Precise and Efficient Points-to Analysis
via New Context-Sensitivity
and Heap Abstraction

by

Tian Tan

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SCHOOL
OF
COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY • AUSTRALIA

Wednesday 19th April, 2017

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

© Tian Tan 2017
Abstract

Points-to analysis addresses a fundamental problem in program analysis: determining statically which objects a variable or reference can point to. As a fundamental technique, many real-world clients such as bug detection, security analysis, program understanding, compiler optimization and program verification, depend on the results of points-to analysis.

A long-standing problem in points-to analysis is the balance between precision and efficiency. This thesis aims to improve both ends of the balance respectively.

- For precision, object-sensitivity is usually considered as the most precise context-sensitivity for points-to analysis for object-oriented languages, such as Java. However, it suffers from the scalability problem when increasing the context length and thus it is hard to further improve its precision. We present BEAN, a new object-sensitivity approach for points-to analysis. By identifying and eliminating the redundant context elements which contribute nothing to the precision, BEAN is able to improve the precision of any $k$-object-sensitive analysis by still using a $k$-limiting context abstraction.

- For efficiency, targeting the type-dependent clients such as call graph construction, devirtualization and may-fail casting, we present MAHJONG, a new heap abstraction approach for points-to analysis. By merging equivalent automata representing type-consistent objects that are created by the
allocation-site abstraction, MAHJONG enables an allocation-site-based points-to analysis to run significantly faster while achieving nearly the same precision for type-dependent clients.

We extensively evaluate BEAN and MAHJONG against the state-of-the-art points-to analysis for Java with large real-world Java applications and library. The results demonstrate that both BEAN and MAHJONG have met their goals of design. BEAN has succeeded in making points-to analysis more precise at only small increases in analysis cost. MAHJONG enables points-to analysis to run significantly faster while achieving nearly the same precision for type-dependent clients. We have released BEAN and MAHJONG as open-source tools.
Publications

- **Tian Tan**, Yue Li, and Jingling Xue. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’17).*

- Yifei Zhang, **Tian Tan**, Yue Li and Jingling Xue. Ripple: Reflection Analysis for Android Apps in Incomplete Information Environments. *7th ACM Conference on Data and Applications Security and Privacy (CODASPY’17).*

- **Tian Tan**, Yue Li and Jingling Xue. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. *23nd International Static Analysis Symposium (SAS’16).*

- Yue Li, **Tian Tan**, Yifei Zhang and Jingling Xue. Program Tailoring: Slicing by Sequential Criteria. *30th European Conference on Object-Oriented Programming (ECOOP’16).* **Distinguished Paper Award.**

- Yue Li, **Tian Tan** and Jingling Xue. Effective Soundness-Guided Reflection Analysis. *22nd International Static Analysis Symposium (SAS’15).*

- Yue Li, **Tian Tan**, Yulei Sui and Jingling Xue. Self-Inferencing Reflection Resolution for Java. *28th European Conference on Object-Oriented Programming (ECOOP’14).*
Acknowledgements

First, I would like to thank my supervisor, Professor Jingling Xue, for supporting me with freedom and guidance throughout my Ph.D. studies. His serious attitude, diligence and enthusiasm to the research influence me and will continue to benefit me in my future career.

My special thanks go to Yue Li, a great research partner and the best friend of mine. I worked with Yue during the whole period of my Ph.D. studies. He is like my elder brother and gives me great help in both research and life. I am very very very lucky to have him as my collaborator and friend.

I also thank my labmates and colleagues from the CORG at UNSW, Peng Di, Yulei Sui, Sen Ye, Yu Su, Ding Ye, Hao Zhou, Xiaokang Fan, Hua Yan, Feng Zhang, Yifei Zhang, Jieyuan Zhang, Jie Liu, Diyu Wu and Jingbo Lu. It was a pleasant experience to get along with them during these unforgettable days.

I thank the DOOP team for making DOOP (such a good points-to analysis framework for Java) public available. DOOP is well-designed so that I can easily integrate both my thesis work BEAN and MAHJONG into it to perform evaluation.

Many thanks go to the two international examiners of my thesis, Prof. Yannis Smaragdakis at University of Athens and Prof. Ondřej Lhoták at University of Waterloo, for their valuable time and comments.

Last, but not least, I would like to express my gratefulness to my family. I
am deeply indebted to my parents for their fantastic love, endless support and everything else they have given me all through these years. Without them, I would not have been where I am now. I dedicate my dissertation and love to them.
# Contents

Abstract

Acknowledgements

1 Introduction

1.1 Challenges

1.2 Contributions

1.3 Organization

2 Background: Context-Sensitive Points-to Analysis

2.1 Preliminary

2.2 Formalism

2.2.1 Notations

2.2.2 Context-Sensitive Points-to Analysis

2.2.3 Context-Sensitivity

3 BEAN: Precise Points-to Analysis via Object Allocation Graph

3.1 Overview

3.2 Motivation

3.2.1 Redundant Elements in Method Contexts

3.2.2 Redundant Elements in Heap Contexts
3.2.3 Discussion ...................................................... 24
3.3 Bean ............................................................... 25
  3.3.1 Object Allocation Graph ..................................... 26
  3.3.2 Context Selection ............................................. 27
  3.3.3 Discussion ................................................... 28
3.4 Formalism .......................................................... 29
  3.4.1 Object Allocation Graph ..................................... 29
  3.4.2 Context Selection ............................................. 31
  3.4.3 Bean-directed Object-Sensitive Points-to Analysis ....... 36
  3.4.4 Properties ................................................... 36
3.5 Implementation .................................................... 38
3.6 Evaluation .......................................................... 38
  3.6.1 Experimental Setting ......................................... 39
  3.6.2 RQ1: Precision and Performance Measurements .......... 40
  3.6.3 RQ2: A Real-World Case Study .............................. 43
3.7 Related Work ....................................................... 46

4 Mahjong: Efficient Points-to Analysis by Merging Equivalent Automata 

  4.1 Overview .......................................................... 49
  4.2 Motivation .......................................................... 52
    4.2.1 Allocation-Type Abstraction: A Naive Solution .......... 53
    4.2.2 Mahjong: Our Solution ....................................... 54
  4.3 Mahjong ........................................................... 60
    4.3.1 Overview of Mahjong ......................................... 60
    4.3.2 The NFA Builder .............................................. 61
    4.3.3 The DFA Converter ............................................ 61
4.3.4 The Automata Equivalence Checker .......................... 62
4.3.5 The Heap Modeler ............................................. 62
4.3.6 MAHJONG-based Points-to Analysis ......................... 62
4.4 Algorithms ....................................................... 66
  4.4.1 MAHJONG .................................................. 66
  4.4.2 The NFA Builder ............................................. 68
  4.4.3 The DFA Converter ......................................... 69
  4.4.4 The Automata Equivalence Checker ......................... 70
  4.4.5 The Heap Modeler ........................................... 72
4.5 Implementation .................................................. 72
4.6 Evaluation ....................................................... 73
  4.6.1 RQ1: MAHJONG’s Effectiveness as a Pre-Analysis ....... 75
  4.6.2 RQ2: MAHJONG-based Points-to Analysis .................. 78
4.7 Related Work ................................................... 85

