaluating points-to analysis. We have taken such client analyses used in literature that are rather easy to implement on top of static points-to analysis as well as with dynamic analysis.

We have also pointed out that it is important to document not only what program, but also what version of the program has been used for evaluation. This includes the versions of libraries (including the Java standard library) that have been used for analysis, as those can influence the analysis results even if not part of the application code.

We have also argued that static analysis results should always be compared to dynamic analysis results for validation purposes. Good dynamic coverage is important, and we have discussed a few measures regarding how to create good dynamic coverage.

In particular, the steps required to create a benchmark suite are: Select a number of client analyses and accompanying metrics, select a number of
benchmark programs and define their application name filters, define, for each program, the set of native methods required to analyze its resulting sub-parts conservatively, and provide dynamic analysis results for all benchmark programs and client analyses.

Note that we deliberately did not propose a definite benchmark suite in this chapter. A commonly accepted and used benchmark suite is required to allow direct comparability between different papers, and we believe that such acceptance requires a benchmark suite to evolve from agreement within the research community. Until then, ad-hoc benchmark suites will have to suffice in practice. Instead, we describe how to create and use such a benchmark suite, and we will also create such an ad-hoc benchmark suite for our evaluation in the next chapter.

In Chapter 8, we will present an experimental comparison of three points-to analysis implementations based on this benchmarking methodology.

Based on the guidelines presented in this chapter, we will present a benchmarking platform that supports applying these guidelines in Chapter 7. The platform can be used for creating a benchmark suite.
Chapter 6

A Shared Effort Towards a Gold Standard

In this chapter, we describe a process for a shared effort towards a Gold Standard and show means to work towards it. This chapter is organized as follows: In Section 6.1, we motivate the need for a Gold Standard. Then we describe the process to work towards it in Section 6.2. This process is based on iteratively improving over- and under-approximations of the Gold Standard by improving accuracy of static and dynamic analysis. Therefore, in Section 6.3, we describe new measures to improving the precision of static points-to analysis. Dynamic analysis, on the other hand, must be improved by running programs with more input. To reduce the time required for running dynamic analysis, we show an approach to reducing the runtime overhead of dynamic analysis in Section 6.4. We conclude this chapter in Section 6.5.

6.1 Motivation

Remember from Section 4.1 that a Gold Standard is required to compare the accuracy of two static may-dataflow analyses experimentally if at least one of them is not conservative, i.e., when evaluating general analyses. For points-to analysis, this is rather often the case, especially for new (research) implementations. Additionally, a Gold Standard can be used to assess the exact accuracy of points-to analysis, not only (under-)approximations. This is of general interest to researchers, as it helps identify sources of imprecision and thus points out where further improvements to points-to analysis is worth it.

Using conservative analysis is required to create the Gold Standard (otherwise, the effort to manually creating it would know no limit), so it is unlikely that there will ever be a Gold Standard for a benchmark program for which no conservative analysis exists. However, the step from conservative to general analysis may be deliberately made in order to either reduce the analysis costs, improve accuracy (tradeoff precision vs. recall), or both. However, in order to quantify accuracy properly, a Gold Standard is required after all. With the help of a Gold Standard, the tradeoff between precision and re-
call could be evaluated when going from a conservative to a general static analysis.

Finally, with the help of a Gold Standard, the usefulness of client analyses can be quantified. If a Gold Standard is not good enough to be useful for a given client analysis, then the client analysis itself may lack usefulness.

Remember from Section 3.2 that Gold Standards are often “best efforts” that have not been proven correct and would thus have to be defined as $G'$. However, for simplicity, we assume $G = G'$ in this chapter.

### 6.2 Process

In this section, we describe our proposed process for creating a Gold Standard for points-to analysis. The basic idea for creating a Gold Standard is to bring under- and over-approximations (through optimistic dynamic analysis and conservative static analysis) closer and closer together until they meet. The process is defined as follows:

1. Select a benchmark program and a client analysis for which a Gold Standard is to be created. Also name the application entity filter(s).

2. Define the requirements for a points-to analysis implementation to compute conservative results (supported native methods and language features).


4. The under-approximation is augmented with additional optimistic analysis results. The over-approximation is reduced through more precise conservative analysis results. This is done iteratively until both, ideally, meet. If, after improving either static or dynamic analysis, the under-approximation is not a subset of the over-approximation, a computation error has occurred that has to be found and corrected.

Researchers, therefore, have to contribute with more precise conservative results and more accurate optimistic results. However, since a Gold Standard cannot be computed automatically, human insight is required to improve static analysis. For this, researchers also need to contribute with manual absence proofs, e.g., reasoning that a certain cast can never fail, a certain call never be made, etc; this is a special case of improving static analysis results. Note that formal proofs are not expected here, as that would be infeasible for the amount of data that has to be dealt with.
Figure 6.1: The general idea of how to create a Gold Standard (G): Improve precision of conservative analysis (Cons), increase dynamic coverage (Dyn), and provide manual absence proofs (AP). Results in the gray area still need to be investigated further.

Figure 6.1 illustrates this general idea informally: Conservative static results as well as dynamic results – both, of course, as accurate as possible – form the starting basis for creating a Gold Standard. There is no “single result set” that forms the basis; as, in general, multiple static (dynamic) result sets can be combined into a more precise static (more accurate dynamic) result set by simply intersecting (merging) the sets (cf. Section 6.3.2). Over time, more precise analysis results can be simply added to the computation of the Gold Standard. Additionally, entities contained in manual absence proofs are to be removed from the combined conservative result sets.

It should be noted that result sets may have dependencies, e.g., to correctness of points-to analysis implementations (naturally), the set of supported native methods (in case an error has been made in step 2 of the process), manual absence proofs, or other analysis results (cf. the discussion above as well as feedback-driven points-to analysis in Section 6.3.3). Note that such dependencies may form a tree, as a result set may be dependent on a result set that is dependent on another result set, etc.

Manual proofs shall be reviewed independently by at least a second researcher.

It may happen that, by improving static and/or dynamic analysis, analysis results suddenly contradict each other. Then, one of them is obviously faulty, which has to be investigated manually so that the root cause may be found, e.g., an implementation error in dynamic/static analysis, a missing dependency on a native method, or an incorrect manual absence proof. One of the results has be withdrawn, as might also dependent results, see above. Also, it should be checked if other result sets may suffer from the same cause.

Over time, the dynamic and static results should converge and ideally meet. The benchmarking platform presented in Chapter 7 can help in this process, and we show an initial attempt towards a Gold Standard in Sec-
6.3 Improvements to Points-to Analysis

Note that, unlike the process done for treebanks (cf. Section 3.2), the process is not limited to a closed group of researchers but encourages all researchers in points-to analysis to contribute through the benchmarking platform.

6.3 Improvements to Points-to Analysis

In this section, we present improvements to points-to analysis that help work towards a Gold Standard.

In Section 6.3.1, we present a set of replacement classes for the Java Collections Framework. In Section 6.3.3, we present feedback-driven points-to analysis, which includes “manual absence proofs” as briefly addressed in Section 6.2. In Section 6.3.2, we briefly discuss the theory behind combining points-to analysis results.

6.3.1 Collection Classes Replacements

Collections frameworks are provided by most modern programming language infrastructures, e.g., the .NET family and Java. Being used by many applications, they are frequently analyzed by points-to analysis. However, collection classes are very difficult to analyze statically. As we argue, they spoil both performance and analysis precision. It is, therefore, worthwhile to treat collection classes with special care in points-to analysis.

In this section, we contribute the following:

- We describe why collection classes are expensive to analyze statically, and why they contribute to imprecision.
- We present a way to overcome these obstacles by means of replacement classes that are, with respect to points-to analysis, sound.
- We provide such replacement collection classes for the Java Collections Framework.

The evaluation of the replacement classes is presented in Chapter 8.

6.3.1.1 Collections Frameworks

Collections frameworks are available for many programming languages, e.g., Java and the .NET family, and should be familiar to every programmer in the Java or .NET world. We discuss here why they cause imprecision in points-to analysis. As an example, consider the object-oriented pseudo code in Figure 6.2: First, a sketch of a typical array-backed list implementation is given (Figure 6.2(a)). If the size of the backing array is insufficient, a new array is created at line 6. Assume, for
simplicity, that this is the only assignment to field elem. Figure 6.2(b) shows a class using this implementation. Two lists are created (lines 5 and 6) and to each of them, an object of type A and B, respectively, is added (lines 7 and 8). For each object in x1, which contains the object of type A, the polymorphic method A.foo() is invoked (line 10). Since there is provably no object of type B in this list, method B.foo() is not reachable in this program. Thus, one would expect that points-to analysis finds that method B.foo() is identified as not reachable. However, only points-to analyses with a context-sensitive naming scheme are capable of identifying this. We discuss why in the following.

Consider now the points-to graphs in Figures 6.2(c) and (d). The circles denote abstract objects, while the arrows points-to relations through the abstract objects’ elem fields. \( M_5 : A \ell \) means that the abstract object is created at line 5 in the code of M, and that its type is AL.

Figure 6.2(c) shows a points-to graph for an analysis with a syntactic creation site naming scheme. This naming scheme can distinguish the AL instances assigned to x1 and x2. A context-sensitive points-to analysis can further distinguish that add() is called once on x1 with the abstract object created at line 3 as this-argument, and once on x2 with the abstract object created at line 4 as this-argument. However, the contents of the two lists are still merged because both abstract AL objects share the same abstract array object that is backing the list – the array created at line 6 in class AL.

Consider now the points-to graph for an analysis with a context-sensitive naming scheme in Figure 6.2(d). Here, one backing abstract array object is created for each instance of AL, and annotated in parentheses with the context the abstract objects are created for (one for calling context \( M_5 \) and one for \( M_6 \)).

Even a simple example like this requires a context-sensitive naming scheme for being as accurate as expected.

In the following, we look at more issues that make analyzing collection classes with points-to analysis expensive and/or imprecise. Examples are taken, without loss of generality, from the Java Collections Framework.

A convenient feature that many collection classes offer is to provide access to the contents of a collection object through different interfaces. We refer to them as views in the following. Examples are iterators for linear access of objects in a set, views on only parts of a list, and views showing only the keys of a map. For example, an ArrayList in the Java Collections Framework has methods iterator(), listIterator(), and subList() that create new objects of types Itr, ListItr, and SubList, respectively, which – from a points-to analysis point of view – just hold references to their owning ArrayList instance and can thus be considered aliases as they always expose the same elements, which, from the point of view of points-to analysis, is all that matters. A SubList of an ArrayList contains, in turn, these same
6.3. Improvements to Points-to Analysis

(a) Array-backed list implementation

(b) Class using array-backed list implementation

(c) Pointer graph for context-insensitive naming scheme

(d) Pointer graph for context-sensitive naming scheme

Figure 6.2: Example: Imprecision due to Collection Classes
methods. Thus, there are potentially six additional abstract objects containing just references to the list itself, which blows up analysis time and memory consumption.

Outlines of the classes `HashMap` and `HashSet` are depicted in Figure 6.3. The class `HashMap` contains, like `ArrayList`, views of its content; when we looked at the real implementation of this class, we found that, in the worst case, 13 aliases are required for each abstract `HashMap` object. Points-to analysis would have to create 13 abstract objects for each abstract `HashMap` object in order to not mix analysis results of different `HashMap` objects. This requires a depth-2 context-sensitive naming scheme, as the backing data structure must also be distinguished.

The class `HashSet` is implemented so that it is backed by a `HashMap`, which means that it also has these 13 aliases. In addition, a more expensive (depth 3) context-sensitive naming scheme is required for distinguishing these: In order to distinguish the `KeyIterator` of different abstract `HashSet` objects, the encapsulating `HashSet` of the `HashMap` must be known as well. An analysis with a context-insensitive naming scheme further merges the contents of all `HashSets` and `HashMaps` through the same abstract `Entry[]` object.

Collections frameworks often contain static helper methods, which only change the order of elements, e.g., for sorting or reversing a list. The order of elements is of no interest to points-to analysis, but abstract collection objects and their contents are mixed here anyway.

In summary, we identified the following problems regarding analysis precision and cost that collections frameworks cause to points-to analysis: (1) Data structures backing collection classes, e.g., arrays, (2) views provided by...
collection classes, and (3) static helper methods.

### 6.3.1.2 Approach Overview

A naive approach to overcome the above-mentioned obstacles is the following: Each abstract collection object is handled as an abstract object of a class with a single field, and calls to methods of collection classes are replaced with corresponding field accesses. In particular, calls to methods adding elements to collection objects are replaced with field `Store`-operations, and calls to methods obtaining elements from collection objects are replaced with field `Load`-operations. For example, a call `coll.add(obj)` is replaced with a field `Store`-operation `coll.f = obj`, a call `obj = coll.get(idx)` is replaced with a field `Load`-operation `obj = coll.f`. This idea basically follows the approach of Liang et al. [46], but it has a major drawback: methods of collection classes often have callbacks into application code, a prominent example being `HashSet.add()`, which calls `hashCode()` and `equals()`. Thus, the analysis is no longer sound, as those methods may no longer be reachable.

We therefore propose to provide replacement implementations for collections frameworks, e.g., of the Java Collections Framework, targeted at use with points-to analysis. The resulting *replacement classes* have the following goals:

I. No backing data structures that lead to imprecision.

II. Avoiding distinct objects for views. This aims at improving analysis precision and cost and at removing the need for context-sensitive naming schemes to precisely analyzing collection classes.

III. Soundness with respect to points-to analysis by following the original collections framework’s contracts where relevant to points-to analysis.

IV. The approach shall not be limited to a single points-to analysis implementation but, instead, will be able to plug into different implementations.

Goals (I) and (II) address the problems of analyzing collections frameworks with points-to analysis that we have identified the previous section. Goal (III) is a natural requirement for being of any use for almost every dataflow analysis. Goal (IV) shall allow the replacement classes to be easily integrated into any given points-to analysis implementation.

Since values of primitive types are ignored by (almost all) points-to analyses anyway, we also omit computations regarding them as far as possible. This makes the code of the replacement classes more compact.

The remainder of this section is as follows: Sections 6.3.1.3 and 6.3.1.4 present the design of such replacement classes, addressing explicitly goals (I) and (II) above, respectively. Section 6.3.1.5 discusses aspects of the soundness
of our approach, i.e., goal (III). Then, section 6.3.1.6 discusses how inlining calls to methods of collection classes can further improve analysis precision. Finally, in Section 6.3.1.7, we present how to incorporate the replacement classes into any points-to analysis implementation (goal (IV)).

6.3.1.3 Replacing Arrays with Base Type Fields As described in Section 6.3.1.1, a context-sensitive naming scheme is required for distinguishing the backing data structures of different abstract collection objects. The basic idea of our proposed replacement classes is to replace the backing data structures with base type fields, i.e., instead of a backing field of type T[], a backing field of type T is used. This is sound with respect to points-to analysis under the following preconditions: (1) An (intermediate) analysis result is never overwritten by the analysis (no strong updates), and (2) the backing array objects do not escape the collection classes. Precondition (1) is generally fulfilled by the nature of points-to analysis, whereas (2) must be checked for each collections framework separately, e.g., by static program analysis; however, it is good object-oriented practice to encapsulate a class’s data and only allow access to its data via methods.

For Java, fields referencing the internal data structures (arrays) of the classes of the Java Collections Framework are either private or package-private. As long as all collection classes in the package java.util are modified properly, and as long as the internal data structures are both created inside each class and never passed to the outside (two exceptions are discussed in Section 6.3.1.5), our proposed approach is sound.

Replacing arrays with base type fields removes the need for a context-sensitive naming scheme to distinguish contents of different abstract collection objects of the same type, since each abstract object comes with its own set of fields.

6.3.1.4 Addressing Views In order to avoid views and aliases in the replacement classes, we decided to make each collection class also represent the views onto itself. For example, in Java, this means that each List also implements the interface Iterator; the new implementations of iterator() methods return this, and the methods hasNext() and next(), where the latter simply returns the storage field, are added to the class. A sketch of the result for ArrayList is provided in Figure 6.4.

Adding interfaces to classes triggered some return type incompatibilities with some methods. For instance, the interface Queue defines a method Object remove(), which is incompatible with void remove() of the interface Iterator. We therefore changed the return type of the method Iterator.remove() (as well as all its implementors) to Object. All such required changes are listed in Table 6.1. Programs that are analyzed with our replacement collection classes must be transformed to comply with these
6.3. Improvements to Points-to Analysis

class ArrayList extends AbstractList implements Iterator {

private Object elems; // non-array field
Object get(int i) { return elems; }
void add(Object o) { elems = o; }

// ...
Iterator iterator() { return this; }

// These methods are required for
// java.util.Iterator:
boolean hasNext() { return true; }
Object next() { return elems; }
}

Figure 6.4: Layout of ArrayList replacement

<table>
<thead>
<tr>
<th>class and method</th>
<th>return type change</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection.remove(Object)</td>
<td>boolean → Object</td>
<td>Map.remove(Object)</td>
</tr>
<tr>
<td>Iterator.remove()</td>
<td>void → Object</td>
<td>Queue.remove()</td>
</tr>
<tr>
<td>ListIterator.add(Object)</td>
<td>void → boolean</td>
<td>Collection.add(Object)</td>
</tr>
</tbody>
</table>

Table 6.1: Changed method return types

changes. We have implemented a bytecode transformation tool that performs these transformations. It will be presented in Section 6.3.1.7. Note that the points-to analysis results for such a transformed program can be mapped back to the original program.