5 Conclusions and Future Work ................................. 88
  5.1 Conclusions .................................................... 88
  5.2 Future Work .................................................. 89

Bibliography .................................................... 90
List of Figures

2.1 Notations................................................................. 11
2.2 Context-sensitive points-to analysis............................ 12

3.1 Method contexts for 2-obj and BEAN.......................... 21
3.2 Heap contexts for 2-obj and BEAN............................... 23
3.3 Overview of BEAN................................................... 25
3.4 The OAGs for the two motivating programs in Figures 3.1 and 3.2. 26
3.5 Notations for object allocation graph.......................... 29
3.6 Rules for building the OAG, $G = (N, E)$, for a program based on a pre-analysis.............................................. 30
3.7 Rules for basic relations in an OAG, $G = (N, E)$............... 31
3.8 Rules for context selection in an OAG, $G = (N, E)$............ 32
3.9 An OAG.................................................................. 33
3.10 Three Cases marked for [Hctx-Div] and [Hctx-Cyc] in Figure 3.8. . 35
3.11 A real-world application for using java.util.HashSet.... 44
3.12 Part of OAG related to HS/1 and HS/2......................... 45

4.1 An example program illustrating object merging............... 53
4.2 Field points-to graph rooted at $o_1^T$ and $o_2^T$................. 55
4.3 Illustrating Condition 2 in Definition 1........................ 57
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>The mapping of a field points-to graph rooted at an object to a</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>sequential automaton.</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Overview of MAHJONG.</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td>Illustrating <em>the null-field problem</em>.</td>
<td>65</td>
</tr>
<tr>
<td>4.7</td>
<td>Number of abstract objects created by the allocation-site abstraction</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>and MAHJONG.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Object merging in <em>checkstyle</em>.</td>
<td>76</td>
</tr>
<tr>
<td>4.9</td>
<td>Precision gains of <em>M-k-type over k-type</em>.</td>
<td>84</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Five kinds of core statements in points-to analysis. . . . . . . . . . 8

3.1 Precision and performance results for all the five analyses. The two
precision metrics shown are the number of variable pairs that may be
aliases generated by \textit{may-alias} ("may-alias pairs") and the number
of casts that cannot be statically proved to be safe by \textit{may-fail-cast}
("may-fail casts"). In both cases, \textit{smaller is better}. One performance
metric used is the analysis time for a program. . . . . . . . . . . . 41

3.2 Pre-analysis times of \texttt{Bean} (secs). For a program, its pre-analysis
time comes from three components: (1) a context-insensitive points-
to analysis ("CT"), (2) OAG construction per Figure 3.6 (OAG),
and (3) object-sensitive context computation per Figure 3.8 ("CTX-
COMP"). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

4.1 Some equivalence classes in \texttt{checkstyle}. . . . . . . . . . . . . . . 77

4.2 Efficiency and precision metrics for all programs and analyses with
and without \texttt{MAHJONG}. In \textit{all cases} (except \textit{speedup}), \textit{lower is bet-
ter}. Symbol $\infty$ is used in \textit{speedup} when a baseline analysis is not
scalable but \texttt{MAHJONG} is scalable. . . . . . . . . . . . . . . . . . 81
Chapter 1

Introduction

Points-to analysis is a static analysis technique that addresses a fundamental problem in program analysis: at compile-time, it determines which memory locations a pointer can point to at runtime. For object-oriented languages, such as Java, points-to analysis focuses on heap locations, i.e., it statically determines which heap objects a variable or reference can point to dynamically.

The results of points-to analysis are required by a wide range of client applications, including bug detection [57, 56, 11, 90, 91, 103, 9], security analysis [47, 6, 31, 27], program understanding [26, 29, 62, 79], compiler optimization [66, 19, 18, 88] and program verification [79, 12], as well as other program analyses, such as program slicing [44, 81, 43], call graph construction [54, 38, 3], reflection analysis [41, 42, 74, 107] and escape analysis [15, 100, 97, 68]. Hence effective points-to analyses are highly demanded as they can benefit many client applications and other fundamental analysis techniques.

Two metrics are usually considered to measure the effectiveness of points-to analysis: precision and efficiency. In terms of a client, a more precise points-to analysis enables less false bugs to be reported or more program properties to be
proved safely. A more efficient points-to analysis enables more time-critical clients (e.g., compiler optimizations) to be applied or less turnaround time to wait.

In this thesis, we aim to improve the precision and efficiency of points-to analysis respectively, in a significant way. However, this is not trivial as discussed below.

1.1 Challenges

Although many techniques for improving the precision and efficiency of points-to analysis have been proposed in the last decades [5, 53, 7, 82, 101, 75, 92, 69, 34, 48, 83, 84, 49, 93, 21, 4, 78, 87, 89, 85], how to make a good balance between precision and efficiency remains a long-standing problem: the techniques which achieve higher precision usually sacrifice efficiency, and vice versa, which limits the effectiveness of points-to analysis.

To make a good precision and efficiency trade-off, existing points-to analyses mainly focus on two dimensions: modeling the control-flow and modeling the heap. For object-oriented programs, context-sensitivity is known to model control-flow with tractable and useful precision [7, 39, 72, 75, 82, 79]. Compared to the storeless heap abstraction (e.g., access paths), store-based heap abstraction (e.g., allocation-site-based and allocation-type-based heap modeling) is the one predominately adopted by points-to analysis [32, 72, 80, 45, 94]. However, both techniques suffer from the balance problem, especially for large object-oriented programs.

Context-sensitive points-to analysis separately analyzes the methods under different calling contexts to improve precision by preventing the merging of the points-to information in different contexts. To address the problem of infinite calling contexts due to the recursive calls and also to further improve the analysis scalability in practice, traditional approaches usually limit the length of contexts by a given
parameter, $k$ [71, 53, 55, 75]. As a result, if the length of the contexts is no larger than $k$, the contexts can be distinguished; otherwise, the contexts ending with the same $k$ suffixes cannot be further distinguished and the points-to information under these contexts would be merged. Given a larger $k$-limiting, the analysis can partition the context space with finer grain and achieve better precision, but it may also cause the analysis to dramatically slow down or even become unscalable.

Heap abstraction [32], which partitions the infinitely-sized heap into a finite number of (abstract) objects, is particularly important to the analysis for object-oriented languages such as Java. Generally, heap abstractions for points-to analysis for Java may use one abstract object per class (allocation-type heap abstraction) [99], or one abstract object per allocation site (allocation-site heap abstraction) [38], which can be further separated context-sensitively in an orthogonal manner [53, 75, 34]. Points-to analyses with different heap abstractions may exhibit significant differences in precision and efficiency [99, 75, 32]. Allocation-site heap abstraction, as a finer abstraction, makes points-to analysis usually notably more precise than the ones using allocation-type heap abstraction. However, allocation-site-based points-to analysis usually runs significantly slower and its performance will dramatically become worse if context-sensitivity is applied to further distinguish different heap objects for better precision.

1.2 Contributions

In this thesis, we present a new context-sensitivity approach called BEAN [95] and a new heap abstraction approach called MAHJONG [96], to improve the precision and efficiency of points-to analysis respectively. Specifically, this thesis makes the following contributions.
• We introduce **Bean** to improve the precision of points-to analysis with small overhead increases.

  - **Bean** is a new object-sensitivity approach for points-to analysis. Object-sensitivity \([53, 55, 75]\) is usually considered as the most precise context-sensitivity for points-to analysis for Java \([34]\). **Bean** further improves its precision by automatically identifying and eliminating the redundant context elements in distinguishing contexts. Unlike traditional object-sensitivity, which obtains better precision by increasing the limit of context length \(k\) with usually significantly worse efficiency, **Bean** is able to achieve better precision with still the same \(k\)-limiting at only small increases in analysis cost.

  - We thoroughly evaluate the effectiveness of **Bean** by comparing it with traditional object-sensitivity \([53, 75]\) and the state-of-art hybrid context-sensitivity \([34]\) for points-to analysis. Our evaluation shows that **Bean** can always achieve better precision with small increases in analysis cost for real-world Java programs in practice.

• We introduce **Mahjong** to significantly improve the efficiency of points-to analysis while maintaining nearly the same precision for type-dependent clients such as call graph construction.

  - **Mahjong** is a new heap abstraction that models the heap by identifying and merging type-consistent objects, which are distinguished by the mainstream allocation-site-based points-to analysis. However, the allocation-site-based heap abstraction often over-partitions the heap without improving the precision much for an important class of type-dependent clients such as call graph construction, devirtualization and
may-fail casting. By merging type-consistent objects, MAHJONG improves the efficiency of points-to analysis with nearly the same precision for type-dependent clients.

– We extensively evaluate the effectiveness of MAHJONG by applying it on three kinds of mainstream context-sensitivity for points-to analysis, i.e., call-site-sensitivity [71, 99, 34], object-sensitivity [53, 55] and type-sensitivity [75]. The evaluation demonstrates that MAHJONG meets its goal of design and is versatile to improve all these three kinds of context-sensitive points-to analysis for the type-dependent clients.

• We implement both BEAN and MAHJONG as standalone tools which are designed to work well with various points-to analysis frameworks for Java. We have released BEAN as an open-source tool at

http://www.cse.unsw.edu.au/~corg/bean

and we also have released MAHJONG as an open-source tool at


1.3 Organization

The remainder of this thesis is organized as follows.

In Chapter 2, we introduce context-sensitive points-to analysis for object-oriented languages. We first give a generic formalism for context-sensitive points-to analysis. Based on this basic formalism, we then explain and formally present three kinds of mainstream context-sensitivity for object-oriented languages, i.e., call-site-sensitivity [71, 99, 34], object-sensitivity [53, 55] and type-sensitivity [75].
In Chapters 3 and 4, we introduce BEAN and MAHJONG respectively. These two chapters share the same structures, i.e., they both present the techniques following the order of overview, motivation, methodology, formalism, implementation and evaluation. At the end of both chapters, we discuss their related work.

In Chapter 5, we give the conclusion and suggest some further work.
Chapter 2

Background: Context-Sensitive Points-to Analysis

In this chapter, we provide the background information about context-sensitive points-to analysis that will be necessary to understand the remainder of this thesis. In Section 2.1, we present the preliminary knowledge about context-sensitive points-to analysis. In Section 2.2, we give a standard formulation of context-sensitive points-to analysis and three kinds of mainstream context-sensitivity that are used in the following chapters.

2.1 Preliminary

This section gives introductory information about points-to analysis for Java and context-sensitivity.

Points-to Analysis Points-to analysis is a fundamental static analysis that computes the over-approximation of the heap locations that each program pointer may point to during executions. For object-oriented languages such as Java, a pointer
can be a local variable or an instance field, and heap locations usually correspond to heap objects. The results of points-to analysis for each pointer are a set of heap objects that are pointed to by the pointer, called points-to set.

In Java programs, the heap objects can be created infinitely during execution in the presence of loop and recursion. For decidability and scalability, a points-to analysis must use a heap abstraction which partitions the infinitely-sized heap into a finite number of abstract objects. In points-to analysis for Java, allocation-site abstraction, which uses an allocation site to represent all the heap objects created at the site, is the most commonly used heap abstraction [38, 53, 55, 75, 34]. We will see later that the heap can be further partitioned context-sensitively in an orthogonal manner (Section 2.2) or abstracted with other granularities to improve the effectiveness of points-to analysis (Chapter 4).

Core Statements in Points-to Analysis Table 2.1 lists the five kinds of core statements that points-to analysis for Java needs to deal with, including object allocation (New), variable assignment (Assign), instance field read (Load), instance field write (Store) and virtual method invocation (Call). In this thesis, we only discuss these statements for brevity as in [38, 53, 55, 75, 76]. Other kinds of statements (e.g., static field accesses and static method invocations) are handled

<table>
<thead>
<tr>
<th>Statement</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = new T()</td>
<td>New</td>
</tr>
<tr>
<td>x = y</td>
<td>Assign</td>
</tr>
<tr>
<td>x = y.f</td>
<td>Load</td>
</tr>
<tr>
<td>x.f = y</td>
<td>Store</td>
</tr>
<tr>
<td>x = y.g(arg1, ..., argn)</td>
<td>Call</td>
</tr>
</tbody>
</table>

Table 2.1: Five kinds of core statements in points-to analysis.
in a similar fashion.

**Context-Sensitivity for Points-to Analysis**  Context-sensitivity is shown to be useful for improving the precision of points-to analysis for Java programs in practice [99, 39, 79, 75, 41, 42, 72, 43]. A traditional context-sensitive points-to analysis analyzes the same methods and heap objects that under different **calling contexts** separately. By this way, the analysis aims to avoid the precision loss caused by the conflation of the behaviors across different interprocedural dynamic execution paths. In theory, the calling context can be generalized as certain abstraction of the program states regarding to the control flow at the call site. As a result, different variants of context-sensitivity are available and they are usually distinguished by the different context elements used.

For object-oriented languages such as Java, three kinds of context-sensitivity are commonly used, i.e., **call-site-sensitivity** [71, 99, 34], **object-sensitivity** [53, 55] as well as **type-sensitivity** [75], and their contexts consist of sequences of call sites, abstract heap objects (allocation sites) and types, respectively. They will be explained and formalized in Section 2.2 and used in Chapters 3 and 4.

In a program, the calling contexts of its methods and heap objects can be infinite in the presence of recursive calls. Even without recursion, the space of calling contexts can be tremendous in the programs with numerous execution paths. To ensure termination and improve scalability, context-sensitive points-to analyses usually limit the length of contexts by a given parameter \( k \) and thus restrict the space of contexts during the analysis. As a result, a context-sensitive points-to analysis can be parameterized and tuned by \( k \).

When analyzing Java programs, there are two types of context, a **method context** for local variables in methods and a **heap context** for heap objects. The relationship between context parameter \( k \) and the two types of context must be clarified. In the
rest of this thesis, we adopt the convention of points-to analysis as in [75], i.e., a $k$-context-sensitive points-to analysis refers to the analysis whose method contexts consist of $k$ context elements and its heap contexts consist of $k-1$ context elements. For example, in 2-call-site-sensitive points-to analysis, each method context consists of 2 call sites and each heap context consists of 1 call site.