To maintain soundness, callbacks into non-collection code must be preserved. As an example, consider the layout of the replacement for class HashMap in Figure 6.5. Method get() simply returns the field valuesField (line 8), because a points-to analysis usually cannot infer anything about the mappings between keys and values. However, some application methods may be reachable via callbacks here, so these calls need to be retained (lines 6 and 7).

Writing a replacement for a collection class that provides several views using the same abstraction (i.e., interfaces in the object-oriented world) is more complicated. In order to get a sound replacement that does not require a context-sensitive naming scheme, these views need to be merged into a single view. In Java, a prominent example is, again, the Map interface. Views here include keySet(), values(), and entrySet(), which are all Sets; in turn, there are views on these views (i.e., iterator()).

An example is again the replacement class for HashMap in Figure 6.5: the next() method of the iterator returns the merger of the keys, the values, and the entries in the HashMap; the if-statements in the source code (lines 14 and 16) are arbitrary and present only to make the source code
implementation type fields as well as changing the type hierarchy, are correct with respect to the source of imprecision since the keys and values of a HashMap cannot be distinguished any longer if accessed via such a view. However, if the types of keys and values of a map are not related, the imprecision is often instantly removed by subsequent casts. We believe that the ability to distinguish different abstract map instances outweighs this, and our experimental results confirm that this is almost always the case (cf. Section 8.7.2).

Finally, we have written the replacement classes in such a way that relevant contracts of the Java Collections Framework are followed, so that application classes extending collection classes work with our approach. For example, like in the original Java implementation, AbstractMap.get() is implemented to use entrySet() instead of directly accessing the base type fields introduced in our approach.

6.3.1.5 Soundness A formal proof for the approach is out of the scope of this thesis. Such a proof would require to show that the changes, using base type fields as well as changing the type hierarchy, are correct with respect to the semantics of points-to analysis. Providing a proof of soundness for the implementation of the replacement classes is infeasible, as the code is just too big for a formal proof using some calculus.
6.3. Improvements to Points-to Analysis

We have, however, argued informally why using base type fields instead of arrays is sound with respect to points-to analysis. We have further described our efforts to maintain the API contracts as well as implementations of abstract classes (e.g., `AbstractMap.get()`) to retain all possible callbacks into application code. In the following, we will discuss two assumptions that we have made and argue why they, when care is taken, guarantee that soundness of the analysis is preserved.

First, the backing data structures of collection classes may not be entirely encapsulated from the outside. We found two such cases in the Java Collection Framework: The class `Vector` stores vector elements in a field `elementData` that is protected, which means that it is accessible to the outside of the package `java.util`. Thus, class `Vector` needs to be removed from the replacement classes if a user-based subclass directly accesses this field. Second, class `ArrayList` uses `Collection.toArray()` to initialize its backing array in some cases. This means that, while the internal array is never passed to the outside, it may come from the outside, from where it may be referenced. However, we assume that the contract of the method `Collection.toArray()` is followed, which says that “The returned array will be “safe” in that no references to it are maintained by this collection” (Source: Javadoc of the method `Collection.toArray()`).

The changes to the type hierarchy of the collection classes must not interfere with any optimization in the points-to analysis algorithms. Operations in question in Java bytecode are `cast` and `instanceof`. The former is affected by standard points-to analysis implementations only in the sense that more objects will pass the cast, which is a potential source of imprecision but does not affect soundness. The latter operation is usually disregarded by points-to analysis. However, remember the presentation of path sensitivity in P2SSA in Section 2.9: here, an if-then-else statement with a guard of form `x instanceof T` restricts the points-to sets of `x` in each conditionally executed block: in the then-block to objects of type `T` – which, like in the case of cast, just lets through additional objects – and in the else-block to objects not of type `T`. If `T` is now a type with a changed type-hierarchy, then the optimization becomes unsound. However, both operations are rarely used on types in question outside the collection framework itself. Our bytecode transformation tool, see Section 6.3.1.7, warns if this is the case.

None of the above issues occurs with the programs that we used for the experimental evaluation. The overall lesson learned here is that one needs to be aware of such issues before applying our approach.

6.3.1.6 Inlining Calls As an additional step, we inline calls to collection classes. This allows context-insensitive analysis to benefit from the replacement classes as well, because collection objects and elements stored in such objects are no longer mixed through calls like `add()`, `get()`. Context-
sensitive analysis already distinguishes different calls to such methods and thus does not benefit to the same extent, but argument-mixing within a context is still avoided.

Inlining is straightforward in our setting as methods in the collection classes are never recursive. Polymorphic calls are inlined as a sequence of monomorphic calls, where each monomorphic call has a filter for the implicit `this` attached.

6.3.1.7 Bytecode Transformation Tool We implemented a bytecode transformation tool that does the following two tasks: (1) it checks for the traps regarding `Vector` and `instanceof` outlined in Section 6.3.1.5 and (2) it performs the method signature changes listed in Table 6.1.

The first task is straightforward: Code is checked for accesses to the field `Vector.elementData` and operations `instanceof T` where T is of a type that has its type hierarchy changed.

The latter task works as follows: In the first step, the methods for which the signatures are changing are determined, i.e., all methods implementing any of the methods listed in Table 6.1 are identified. This is done based on a simple class hierarchy analysis. Then, the code is transformed such that (1) all method declarations are changed accordingly, and (2) all method bodies are transformed to comply with the new return types. Step (2) is required to keep the bytecode valid. Otherwise, analysis tools will not work because the stack states before `returns` in transformed methods are invalid, as are the stack states after calls to those methods. For `ListIterator.add()` and `Iterator.remove()`, the code is changed as follows: Each `return` instruction is changed to `return null` or `return false`, respectively, and, after callers of those methods, a `pop` instruction is inserted to clean up the stack. Method bodies of implementors of `Collection.remove()` are changed by replacing each `return booleanExpr` with `return null`. After calls to these methods, the stack is cleaned up by popping the result and pushing a boolean constant. This assumes that the subsequently used analysis does not evaluate boolean constants, but some engineering work can make the result random, if required.

The tool is based on the ASM framework\(^1\). It takes the complete set of `.jar` files that are to be analyzed in a subsequent program analysis and generates transformed `.jar` files. It also generates a list of those methods whose signatures have been changed, which can be used for mapping back points-to analysis results from the transformed program to the original program. The transformed set of `.jar` files is then used instead of the original set of `.jar` files for the subsequent analysis. This subsequent analysis does not require any knowledge regarding the fact that the code has been transformed.

\(^1\)See http://asm.ow2.org.
6.3. Improvements to Points-to Analysis

6.3.1.8 Summary
An initial attempt to replace calls to collection classes with simple field accesses, which has been presented in previous literature in a similar manner, is not sound because callbacks into non-collection classes are removed. It also needs to be implemented for each points-to analysis individually. We propose to, instead, use simplified replacement collection classes that are sound with respect to points-to analysis. The implementation is basically changed in the following way: (1) Fields of base types are used instead of array types for storing elements in collections, (2) the type-hierarchy is adapted so that collection classes can also represent views of themselves, and (3) computations on primitive types are removed. Possible callbacks into non-collection classes are maintained. An evaluation of the benefits of these replacement classes is presented in the next section.

The replacement collection classes as well as the complementing bytecode transformation tool are made available at http://p2abench.sf.net/.

6.3.2 Combining analysis results
Combining dynamic analysis results by computing the set union and conservative static analyses by computing the set intersection are obvious improvements. A theoretical discussion also with respect to combining general analyses can be found in [32]. Here, we discuss the benefits of combining analysis results with respect to creating a Gold Standard.

Combining analysis results comes in handy as different points-to analysis algorithms (and variations thereof) are usually not strictly ordered in terms of precision. This holds, for instance, for different context-sensitive approaches, as shown for example by Milanova et al. [61] and Lundberg et al. [52]. More importantly, specific features of different implementations can be exploited: Usually, different points-to analysis implementations support a different set of optimizations for precision, performance, and memory consumption. For example, Paddle may be able to analyze larger projects than P2SSA with memory-expensive context-sensitive naming schemes due to its BDD-based approach. On the other hand, P2SSA is flow-sensitive and supports filter operations. Combining the results of both analyses brings the benefits of flow sensitivity, filter operations, and context-sensitive naming schemes together. Of course, a flow-sensitive and BDD-based points-to analysis implementation would be even more desirable, but there is no such implementation known to us.

6.3.3 Feedback-driven Points-to Analysis
In this section, we present feedback-driven points-to analysis. Feedback-driven points-to analysis means to perform (any classical) points-to analysis with the points-to results at certain program points guarded by a-priori upper bounds.
Chapter 6. A Shared Effort Towards a Gold Standard

There are two sources of predefined upper bounds for points-to analysis: complete results obtained from conservative points-to analysis, which allow for automated feedback, and manual absence proofs, i.e., human insight that certain points-to information is infeasible. We look at the two in Sections 6.3.3.1 and 6.3.3.2, respectively.

6.3.3.1 Automated Feedback Automated feedback-driven points-to analysis is desirable when different approaches are not strictly ordered in terms of accuracy. This is, for example, the case for the context sensitivities presented in Section 2.5.1. While it is of course possible to define a more precise combined context sensitivity, e.g., a combination of ObjSens and CallSite, this would increase the number of contexts dramatically and can lead to practical problems.

Our approach is now to set upper bounds for the operands to two kinds of operations: Store-operations that write values to memory slots on the heap, and Call-operations.

Each Store-operation $S.f$ has, next to the (fixed) field $f$ that is being accessed, two operands that depend on intermediate analysis results: the address $A \subseteq O$ (where to store), and the value $V \subseteq O'$ (what to store). Each of them gets assigned an upper bound that is dependent on previously computed results. These usually stem from a context-sensitive approach, i.e., there is a pair $(A_{ctx}, V_{ctx})$ computed for $S.f$ for each context $ctx$ the method containing $S.f$ is analyzed in.

For example, assume that the previously computed results for $S.f$ are computed under two different contexts and that $A_1 = \{o_1\}, V_1 = \{o_2\}, A_2 = \{o_3\}, V_2 = \{o_4\}$. The heap then contains the relations $(f, o_1) \leftarrow o_2$ and $(f, o_3) \leftarrow o_4$.

When a new analysis with a different context definition updates $S.f$, we use the set of previously computed pairs as upper bounds. In the new analysis, we can identify the same Store-operation $S.f$, the same field $f \in F$, and the abstract objects $O$ used as addresses or values (provided the new analysis uses the same object abstraction). The different contexts of the previous analysis, however, are not known any longer. But, regardless of the context definition, each new pair $(A_{ctx'}, V_{ctx'})$ analyzed as addresses and values by the new analysis needs to be consistent with the previously computed upper bound. This means that each address value element $(a, v) \in A_{ctx'} \times V_{ctx'}$ must be contained in at least one pair $(A_{ctx}, V_{ctx})$ of the upper bound, i.e., $(a, v) \in A_{ctx} \times V_{ctx}$ or it is filtered otherwise. Then, the less precise analysis has – thanks to the upper bounds – at least as good precision as the previous analysis.

Continuing on our example, if a new analysis updates $S.f$ with the input pair $A_{ctx'} = \{o_1, o_3\}, V_{ctx'} = \{o_2, o_4\}$, then, without filtering, the heap is updated with four relations $(f, o_1) \leftarrow o_2, (f, o_1) \leftarrow o_4, (f, o_3) \leftarrow o_2$, and
6.3. Improvements to Points-to Analysis

\((f, o_3) \leftarrow o_4\). This is larger than the previously computed upper bound \(\{ (A_1, V_1), (A_2, V_2) \} = \{ \{ (o_1, o_2) \}, \{ (o_3, o_4) \} \} \) that does not contain the pairs \((o_1, o_4)\) and \((o_3, o_2)\), which can be filtered in the new analysis.

For Call-operations, the same could be done for the tuples of this-values \(T\) and corresponding arguments \(A_i\). More precisely, for each context \(ctx\), we could capture and distinguish the tuples \((T_{ctx}, A_{ctx}^1, \ldots, A_{ctx}^n)\) as bound but drop the actual context definitions \(ctx\). For a new analysis and a tuple \((T_{ctx'}, A_{ctx'}^1, \ldots, A_{ctx'}^n)\) analyzed for the same call operation, each element \((t, a^1, \ldots, a^n) \in T_{ctx'} \times A_{ctx'}^1 \times \ldots \times A_{ctx'}^n\) must be contained in at least one tuple of the upper bound or it would be filtered otherwise.

For a Store-operation, we just perform an abstract heap update for all (non-filtered) tuples. For a Call-operation, we would have to analyze the target method for each (non-filtered) tuple. This would just lead to mimicking another, very fine-grained context sensitivity, i.e., a different context-sensitive approach with many distinguished contexts (one for each non-filtered element). As mentioned above, this might not be desirable due to performance issues, and early tests that we performed confirm this. Thus, we merge upper bounds for those tuples and accept the loss of precision.

Assume, for example, \(T_1 = \{ o_1 \}, A_1^1 = \{ o_2 \}, T_2 = \{ o_3 \}, A_2^1 = \{ o_4 \}\) as the bound computed in one analysis and \(T_{ctx'} = \{ o_1, o_3 \}, A_{ctx'}^1 = \{ o_2, o_4 \}\) the unfiltered update for a second analysis. In this case, we only check the relaxed bounds \(T_{ctx'} \subseteq T_1 \cup T_2\) and \(A_{ctx'}^1 \subseteq A_1^1 \cup A_2^1\) and accept the less precise update.

Note that this limitation does not have an impact on the set of reachable methods, as invalid target objects are still filtered completely.

6.3.3.2 Manual Feedback

Another source of upper bounds comes from manual proofs, which are indispensable for creating a Gold Standard. In addition to the upper bounds from automated feedback, we allow even coarse-grained hints that are more accessible to human insight, i.e., (1) which code can be considered dead, (2) what arguments to or return values of methods are infeasible, (3) what call edges can not be taken. Figure 6.6 shows a short code example of where such manual feedback is trivial but that many points-to analysis algorithms cannot deduct: (1) The user is able to prove that method \(f()\) never returns \(-1\), so that the code at line 6 can be considered dead. (2) Human insight can determine that the call \(x.h(y)\) at line 9 must return \(o_2\), so that the use of the local variable \(x\) can be restricted to \(pt(x) = \{ o_2 \}\) in the following.

For dead code (1) and impossible return values (2), human insight refines the upper bounds as provided by automated feedback; then, both kinds of feedback are treated just alike. All operations that stem from dead code are annotated with empty upper bounds. Impossible return values are removed from the upper bounds of Call-operation results. Finally, for a call edge
analysis by creating proper input to the program, or prove that the methods attempted to either trigger the execution of methods not covered by dynamic and static points-to analysis results on a given program, compared needed to understand where imprecision originates. We thus compared dynamic and static points-to analysis results on a given program, compared the sets of reachable methods, and investigated why these differ. We then excluded at a Call-operation (3), the corresponding target method is simply skipped in the analysis.

Manual feedback can be combined with automated feedback, which comes in very handy when performing manual absence proofs: In the first step, results for the most precise (but usually also most expensive) available analysis are computed. When iteratively adding and evaluating new manual absence proofs, a faster analysis is used with those results as upper bounds. This way, faster iteration steps are possible while still having the more precise results as a reference.

Note that the human expert stands for the correctness of his manual feedback. If, after adding some feedback, the dynamic points-to information is no longer a subset of the static points-to information, the feedback is obviously not sound. On the other hand, if this is not the case, then this is still not a proof of the correctness of the manual feedback. However, additional test cases that are provided later may uncover a given manual feedback as invalid. Also, the manual absence proofs can be made public on the benchmarking platform and thus peer-review from other researchers is possible, cf. Section 6.2 for details.

In order to improve the precision of static points-to analysis, we first needed to understand where imprecision originates. We thus compared dynamic and static points-to analysis results on a given program, compared the sets of reachable methods, and investigated why these differ. We then attempted to either trigger the execution of methods not covered by dynamic analysis by creating proper input to the program, or prove that the methods

Figure 6.6: Example code with annotations of possible manual feedback.

```java
1: class A {
2:     static void main(String[] args) {
3:         A x = new A(); // abstract object o1
4:         A y = new A(); // abstract object o2
5:         if (x.f() == -1) {
6:             // dead code since x.f() != -1
7:         }
8:         x.h(x);
9:         x = x.h(y); // return value always o2
10:        //... use of x
11:    }
12:    int f() { return 0; }
13:    A h(A p) { return p; }
14:}
```

14:}
6.3. Improvements to Points-to Analysis

are indeed not reachable. As an item to investigate for this initial case study, we selected the Java 5 to 4 backporting tool that is part of the distribution of recoder (http://recoder.sf.net), a source code analysis and transformation framework for Java. The author of this thesis is well familiar with the implementation of recoder, which is also complex enough to imply a challenge to our research. We discuss the patterns of sources of imprecision that we found in the following.

**Dead Code** If an application is built on top of a more general framework, unused parts of the framework are often still deployed with the application, be it for convenience, inaccessibility of the source code of the framework, or because instantiation options are delayed until runtime by means of dynamic binding or other means of configuring the framework at runtime. The latter is usually not recognizable by points-to analysis. Concretely, the tool that we analyze does not change many configuration options provided by the recoder framework, so that we can mark code as dead if it is triggered only when the default values of these options are changed.