### 2.2 Formalism

This section gives a standard formulation of context-sensitive points-to analysis for Java. We present the notations used in Section 2.2.1. In Section 2.2.2, we introduce a generic formulation of context-sensitive points-to analysis. Finally, we formalize three specific kinds of context-sensitivity in Section 2.2.3.

#### 2.2.1 Notations

In Figure 2.1, the notations include three parts. The top part lists the basic domains of context-sensitive points-to analysis. The middle part gives two domains about context. As mentioned in Section 2.1, there are two types of context, i.e., method context ($\mathbb{C}$) and heap context ($\mathbb{HC}$) for decorating local variables and heap objects respectively. These two domains are both the composition of the basic domains in the top part, and they vary with the different kinds of context-sensitivity used.

The last part gives the five key relations in context-sensitive points-to analysis. $pt$ and $fpt$ store the analysis results: $pt(c, x)$ represents the points-to set of variable $x$ under context $c$ and $fpt(hc, o_i, f)$ represents the points-to set of field access $o_i.f$ where $o_i$’s heap context is specified by $hc$.

$mtdCtxSelector$ and $heapCtxSelector$ are the core of context-sensitivity. They choose the method contexts for the local variables in the methods and heap contexts.
for object allocations. Specifically, \textit{mtdCtxSelector} uses the information available at a call site, i.e., the method context for the method containing the call site (C), the label of the call site (I), the receiver object of the call site (H) and the heap context for the receiver object (HC), to select a new method context for the local variables in the callee of the call site.

Which input elements are used to select a context by the selector depends on the specific context-sensitivity used. For example, the selector of call-site-sensitivity only uses the first two input elements, i.e., a method context (C) and a label of the call site (I), to select a new context. But the selector of object-sensitivity only considers the last input elements to select contexts.

\textit{heapCtxSelector} uses the information available at an allocation site of a heap object (i.e., the method context for the method containing the allocation site (C) and the heap object (H) allocated at the site) to select a new heap context for the heap object.

Finally, \textit{contextsOf} maps a method to its method contexts.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Notation</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>class type</td>
<td>t \in T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td>x, y \in V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heap object</td>
<td>o_i, o_j \in H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>field</td>
<td>f \in F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>method</td>
<td>m \in M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>method invocation</td>
<td>l \in I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>method context</td>
<td>c \in C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heap context</td>
<td>hc \in HC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{pt}</td>
<td>\text{C} \times \text{V} \rightarrow \mathcal{P}((\text{HC} \times \text{H}))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{fpt}</td>
<td>\text{HC} \times \text{H} \times \text{F} \rightarrow \mathcal{P}((\text{HC} \times \text{H}))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{mtdCtxSelector}</td>
<td>\text{C} \times \text{I} \times \text{HC} \times \text{H} \rightarrow \text{C}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{heapCtxSelector}</td>
<td>\text{C} \times \text{H} \rightarrow \text{HC}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{contextsOf}</td>
<td>\text{M} \rightarrow \mathcal{P}(\text{C})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: Notations.
2.2.2 Context-Sensitive Points-to Analysis

Figure 2.2 gives a generic formulation of context-sensitive points-to analysis that is parameterized by the two context selectors, *mtdCtxSelector* and *heapCtxSelector*. If the two selectors are defined to always return a constant value, then the analysis will be reduced to a context-insensitive points-to analysis. We will see three sets of instances of *mtdCtxSelector* and *heapCtxSelector* in Section 2.2.3 that correspond to three kinds of context-sensitivity. Now let us go through the rules in Figure 2.2.

**NEW**

\[ i: x = \text{new } T() \quad c \in \text{contextsOf}(m) \quad hc = \text{heapCtxSelector}(c, o_i) \]

\[ \langle hc, o_i \rangle \in pt(c, x) \]

**ASSIGN**

\[ x = y \quad c \in \text{contextsOf}(m) \]

\[ pt(c, y) \subseteq pt(c, x) \]

**LOAD**

\[ x = y.f \quad c \in \text{contextsOf}(m) \]

\[ \langle hc, o_i \rangle \in pt(c, y) \]

\[ fpt(hc, o_i, f) \subseteq pt(c, x) \]

**STORE**

\[ x.f = y \quad c \in \text{contextsOf}(m) \]

\[ \langle hc, o_i \rangle \in pt(c, x) \]

\[ pt(c, y) \subseteq fpt(hc, o_i, f) \]

**CALL**

\[ l: x = y.g(\text{arg}_1, \ldots, \text{arg}_n) \quad c \in \text{contextsOf}(m) \]

\[ \langle hc, o_i \rangle \in pt(c, y) \]

\[ m' = \text{dispatch}(o_i, g) \quad c' = \text{mtdCtxSelector}(c, l, hc, o_i) \]

\[ c' \in \text{contextsOf}(m') \]

\[ \langle hc, o_i \rangle \in pt(c', m'_\text{this}) \]

\[ \forall 1 \leq k \leq n : pt(c, \text{arg}_k) \subseteq pt(c', m'_\text{arg}_k) \]

\[ pt(c', m'_\text{ref}) \subseteq pt(c, x) \]

Figure 2.2: Context-sensitive points-to analysis.

In [NEW], object creation is handled. This rule identifies uniquely the abstract object \( o_i \) created as an instance of \( T \) at allocation site \( i \), with heap context \( hc \) selected by *heapCtxSelector* and puts \( o_i \) (with \( hc \)) into the points-to set of the variable \( x \) under context \( c \). Here, *heapCtxSelector* uses the context of the method \( m \) which contains the allocation site \( i \) and \( i \) itself to compound a heap context \( hc \) for \( o_i \). Then the abstract object \( o_i \) will be analyzed under \( hc \) and distinguished
from the other $o_i$ that are allocated at $i$ but with different heap contexts.

**[Assign]** deals with local assignment. It makes the points-to set of variable $y$ as the subset of the points-to set of variable $x$ under context $c$.

In **[Load]** and **[Store]**, field read and write are handled respectively. Note that we analyze an array object with its elements collapsed to one pseudo-field, denoted $\text{arr}$ as in [80, 42]. Hence, $x = y[i]$ ($x[i] = y$) is handled as $x = y.\text{arr}$ ($x.\text{arr} = y$) by **[Load]** (**[Store]**).

**[Call]** handles method invocation. It uses function $\text{dispatch}(o_i, g)$ to resolve the virtual dispatch of method $g$ on the receiver object $o_i$ to be $m'$. Here, $\text{mtdCtxSelector}$ is used to leverage the information available at call site $l$, i.e., the context $c$ of the method $m$ that contains $l$, the call site $l$ itself, the receiver object $o_i$ and its heap context $hc$, to work out a context $c'$ for the callee $m'$, then $m'$ will be analyzed under context $c'$.

**[Call]** also deals with the context-sensitive interprocedural assignment between call site $l$ and callee $m'$. We assume that $m'$ has a formal parameter $m'_\text{this}$ for the receiver object and $m'_{p1}, ..., m'_{pn}$ for the remaining parameters, and a pseudo-variable $m'_{\text{ret}}$ is used to hold the return value of $m'$. **[Call]** assigns the abstract objects pointed by arguments $y$ and $arg1, ..., argn$ at call site $l$ (under context $c$) to the parameters of callee $m'$, i.e., $m'_\text{this}$ and $m'_{p1}, ..., m'_{pn}$ (under context $c'$). In addition, **[Call]** makes the points-to set of $m'_{\text{ret}}$ (under context $c'$) as the subset of the points-to set of return variable $x$ (under context $c$).

### 2.2.3 Context-Sensitivity

In this section, we present three mainstream variants of context-sensitivity and express them as the instances of $\text{mtdCtxSelector}$ and $\text{heapCtxSelector}$.

In each context-sensitivity, $\text{heapCtxSelector}$ simply selects the context of the
method containing the allocation site as the heap context for the abstract object created at that site (we write “_” to denote an unused arguments):

\[ \text{heapCtxSelector}(c, _) = c \]

It implies that the domains of method context (\(C\)) and heap context (\(HC\)) are the same. For brevity, we omit heapCtxSelector in the rest of this section and focus on mtdCtxSelector.

### 2.2.3.1 Call-Site-Sensitivity

For a method, call-site-sensitivity [71, 99, 34], also referred to as \(k\)-CFA [71, 52] or call-string-sensitivity [70, 60, 80], distinguishes its contexts by sequences of call sites on the call stacks of method invocations that lead to the method. In this case, the domain of method context \(C\) consists of sequences of call sites:

\[ C = I^0 \cup I^1 \cup I^2 \ldots \]

When dealing with a method call, call-site-sensitivity concatenates the context \(c\) of the method containing the current call site and the call site \(l\) itself as the method context for the callee method (\(+\) is a concatenation operator):

\[ \text{mtdCtxSelector}(c, l, _, _) = c + + l \]

### 2.2.3.2 Object-Sensitivity

In contrast to call-site-sensitivity, object-sensitivity distinguishes method contexts by the receiver objects at each call site [53, 55, 75]. The intuition behind object-sensitivity is that in object-oriented languages such as Java, one critical role of instance methods is to manipulate the receiver objects on which they are invoked. Therefore object-sensitivity aims to separately analyze the operations performed on different (receiver) objects to improve the analysis precision.
In object-sensitivity, the context elements are (abstract) heap objects:

\[ C = H^0 \cup H^1 \cup H^2 \ldots \]

The `mtdCtxSelector` for object-sensitivity is defined as above. At a call site, it selects the receiver object \( o_i \) together with its heap context \( hc \) as the context of the callee method:

\[ mtdCtxSelector(-,-, hc, o_i) = hc + o_i \]

In this way, the method contexts of a method totally depend on its receiver objects (with their heap contexts), and the methods invoked on different objects will be analyzed under different contexts.

### 2.2.3.3 Type-Sensitivity

Type-sensitivity [75] is a kind of practical context-sensitivity as it achieves a large part of the precision of object-sensitivity while being more efficient. Technically, it is an approximation of object-sensitivity. Object-sensitivity uses the receiver objects at a call site as the contexts for the callee method. Instead, type-sensitivity replaces the receiver objects in the contexts by the class types which enclose the allocation sites of the receiver objects. Therefore, type-sensitivity approximates object-sensitivity by merging the objects allocated in the same types as contexts. Thus the domain of context in type-sensitivity is composed by sequences of class types:

\[ C = T^0 \cup T^1 \cup T^2 \ldots \]

To formalize type-sensitivity, we define a function `enclosingType : H \rightarrow T` which maps a heap object to a class type enclosing the allocation site of the object. Now we can instantiate `mtdCtxSelector` for type-sensitivity:
As explained in the beginning of Section 2.2.3, heapCtxSelector selects the context of the method which contains the allocation site $o_i$ as $o_i$’s heap context $hc$. So in mtdCtxSelector, the heap context $hc$ of $o_i$ also consists of a sequence of types.
Chapter 3

**BEAN: Precise Points-to Analysis via Object Allocation Graph**

### 3.1 Overview

Two major dimensions for improving precision of points-to analysis are flow-sensitivity and context-sensitivity. For C/C++ programs, flow-sensitivity is needed by many clients [28, 37, 105, 104, 86]. For object-oriented programs, e.g., Java programs, however, context-sensitivity is known to deliver tractable and useful precision [39, 41, 42, 43, 72, 75, 79], in general.

We have introduced several kinds of commonly-used context-sensitivity in Chapter 2. Among all the context abstractions proposed, object-sensitivity is generally the most precise one in practice and is regarded as arguably the best for points-to analysis in object-oriented languages [39, 75, 34]. This can be seen from its widespread adoption in a number of points-to analysis frameworks for Java, such as DOOP [22, 14], CHORD [16] and WALA [98]. In addition, object-sensitivity has also been embraced by many other program analysis tasks, including typestate
verification \cite{25, 106}, data race detection \cite{57}, information flow analysis \cite{6, 27, 50}, and program slicing \cite{43}.

Despite its success, a $k$-object-sensitive points-to analysis, denoted $k$-obj, which uses a sequence of $k$ allocation sites (as $k$ context elements) to represent a calling context of a method call, may end up using some context elements redundantly in the sense that these redundant context elements fail to induce a finer partition of the space of (concrete) calling contexts for the method call. As a result, many opportunities for making further precision improvements are missed.

In this chapter, we introduce BEAN, a general approach for improving the precision of a $k$-object-sensitive points-to analysis, denoted $k$-obj, for Java, by avoiding redundant context elements in $k$-obj while still maintaining a $k$-limiting context abstraction. BEAN can also be considered as a new context-sensitivity approach, which breaks the informed opinion about the consecutive context elements as inherited in the traditional $k$-CFA-based context-sensitivity. The novelty of BEAN lies in identifying redundant context elements by solving a graph problem on an Object Allocation Graph (OAG), which is built based on a pre-analysis (e.g., a context-insensitive Andersen’s analysis) performed initially on a program, and then avoid them in the subsequent $k$-object-sensitive analysis. By construction, BEAN is generally more precise than $k$-obj, with a precision that is guaranteed to be as good as $k$-obj in the worst case.

We have implemented BEAN and applied it to refine two state-of-the-art (whole-program) points-to analyses, $2$-obj and $S$-$2$-obj \cite{34}, provided in DOOP \cite{22}, resulting in two BEAN-directed points-to analyses, $B$-$2$-obj and $B$-$S$-$2$-obj, respectively. We have considered \textit{may-alias} and \textit{may-fail-cast}, two representative clients used elsewhere \cite{23, 75, 79} for measuring the precision of a points-to analysis on a set of nine large Java programs from the DaCapo benchmark suite \cite{8}. Our results show
that $B$-$2$-$obj$ ($B$-$S$-$2$-$obj$) is more precise than $2$-$obj$ ($S$-$2$-$obj$) for every evaluated benchmark under each client, at some small increases in analysis cost.