**Polymorphic Calls in Branching Conditions** When a branching condition compares the result of a polymorphic call with a given constant value, some possible target methods of the polymorphic call may provably never fulfill the given condition. This can be the case if such a method returns a constant value (but other related methods do not). Then, the value of the receiver object can be restricted in the then-block of the condition: abstract objects of the types that never fulfill the condition can be filtered. We found examples for this in some central places of recoder: Conditions of branch statements look like in the following source code excerpt, where ClassType is an interface common to many concrete classes, and getA() returns a list.

```java
recoder.abstraction.ClassType x = ...;
if (x.getA().size() > 0) { use of x... }
```

Some of the classes that implement the interface ClassType always return an empty list, so that instances of those classes can never reach the body of the if-statement. We manually told our analysis to remove instances of such classes from method arguments, field accesses, etc. within the then-block. For future work, it should be possible to perform this optimization automatically by inserting special filter-operations (similar to instanceof filters as used for path sensitivity in P2SSA) into the analysis’ program representation.

**Shared Caches** A program may have general, untyped general-purpose object caches that are shared by different parts of the program. If it is guaranteed that each part of the program retrieves only those objects that it puts into a cache itself, the caches can be considered as being logically partitioned. However, points-to analysis cannot, in general, recognize this, and thus assumes that all objects stored in the cache may be read by any part of the program, which introduces a source of imprecision.
Recoder uses a single *HashMap* instance to cache previously computed mappings from type-, variable-, etc. references to resolved program elements. Through the cache, resolving the different kinds of references gets mixed in the points-to analysis results. We solved this problem by removing the caches from the analysis: Since abstract objects retrieved from the cache (and instantly returned) by the methods in question are computed anyway within the very same methods, this does not pose a threat to the analysis being conservative.

### 6.4 Fast Dynamic Analysis

In order to gain very precise dynamic coverage for benchmark programs that are meant to become part of a Gold Standard, it is desirable to monitor the execution of the programs in real life, e.g., when researchers use the benchmark programs themselves. Then, however, performance overhead must be kept as small as possible. Therefore, we present an experimental approach on how performance overhead can be kept low for capturing dynamic analysis information related to points-to information in this section.

Our dynamic analysis tool captures the occurrence of events, e.g., invocations of methods, thrown exceptions, etc. A priori, it is not known which or how many distinct events may possibly occur, which makes it necessary to have a general data structure that is able to capture any event. This data structure must be dynamically expanded to be able to capture events that happen. However, when conservative static analysis is available, the possibly occurring events are known to be limited prior to executing the program so that the data structure can be statically optimized for the given task and accessed with higher performance.

The remainder of this section is organized as follows: In Section 6.4.1, we lay out the overall engineering process of our approach. In Section 6.4.2, we then describe what efficient instrumentation code may look like. Analysis of multi-threaded programs is discussed in Section 6.4.3. Finally, Section 6.4.4 lists some limitations of our current prototypical implementation.

Note that we will keep the description of concepts independent of points-to analysis, but we will use the client analysis “object call graph” as a running example in this section.

#### 6.4.1 Engineering Process

The general approach that we suggest works as depicted in Figure 6.7: First, the original code is used as input to a *static analysis* that produces *static analysis results* for the same task as the dynamic analysis tool’s task. The *instrumentation tool* then uses these results as upper bounds for events that can occur at runtime, and generates instrumented code *code’* from the original
6.4. Fast Dynamic Analysis

Figure 6.7: Engineering Process

code. This instrumented code can then be deployed instead of the original code.

6.4.2 Instrumentation Code

At runtime, events correspond to concrete program states. A static analysis can, by its nature, only reason about abstract program states. Therefore, if dynamic and static analyses compute the same type of analysis result, a mapping from concrete program states to abstract program states is essential to our approach. For points-to analysis, this means mapping from concrete runtime objects to abstract objects according to a naming scheme. The amount $|S|$ of distinct events $S$ possibly occurring at runtime is bounded by static analysis, and we can describe each event with an abstract program state. A bijective function $f(S) \rightarrow [0, |S|]$ maps from a currently occurring abstract program state to an abstracted event ID $i$. The function $f$ is thus a minimal perfect hash function mapping concrete events to abstract events (actually, their IDs). Implementation-wise, the value $i = f(s), s \in S$ must then be computed for the current abstracted runtime state, and the occurrence of the event must be stored into a result data structure.

In our case, we are not interested in how often a given event (i.e., object call graph edge) occurs, but only if ever, so our result data structure is a byte-array, and storing the occurrence of an event is an access like $res[i] = 1$. Further, the function $f$ takes three arguments: The call site where a call is occurring, the abstract object corresponding to the “this” object of the currently executing method, and the abstract object corresponding to the “this” object of the target. Since the call site is fixed for each program point where an event can occur (defined by the syntactic location of the call site), a set of functions $f_s(o_c, o_t)$, with $o_c$ being the caller and $o_t$ being the target object, can be defined.

The mapping functions need to abstract from the current runtime state, so that dynamic and static analysis results become comparable. How this is done in general depends on the client analysis; we therefore describe herein
how this is done for our client analysis *object call graph*: Our tool instruments each object creation site so that all objects created at this program point get a special *tag* value which is a unique integer value. This corresponds to the *creation site naming scheme* described in Section 2.3. This value can be accessed via a special `getTag()` method that simply returns this value. This tag-number can be mapped back to the class and line number where the object is created.

If each abstract program state can be defined as an ordered tuple \((s_1, ..., s_n)\), where \(s_i, i \in [1, n]\) can be computed to integer values, then each such tuple can be assigned an event ID \(e_k\). So, the mapping function from abstract state to event is defined by the ordered tuples \((s_1, ..., s_n, e_k)\). Those tuples are derived from static analysis. For example, our instrumentation tool reads the statically computed possible object call graph edges from an XML file, assigns each abstract object a unique tag-number, and creates a set of such tuples for each syntactical call site in the program. As an example, a call site may have the tuples \((0, 13, 28), (0, 28, 14)\), and \((7, 13, 29)\) defining its mapping function.

At runtime, the function must compute the event ID as efficiently as possible. We have experimented with two ideas that are presented in the following. We explain the two different approaches by means of our object call graph analysis, as it is hard to generalize to all other possible program analysis tasks. However, we believe that the general idea can be applied to other program analyses as well.

**6.4.2.1 Approach One: Switch-statements** Our first idea is to encode the tuples directly into the control flow of the program by means of (nested) *switch*-statements. Consider the program in Figure 6.8. It contains a single method `foo()`, which, in turn, contains just one call to another method `bar()`. This call is instrumented as follows: Depending on the tag of both the caller object and the target object (*this* and *a*, respectively), an index in the storage array (`Result.ogc`) is selected and set to value 1, indicating that this call (event ID corresponds to array index) has occurred at some time during the program run. Note that, if no matching pairs for the two tags are found, i.e., one of the *switch*-statements goes into the “default” branch, then this needs special handling. In our case, this indicates incomplete results from static analysis; sometimes, this may be the case due to unsupported dynamic class loading or unsupported native methods by the static analysis. A fallback to general purpose dynamic analysis can then be done.

**6.4.2.2 Approach Two: Formulas** The more imprecise the static analysis, the more *case*-statements are required in the instrumented code. This increases the code size and decreases the performance. It is therefore interesting to describe the hash function with a formula, which is our second
6.4. Fast Dynamic Analysis

```c
void foo(A a) {
    ///// start instrumentation /////
    oThis = this.getTag();
    oTgt = a.getTag();
    switch (oThis) {
        case 0:
            switch (oTgt) {
                case 13: Result.occ[28] = 1; break;
                case 28: Result.occ[14] = 1; break;
                default: handleDefault(); break;
            }
        case 7:
            if (oTgt == 13) Result.occ[29] = 1;
            else handleDefault();
            break;
        default: handleDefault();
    }
    ///// end instrumentation /////
    a.bar();
}
```

Figure 6.8: Instrumentation: Nested switch-statements

approach. Unfortunately, there is no obvious functional connection between the abstract objects reaching this call and the event ID. However, we implemented a heuristic as follows: (1) Tag-numbers of abstract objects are ordered according the type hierarchy. This way, abstract objects that can reach the same call sites should have tag-numbers “close by” each other. (2) For each call site \(s\), the range of tag-numbers of abstract objects for both the caller \(c\) and target \(t\) are defined: \(\min_{c,t}\) (\(\max_{c,t}\)) define the minimum (maximum) tag-number of all abstract objects possibly reaching the current call site as caller or target, respectively. Then we define the range \(r_t\) of \(t\) as \(r_t = \max_t - \min_t + 1\). Finally, we define a formula as follows: \(f_s(o_c, o_t) = (o_c - \min_c) \times r_t + (o_t - \min_t) + O_s\). Note that this means that event IDs are reordered according to the formulas. Since another call site may already “use” some of the event IDs that \(f_s\) can map to (note that it is not the full range of values, as not all argument values are possible combinations), we add an offset \(O_s\) for each call site, which is computed so that all possible function results are still “free”; then, those event IDs are marked as being “in use”. The size of the result array must, in general, be increased for this approach, so that the perfect hash function is no longer minimal.

In our above example, the abstract objects \(o_0, o_7, o_{13}\), and \(o_{28}\) may get their tag-numbers adjusted to \(o_5, o_6, o_7\) and \(o_8\). Then, \(\min_c = 5, \max_c = 6, \min_t = 7, \max_t = 8, r_t = 2\) and the formula will be \(f_s(o_c, o_t) = (o_c - 5) \times 2 + (o_t - 7) + O_s\). Slot \(3 + O_s\) in the result array remains unused, because the reordered tuple \((o_6, o_8)\) at this call site cannot occur at runtime, according to static analysis.

Obviously, this second approach comes with more memory overhead – the
result array may be significantly bigger – but possibly reduced performance overhead than when inlining the function into the control flow by means of switch-statements. Trying to find formulas that are not bound to the structure presented above may reduce the need for increasing the size of the result array. We will evaluate both approaches in Section 8.6.

### 6.4.3 Analyzing Multi-thread Programs

In general, when analyzing multi-threaded programs, synchronization might be required to access the common data structures, e.g., when increasing an event counter. Optionally, the data structures could be cloned for each thread in order to reduce synchronization, at the expense of higher memory cost and thread creation time. This approach is successfully applied by Binder et al. [15]. In our concrete case, where we are interested in what happens, but not how often it happens, synchronization is not necessary as our event-capturing code simply writes a “1” into an array (and never any other value). Race conditions thus simply do not matter in our special case.

### 6.4.4 Limitations

Our approach suffers from the general limitations that every instrumentation-based approach suffers from: First, native code cannot be analyzed. That is, objects created in native code will not be tagged, and thus the results of the dynamic analysis will be incomplete. However, Binder et al. point out that programs usually spend only little time in native code [14]. Second, the size of a method’s bytecode is restricted by the Java Virtual Machine Specification [50]. If now, through instrumentation, the bytecode exceeds this limit, the program will no longer work. In order to avoid this as much as possible, we put the switch-statements into separate methods (one method for each call site) and add calls to those methods at the original call sites. The problem then still exists in theory, but the added instrumentation code is much smaller; therefore we agree with Binder et al. who state that “this problem is very unlikely to occur with normal, hand-crafted Java code” [15].

### 6.5 Conclusion

In this chapter, we have presented a distributed process to create a Gold Standard for points-to analysis. This process is based on iteratively improving static and dynamic analysis, until results of both meet and thus define the Gold Standard. Static analysis must, here, be supported by manual feedback, i.e., expert knowledge about programs.

We have also presented a number of improvements to static points-to analysis: replacement classes for collection classes, combining analysis results,
and feedback-driven points-to analysis (manual and automated). We will experimentally investigate their impact in Section 8.7.

Finally, dynamic analysis must be improved by running programs with more input. In order to reduce the overhead when running the programs, we have presented an approach that utilizes results obtained from static analysis to custom-make instrumentation code for each benchmark program. We present an experimental evaluation of this approach in Section 8.6.
Chapter 7

Benchmarking Platform

In this chapter, we present a benchmarking platform that supports researchers in applying the benchmarking methodology presented in Chapter 5 as well as the process for creating a Gold Standard presented in Chapter 6.

The benchmarking platform is basically a database that contains client analysis results from different static points-to analyses as well as dynamic analysis. Results stored in it can be compared qualitatively, i.e., differences in the result sets can be shown, and the platform computes approximated accuracy results ($P^-$ respectively $\hat{P}$, $\hat{R}$ etc.; cf. Chapter 4) for the results of static analyses. Each points-to analysis implementation that implements at least one of the client analyses from Section 5.4 and follows the other guidelines from Chapter 5 can be connected to this platform by exporting client analysis results to a special file format.

This chapter is organized as follows: In Section 7.1, we discuss the user stories, i.e., the requirements, of the benchmarking platform on a high level. The design of the benchmarking platform is then described in Section 7.2. In Section 7.3, we describe the file format that is used to upload client analysis results to the benchmarking platform. We conclude this chapter in Section 7.4.

7.1 User Stories

In this section, we present the user stories that the benchmarking platform has to provide in order to support researchers in applying our benchmarking methodology and in helping researchers create both a benchmark suite as well as a Gold Standard for points-to analysis.

7.1.1 Creating a Benchmark Suite

In order to create a benchmark suite for points-to analysis, the following steps – as summarized in Section 5.6 – must be performed: A number of client analyses and accompanying metrics must be defined, a number of benchmark programs selected, and, for each benchmark program, (1) its application name filter(s) must be defined, (2) its set of required supported native methods in
order to get conservative analysis results for its application client analyses must be specified, and finally (3) dynamic analysis results must be provided.

The benchmarking platform has a built-in support for seven client analyses, namely all of those defined in Section 5.4. These are fixed for the benchmarking platform, however, not all of them have to be used for a benchmark suite and can just be ignored. If a currently unsupported client analysis should be part of a benchmark suite, code changes to the platform are required. However, we will show in Section 8.8.3 that those code changes are rather small.

In the following, we describe how the other steps are performed on the benchmarking platform.

7.1.1.1 Registering a new Benchmark Program When registering a new benchmark program, the user specifies the name of the program and its version as well as an optional link to the homepage of the benchmark program. The user must also provide the name filters for identifying application entities (cf. Section 5.1) that are to be used with this benchmark program. Additionally, the application entry point (the class containing the \texttt{main}-method) for analysis must be specified.

The user should also upload the source and/or bytecode, including third-party libraries, of the benchmark program. If that is not possible, an exact description of where to obtain the code must be provided. The user must also specify what Java runtime version this benchmark program is to be analyzed with. This is important for repeatability and comparability, as described in Section 5.5. The code can be downloaded by other users, so that they can perform their own experiments with the exact same benchmark program.

7.1.1.2 Annotating Benchmark Programs Each benchmark program should be annotated with a set of native methods that must be supported by a points-to analysis implementation in order to analyze it conservatively. How to obtain this set of native methods is described in Section 5.2.

A benchmark program can be annotated with an extensible set of soundness features that are required to analyze its application part conservatively. Such features are mostly different native methods, but can also be language features such as finalizers and threads, which may not be supported by all points-to analysis implementations.

7.1.1.3 Providing Dynamic Results In order to provide dynamic analysis results, the user first selects the benchmark program to which the results belong. The user then uploads the results file, cf. Section 7.3, and the benchmarking platform takes care of the rest. Multiple result files can be uploaded this way, and the benchmarking platform combines them by merging them.
Chapter 7. Benchmarking Platform

7.1.2 Applying the Benchmarking Methodology

Applying our proposed benchmarking methodology requires to first create a benchmark suite, as described in the previous section. Then, users must register their points-to analysis and submit analysis results for this points-to analysis. Submitted results can then be compared, both to dynamic analysis results (including calculations of $P^-$ etc.) and other static analysis results for the same benchmark programs.

These three steps are described in detail in the following.

7.1.2.1 Registering a new Points-to Analysis

A points-to analysis is identified by its name and version. A link to the homepage of the points-to analysis and/or research group developing it can be provided. A textual description can be provided as well as a set of soundness features that the analysis supports. Additional to the soundness features, the benchmark platform allows to annotate points-to analyses additionally with a set of supported accuracy features (e.g., different context sensitivities). Those are an informal categorization and merely for informational purposes to the users of the benchmarking platform.

The benchmarking platform provides a micro benchmark suite to assess the set of supported soundness features of a points-to analysis implementation. It is used as follows:

1. The micro benchmark suite is downloaded from the benchmarking platform and analyzed with points-to analysis.
2. The results (providing the call graph / set of reachable methods is sufficient) are uploaded to the platform.
3. The platform determines, based on the methods deemed reachable by the points-to analysis, which soundness features are supported.

That is, the micro benchmark suite is written so that certain methods are reachable if and only if certain native methods or language features are supported.

7.1.2.2 Uploading Static Analysis Results

When providing static analysis results, the user first decides whether the results are to be treated as conservative or general. Note that the distinction between conservative and general analysis must be made by human insight and on a per-benchmark program basis. Only in cases where dynamic analysis proves a static analysis to be not sound does the platform help in this decision.

The user then selects the points-to analysis with which the results were obtained. The user can also select which of the accuracy features that the points-to analysis supports were used for this analysis run.
Finally, the user should provide as exact details as possible on how the experiment was performed and how it can be repeated. This can be done with a textual description.

After submitting the results file, the user is directed to the analysis results page for this points-to analysis; see below.

### 7.1.2.3 Comparing Results

When selecting a benchmark program, all submitted analysis result sets from static analysis are listed. The user can select any number of them for comparison. Then, for each client metric, the absolute numbers of dynamic analysis as well as static analysis are listed.

In case of general analysis, the number of false negatives are listed as well, and the user can follow a link to another web page upon which is listed what exactly is contained in the dynamic result sets, but not in this specific static result set.

If exactly two static analysis result sets are selected for comparison, the user can also follow a link to show the differences in the client analysis results, i.e., entities contained in the first but not the second result set, and vice versa. This is for informative purposes and can aid researchers in identifying sources of inaccuracy in their own points-to analysis implementation, thus pointing out ways to improve it.

### 7.1.3 Working Towards a Gold Standard

The benchmarking platform supports the different steps for our proposed process to work towards a Gold Standard, cf. Chapter 6. The basis is the same as creating a benchmark suite (select programs, specify entry points, name filters, required soundness features, and provide dynamic analysis results) and using it (submitting conservative static analysis results). One difference is that dynamic analysis results are continuously submitted, whereas when using a benchmark suite, only static analysis results are submitted after the suite has been created.

The benchmarking platform also supports managing absence proofs, as well as combining conservative analysis results, and handles submitting of results a bit differently. These are described in the following.

#### 7.1.3.1 Managing Absence Proofs

The benchmarking platform allows users to upload manual absence proofs. Additionally, other users can review them and mark them as “verified” on the platform. This feature is for informational use only but will usually strengthen the acceptance of a manual proof.

#### 7.1.3.2 Combining

Combining analysis results (cf. Section 6.3.2) comes naturally to the benchmarking platform: Since client analysis results are stored on the platform, the current best effort towards a Gold Standard can
be computed by intersecting all conservative static results and merging all dynamic (optimistic) results for a given benchmark program.

Note that manual absence proofs are not taken into consideration here, although they could be removed from the best effort of static analysis. This is because manual absence proofs can contain non-client analysis information such as dead code; cf. Section 6.3.3. Instead, feedback-driven analysis results created with help of the absence proofs should be submitted.

### 7.1.3.3 Submitting Results

Submitting analysis results is done in the same way as before, but users can now specify whether the results have dependencies to previously submitted results or manual absence proofs.

After submitting, the platform checks whether the under-approximation of the Gold Standard is still a subset of the over-approximation of the Gold Standard. If not, the newly submitted results are either incorrect or prove previously submitted results to be incorrect. The benchmarking platform requests the user to sort this out.

## 7.2 Platform Design

In this section, we present the software design of the benchmarking platform. First, we discuss the design of the backing database of the benchmarking platform in Section 7.2.1. We then describe the XML file format in which results have to be uploaded to the platform in Section 7.3. Finally, the more high-level implementation details of the platform are presented in Section 7.2.2.

### 7.2.1 Entities

The benchmarking platform design is based on the following entities:

- Users
- Points-to analysis implementations
- Features
- Benchmark programs
- Source code entities, e.g., methods and allocation sites, as basis for client analyses
- Client analyses
- Analysis runs
User management will not be discussed in the remainder of this thesis as it is a necessity for Web-based platforms without otherwise restricted access, but is not a key concept of the benchmarking platform.

Before submitting analysis results, the points-to analysis implementation that was used for conducting the experiments must be registered on the benchmarking platform as such. It is simply registered with its name and version/build date. Optionally, yet also recommended, it can be annotated with information on how to obtain its code, so that other researchers can use this points-to analysis implementation as well. The latter can be done either by specifying URLs, links to revision control systems (including the revision used for the actual results), or by uploading (binary) data files. Optionally, researchers can provide a maven “pom.xml” file that specifies the exact version of all involved software and that allows users to easily download and set up the software.

A points-to analysis can also be annotated with a set of features that it supports. Such features are divided into soundness features, e.g., different native methods or multi-threading support, and accuracy features, e.g., different context sensitivities, inter-/intra-procedural flow sensitivity, and other potential optimizations.

A benchmark program must be registered with its name and version, accompanied by an exact description of how and where to obtain the code, as well as what third-party libraries (including what version of the Java runtime) are to be used for analysis. The platform even allows users to upload the code of the benchmark program itself, but the uploader is responsible for checking if the program’s license allows for this. A benchmark program can also be annotated with required soundness features.

Source code entities are basic entities of which the client analyses, see below, are composed. The benchmarking platform knows the source code entities allocation sites, methods, fields, classes, type casts, and member (field and method) accesses. Each source code entity has relevant information for uniquely identifying it, e.g., a method its bytecode name and description, a field its fully qualified name, a member access its target as well as the line number and its enclosing class. Note that sometimes two source code entities cannot be distinguished, for example, two casts to the same type at the same source code line; as discussed in Section 5.4, this requires a slightly different interpretation of client analyses, but is of no other consequence. For each benchmark program, a set of source code entities is maintained, and a new entity is added to this set when it is first used in the analysis result of a points-to analysis run.

Client analysis results are modeled as tuples of source code entities. For instance, a call graph node consists of a single method, an object call graph

\footnote{see \url{http://maven.apache.org} for details}
7.3 The XML Exchange Format

In the following, we describe the concepts of the file formats; both complete file formats, as documented .XSD specifications and with examples, can be found on [http://p2abench.sf.net](http://p2abench.sf.net).

Each file is divided into two parts: The header and the body. The main purpose of the header is to specify which client analyses are contained in the file. Additionally, the name of the analyzed benchmark program and the used name filters are contained as well. The benchmarking platform validates that the latter two match the configuration on the platform, in order to make sure that correct data is uploaded. The header may contain a few optional elements: the analysis setup description and the analysis kind (optimistic, conservative, or general). If not present, these are specified by the user via a web form when submitting the results to the benchmarking platform.

The body contains the actual client analyses results. Source code entities are specified either as complex types or attributes. An example of the former are allocation sites, which are defined through their type, enclosing class, and line number. An example of the latter are methods, which are defined through their bytecode signature.

The formats are intended to be as nonrestrictive as possible to make implementing export adapters for different points-to analyses as simple as possible. There is no need to list the different client analyses in a specific order. There is also no need to take care not to define any element of any client analysis result set multiple times. This allows users, for instance, to traverse the points-to data structures and write all relevant information to the file. The benchmarking platform then sorts out duplicates and also deals with certain specifics of the client analyses, e.g., removing calls to static initializers.

Allocation sites as the most often used source code entity can be specified in two ways: As an explicit list, or as a reference to a predefined set of allocation sites (for reusability of common sets, intended for writing results for static analysis). Sets of allocation sites must be defined prior to being referenced.

Each xml element describes one or more elements of the result sets of the client analyses, where the format is specific to each client analysis. Details can be found on [http://p2abench.sf.net](http://p2abench.sf.net).

Note that if an analysis does not support the object call graph client analysis but only the call graph client analysis, then this is indicated in the header in the first place, and the elements indicating abstract objects are simply omitted from the body.

7.2.2 Implementation Details

Our benchmarking platform is developed as a JavaEE application. We have chosen JBoss Application Server 7 as application server, hibernate as persistence provider, and MySQL as backing database. For the presentation layer, Java Server Faces (JSF)5, with the additional usage of some RichFaces components, have been used.

The choice has been made by the simple fact that we are familiar with these technologies. In case that any of these seem unfit for the platform in the future, they can be easily replaced due to the high level of standardization of JavaEE.

7.3 The XML Exchange Format

Client analysis results are uploaded in a specific XML file format, or rather, two different versions thereof. One version is targeted for use by static points-to analysis and includes concepts such as contexts for easy dumping of points-to information: The XML file export code can be written to traverse one context (or reachable method in case of context-insensitive analysis) at a time; when writing the results file, calls do not have to be followed, so all information for the given context/method can be written at the same time. The other version is mainly targeted at being used by dynamic analysis, where execution naturally interrupts at method calls and the same bundling is not possible.

---

2http://www.jboss.org
3http://www.hibernate.org/
4http://www.mysql.com
5http://javaserverfaces.java.net/
6http://www.jboss.org/richfaces
7.3. The XML Exchange Format

In the following, we describe the concepts of the file formats; both complete file formats, as documented .xsd specifications and with examples, can be found on http://p2abench.sf.net.

Each file is divided into two parts: The header and the body. The main purpose of the header is to specify which client analyses are contained in the file. Additionally, the name of the analyzed benchmark program and the used name filters are contained as well. The benchmarking platform validates that the latter two match the configuration on the platform, in order to make sure that correct data is uploaded. The header may contain a few optional elements: the analysis setup description and the analysis kind (optimistic, conservative, or general). If not present, these are specified by the user via a web form when submitting the results to the benchmarking platform.

The body contains the actual client analyses results. Source code entities are specified either as complex types or attributes. An example of the former are allocation sites, which are defined through their type, enclosing class, and line number. An example of the latter are methods, which are defined through their bytecode signature.

The formats are intended to be as nonrestrictive as possible to make implementing export adapters for different points-to analyses as simple as possible. There is no need to list the different client analyses in a specific order. There is also no need to take care not to define any element of any client analysis result set multiple times. This allows users, for instance, to traverse the points-to data structures and write all relevant information to the file. The benchmarking platform then sorts out duplicates and also deals with certain specifics of the client analyses, e.g., removing calls to static initializers.

Allocation sites as the most often used source code entity can be specified in two ways: As an explicit list, or as a reference to a predefined set of allocation sites (for reusability of common sets, intended for used when writing results for static analysis). Sets of allocation sites must be defined prior to being referenced.

Each xml element describes one or more elements of the result sets of the client analyses, where the format is specific to each client analysis. Details can be found on http://p2abench.sf.net.

Note that if an analysis does not support the object call graph client analysis but only the call graph client analysis, then this is indicated in the header in the first place, and the elements indicating abstract objects are simply omitted from the body.
Chapter 7. Benchmarking Platform

Figure 7.1: Presentation of comparison results: The raw metrics values are presented, and the user can select to show the false negatives (“missing”) of either of the static analyses. The static analyses can be compared qualitatively as well.

7.4 Conclusion

In this chapter, we have presented a benchmarking platform that supports the creation of benchmark suites for points-to analysis, their application, and the process to create a Gold Standard for points-to analysis as presented in Chapter 6.

The code for the benchmarking platform is accessible through http://p2abench.sf.net, where a link to a publicly accessible instantiation can also be found.

Figure 7.1 shows how the comparison of two static analysis (including comparison to dynamic analysis) is presented on the benchmarking platform: The metrics values for dynamic as well as static analysis are presented, and the number of identified false negatives are shown as well. The user can choose to see the identified false negatives (“misses”), or see a qualitative comparison of the two static analysis results.
In this chapter, we evaluate those goal criteria that can be evaluated experimentally. In particular, we show the practicability of our benchmarking methodology (goal criterion 3) by comparing three different points-to analysis implementations experimentally, in different setups, and we show the feasibility of our process for creating a Gold Standard (goal criterion 6) by showing a first attempt towards a Gold Standard. We will also exemplify the usability and extensibility of our benchmarking platform, which is part of goal criteria 2 and 5 (“efficiency”). The other goal criteria cannot be evaluated experimentally and will be discussed in the next chapter.

We also show experimental results that strengthen the motivation for this thesis by showing that our theoretical considerations are relevant in praxis. In particular, we experimentally confirm our theoretical considerations from Chapter 4, which state that general analyses cannot be compared experimentally in the absence of a Gold Standard (cf. Section 4.1) and that it is not possible to evaluate the impact of a sound improvement to points-to analysis based on a general baseline analysis (cf. Section 4.2). We also show experimental results that different approximations of the F-Score ($\tilde{F}$) through different under-approximations of the Gold Standard ($G^-$) can lead to different order in terms of accuracy of the same set of points-to analyses. That is, we show that conservative analysis or a Gold Standard is required to compare the accuracies of points-to analyses, even in praxis.

This chapter is organized as follows:

In Section 8.1, we list the benchmark programs that we use in the remainder of this chapter.

In Section 8.2, we show that different approximations of the Gold Standard lead to different accuracy assessments when analyzing general analysis. That is, we show that approximations of the Gold Standard are not good enough to assess accuracy of general analyses, confirming our theoretical considerations from Section 4.1. As a side effect, we also show that the dynamic coverage of benchmark programs in performance benchmark suites can be improved a lot.

In Section 8.3, we investigate whether improvements to a general baseline analysis do, in praxis, result in worse recall. This is to confirm our theoretical
considerations from Chapter 4.2. For this, we compare: a) the numbers of false negatives of results from Paddle in a context-insensitive setup with Paddle in an object-sensitive setup, and b) the numbers of false negatives of results from P2SSA in a context- and path-insensitive setup with P2SSA in an object- and path-sensitive setup.

Having established that accuracy of general analysis cannot be assessed properly without a Gold Standard, we look, in the remainder of the chapter, at conservative static analyses only.

In Section 8.4, we investigate whether our benchmarking methodology works in practice. Theoretically expected results should thus show in praxis, i.e., a theoretically more precise analysis should produce better benchmark results than a theoretically less precise analysis, whereas theoretically equally precise analyses should produce equal results. For this, we compare the benchmark results of Spark (context-insensitive), Paddle (context-insensitive and object-sensitive) and P2SSA (context-insensitive and object-sensitive). We expect Spark and Paddle in a context-insensitive setup to be equally precise as they are otherwise both field-sensitive and flow-insensitive, whereas P2SSA (in a context-insensitive setup) should be more precise as it is flow-sensitive. For the object-sensitive setups, we expect P2SSA to be more precise than Paddle for the same reason.

In Section 8.5, we investigate the impact of using different Java standard library classes for analyzing otherwise identical benchmark programs, both with dynamic and static analysis. This experiment shall confirm our reasoning that it is important to document the Java standard library versions used with benchmark programs (as we have seen in Section 3.4, this is not always done in literature).

In Section 8.6, we evaluate our custom-made dynamic analysis approach that we have presented in Section 6.4.

In Section 8.7, we present a first attempt towards a Gold Standard based on the process and the enhancements presented throughout Chapter 6.

Finally, in Section 8.8, we exemplify the extensibility of the benchmarking platform that was presented in Chapter 7. We conclude this chapter in Section 8.9.

### 8.1 Benchmark Programs

We have chosen benchmark programs from two sources: First, we have used programs that have been used frequently by other authors, in particular programs that are part of (performance) benchmark suites. We have used the same versions of the programs as can be found in such suites, although we did not use the test harnesses from those suites but instead used main()-methods as entry points to the programs. The reason for this is that the
harnesses either do not provide \texttt{main()}-methods or, in cases where this is the case, input is statically limited to the benchmark test input so that better dynamic coverage cannot be achieved. Sometimes, benchmark suites use frameworks without explicit \texttt{main()}-methods; instead, code utilizing the core features of the framework is provided in the test harness. We did not include such benchmarks in our selection. We used versions downloaded from the programs’ homepages instead of the code provided in the benchmark suites, as there the code may be modified compared to its original version.

As a second source of benchmark programs, we have used programs that are well-known to us. Table 8.1 lists the benchmark programs used in this chapter. For each benchmark program, we list its version, homepage, entry point used for analysis, which JDK version we used for analysis, and application name filters, as well as what motivated us to use the program. The programs are ordered so that the first five can be analyzed conservatively by all the used points-to analyses, whereas the next one only by P2SSA (Spark and Paddle lack support for one required native method). The last five programs cannot be analyzed conservatively by any of the utilized points-to analyses.

8.2 Approximating $\bar{F}$ with help of different $G^-$

In this section, we show that approximating $\bar{F}$ does not yield conclusive results. For this, we take two under-approximations of a Gold Standard and compute $\bar{F}$ for the different client metrics.

In order to assure practical relevance, we do not manually create the approximations of the Gold Standard (in order to confirm our theories, we could just create an initial $G^-$ and derive $G^{--}$ by removing all known true positives, thus reducing recall to 0). Instead, we use two under-approximations of the Gold Standard that we have created in the following ways: For the first one, we took dynamic input for the programs that were also used in performance benchmark suites. For the second one, we created additional input for dynamic analysis as described in Section 5.3, e.g., we varied command line options and used examples that had been provided along with the programs. As a side effect, we also show that dynamic coverage of performance benchmark suites can be improved with little effort.

First, we quantify the size of the two approximations of the Gold Standard. The initial, absolute metrics values, as well as the respective relative increased values, are listed in Table 8.2. The increases in percent are listed in the second row of each benchmark program.

The number of reachable methods, i.e., metric $N$, increased by 11.8% (antlr) to 37.6% (jython). For the other metrics, the relative increase is always above 13.0% and reaches as high as 189.8% (OE, pmd). There are
Approximating $\hat{\mathcal{F}}$ with help of different $G$-program.

Table 8.2: Metrics Results of Two Approximations of the Gold Standard.

Many reasons for pmd's strong increases. One of them is that pmd can render its results in different formats, e.g., plain text, html, xml, and more, but the benchmark suite uses the plain text renderer only.

We conclude that our findings agree with the ones by Brown et al. [21]: Dynamic coverage of performance benchmark suites is unsatisfying for the comparative evaluation of static analysis. Note that this is by no means criticism of the performance benchmark suites, as they intend to capture typical program usage. However, for evaluation of static analysis in general and points-to analysis in particular, a dedicated benchmark suite for the task containing very good dynamic coverage is desirable.

Now, in order to assess whether assessing general analyses with help of under-approximations of the Gold Standard might be permissible, we compare approximated accuracies $\hat{\mathcal{F}}$ for the metric with both the initial and the final input. These results are listed in Table 8.3. Looking at this metric is sufficient for our purposes, so the results for metrics $E, ON, OE, H$ are listed for informative purposes in Tables B.6 and B.7 in Appendix B.