This chapter presents and validates a new idea on improving the precision of object-sensitive points-to analysis by exploiting an object allocation graph. Considering the broad applications of object-sensitivity in analyzing Java programs, we expect more clients to benefit from BEAN, in practice. Specifically, this chapter makes the following contributions:

- We introduce a new approach, BEAN, for improving the precision of any $k$-object-sensitive points-to analysis, $k$-$obj$, for Java, by avoiding its redundant context elements while maintaining still a $k$-limiting context abstraction.

- We introduce a new kind of graph, called an OAG (object allocation graph), constructed from a pre-analysis for the program, as a general mechanism to identify redundant context elements used in $k$-$obj$.

- We have implemented BEAN as a open-source tool and make it public available at http://www.cse.unsw.edu.au/~corg/bean. BEAN is expected to work well with various points-to analysis frameworks for Java. Currently, we have integrated BEAN with DOOP.

- We have applied BEAN to refine two state-of-the-art object-sensitive points-to analyses for Java. BEAN improves their precision for two representative clients on a set of nine Java programs in DaCapo at small time increases.

The rest of this chapter is organized as follows. We first describe the motivation of BEAN in Section 3.2. Then, in Section 3.3, we present the methodology of BEAN. In Section 3.4, we formalize BEAN, and give several key properties of BEAN. The implementation of BEAN is introduced in Section 3.5. In Section 3.6, we evaluate
the effectiveness of BEAN by comparing BEAN-directed analyses against two state-of-the-art points-to analysis. In Section 3.7, we discuss the work related to BEAN.

3.2 Motivation

When analyzing Java programs, there are two types of context, a method context for local variables and a heap context for object fields. In k-obj, a k-object-sensitive points-to analysis [55, 75], a method context is a sequence of k allocation sites and a heap context is typically a sequence of k – 1 allocation sites as mentioned in Section 2.1. Given an allocation site at label ℓ, ℓ is also referred to as an abstract object for the site.

Currently, k-obj, where k = 2, represents a 2-object-sensitive analysis with a 1-context-sensitive heap (with respect to allocation sites), denoted 2-obj [34], which usually achieves the best trade-off between precision and scalability and has thus been widely adopted in points-to analysis for Java [23, 43, 75]. In 2-obj, a heap context for an abstract object ℓ is a receiver object of the method that made the allocation of ℓ (known as an allocator object), and a method context for a method call is a receiver object of the method plus its allocator object.

Below we examine the presence of redundant context elements in 2-obj, with two examples, one for method contexts and one for heap contexts. This serves to motivate the BEAN approach proposed for avoiding such redundancy.

3.2.1 Redundant Elements in Method Contexts

We use an example in Figure 3.1 to illustrate how 2-obj analyzes it imprecisely due to its use of a redundant context element in method contexts and how BEAN avoids the imprecision by avoiding this redundancy. We consider a may-alias client
Chapter 3. **BEAN**: Precise Points-to Analysis via Object Allocation Graph

```java
1  void main(Object[] args) {
2      A a1 = new A(); // A/1
3          Object v1 = a1.foo(new Object()); // O/1
4  
5      A a2 = new A(); // A/2
6          Object v2 = a2.foo(new Object()); // O/2
7  }
8  class A {
9      Object foo(Object v) {
10             B b = new B(); // B/1
11                 return b.bar(v);
12          }
13     }
14  class B {
15      Object bar(Object v) {
16             C c = new C(); // C/1
17                 return c.identity(v);
18          }
19     }
20  class C {
21      Object identity(Object v) { return v; }
22  }
```

(a) Program

(b) Context-sensitive call graph by 2-obj

(c) Context-sensitive call graph by **BEAN**

Figure 3.1: Method contexts for 2-obj and **BEAN**.
that queries for the alias relation between variables \( v_1 \) and \( v_2 \).

In Figure 3.1(a), we identify the six allocation sites by their labels given in their end-of-line comments, i.e., \( A/1 \), \( A/2 \), \( O/1 \), \( O/2 \), \( B/1 \), and \( C/1 \).

In Figure 3.1(b), we give the context-sensitive call graph computed by 2-\textit{obj}, where each method is analyzed separately for each different calling context, denoted by \([\ldots]\). \texttt{C.identity()} has two concrete calling contexts but is analyzed only once under \([B/1,C/1]\). We can see that \( B/1 \) is redundant (relative to \( C/1 \)) since adding \( B/1 \) to \([C/1]\) fails to separate the two concrete calling contexts. As a result, variables \( v_1 \) and \( v_2 \) are made to point to both \( O/1 \) and \( O/2 \) at the same time, causing \textit{may-alias} to report a spurious alias. During any program execution, \( v_1 \) and \( v_2 \) can only point to \( O/1 \) and \( O/2 \), respectively.

In Figure 3.1(c), we give the context-sensitive call graph computed by \texttt{Bean}, where \texttt{C.identity()} is now analyzed separately under two different contexts, \([A/1,C/1]\) and \([A/2,C/1]\). Due to the improved precision, \( v_1 \) (\( v_2 \)) now points to \( O/1 \) (\( O/2 \)) only, causing \textit{may-alias} to conclude that both are no longer aliases.

### 3.2.2 Redundant Elements in Heap Contexts

We now use an example in Figure 3.2 to illustrate how 2-\textit{obj} analyzes it imprecisely due to its use of a redundant element in heap contexts and how \texttt{Bean} avoids the imprecision by avoiding this redundancy. Our \textit{may-alias} client now issues an alias query for variables \texttt{emp1} and \texttt{emp2}. In Figure 3.2(a), we identify again its six allocation sites by their labels given at their end-of-line comments.

Figure 3.2(b) shows the context-sensitive field points-to graph computed by 2-\textit{obj}, where each node represents an abstract heap object created under the corresponding context, denoted \([\ldots]\), and each edge represents a field points-to relation with the corresponding field name being labeled on the edge. An array object is
Chapter 3. **BEAN**: Precise Points-to Analysis via Object Allocation Graph

1. `void main(String[] args) {`
2. `Company comp1 = new Company(); // Co/1`
3. `comp1.addEmployee(new Employee()); // Emp/1`
4. `Employee emp1 = comp1.getEmployee(0);`
5. `Company comp2 = new Company(); // Co/2`
6. `comp2.addEmployee(new Employee()); // Emp/2`
7. `Employee emp2 = comp2.getEmployee(0);`
8. `}
9. `class Employee {...}
10. `class Company {
11. `private ArrayList emps;
12. `Company() { emps = new ArrayList(); } // AL/1
13. `void addEmployee(Employee emp) { emps.add(emp); }
14. `Employee getEmployee(int i) {
15. `return (Employee) emps.get(i);
16. `}
17. `}
18. `}
19. `class ArrayList {
20. `private Object[] elems;
21. `private int size = 0;
22. `ArrayList() { elems = new Object[10]; } // Obj[]/1
23. `void add(Object e) { elems[size++] = e; }
24. `Object get(int i) { return elems[i]; }
25. `}

(a) Program

(b) Context-sensitive field points-to graph by 2-obj

(c) Context-sensitive field points-to graph by BEAN

Figure 3.2: Heap contexts for 2-obj and BEAN.
analyzed with its elements collapsed to one pseudo-field, denoted \texttt{arr}. Hence, \( x[i] = y \) (\( y = x[i] \)) is handled as \( x.\texttt{arr} = y \) (\( y = x.\texttt{arr} \)).