There are, in fact, a few cases where $\hat{\mathcal{F}}_f$ is lower than $\hat{\mathcal{F}}_i$ (in bold in the table), whereas it is the other way around otherwise. This is, however, not consistent for either points-to analysis or benchmark program. Additionally, Spark, which is expected (and experimentally proven in Section 8.4) to be less precise for conservative analysis than P2SSA, has, for example for antlr, a better $\hat{\mathcal{F}}$-score with the second approximation of the Gold Standard, reversing the results compared to the first approximation of the Gold Standard. Also noteworthy is that Paddle, for project pmd, has a worse $\hat{\mathcal{F}}$ score with object sensitivity than with context insensitivity, but only with the second approximation of the Gold Standard.

We can thus conclude that different under-approximations of the Gold Standard may yield different accuracy assessments, so no statement can be made about whether or not a better approximation of the Gold Standard actually yields a better approximation of the accuracy.
8.2. Approximating $\tilde{F}$ with help of different $G^-\tilde{F}$

<table>
<thead>
<tr>
<th>program</th>
<th>$N$</th>
<th>$E$</th>
<th>$ON$</th>
<th>$OE$</th>
<th>$H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>1104 +130</td>
<td>3730 +597</td>
<td>2202 +286</td>
<td>5662 +912</td>
<td>652 +87</td>
</tr>
<tr>
<td></td>
<td>11.8%</td>
<td>16.0%</td>
<td>13.0%</td>
<td>16.1%</td>
<td>13.3%</td>
</tr>
<tr>
<td>jython</td>
<td>2081 +783</td>
<td>3254 +2552</td>
<td>7450 +3378</td>
<td>7089 +5334</td>
<td>3307 +1689</td>
</tr>
<tr>
<td></td>
<td>37.6%</td>
<td>78.4%</td>
<td>45.3%</td>
<td>75.2%</td>
<td>51.1%</td>
</tr>
<tr>
<td>lucene</td>
<td>674 +172</td>
<td>758 +300</td>
<td>884 +269</td>
<td>891 +299</td>
<td>229 +83</td>
</tr>
<tr>
<td></td>
<td>25.5%</td>
<td>39.6%</td>
<td>30.4%</td>
<td>33.6%</td>
<td>36.2%</td>
</tr>
<tr>
<td>pmd</td>
<td>1643 +618</td>
<td>2863 +1303</td>
<td>4756 +6807</td>
<td>10038 +19050</td>
<td>562 +677</td>
</tr>
<tr>
<td></td>
<td>27.3%</td>
<td>45.5%</td>
<td>143.1%</td>
<td>189.8%</td>
<td>120.5%</td>
</tr>
</tbody>
</table>

Table 8.2: Metrics Results of Two Approximations of the Gold Standard.

many reasons for pmd’s strong increases. One of them is that pmd can render its results in different formats, e.g., plain text, html, xml, and more, but the benchmark suite uses the plain text renderer only.

We conclude that our findings agree with the ones by Brown et al. [21]: Dynamic coverage of performance benchmark suites is unsatisfying for the comparative evaluation of static analysis. Note that this is by no means criticism of the performance benchmark suites, as they intend to capture typical program usage. However, for evaluation of static analysis in general and points-to analysis in particular, a dedicated benchmark suite for the task containing very good dynamic coverage is desirable.

Now, in order to assess whether assessing general analyses with help of under-approximations of the Gold Standard might be permissible, we compare approximated accuracies $\tilde{F}$ for the metric $N$ with both the initial and the final input. These results are listed in Table 8.3. Looking at this metric is sufficient for our purposes, so the results for metrics $E, ON, OE,$ and $H$ are listed for informative purposes in Tables B.6 and B.7 in Appendix B.

There are, in fact, a few cases where $\tilde{F}_f$ is lower than $\tilde{F}_i$ (in bold in the table), whereas it is the other way around otherwise. This is, however, not consistent for either points-to analysis or benchmark program. Additionally, Spark, which is expected (and experimentally proven in Section 8.4) to be less precise for conservative analysis than P2SSA, has, for example for antlr, a better $\tilde{F}$-score with the second approximation of the Gold Standard, reversing the results compared to the first approximation of the Gold Standard. Also noteworthy is that Paddle, for project pmd, has a worse $\tilde{F}$ score with object sensitivity than with context insensitivity, but only with the second approximation of the Gold Standard.

We can thus conclude that different under-approximations of the Gold Standard may yield different accuracy assessments, so no statement can be made about whether or not a better approximation of the Gold Standard actually yields a better approximation of the accuracy.
8.3 Evaluating Improvements Based on General Analysis

In Chapter 4, we have stated that, from a theoretical perspective, a Gold Standard is required in order to compare the accuracy of general analysis. In particular, we have stated in Section 4.2 that improvements that are sound for conservative analysis must not be evaluated based on a general baseline analysis, as an optimization may not only optimize false positives, but even true positives away, i.e., leading to a larger number of false negatives. In this section, we investigate if this happens in praxis.

For this, we look at experimental results of the following benchmark programs, for which our analyses are general: antlr, javadoc, jython, lucene, pmd, and recoder. We analyzed these benchmark programs with P2SSA and Paddle context-insensitively. Then, for P2SSA, we compare the number of false negatives with results obtained through object-sensitive analysis and with filter operations for path-sensitivity. For Paddle, we compare the results with object-sensitive analysis. Note that for recoder, we disabled support for the native method `Arrays.newInstance()` in P2SSA, the method that Paddle also lacks support for in order to analyze this program conservatively.

The number of false negatives are listed in Tables 8.4 and 8.5 for P2SSA and Paddle, respectively.

For Paddle, the number of identified false negatives increases for each benchmark program with the exception of recoder. When looking into the reason for this for benchmark program lucene, we found one source for this: object-flow from unreachable methods are included to some extend in context-insensitive analysis, i.e., in context-insensitive analysis, Paddle does not exclude unreachable methods from analysis. In particular, the points-to set of a call to a non-reachable method `m()` with a return expression like `new T();` would contain this abstract object. Only in object-sensitive analysis.

<table>
<thead>
<tr>
<th>Program</th>
<th>P2SSA</th>
<th>Spark</th>
<th>Paddle</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>0.642</td>
<td>0.649</td>
<td>0.642</td>
</tr>
<tr>
<td>javadoc</td>
<td>0.642</td>
<td>0.649</td>
<td>0.642</td>
</tr>
<tr>
<td>jython</td>
<td>0.646</td>
<td>0.644</td>
<td>0.646</td>
</tr>
<tr>
<td>lucene</td>
<td>0.646</td>
<td>0.695</td>
<td>0.696</td>
</tr>
<tr>
<td>pmd</td>
<td>0.704</td>
<td>0.767</td>
<td>0.704</td>
</tr>
<tr>
<td>recoder</td>
<td>0.646</td>
<td>0.644</td>
<td>0.646</td>
</tr>
</tbody>
</table>

Table 8.3: \( \hat{T} \) scores computed with initial input (subscript i) and final input (subscript f) for metric \( N \).
8.3 Evaluating Improvements Based on General Analysis

In Chapter 4, we have stated that, from a theoretical perspective, a Gold Standard is required in order to compare the accuracy of general analysis. In particular, we have stated in Section 4.2 that improvements that are sound for conservative analysis must not be evaluated based on a general baseline analysis, as an optimization may not only optimize false positives, but even true positives away, i.e., leading to a larger number of false negatives. In this section, we investigate if this happens in praxis.

For this, we look at experimental results of the following benchmark programs, for which our analyses are general: antlr, javadoc, jython, lucene, pmd, and recoder. We analyzed these benchmark programs with P2SSA and Paddle context-insensitively. Then, for P2SSA, we compare the number of false negatives with results obtained through object-sensitive analysis and with filter operations for path-sensitivity. For Paddle, we compare the results with object-sensitive analysis. Note that for recoder, we disabled support for the native method `Arrays.newInstance()` in P2SSA, the method that Paddle also lacks support for in order to analyze this program conservatively.

The number of false negatives are listed in Tables 8.4 and 8.5 for P2SSA and Paddle, respectively.

For Paddle, the number of identified false negatives increases for each benchmark program with the exception of recoder. When looking into the reason for this for benchmark program lucene, we found one source for this: object-flow from unreachable methods are included to some extend in context-insensitive analysis, i.e., in context-insensitive analysis, Paddle does not exclude unreachable methods from analysis. In particular, the points-to set of a call to a non-reachable method `m()` with a return expression like `new T();` would contain this abstract object. Only in object-sensitive analysis

<table>
<thead>
<tr>
<th>program</th>
<th>N</th>
<th>E</th>
<th>ON</th>
<th>OE</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>612</td>
<td>-</td>
<td>2797</td>
<td>-</td>
<td>1169</td>
</tr>
<tr>
<td>javadoc</td>
<td>606</td>
<td>-</td>
<td>1365</td>
<td>-</td>
<td>1310</td>
</tr>
<tr>
<td>jython</td>
<td>1376</td>
<td>+1</td>
<td>2813</td>
<td>+4</td>
<td>8063</td>
</tr>
<tr>
<td>lucene</td>
<td>155</td>
<td>-</td>
<td>302</td>
<td>-</td>
<td>235</td>
</tr>
<tr>
<td>pmd</td>
<td>610</td>
<td>-</td>
<td>1284</td>
<td>-</td>
<td>5800</td>
</tr>
<tr>
<td>recoder</td>
<td>4</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>59</td>
</tr>
</tbody>
</table>

Table 8.4: False negatives in P2SSA. The first row for each metric denotes the absolute number for context-insensitive analysis and without filter operations; the second row for each metric indicates the relative results for object-sensitive analysis and with filter operations.

8.3 Evaluating Improvements Based on General Analysis

In Chapter 4, we have stated that, from a theoretical perspective, a Gold Standard is required in order to compare the accuracy of general analysis. In particular, we have stated in Section 4.2 that improvements that are sound for conservative analysis must not be evaluated based on a general baseline analysis, as an optimization may not only optimize false positives, but even true positives away, i.e., leading to a larger number of false negatives. In this section, we investigate if this happens in praxis.

For this, we look at experimental results of the following benchmark programs, for which our analyses are general: antlr, javadoc, jython, lucene, pmd, and recoder. We analyzed these benchmark programs with P2SSA and Paddle context-insensitively. Then, for P2SSA, we compare the number of false negatives with results obtained through object-sensitive analysis and with filter operations for path-sensitivity. For Paddle, we compare the results with object-sensitive analysis. Note that for recoder, we disabled support for the native method `Arrays.newInstance()` in P2SSA, the method that Paddle also lacks support for in order to analyze this program conservatively.

The number of false negatives are listed in Tables 8.4 and 8.5 for P2SSA and Paddle, respectively.

For Paddle, the number of identified false negatives increases for each benchmark program with the exception of recoder. When looking into the reason for this for benchmark program lucene, we found one source for this: object-flow from unreachable methods are included to some extend in context-insensitive analysis, i.e., in context-insensitive analysis, Paddle does not exclude unreachable methods from analysis. In particular, the points-to set of a call to a non-reachable method `m()` with a return expression like `new T();` would contain this abstract object. Only in object-sensitive analysis
Chapter 8. Experimental Evaluation

<table>
<thead>
<tr>
<th>program</th>
<th>N</th>
<th>E</th>
<th>ON</th>
<th>OE</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>602</td>
<td>+10</td>
<td>2788</td>
<td>+5</td>
<td>1150</td>
</tr>
<tr>
<td>javadoc</td>
<td>601</td>
<td>+5</td>
<td>1357</td>
<td>+8</td>
<td>1296</td>
</tr>
<tr>
<td>jython</td>
<td>1141</td>
<td>+234</td>
<td>2438</td>
<td>+375</td>
<td>7054</td>
</tr>
<tr>
<td>lucene</td>
<td>81</td>
<td>+73</td>
<td>187</td>
<td>+113</td>
<td>135</td>
</tr>
<tr>
<td>pmd</td>
<td>275</td>
<td>+333</td>
<td>771</td>
<td>+512</td>
<td>5313</td>
</tr>
<tr>
<td>recoder</td>
<td>4</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 8.5: False negatives in Paddle. The first row for each metric denotes the absolute number for context-insensitive analysis, the second row for each metric the relative results for object-sensitive analysis.

is it checked whether the points-to set of the implicit this-parameter to a method actually is not empty. Some of these methods – which are, in fact, reachable in concrete executions – return abstract objects that are no longer propagated to other places of the program in object. The actual optimization leading to additional false negatives is therefore not object-sensitivity in itself, but rather a form of dead code elimination. Dead code elimination is surely a valid optimization for conservative analysis, but this case shows that it can influence recall negatively in case of general analysis.

We conclude that evaluation of improvements to points-to analysis based on general analysis is questionable and should best be avoided. The practical experiments presented in this section confirmed our theoretical considerations from Section 4.2.

8.4 Comparing Points-to Analyses

We evaluate the usability of our benchmarking methodology by qualitatively comparing different points-to analysis implementations in different variations. We performed context-insensitive with all three points-to analysis implementations (P2SSA, Spark, Paddle), and object-sensitive analysis with P2SSA and Paddle. Both Paddle and P2SSA support a variety of additional setups. However, using the same context-sensitivity means that differences in analysis results must be explainable by the underlying analysis algorithms and the different sets of supported language features and native methods. If we then find differences in analysis results and can explain them with the implementation differences, we show that our benchmarking methodology works.

For the sake of readability, we have moved the complete metrics results to Appendix B. Each table contains, for each benchmark program, the metrics for dynamic analysis as well as for each static analysis setup and, where applicable, the number of identified false negatives (misses).

For starters, the benchmark programs antlr, javadoc, jython, lucene, and
pmd cannot be analyzed conservatively, so our benchmarking methodology does not work here. Additionally, Spark and Paddle cannot analyze recoder conservatively (cf. also the previous section). We do not consider these programs any further in this section.

In the following, we will pick up some of the numbers for the projects where the analyses are conservative and explain the reasons for the differences among them.

First, we expected the flow- and context-insensitive Spark and Paddle to have the same metrics results, as we were not aware of any feature in either of the two that would influence precision. However, Paddle is, most of the time, more precise than Spark (Spark is more precise only for lucene and metrics heap, which we will explain shortly). Looking for the reason for this, we found that Spark has a source of imprecision when it comes to handling of points-to sets of virtual calls. Assume a type T with method m() and a subtype U of T that overrides m(). Assume now a call e.m() where pt(e) = {o} with o being of type U. Spark now propagates o not only to U.m(), but even to T.m(), which – among other things – incorrectly marks T.m() as reachable. Paddle does not have this source of imprecision, which explains why the context-insensitive variant of Paddle is more precise than Spark.

The aforementioned exception where Spark has a smaller result set (lucene, metric H) is the same as in the previous section: An abstract object that “escapes” a non-reachable method. However, in this case it does not cascade unsound results, but simply introduces imprecision.

P2SSA is, both in context-insensitive as well as in object-sensitive setups, as expected more precise than Paddle and Spark. This is explained by flow-sensitivity: P2SSA can, due to flow-sensitivity, strictly distinguish between analyzing system initialization code and program code (code reachable from the main()-method). In particular, certain methods called on collection objects (toString(), equals(), hashCode()) may call back from the system initialization code into the application code. Usually, no application objects should be stored in any abstract collection object during system initialization, so calls to those methods should not call back into the application code. However, in flow-insensitive analysis, application objects end up stored in abstract collection objects that are used during system initialization. In turn, those method calls are analyzed again with these application objects, which does not happen in flow-sensitive analysis.

In conclusion, our benchmarking methodology presented in the previous chapter seems to work: Theoretically more precise analysis shows better analysis results, and the differences in theoretically equally precise analyses are explainable through implementation details. Note that the insights gained here should show possibilities to improve precision for both Paddle and Spark.
8.5 Impact of Using Different Java Standard Libraries

As we have seen in Section 3.4, many authors do not document what versions of the Java standard library they use for their evaluation. We have stated in Section 5.5 that this is important, as analysis results may be impacted.

We investigate in the following the impact on static as well as the impact on dynamic analysis.

8.5.1 Impact on Static Analysis

In order to evaluate the impact on client analyses when using different Java standard libraries, we have analyzed the benchmark programs that work with at least Java 1.4.2_19 (antlr, bloat, javac, javacc, javadoc, jython) statically with Java 1.3.1_28, 1.4.2_19, 1.5.0_22, 1.6.0_38, and 1.7.0_09 (javacc and jython cannot be analyzed with Java 1.3.1_28) and compared the results of the client analyses. For this experiment, we used Spark. In the following, we summarize the most significant differences that we observed in the metrics results.

In general, not a single metric has stable results for all used Java versions; however, the differences are often rather small (around .1 to .2%). The biggest difference for the metric N is observed for bloat, where the metrics results differ up to 1.1%. The biggest overall difference we have seen is for antlr, where the metric H differed up to 1.6%.

It is thus important to document not only the versions of programs, but also the Java runtime library and other libraries used in the analysis.

8.5.2 Impact on Dynamic results

For evaluating the impact of different Java versions on dynamic analysis results, we have analyzed all benchmark programs dynamically with Java 1.6.0_38 and 1.7.0_09. We could not use earlier versions as the dynamic analysis tool that we use is based on an API that was introduced with Java 1.6.

When running bloat dynamically with the two different Java versions, we were surprised to find rather relevant differences, for example, in the set of reachable methods. When examining those, we found that bloat, a bytecode optimization tool, loads classes from the execution environment (classes within rt.jar). Those classes differ so much that the actual runtime behavior changes. However, since we (unknowingly) implicitly changed the input to bloat, this change is not caused by using a different Java version for analysis. We could indeed use the gained insight to improve the dynamic input to running bloat for increasing the dynamic coverage, after which running dynamic
analysis with both Java versions was the same. For the other programs, there were no differences when using either Java 6 or Java 7 for dynamic analysis.