In this example, two companies, \texttt{Co/1} and \texttt{Co/2}, maintain their employee information by using two different \texttt{ArrayLists}, with each implemented internally by a distinct array of type \texttt{Object[]} at line 22. However, \texttt{2-obj} has modelled the two array objects imprecisely by using one abstract object \texttt{Obj[]} under \[ \texttt{AL/1} \]. Note that \texttt{AL/1} is redundant since adding it to \[ \] makes no difference to the handling of \texttt{Obj[]} under \[ \texttt{AL/1} \]. As a result, \texttt{emp1} and \texttt{emp2} will both point to \texttt{Emp/1} and \texttt{Emp/2}, causing \textit{may-alias} to regard both as aliases conservatively.

Figure 3.2(c) shows the context-sensitive field points-to graph computed by \texttt{Bean}. This time, the \texttt{Object[]} arrays used by two companies \texttt{Co/1} and \texttt{Co/2} are distinguished under two distinct heap contexts \[ \texttt{Co/1} \] and \[ \texttt{Co/2} \]. As a result, our \textit{may-alias} client will no longer report \texttt{emp1} and \texttt{emp2} to be aliases.

### 3.2.3 Discussion

As illustrated above, \textit{k-obj} selects blindly a sequence of \textit{k}-most-recent allocation sites as a context. To analyze large-scale software scalably, \textit{k} is small, which is 2 for a method context and 1 for a heap context in \texttt{2-obj}. Therefore, redundant context elements, such as \texttt{B/1} in \[ \texttt{B/1,C/1} \] in Figure 3.1(b) and \texttt{AL/1} in \[ \texttt{AL/1} \] in Figure 3.2(b), should be avoided since they waste precious space in a context yet contribute nothing in separating the concrete calling contexts for a call site.

This chapter aims to address this problem in \textit{k-obj} by excluding redundant elements from its contexts so that their limited context positions can be more profitably exploited to achieve better precision, as shown in Figures 3.1(c) and 3.2(c).
We introduce a new approach, BEAN, as illustrated in Figure 3.3, to improving the precision of a $k$-object-sensitive points-to analysis, $k$-obj. The basic idea is to refine $k$-obj by avoiding its redundant context elements while maintaining still a $k$-limiting context abstraction. An element $e$ in a context $c$ for a call or allocation site is redundant if $c$ with $e$ removed does not change the context represented by $c$. For example, B/1 in [B/1,C/1] in Figure 3.1(b) and AL/1 in [AL/1] in Figure 3.2(b) are redundant.

BEAN proceeds in two stages. In Stage 1, we aim to identify redundant context elements used in $k$-obj. To achieve this, we first perform usually a fast but imprecise pre-analysis, e.g., a context-insensitive Andersen’s points-to analysis on a program to obtain its points-to information. Based on the points-to information discovered, we construct an object allocation graph (OAG) to capture the object allocation relations in $k$-obj. Subsequently, we traverse the OAG to select method and heap contexts by avoiding redundant context elements that would otherwise be used by $k$-obj. In Stage 2, we refine $k$-obj by avoiding its redundant context elements. Essentially, we perform a $k$-object-sensitive analysis in the normal way, by using the contexts selected in the first stage, instead.
3.3.1 Object Allocation Graph

The OAG of a program is a directed graph, $G = (N, E)$. A node $\ell \in N$ represents a label of an (object) allocation site in the program. An edge $\ell_1 \rightarrow \ell_2 \in E$ represents an object allocation relation. As $G$ is context-insensitive, a label $\ell \in G$ is also interchangeably referred to (in the literature) as the (unique) abstract heap object that models all the concrete objects created at the allocation site $\ell$. Given this, $\ell_1 \rightarrow \ell_2$ signifies that $\ell_1$ is the receiver object of the method that made the allocation of $\ell_2$. Therefore, $\ell_1$ is called an allocator object of $\ell_2$ [75].

Figure 3.4 gives the OAGs for the two programs in Figures 3.1 and 3.2, which are deliberately designed to be isomorphic. In Figure 3.4(a), A/1 and A/2 are two allocators of B/1. In Figure 3.4(b), AL/1 is an allocator of Obj[]/1. Some objects, e.g., those created in main() or static initialisers, have no allocators. For convenience, we assume the existence of a dummy node, O_root, so that every object has at least one allocator. The isomorphic OAG in Figure 3.4(c) will be referred to in Example 2.

The concept of allocator object captures the essence of object sensitivity. By definition [55, 75], a context for an allocation site $\ell$, i.e., an abstract object $\ell$,
consists of its allocator object \((\ell')\), the allocator object of \(\ell'\), and so on. The OAG provides a new perspective for object sensitivity, since a context for an object \(\ell\) is simply a path from \(0_{\text{root}}\) to \(\ell\). As a result, the problem of selecting contexts for \(\ell\) can be recast as one of solving a problem of distinguishing different paths from \(0_{\text{root}}\) to \(\ell\). Traditionally, a \(k\)-object-sensitive analysis selects blindly a suffix of a path from \(0_{\text{root}}\) to \(\ell\) with length \(k\).

### 3.3.2 Context Selection

Given an object \(\ell\) in \(G\), BEAN selects its contexts in \(G\) as sequences of its direct or indirect allocators that are useful to distinguish different paths from \(0_{\text{root}}\) to \(\ell\) while avoiding redundant ones that would otherwise be used in \(k\)-obj. The key insight is that in many cases, it is unnecessary to use all nodes of a path to distinguish the path from the other paths leading to the same node. In contrast, \(k\)-obj is not equipped with \(G\) and thus has to select blindly a suffix of each such path as a context, resulting in many redundant context elements being used.

**Method Contexts**  Figure 3.1 compares the method contexts used by \(2\)-obj and BEAN for the first example given. As shown in Figure 3.1(b), \(2\)-obj analyzes \(\text{C.identity}()\) under one context \([B/1, C/1]\), where \(B/1\) is redundant, without being able to separate its two concrete calling contexts. In contrast, BEAN avoids using \(B/1\) by examining the OAG of this example in Figure 3.4(a). There are two paths from \(0_{\text{root}}\) to \(C/1\): \(0_{\text{root}} \rightarrow A/1 \rightarrow B/1 \rightarrow C/1\) and \(0_{\text{root}} \rightarrow A/2 \rightarrow B/1 \rightarrow C/1\). Note that \(2\)-obj has selected a suffix of the two paths, \(B/1 \rightarrow C/1\), which happens to represent the same context \([B/1, C/1]\) for \(\text{C.identity}()\). BEAN distinguishes these two paths by ignoring the redundant node \(B/1\), thereby settling with the method contexts shown in Figure 3.1(c). As a result, \(\text{C.identity}()\) is now analyzed under
Chapter 3. BEAN: Precise Points-to Analysis via Object Allocation

Graph

two different contexts \([A/1, C/1]\) and \([A/2, C/1]\) more precisely.