We conclude that it seems permissible to compare static analysis results obtained with one version of Java standard library classes with dynamic results obtained through another Java version. This is based on the observation that there are no differences in the client analysis between Java 6 and Java 7. The reason for this is that much care is and always has been taken to ensure compatibility between different Java versions.

### 8.6 Fast Dynamic Analysis

Our experimental setup is as follows: We run the dynamic analysis tool described in Appendix A in a version that only supports object call graph on a set of nine benchmark programs and compared the performance overhead to running the programs with generated custom-made instrumentation code. We chose programs for which our points-to analysis implementation is conservative, or we “told” the analysis what classes should be loaded at dynamic class loading points by manually changing the source code. For each of the benchmark programs, we generated three versions with custom-made instrumentation: Two versions with “switch”-statements, whereof one generated with the results of points-to analysis as input, one with “exact” results obtained by the general purpose analysis tool. The latter allows us to evaluate the intrinsic overhead of our approach, and we can make conclusions about the importance of the precision of the underlying static points-to analysis. The third version was generated with the “formula” approach.

All experiments were performed on a standard Desktop PC, Intel Core 2 Quad Q9550, 2.83Ghz, 4GB RAM, 32-bit Windows XP, JDK 1.6.0_24.

As entry points to the programs, we created special driver-classes that invoke the different programs’ `main`-methods several times with different input. We do this instead of running the program over and over again with different input because we want to evaluate the performance overhead on long-running programs where JVM optimizations can start kicking in; for short running programs, it would not really matter if a program runs, e.g., one or three seconds. All timing results are the average over three runs.

#### 8.6.1 Performance Overhead

Figure 8.1 shows the relative execution times of all setups as a factor to the execution time of the un-monitored setup. For projects `bloat` and `recoder`, the “formula” approach failed because the static analysis was too imprecise and the resulting formulas spread over such a large range that the result array
required too much memory for the instrumented programs to run on a 32-bit Java.

The average performance overhead for the nine benchmark programs that we investigated is reduced from factor 4.2 (general purpose analysis tool) to 2.5 (“switch” approach, 2.3 when using dynamic results for generating the instrumentation code) to 1.5 (“formula” approach), respectively. Note that the latter is possibly a little higher if recoder and bloat count as well. In all cases, the programs run with custom-made instrumentation code are faster than when using the general purpose analysis tool, with the “formula”-approach being the fastest.

Looking at extremes, bloat (factor 4.2) and recoder (3.6) suffer the biggest slowdown with custom-made instrumentation. Those two programs are very method-invocation intensive. Especially, getter and setter methods, which contain very little computations but are frequently called, become much slower due to instrumentation. jlayer experiences hardly any overhead with custom-made instrumentation. This program, an MPEG audio to WAV converter, contains a lot of numerical computations, which dominate the execution time of the program. Our analysis task does not require those computations to be instrumented. The search-engine lucene also runs very fast, because it contains many (non-instrumented) String- and File-operations. Finally, the very strong speed-up through custom-made instrumentation for antlr surprises us somewhat, something that needs further investigation.

When using the dynamic analysis results for generating the custom-made instrumentation (Figure 8.1, third data set), the number of switch- and corresponding case-statements is minimal but sufficient for our benchmark setup. The slowdown factor decreases then, on average, from 2.5 to 2.3. The precision of static analysis thus has a measurable effect on the performance overhead of custom-made instrumentation. However, the effect is not so big that the underlying static points-to analysis must be improved at any cost.

### 8.6.2 Memory and I/O Overhead

The I/O overhead, i.e., additional runtime for writing the dynamic analysis results to disk, is negligible for all benchmark programs that we used in the evaluation. The largest result file that we obtained was 21MB (recoder). Writing this to a file is done in less than two second on today’s computers. Further, this format is our XML file format, which is not at all optimized for space-efficient storage but aims at readability by humans, which allows for an even further reduction of the runtime overhead.

The memory overhead of our approach comes from two sources: The byte array that is kept in memory for storing the observed results, and the larger bytecode, which stems from the instrumentation.

We first look at the memory overhead for the “switch”-approach. The
8.6. Fast Dynamic Analysis

byte-array for each client analysis is as large as the size of the statically computed result. The worst case we observed for this was for *bloat*, with almost one million object call graph edges. This yielded one megabyte of memory. For measuring the memory overhead due to larger bytecode, we compared the size of each original project’s *.class*-files with the size of its instrumented class files. Again it is *bloat* which suffers from the largest memory-overhead: Originally 1.5 MB of *.class*-files become 16 MB in the instrumented code. In both cases, the absolute memory overhead is insignificant on modern computers.

The memory overhead for the “formula” approach depends on how tight the formulas distribute possible results. As mentioned before, this did not work for projects *bloat* and *recoder* – these projects required result arrays larger than allowed by a 32-bit Java. This can, of course, be fixed by improving the algorithm that computes the formulas. The other projects showed memory overhead between .5 and 200MB. The latter is, of course, still too high; again, with a better algorithm for finding the formulas, this should be possible to fix. The size of the instrumented code is increased much less for this approach as for the switch-approach and thus does not play a significant role here either.

Figure 8.1: Performance overhead for different setups, as factor to normal runtime.
8.7 A First Attempt Towards a Gold Standard

In this section, we evaluate our attempts towards a first attempt for a Gold Standard for points-to analysis. It also serves as an evaluation for the proposed improvements to points-to analysis presented in Section 6.3.

For evaluation, we use the six benchmark programs for which we have conservative analysis at hand: bloat, javac, javacc, jlayer, recoder, and sablecc.

The remainder of this section is organized as follows: In Section 8.7.1, we present our efforts to increasing the dynamic coverage of the benchmark programs. In Section 8.7.2, we present the effects of using the collection classes replacements. In Section 8.7.3, we present the effects of combining different analyses. In Section 8.7.4, we present the effects of automatic feedback. In Section 8.7.5, we present the effects of manual absence proofs.

Because P2SSA has proven a bit more precise than Paddle due to its flow sensitivity, all remaining experiments (with the exception of combining analysis results in Section 8.7.3) in this chapter have been performed with P2SSA, for which we also have implemented feedback-driven analysis and inlining of calls to methods of collection classes.

Note that the following sections present the results of an incremental process. That is, we have not performed the single steps in a particular order but have, instead, constantly investigated the differences between over- and under-approximations of reachable methods and attempted to either trigger their dynamic execution or provide an absence proof for the root source for why they are deemed reachable by static analysis. However, we present the results as if the steps were performed in the above-listed order. That is, we start with P2SSA in with filter operations for path sensitivity and object sensitivity as a basis, which is the most precise initial configuration.

Note that, as we had no success with providing manual absence proofs for either bloat or sablecc due to missing expert knowledge on the programs (cf. Section 8.7.5), we have not tried as hard as for the other programs to improve dynamic coverage. Later on, we have then focused our efforts on jlayer and recoder. The former of the two is the smallest program, so we deemed the chances to create a Gold Standard highest for this program. We have used recoder as it is a bigger program and because we possess the most expert knowledge for it, so we expected to achieve good overall improvements with manual feedback.

We will focus on the client analysis “call graph” and there especially the metric N (“reachable methods”), as it is also the most promising to create a Gold Standard for. It also seems most natural to focus on reachable methods first and try to create a Gold Standard for them, as improving this improves all other client analyses as well.
8.7. A First Attempt Towards a Gold Standard

8.7.1 Increasing Coverage

We now describe how we improved the dynamic coverage of the benchmark programs additionally to the measures described in Section 5.3. This was mostly done when trying to determine if certain methods are reachable or not, for which we studied the source code of the benchmark programs and, thus, obtained some expert knowledge on these programs.

For bloat, we discovered that bytecode compiled for Java 7 triggers the execution of a number of additional methods. This we actually found when investigating whether running dynamic analysis with different Java versions matters, as described in Section 8.5.2.

For javacc, we realized that its parser to read grammar specification files contains code to parse large parts of the Java language. Adding Java code to the appropriate places in the grammar files increased the number of dynamically reachable methods considerably.

For jlayer, we have added .mp1 and .mp2 as well as corrupted files, and we also manually edited files on a binary level to trigger certain exceptions in the code.

For recoder, we used a collection of 72 programs\(^1\) as input. We also created input to trigger execution of methods based on our expert knowledge.

For javac and sablecc, we have not been able to improve dynamic coverage after applying the measures described in Section 5.3.

The initial and final dynamic metrics results are shown in Table 8.6.

8.7.2 Collection Classes

Table 8.7 lists the metrics results (for call graph metrics only) before and after applying the collection replacement classes presented in Section 6.3.1, without (subscript \(r\)) and with (subscript \(ri\)) additionally inlining calls to methods of collection classes.

There is no effect on either javac or jlayer as both do not use the java collection classes. javac uses, in fact, its own set of collection classes. The four other benchmark programs benefit from the collection classes, while two of them (recoder, sablecc) even from inlining calls. The latter two are also those that benefit the most, with the number of reachable methods for sablecc being reduced by 4.4%.

Note that the collection replacement classes also reduce the required analysis time and work quite well for Paddle, cf. [36] for a more detailed evaluation.

\(^1\)http://softlang.uni-koblenz.de/explore-API-usage/
Table 8.6: Initial (subscript i) and final (subscript f) dynamic coverage for all metrics.

<table>
<thead>
<tr>
<th>Program</th>
<th>bloat</th>
<th>javac</th>
<th>javacc</th>
<th>jlayer</th>
<th>recoder</th>
<th>sablecc</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>946</td>
<td>1190</td>
<td>732</td>
<td>153</td>
<td>4311</td>
<td>1381</td>
</tr>
<tr>
<td>f</td>
<td>1408</td>
<td>1208</td>
<td>834</td>
<td>222</td>
<td>4356</td>
<td>1381</td>
</tr>
</tbody>
</table>

Table 8.7: Call graph nodes and edges: Dynamic results (subscript d), with initial points-to analysis results (i), after using replacement collection classes (r), and with additionally inlining calls to methods of collection classes (ri).

Table 8.8: Call graph nodes and edges: Dynamic results (subscript d), with initial points-to analysis results from the previous section (i), and final results after combining different results (f).

8.7.3 Combining Analysis Results

In order to evaluate the effect of combining analysis results, we run P2SSA with ThisSens, ObjSens, and CallString (cf. Section 2.2). We also run Paddle with a context-sensitive naming scheme. We run these experiments with and without collection classes replacement (as mentioned in Section 6.3.1, there is one design decision that does not strictly improve precision).

Table 8.8 lists the metrics results before and after combining the analysis results. There are a few improvements when combining analysis results. For bloat, we could trace this back to the fact that the collection replacement classes do not, in fact, strictly improve precision: With the replacement classes, one new method is deemed reachable; cf. Section 6.3.1 for an explanation to this. Combining the result thus makes the result more precise here. For sablecc, the same happened. Additionally, CallString proved more precise than the other context sensitivities, so combining its results with results from object-sensitive analysis improved the overall precision here. For recoder, it is the case that all three context-sensitive approaches indeed produce results where no result is a subset of the others. The otherwise...
8.7. A First Attempt Towards a Gold Standard

Table 8.7: Call graph nodes and edges: Dynamic results (subscript d), with initial points-to analysis results (i), after using replacement collection classes (r), and with additionally inlining calls to methods of collection classes (ri). A dash indicates that no improvement was made.

<table>
<thead>
<tr>
<th>Program</th>
<th>(N_d)</th>
<th>(N_i)</th>
<th>(N_r)</th>
<th>(E_d)</th>
<th>(E_i)</th>
<th>(E_r)</th>
<th>(E_{ri})</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>1639</td>
<td>2413</td>
<td>2410</td>
<td>5674</td>
<td>12040</td>
<td>11945</td>
<td>-</td>
</tr>
<tr>
<td>javac</td>
<td>1208</td>
<td>1460</td>
<td>-</td>
<td>3620</td>
<td>5860</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>javacc</td>
<td>834</td>
<td>1018</td>
<td>1015</td>
<td>1893</td>
<td>2654</td>
<td>2648</td>
<td>-</td>
</tr>
<tr>
<td>jlayer</td>
<td>222</td>
<td>231</td>
<td>-</td>
<td>315</td>
<td>334</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>recoder</td>
<td>4356</td>
<td>5120</td>
<td>5089</td>
<td>5056</td>
<td>13078</td>
<td>21719</td>
<td>20682</td>
</tr>
<tr>
<td>sablecc</td>
<td>1381</td>
<td>1763</td>
<td>1697</td>
<td>1686</td>
<td>2595</td>
<td>5329</td>
<td>4922</td>
</tr>
</tbody>
</table>

Table 8.8: Call graph nodes and edges: Dynamic results (subscript d), with initial points-to analysis results from the previous section (i), and final results after combining different results (f).

<table>
<thead>
<tr>
<th>Program</th>
<th>(N_d)</th>
<th>(N_i)</th>
<th>(N_f)</th>
<th>(E_d)</th>
<th>(E_i)</th>
<th>(E_f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>1639</td>
<td>2410</td>
<td>2409</td>
<td>5674</td>
<td>11945</td>
<td>11900</td>
</tr>
<tr>
<td>javac</td>
<td>1208</td>
<td>1460</td>
<td>-</td>
<td>3278</td>
<td>5860</td>
<td>-</td>
</tr>
<tr>
<td>javacc</td>
<td>834</td>
<td>1015</td>
<td>-</td>
<td>1893</td>
<td>2648</td>
<td>-</td>
</tr>
<tr>
<td>jlayer</td>
<td>222</td>
<td>231</td>
<td>-</td>
<td>315</td>
<td>334</td>
<td>-</td>
</tr>
<tr>
<td>recoder</td>
<td>4356</td>
<td>5056</td>
<td>5055</td>
<td>13078</td>
<td>20515</td>
<td>20489</td>
</tr>
<tr>
<td>sablecc</td>
<td>1381</td>
<td>1686</td>
<td>1684</td>
<td>2595</td>
<td>4864</td>
<td>4838</td>
</tr>
</tbody>
</table>

8.7.3 Combining Analysis Results

In order to evaluate the effect of combining analysis results, we run P2SSA with ThisSens, ObjSens, and CallString (cf. Section 2.2). We also run Paddle with a context-sensitive naming scheme. We run these experiments with and without collection classes replacement (as mentioned in Section 6.3.1, there is one design decision that does not strictly improve precision).

Table 8.8 lists the metrics results before and after combining the analysis results. There are a few improvements when combining analysis results.

For bloat, we could trace this back to the fact that the collection replacement classes do not, in fact, strictly improve precision: With the replacement classes, one new method is deemed reachable; cf. Section 6.3.1 for an explanation to this. Combining the result thus makes the result more precise here.

For sablecc, the same happened. Additionally, CallString proved more precise than the other context sensitivities, so combining its results with results from object-sensitive analysis improved the overall precision here.

For recoder, it is the case that all three context-sensitive approaches indeed produce results where no result is a subset of the others. The otherwise
Chapter 8. Experimental Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>$N_d$</th>
<th>$N_i$</th>
<th>$N_f$</th>
<th>$E_d$</th>
<th>$E_i$</th>
<th>$E_f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac</td>
<td>1208</td>
<td>1460</td>
<td>1395</td>
<td>3620</td>
<td>5860</td>
<td>5571</td>
</tr>
<tr>
<td>jlayer</td>
<td>222</td>
<td>231</td>
<td>222</td>
<td>315</td>
<td>334</td>
<td>321</td>
</tr>
<tr>
<td>recoder</td>
<td>4356</td>
<td>5055</td>
<td>5025</td>
<td>13078</td>
<td>20489</td>
<td>19603</td>
</tr>
</tbody>
</table>

Table 8.9: Call graph nodes and edges: Dynamic results (subscript $d$), with initial points-to analysis results from the previous section ($i$), and final results after providing manual feedback ($f$).

observed effect of the collection framework replacement classes is not observed here.

In summary, combining analysis results does not lead – at least in our setting – to strong precision improvements. Note that there are some small improvements for the more fine-grained client analyses due to combining analysis results, but as those are not of immediate interest to creating a Gold Standard, we do not list them here explicitly.

### 8.7.4 Automatic Feedback

For automated feedback, we run P2SSA initially configured with ObjSens and then run ThisSens, CallString, and then re-run ObjSens, etc. with the previous results as input to automatic feedback until a fixed point is reached. We have seen some improvements for the client analyses heap and object call graph, but not for the client analysis call graph, which we currently focus on, so we do not list analysis results here.

### 8.7.5 Manual Feedback

Table 8.9 lists the metrics results before and after providing manual feedback.

For bloat, javacc, and sablecc, we were not able to find any manual feedback after spending some hours looking at the code for all of them. We then decided that spending time on the projects where we did succeed was more worth it.

We found that javac contains code for serializing its AST, e.g., for writing source code back to the file system. This can be triggered by setting a certain command line flag; however, the command line parser of the `main()` method does not allow the flag to be set in the first place. Instead, the code is intended for use of javac as a framework, not a standalone tool. Thus, we can ensure that this is dead code. We also found some smaller improvements, but they did not improve on precision as much. Manual feedback increased $P^−$ for metric $N$ from .827 to .866.

For jlayer, we have been able to compute the exact set of reachable methods, i.e., its Gold Standard for part of client analysis “call graph”. There are
also only six “call graph edges” in which the final dynamic and static analysis results differ; five of those turn out to be related to exception handling and would be triggered by appropriate IO-exceptions; we are confident that these can be triggered as well, but this would require removal of access to files that are already opened by the program, which we have not yet found the means to do. Overall, it turned out that creating proper dynamic input was the major part of this achievement, cf. Section 8.7.1, and only 9 methods had to be removed from the static analysis result set.

We used recoder for our kickoff study for manual feedback, cf. Section 6.3.3.2. Despite the fact that we spent most time on this program, $P^-$ for metric $N$ increased only from .862 to .867.

8.7.6 Conclusion

We have succeeded in computing the exact set of reachable methods for jlayer. This was mostly done by increasing dynamic input (for example, plus 45.1% reachable methods), but we also had to provide manual absence proofs for nine methods. For the other benchmark programs used in this chapter, we have better approximated the Gold Standard by either improving dynamic analysis, improving static analysis, providing manual absence proofs, or a combination of those measures.

It turns out that improving dynamic analysis yields the biggest effect – improving the results by 5 to 45% reachable methods is something that the measures for improving static analysis do not achieve together. However, in order to create a Gold Standard, all measures that improve static analysis precision are relevant. An exception here is perhaps automated feedback, which in itself contributes little to improved precision. However, its main benefit lies elsewhere, namely in speeding up the iterative process of manual feedback. Manual feedback requires human interaction and is done iteratively, so it is desirable to use a fast and inexpensive analysis for evaluating the impact of single manual absence proofs. However, with the help of automatic feedback, upper bounds based on the most precise (and likely most expensive) available analysis can be set and thus be taken into consideration when evaluating these manual absence proofs.

8.8 Evaluating the Benchmarking Platform

In this section, we evaluate different aspects of the benchmarking platform. In Section 8.8.1, we discuss how the benchmarking platform can reduce the work load for comparing points-to analyses experimentally and for creating a Gold Standard for points-to analysis. In Section 8.8.2, we assess how much work is required to connect a new points-to analysis implementation to the
benchmarking platform. Finally, in Section 8.8.3, we assess the amount of work required for adding a new client analysis to the benchmarking platform.

8.8.1 Reducing Work Load for Performing Experiments

Using the benchmarking platform can reduce the time effort for comparing different points-to analysis implementations significantly. The following points speak strongly for this statement: Once a new points-to analysis implementation is connected to the benchmarking platform, it can be instantly compared to all other points-to analysis implementations already connected, provided that they a) implement a common subset of client analyses and b) analysis results for the same benchmark programs are available. We have already connected three points-to analysis implementations to the benchmarking platform that implement three common client analyses (five metrics). Since call graph construction is a natural client analysis of points-to analysis, condition a) is very easily fulfilled. Condition b) is also rather easily fulfilled, as the benchmarking platform allows for uploading the code for the used benchmark programs, so it is simple to download those programs and perform the analysis for the same benchmark programs.

Also, the dynamic analysis results already submitted to the benchmarking platform become available for use by researchers at no charge. That is, researchers do not need to implement an own dynamic analysis tool and perform experiments on their own.

8.8.2 Writing XML file format export code

The XML export code for P2SSA is 430 lines long, including comments. It was rather easy to write, considering that we have very good knowledge of the internals of this points-to analysis implementation.

The XML export code for Spark and Paddle shares some code and is, in total, 956 lines long. It was more difficult to write this code, as we are not familiar with the internals of these points-to analysis implementations and thus initially did not know how to access the internal data structures – the public interfaces of the Soot framework, on which both of these points-to analyses are based, do not allow access to abstract objects, only to the call graph. However, an initial prototype of the final version of the Spark extractor for Spark was written within a day’s work, while the Paddle extractor took two days to implement and required a number of subsequent bug fixes until it started working reliably.

However, the general case should be that the maintainers of a points-to analysis write this code, so we do not expect it to take more than one or two working days. Also, a fully functional testing environment exists with the benchmarking platform, and reference implementations with extractors
for P2SSA, Spark, and Paddle that should make an implementation rather straightforward.

### 8.8.3 Adding a Client Analysis

In order to evaluate the effort required for adding a new client analysis, we defined the client analysis “escape analysis” and measured the required implementation effort in the benchmarking platform:

**Definition** For a given program $P$, let $M$ be the set of all methods in $P$ and $O$ be the set of all object creation sites in $P$. Let then the **exact set of escaping objects** consist of all the tuples $(o_i, m_j) \in O \times M$ where there exists a concrete execution of $P$ so that (1) $o_i$ is allocated in $m_j$ and, after returning from $m_j$, $o_i$ cannot be garbage-collected.

The **exact set of application escaping object** is the subset of the exact set of escaping objects restricted to abstract application objects created in application methods.

Adding this new client analysis to the comparison platform requires adding new entity table to the database, an extension of the .xsd file format specification (as well as corresponding parsers), and changes in the presentation layer of the benchmarking platform. Implementing the additional dynamic and static analyses is not part of extending the platform and thus not evaluated here.

It turns out that the extensibility of the benchmarking platform made this task rather easy: The implementation was done in three hours. This measured effort is of course subjective due to the fact that a single developer who already knew the platform implementation details performed this task. However, it should give an idea of the flexibility of the benchmarking platform.

### 8.9 Conclusion

In this chapter, we have shown experimental evidence that evaluating general analysis based on under-approximations of a Gold Standard is impossible not only in theory, but even in praxis. It can happen that different approximations of the Gold Standard yield different orders of approximated accuracy ($\hat{F}$) for different points-to analyses. We have also shown that similar holds when evaluating an improvement to points-to analysis based on a general baseline analysis. Thus, our motivation for a benchmarking methodology based on conservative analysis is not only of theoretical nature, but has practical relevance.

We have then shown that our proposed benchmarking methodology, which is based on applying client analyses only on those parts of programs that
can be analyzed conservatively, is able to show differences between different points-to analyses. The observed order of precision is either as expected from theoretical considerations (e.g., flow-sensitive analysis is more precise than flow-insensitive analysis), or the results can be explained by looking at implementation details, as in the case of Spark and Paddle (in context-insensitive setup), which we expected to be equal in precision based on theory, but which do not have a fixed order of precision in praxis. This evaluation confirms that our benchmarking methodology is practicable and thus fulfills our goal criterion 3, practicability.

Then, we have also presented a first attempt towards a Gold Standard by improving both static and dynamic analysis results. For one benchmark program (jlayer) and one metric (N), we have been able to compute the Gold Standard. Considering that we have done this in a small group, we can expect that creating a Gold Standard for more benchmark programs and metrics is possible based on our proposed process; thus, the process is feasible and thus fulfills our goal criterion 6, feasibility.

Both benchmarking and creating a Gold Standard are supported through our benchmarking platform. We have used the platform for evaluating the experiments presented in this chapter, and we have also exemplified the extensibility of the benchmarking platform (both for new client analyses and points-to analysis implementations). Thus, our goal criteria 2 and 5, efficiency, are addressed. Additionally, we have shown that the overhead of dynamic analysis can be reduced significantly based on static analysis results. When incrementally improving dynamic analysis results, this also improves the efficiency of the researcher, as less time for waiting for the results is required. This also improves the efficiency for creating a Gold Standard and thus also addresses goal criterion 5.
Chapter 9

Conclusion and Future Work

9.1 Conclusion

The motivation for this thesis is to make points-to analyses comparable in terms of accuracy. In Section 3.4, we have surveyed evaluation methods used in research in the field of points-to analysis for Java and found that different researchers not only use different benchmark programs and client analyses, they seldom compare their experimental results to third-party implementations. Thus, it is often hard to tell which of two points-to analyses presented in different publications by different research groups is actually more accurate.

Further, implementations often support only subsets of programming languages, e.g., dynamic class loading and native methods are often not at all, or not fully, supported. We show in Chapter 4 that comparing general analyses – i.e., static analyses that are not conservative – cannot be compared without a Gold Standard (a set of benchmark programs with exact analysis results) in theory. More specifically, we show that improvements to points-to analysis that are sound in themselves cannot be evaluated based on a general baseline analysis, as it is unclear if an improvement “optimizes away” good results; we also present experimental proof for this in Section 8.3.

In Chapter 5, we present a methodology to benchmark points-to analysis. This methodology is based on the observation that sub-parts of programs can quite often be analyzed conservatively. In particular, we suggest omitting the evaluation of system initialization, which always contains dynamic class loading and calls many native methods, and focusing on application code. Then, sub-parts of programs that do not use dynamic class loading and call no or few native methods can be analyzed conservatively by many points-to analyses, which allows researchers to compare them experimentally. In Section 8.4, we used this methodology for comparing results from three different points-to analysis implementations. Due to the fact that we found differences in analysis results as expected by theory (or, where this was not the case, we could trace differences back to implementation details), we conclude that our methodology is applicable in praxis to make points-to analyses experimentally comparable.
9.1. Conclusion

Other points-to analysis implementations are already connected to it. We have already connected three points-to analysis implementations to it, and we maintain a public instantiation of the benchmarking platform where both dynamic and static analysis results are available, so efficiency (goal criterion 2) is also fulfilled. Finally, our benchmarking methodology is based on evaluating only parts of the benchmark programs that can be analyzed conservatively. Thus, we worked around our theoretical and practical findings, which state that results of general analyses cannot be used for assessing accuracy without a Gold Standard. The benchmarking methodology is thus theoretically founded, and we consider goal criterion 1 as fulfilled. Overall, we consider our first goal as fulfilled.

The criteria for our second goal, showing how to create a Gold Standard for points-to analysis in order to assess the exact accuracy of points-to analysis, are as follows:

4. Distribution: We define a converging process that allows researchers to collaborate on working towards a Gold Standard for points-to analysis.
5. Efficiency: We provide tool support for applying this distributed process.
6. Feasibility: We show the feasibility of the process by presenting a first attempt towards a Gold Standard.

We have presented a process to create a Gold Standard that allows researchers to work together, and we have used this process to create a first attempt towards a Gold Standard (as presented in the previous chapter), thus fulfilling goal criterion 6, feasibility. The process has not been tested with a large amount of people yet; however, users can work independently on the iterative steps (improving dynamic and static analysis) and only need to synchronize their findings and communicate their current efforts (otherwise, unnecessary double work may occur), so we created the prerequisites for the process to be scalable to a large group of people. However, only future use of the platform can show whether goal criterion 4 is indeed fulfilled. The process is supported through our benchmarking platform, which computes the current best effort towards a Gold Standard and shows the discrepancies between under- and over-approximations, and it also allows to manage manual absence proofs; thus, we consider goal criterion 5, efficiency, as fulfilled.

Overall, we consider our second goal as fulfilled.

In Chapter 6, we discuss the need for a Gold Standard, with which the exact accuracy of points-to analysis can be computed. Creating such a Gold Standard cannot be computed automatically and thus relies on manual feedback based on expert knowledge about programs. We present a process to create such a Gold Standard that is based on incrementally improving both static and dynamic analysis, and we also present a first attempt towards it. For one benchmark program, we computed the exact set of reachable methods, and in general, we improved static and dynamic analysis. Much work remains in order to create a Gold Standard for both more client analyses and benchmark programs, but this was a first step taken by a small group of researchers, and our belief is that the research community must work on this together to ease the workload.

We also present a Web-based benchmarking platform in Chapter 7. This platform aids in applying our benchmarking methodology, and also the process for creating a Gold Standard. A points-to analysis can be connected to this platform by implementing some of the client analyses presented in Section 5.4 and exporting results in a specific XML-based file format. The benchmarking platform acts as a database for points-to analysis results. We also provide a public instantiation of the platform, where we also have submitted results from three points-to analysis implementations in different setups. Therefore, by connecting one’s own points-to analysis implementation, comparison to these points-to analysis and validation with the help of dynamic analysis results can be done at no charge.

9.1.1 Goal Criteria revisited

We now review the goal criteria for our two goals.

The criteria for our first goal, to make points-to analyses comparable, are as follows:

1. Theoretical foundation: We define a theoretically-founded benchmarking methodology for assessing the accuracy of points-to analysis.
2. Efficiency: We provide tool support for applying this benchmarking methodology.
3. Practicability: We show the practicability of the benchmarking methodology by presenting an experimental comparison of at least two existing, fundamentally different points-to analysis implementations.

We have experimentally shown in the previous chapter that our benchmarking methodology is practicable, thus fulfilling goal criterion 3. We have also presented tool support in form of our benchmarking platform: once a points-to analysis is connected to it, it can be experimentally compared to all
other points-to analysis implementations already connected to it. We already connected three points-to analysis implementations to it, and we maintain a public instantiation of the benchmarking platform where both dynamic and static analysis results are available, so efficiency (goal criterion 2) is also fulfilled. Finally, our benchmarking methodology is based on evaluating only parts of the benchmark programs that can be analyzed conservatively. Thus, we worked around our theoretical and practical findings, which state that results of general analyses cannot be used for assessing accuracy without a Gold Standard. The benchmarking methodology is thus theoretically founded, and we consider goal criterion 1 as fulfilled. Overall, we consider our first goal as fulfilled.

The criteria for our second goal, showing how to create a Gold Standard for points-to analysis in order to assess the exact accuracy of points-to analysis, are as follows:

4. Distribution: We define a converging process that allows researchers to collaborate on working towards a Gold Standard for points-to analysis.

5. Efficiency: We provide tool support for applying this distributed process.

6. Feasibility: We show the feasibility of the process by presenting a first attempt towards a Gold Standard.

We have presented a process to create a Gold Standard that allows researchers to work together, and we have used this process to create a first attempt towards a Gold Standard (as presented in the previous chapter), thus fulfilling goal criterion 6, feasibility. The process has not been tested with a large amount people yet; however, users can work independently on the iterative steps (improving dynamic and static analysis) and only need to synchronize their findings and communicate their current efforts (as, otherwise, unnecessary double work may occur), so we created the prerequisites for the process to be scalable to a large group of people. However, only future use of the platform can show whether goal criterion 4 is indeed fulfilled. The process is supported through our benchmarking platform, which computes the current best effort towards a Gold Standard and shows the discrepancies between under- and over-approximations, and it also allows to manage manual absence proofs; thus, we consider goal criterion 5, efficiency, as fulfilled. Overall, we consider our second goal as fulfilled.
Chapter 9. Conclusion and Future Work

9.2 Future Work

9.2.1 Benchmarking

The next natural step is to connect more points-to analysis implementations to the benchmarking platform. However, this is not necessarily our task; rather, it is up to the maintainers of those implementations. However, the platform now needs to be promoted to research groups dealing with points-to analysis.

Some parts of our methodology are to be seen as a basis for future discussion. A discussion should be started in the research community on which client analyses should be used for evaluating points-to analysis, and which programs should be contained in a future benchmark suite for points-to analysis.

In the future, computation time and memory consumption should be taken into consideration for the benchmarking methodology. Currently, precision and recall are measured, whereof recall can, for some benchmark programs, be eliminated as a variable. Adding now two new measures and having them weighted against precision (and possibly recall) seems to be an interesting research topic.

The benchmarking platform serves its purpose. However, with more users, new features will likely be requested from the research community. This may include the support of more client analyses, but also new features are possible. For instance, the benchmarking platform could be extended to a knowledge database for the benchmark programs, i.e., annotating programs with information on which dynamic class loading, reflective calls etc. occur where. Such information can be useful for a variety of research areas, not only points-to analysis.

The general design of the platform should allow to be used for other dataflow analyses as well. In general, it should suffice to replace the existing client analyses and the set of features. However, branching should be avoided and thus the implementation may need to be generalized, but that is an engineering and not a research problem.

9.2.2 Fast Dynamic Analysis

Our prototypical implementation leaves much room for improvement. For instance, switch-statements can be compiled into two different Java bytecode instructions: lookupswitch and tablesswitch (see chapter 7.10 in [50] for details). The former is a general case that is interpreted as a sequence of if-then-else statements. The latter is used if the value range of the case-statements is dense, i.e., there is an efficient representation of the values as indices into a table of target offsets. Our current implementation uses only lookupswitch-instructions, and disregards tablesswitch-instructions, which may
be more efficient in many cases. Then, as discussed in Section 6.4.4, the maximum size of a method's code is limited. Thus, we do not inline `switch`-statements into the code as outlined in Figure 6.8, but generate separate methods containing the `switch`-statements. However, where inlining is possible (i.e., the maximum method code size would not be exceeded), it could improve performance.

If a given event occurs for sure, i.e., it has been observed before, it is no longer necessary to look out for it. Thus, its instrumentation code can be removed, either by re-generating the custom made instrumentation code, or even at runtime by dynamically removing the instrumentation code. However, it is not fully compatible with the "formula" approach; here, only complete instrumentation sites could be removed.

9.2.3 Increasing dynamic coverage

Increasing dynamic coverage of the benchmark programs is essential to creating a Gold Standard. We have already presented and applied some means to increase dynamic coverage in the first part of this thesis. Here, we list two additional ideas.

First, it should be allowed to induce exceptions that do not happen frequently in practice but are theoretically possible into programs, for instance, `OutOfMemoryError`, file system exceptions, etc. This should improve dynamic coverage in application code where such exceptions would be caught by an exception handler.

Our second idea is to utilize dynamic test case generation (also referred to as concolic testing): 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. Dynamic test case generation tools exists for many programming languages, e.g., C [78], PHP [10], and Java [77]. However, concolic testing is usually used for testing single methods, whereas we need to vary input to a main() method, which may include the contents of files.

9.2.4 Where to Apply Manual Absence Proofs?

The differences between over- and under-approximations of a Gold Standard may be quite large, depending on analyzed program and client analysis. 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 will have to prove that the element is infeasible. The client analyses that we used in this thesis interact, i.e., improving the precision of one client analysis may improve the results in other client analyses 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 are 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 researchers in determining at which point it would be useful to perform these proofs.

Lhoták has done work into this direction for call graphs [42]. 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.
Appendix A

Dynamic Agent - Implementation Details

Here, we present implementation details and usage information of our dynamic analysis tool that gathers optimistic analysis results for the client analyses presented in Section 5.4.

The dynamic analysis tool is implemented as an agent based on the java.lang.instrument package\(^1\). In short, such an agent is compiled to a .jar file that is specified on the command line using Java’s -javaagent option. Prior to invoking the actual program’s main()-method, a special method of the agent called premain() is invoked, which then can register itself as a class loader hook, i.e., it can instrument classes that are loaded from now on.

A String-argument which specifies the name filter(s) for identifying application classes has to be provided to the agent. Note that there have to be name filters, and note that the packages java.* and sun.* cannot be instrumented as these contain low-level system classes that are already loaded at time of instrumentation. This is a mere technical restriction of our approach and could be overcome by a different dynamic implementation, but sufficient for our purposes.

Our agents registers such a class loader hook that instruments code so that the following tasks are performed:

1. Mapping concrete runtime objects to abstract objects
2. Writing results to a file
3. Collecting application object call graph information
4. Collecting Heap
5. Collecting ClassCastException
6. Collecting NullPointerExceptions
7. Collecting Exceptions Thrown by Methods

\(^1\)See [http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html](http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html)
We describe each of these tasks in the following subsections. Note that the “polymorphic calls” client analysis is not supported.

A side-goal of the agent is to keep the instrumentation code as small as possible, as there is a maximum size of 65536 bytes for code for each method (a limitation of the class file format, e.g., exception handlers, can address only those first bytes, cf. Section 4.7.3 in [51]) which, if exceeded, will make methods not executable. Since in object-oriented programming methods are kept rather short, this is usually not a problem, but generated code (e.g., from compiler generators) may be rather lengthy.

A.1 Mapping concrete runtime objects to abstract objects

Each application runtime object is, upon its creation, tagged with an integer value identifying its corresponding abstract object, which is identified through its syntactic object creation. This tag can later on be retrieved by the other parts of the dynamic agent; for this, to each application type a new interface is added, namely `agent.ObjectAdapter` which contains a method `int _getTag_AGENT()`.

The special case of reflective object instantiation through the methods `java.lang.Class.newInstance()` and `java.lang.reflect.Constructor.newInstance()` is also handled: A new abstract object is created on each such method invocation, if and only if the (at runtime known!) type of the instantiated class is an application type. Thus, there are in general more than one abstract objects for such an invocation, but when comparing static with dynamic analysis, the abstract objects from dynamic analysis can be mapped to one single object.

However, the actual implementation is far more complicated than the above outline. Assigning tags cannot be done at the same time as creating objects with “new” and before constructor invocation. This is because Java does not allow objects’ identities to be accessed prior to invoking the constructor chain\(^2\). Setting the tag after returning from constructors would then miss monitoring what is happening within the constructor (which, in turn, may invoke a lot more code).

Our dynamic analysis overcomes these issues like this. Directly before a constructor call, the abstract object id is written into a special thread-local tag-field. Each constructor stores this id into a special instance field (which is returned by `_getTag_AGENT()`, see above) after invoking its super-constructor\(^3\).

\(^2\)Experiments with disabling bytecode verification showed that this works very unreliably; sometimes, but not always, the JVM would simply crash.

\(^3\)Note: The seemingly simpler alternative to add an additional id-parameter to each application constructor does not work as it would effectively break all reflective object
Since it is known at the time of instrumentation whether or not this field has been set yet for a local instance, delaying this is not an issue as instead of invoking \_getTag\_AGENT(), the tag can be retrieved from the thread-local tag-field instead.

To handle the case when a super/this-constructor reference instantiates another class (for example, super(new T())), the current value of the thread-local tag-field is saved on the current stack frame.

### A.2 Writing results to a file

The collected information is continuously written to a result file, after checking if the same event has been observed before. This check is required as continuously storing all observed events could lead to very large files, as experiments with an initial implementation have shown. Events are modeled by respective data structures and added to a set; the method \texttt{Set.add()} indicated of an equal object is already in the set, in which case nothing is written to the result file.

The underlying output stream is flushed so that no information is lost upon leaving the program’s \texttt{main()} method or, alternatively, prior to a call to \texttt{System.exit(int)}. These are the two only ways for a program to terminate.

The results file is written in an XML format that is readable by our benchmarking platform, cf. Section 7.3.

### A.3 Collecting application object call graph information

The naive approach to instrument all method invocation sites and collect object call graph information there does not work as there is no explicit information at polymorphic call sites about which method is actually going to be invoked.

We have thus decided the following approach: The agent keeps track of an abstracted call stack for application methods. That is, upon entry to an application method, its pair of tag and method signature is pushed on top of the stack, and popped from the stack before returning from this method. The two top stack elements are then the caller and callee upon entry to an application method.

In case of callbacks from non-application code into application code, there no edge is added to the application call graph but only the node for the callee. In order to identify such cases, the agent keeps track of whether the instantiation.
currently executing code is application code or non-application code. For this, a boolean flag is set to true upon entry to an application method and after returning from calls back into application code, and set to false upon entry to each non-application method.

A.4 Collecting Heap

Each store to fields of application classes are instrumented such that the abstract object ids of both the object owning the field as well as the value stored into the field are received, as described above. Since field accesses are non-polymorphic in Java, the actual field is identified through its static name.

A.5 Collecting ClassCastException

An obvious approach to collect ClassCastException is to add an exception handler around each cast. However, in order to keep the instrumentation code as small as possible, we chose a different approach:

The agent instruments the constructor of ClassCastException that takes one String-parameter, which is the constructor that is being used by the JVM in case a cast fails. The information that have to be collected are type of cast, line number, and enclosing class. The former is not explicitly available but is parsed from the exception’s “message” string, which has the fixed format “T1 cannot be cast to T2”, the latter two are easily obtained through the stack trace (by invoking appropriate methods on java.lang.Throwable.getStackTrace()[0]). ClassCastException stemming from non-application classes are identified by applying the name filters to the enclosing class.

ClassCastException can also be instantiated through a new expression, by sub-typing, and by reflection as well. In order to not falsely collect those instantiations of ClassCastException, the agent performs the following: For the former two cases, an additional constructor with an additional “boolean” parameter that purely serves as a marker is added. It has the same (yet uninstrumentalized) functionality as the original constructor. Then, all explicit references to the original constructor are replaced with the newly added constructor. For the latter case, the reflective call originates come from the sun package. For technical reasons, our agent cannot treat classes from this package as application classes, so the call is ignored automatically by applying name filters.

Implementation note: Instrumenting already loaded classes forbids to add methods, so we provide a re-implementation of ClassCastException. The .jar of the agent must then be prepended (option -Xbootlasspath/p) to the bootclasspath.
A.6 Collecting NullPointerExceptions

For capturing `NullPointerException`, we place an exception handler around each potentially failing access, i.e., non-static member accesses with the exception of constructor invocations and accesses on “this”, which can never be null.

Note that the approach that we took for `ClassCastException` is not possible here as there is no information in the exception object about the member access that has failed. The alternative to allow only one access per source-code line is hardly appealing. Note that this approach may require lots of bytecode instructions and may exceed the maximum method length.

A.7 Collecting Exceptions Thrown by Methods

Collecting the exceptions thrown by methods is done by surrounding each method with an exception handler. Exceptions are logged upon invocation of such exception handlers and are then re-thrown.
Appendix B

Complete Metrics Results

In this chapter, we list the complete metrics results for the experiments described in Chapter 8. Tables B.1 through B.5 list the metrics results for Spark, Paddle (ConIns and ObjSens), and P2SSA (ConIns, ObjSens, and ObjSens with path sensitivity). Tables B.6 and B.7 list the approximated accuracies $\hat{F}$ for metrics $E, ON, OE,$ and $H$ as referenced in Section 8.2.
<table>
<thead>
<tr>
<th>Program</th>
<th>Dynamic</th>
<th>Spark</th>
<th>Paddle</th>
<th>P2SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ConIns</td>
<td>ConIns</td>
<td>ObjSens</td>
<td>ConIns</td>
</tr>
<tr>
<td>antlr</td>
<td>1234</td>
<td>713</td>
<td>713</td>
<td>696</td>
</tr>
<tr>
<td></td>
<td>602</td>
<td>602</td>
<td>612</td>
<td>612</td>
</tr>
<tr>
<td>bloat</td>
<td>1639</td>
<td>2457</td>
<td>2429</td>
<td>2427</td>
</tr>
<tr>
<td>javac</td>
<td>1208</td>
<td>1463</td>
<td>1460</td>
<td>1460</td>
</tr>
<tr>
<td>javacc</td>
<td>834</td>
<td>1021</td>
<td>1021</td>
<td>1019</td>
</tr>
<tr>
<td>javadoc</td>
<td>1248</td>
<td>834</td>
<td>822</td>
<td>821</td>
</tr>
<tr>
<td></td>
<td>601</td>
<td>601</td>
<td>606</td>
<td>606</td>
</tr>
<tr>
<td>jlayer</td>
<td>222</td>
<td>234</td>
<td>233</td>
<td>231</td>
</tr>
<tr>
<td>jython</td>
<td>2864</td>
<td>3049</td>
<td>1974</td>
<td>1903</td>
</tr>
<tr>
<td></td>
<td>1137</td>
<td>1141</td>
<td>1375</td>
<td>1376</td>
</tr>
<tr>
<td>lucene</td>
<td>846</td>
<td>1354</td>
<td>895</td>
<td>894</td>
</tr>
<tr>
<td></td>
<td>81</td>
<td>81</td>
<td>154</td>
<td>155</td>
</tr>
<tr>
<td>pmd</td>
<td>2328</td>
<td>2734</td>
<td>2155</td>
<td>2149</td>
</tr>
<tr>
<td></td>
<td>274</td>
<td>275</td>
<td>608</td>
<td>610</td>
</tr>
<tr>
<td>recoder</td>
<td>4356</td>
<td>5188</td>
<td>5139</td>
<td>5154</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>sablecc</td>
<td>1381</td>
<td>1847</td>
<td>1785</td>
<td>1823</td>
</tr>
</tbody>
</table>

Table B.1: Results for metric N for the different analyses. The upper number in each cell for static analysis denotes the total size where the lower number, where present, denotes the identified misses. The subscript $f$ indicates filter operations in P2SSA.
<table>
<thead>
<tr>
<th>Program</th>
<th>Dynamic</th>
<th>ConIns</th>
<th>ConIns ObjSens</th>
<th>ConIns ObjSens ObjSens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark</td>
<td>Paddle</td>
<td>P2SSA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>antlr</td>
<td>338</td>
<td>2488</td>
<td>2873</td>
<td>2855</td>
</tr>
<tr>
<td>bloat</td>
<td>586</td>
<td>7737</td>
<td>23920</td>
<td>21567</td>
</tr>
<tr>
<td>javac</td>
<td>302</td>
<td>2219</td>
<td>6612</td>
<td>5417</td>
</tr>
<tr>
<td>javacc</td>
<td>375</td>
<td>1156</td>
<td>2081</td>
<td>2080</td>
</tr>
<tr>
<td>javdoc</td>
<td>302</td>
<td>2789</td>
<td>3309</td>
<td>3074</td>
</tr>
<tr>
<td>jlayer</td>
<td>302</td>
<td>361</td>
<td>422</td>
<td>405</td>
</tr>
<tr>
<td>jython</td>
<td>338</td>
<td>10828</td>
<td>95905</td>
<td>76702</td>
</tr>
<tr>
<td>lucene</td>
<td>302</td>
<td>1153</td>
<td>2282</td>
<td>2253</td>
</tr>
<tr>
<td>pmd</td>
<td>302</td>
<td>11715</td>
<td>11419</td>
<td>11197</td>
</tr>
<tr>
<td>recoder</td>
<td>302</td>
<td>13078</td>
<td>23081</td>
<td>22031</td>
</tr>
<tr>
<td>sablecc</td>
<td>302</td>
<td>7841</td>
<td>16999</td>
<td>14726</td>
</tr>
</tbody>
</table>

Table B.2: Results for metric \( \text{E} \) for the different analyses. The upper number in each cell for static analysis denotes the total size where the lower number, where present, denotes the identified misses. The subscript \( f \) indicates filter operations in P2SSA.
Table B.3: Results for metric ON for the different analyses. The upper number in each cell for static analysis denotes the total size where the lower number, where present, denotes the identified misses. The subscript $f$ indicates filter operations in P2SSA.
Table B.4: Results for metric OE for the different analyses. The upper number in each cell for static analysis denotes the total size where the lower number, where present, denotes the identified misses. The subscript f indicates filter operations in P2SSA.
<table>
<thead>
<tr>
<th>Program</th>
<th>Dynamic</th>
<th>Spark</th>
<th>Paddle</th>
<th>P2SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ConIns</td>
<td>ConIns</td>
<td>ObjSens</td>
<td>ConIns</td>
</tr>
<tr>
<td>antlr</td>
<td>739</td>
<td>1717</td>
<td>1717</td>
<td>1656</td>
</tr>
<tr>
<td></td>
<td>279</td>
<td>279</td>
<td>279</td>
<td>280</td>
</tr>
<tr>
<td>bloat</td>
<td>3801</td>
<td>270556</td>
<td>267903</td>
<td></td>
</tr>
<tr>
<td>javac</td>
<td>1972</td>
<td>27547</td>
<td>27523</td>
<td>16110</td>
</tr>
<tr>
<td>javacc</td>
<td>218</td>
<td>1100</td>
<td>1100</td>
<td>996</td>
</tr>
<tr>
<td>javadoc</td>
<td>1362</td>
<td>17297</td>
<td>17295</td>
<td>12959</td>
</tr>
<tr>
<td></td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
<tr>
<td>jlayer</td>
<td>51</td>
<td>135</td>
<td>135</td>
<td>67</td>
</tr>
<tr>
<td>jython</td>
<td>4996</td>
<td>79728</td>
<td>78888</td>
<td>12848</td>
</tr>
<tr>
<td></td>
<td>4328</td>
<td>4328</td>
<td>4437</td>
<td></td>
</tr>
<tr>
<td>lucene</td>
<td>313</td>
<td>718</td>
<td>722</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>90</td>
</tr>
<tr>
<td>pmd</td>
<td>1248</td>
<td>21500</td>
<td>21492</td>
<td>16868</td>
</tr>
<tr>
<td></td>
<td>205</td>
<td>205</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>recoder</td>
<td>9553</td>
<td>69021</td>
<td>69003</td>
<td>50313</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>sablecc</td>
<td>1408</td>
<td>134080</td>
<td>131412</td>
<td>6530</td>
</tr>
</tbody>
</table>

Table B.5: Results for metric H for the different analyses. The upper number in each cell for static analysis denotes the total size where the lower number, where present, denotes the identified misses. The subscript <sub>f</sub> indicates filter operations in P2SSA.
Table B.6: \( P \) scores computed with initial input (subscript i) and final input (subscript f) for metrics \( E \) (up) and \( N \) (below).

<table>
<thead>
<tr>
<th></th>
<th>Spark</th>
<th>Paddle</th>
<th>ConIns</th>
<th>ConIns</th>
<th>ObjSens</th>
<th>ConIns</th>
<th>ObjSens</th>
<th>antlr</th>
<th>jython</th>
<th>lucene</th>
<th>pmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>412.7</td>
<td>63.9</td>
<td>71.3</td>
<td>22.6</td>
<td>1.57</td>
<td>6.37</td>
<td>20.6</td>
<td>1.57</td>
<td>63.9</td>
<td>71.3</td>
<td>412.7</td>
</tr>
<tr>
<td>Final</td>
<td>397.5</td>
<td>60.1</td>
<td>69.7</td>
<td>22.2</td>
<td>1.56</td>
<td>6.36</td>
<td>20.5</td>
<td>1.56</td>
<td>60.1</td>
<td>69.7</td>
<td>397.5</td>
</tr>
</tbody>
</table>

Program Options
Table B.7: $F$ scores computed with initial input (subscript $i$) and final input (subscript $f$) for metrics $OE$ (up) and $H$ (below).

<table>
<thead>
<tr>
<th>program</th>
<th>Spark</th>
<th>Paddle</th>
<th>P2SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ConIns $F_i$ $F_f$</td>
<td>ConIns $F_i$ $F_f$</td>
<td>ObjSens $F_i$ $F_f$</td>
</tr>
<tr>
<td>antlr</td>
<td>.249  .267</td>
<td>.250  .267</td>
<td>.255  .272</td>
</tr>
<tr>
<td>jython</td>
<td>.000  .000</td>
<td>.000  .000</td>
<td>.023  .032</td>
</tr>
<tr>
<td>pmd</td>
<td>.046  .115</td>
<td>.048  .115</td>
<td>.084  .186</td>
</tr>
<tr>
<td></td>
<td>.375  .374</td>
<td>.375  .374</td>
<td>.384  .383</td>
</tr>
<tr>
<td>jython</td>
<td>.012  .015</td>
<td>.012  .015</td>
<td>.051  .062</td>
</tr>
<tr>
<td>lucene</td>
<td>.455  .469</td>
<td>.454  .467</td>
<td>.711  .675</td>
</tr>
<tr>
<td>pmd</td>
<td>.039  .091</td>
<td>.039  .091</td>
<td>.050  .112</td>
</tr>
</tbody>
</table>
Bibliography


Bibliography

symposium on Software testing and analysis, ISSTA ’08, pages 273–282, 2008.


Bibliography


Bibliography


Linnaeus University Dissertations

Below please find a list of recent publications in the series Linnaeus University Dissertations. For a full list and more information: Lnu.se