**Heap Contexts**  Figure 3.2 compares the heap contexts used by 2-obj and BEAN for the second example given. As shown in Figure 3.2(b), 2-obj fails to separate the two array objects created at the allocation site 0bj []/1 for two companies Co/1 and Co/2 by using one context [AL/1], where AL/1 is redundant. In contrast, BEAN avoids using AL/1 by examining the OAG of this example in Figure 3.4(b). There are two paths from 0_root to 0bj []/1: 0_root → Co/1 → AL/1 → 0bj []/1 and 0_root → Co/2 → AL/1 → 0bj []/1. Note that 2-obj has selected a suffix of the two paths, AL/1 → 0bj []/1, which happens to represent the same heap context [AL/1] for 0bj []/1. BEAN distinguishes these two paths by ignoring the redundant node AL/1, thereby settling with the heap contexts shown in Figure 3.2(c). As a result, the two array objects created at 0bj []/1 are distinguished under two different contexts [Co/1] and [Co/2] more precisely.

### 3.3.3 Discussion

BEAN, as shown in Figure 3.3, is designed to be a general-purpose technique for refining k-obj with three design goals, under the condition that its pre-analysis is sound:

- As the pre-analysis is usually less precise than k-obj, the OAG constructed for the program may contain some object allocation relations that are not visible in k-obj. Therefore, BEAN is not expected to be optimal in the sense that it can avoid all redundant context elements in k-obj.

- If the pre-analysis is more precise than k-obj (e.g., in some parts of the program), then the OAG may miss some object allocation relations that are visible in k-obj. This allows BEAN to avoid using context elements that
are redundant with respect to the pre-analysis but not \( k\)-\textit{obj}, making the resulting analysis even more precise.

- \textsc{Bean} is expected to be more precise than \( k\)-\textit{obj} in general, with a precision that is guaranteed to be as good as \( k\)-\textit{obj} in the worst case.

### 3.4 Formalism

Without loss of generality, we formalize \textsc{Bean} as a \( k\)-object-sensitive points-to analysis with a \((k-1)\)-context-sensitive heap (with respect to allocation sites), as in [34]. Thus, the depth of its method (heap) contexts is \( k \) \((k-1)\). We use the formulation and notations of points-to analysis presented in Section 2.2.3.

#### 3.4.1 Object Allocation Graph

Figure 3.5 defines the OAG used for a program: \( o_i \rightarrow o_j \) means that \( o_i \) is an allocator object of \( o_j \), i.e., the receiver object of the method that made the allocation of \( o_j \).

\[
\text{OAG} \quad G = (N, E) \\
\text{node} \quad o_i, o_j \in N \subseteq H \\
\text{edge} \quad o_i \rightarrow o_j \in E \subseteq N \times N
\]

Figure 3.5: Notations for object allocation graph.

Figure 3.6 gives the rules for building the OAG, \( G = (N, E) \), for a program, based on the points-to sets computed by a pre-analysis, which may or may not be context-sensitive. As \( G \) is (currently) context-insensitive, the context information that appears in a points-to set (if any) is simply ignored. [OAG-Node] and [OAG-DummyNode] build \( N \). [OAG-Edge] and [OAG-DummyEdge] build \( E \).

By [OAG-Node], we add to \( N \) all the pointed-to target objects found during the pre-analysis. By [OAG-DummyNode], we add a dummy node \( o_{\text{root}} \) to \( N \).
Chapter 3. BEAN: Precise Points-to Analysis via Object Allocation Graph

Graph 3.6: Rules for building the OAG, \( G = (N, E) \), for a program based on a pre-analysis.

By [OAG-Edge], we add to \( E \) an edge \( o_i \rightarrow o_j \) if \( o_i \) is an allocator object of \( o_j \). Here, \( m_{\text{this}} \), where \( m \) is the name of a method, represents the \texttt{this} variable of method \( m \), which points to the receiver object of method \( m \). By [OAG-DummyEdge], we add an edge from \( o_{\text{root}} \) to every object \( o_i \) without any incoming edge yet, to indicate that \( o_{\text{root}} \) is now a pseudo allocator object of \( o_i \). Note that an object allocated in \texttt{main()} or a static initialiser does not have an allocator object. Due to \( o_{\text{root}} \), every object has at least one allocator object.

Example 1 Figure 3.4 gives the OAGs for the two programs in Figures 3.1 and 3.2. For reasons of symmetry, let us apply the rules in Figure 3.6 to build the OAG in Figure 3.4(a) only. Suppose we perform a context-insensitive Andersen’s points-to analysis as the pre-analysis on the program in Figure 3.1. The points-to sets are: \( pt(v1) = pt(v2) = \{0/1, 0/2\} \), \( pt(a1) = \{A/1\} \), \( pt(a2) = \{A/2\} \), \( pt(b) = \{B/1\} \), and \( pt(c) = \{C/1\} \). By [OAG-Node] and [OAG-DummyNode], \( N = \{o_{\text{root}}, A/1, A/2, B/1, C/1, 0/1, 0/2\} \). By [OAG-Edge], we add \( A/1 \rightarrow B/1 \), \( A/2 \rightarrow B/1 \) and \( B/1 \rightarrow C/1 \), since \( B/1 \) is allocated in \texttt{foo()} with the receiver objects being \( A/1 \) and \( A/2 \) and \( C/1 \) is allocated in \texttt{bar()} on the receiver object \( B/1 \). By [OAG-DummyEdge], we add \( o_{\text{root}} \rightarrow A/1 \), \( o_{\text{root}} \rightarrow A/2 \), \( o_{\text{root}} \rightarrow 0/1 \) and \( o_{\text{root}} \rightarrow 0/2 \).

Due to recursion, an OAG may have cycles including self-loops. This means
that an abstract heap object may be a direct or indirect allocator object of another
heap object, and conversely (with both being possibly the same).

3.4.2 Context Selection

Figure 3.7 establishes some basic relations in an OAG, \( G = (N, E) \), with possibly
cycles. By \( \text{[Reach-Reflexive]} \) and \( \text{[Reach-Transitive]} \), we speak of graph reach-
ability in the standard manner. In \( \text{[Confluence]} \), \( o_i \) identifies a conventional
confluence point. In \( \text{[Divergence]} \), \( o_i < o_t \) states that \( o_i \) is a divergence point, with
at least two outgoing paths reaching \( o_t \), implying that either \( o_i \) is a confluence point
or at least one confluence point exists earlier on the two paths.

\[
\begin{align*}
\frac{o_i \in N}{o_i \sim o_i} \quad & \text{[Reach-Reflexive]} \quad \frac{o_i \rightarrow o_j \in E \quad o_j \sim o_k}{o_i \sim o_k} \quad & \text{[Reach-Transitive]} \\
\frac{o_j \rightarrow o_i \in E \quad o_k \rightarrow o_i \in E \quad o_j \neq o_k}{\forall o_i} \quad & \text{[Confluence]} \\
\frac{o_i \rightarrow o_j \in E \quad o_i \rightarrow o_k \in E \quad o_j \neq o_k \quad o_j \sim o_t \quad o_k \sim o_t}{o_i < o_t} \quad & \text{[Divergence]}
\end{align*}
\]

Figure 3.7: Rules for basic relations in an OAG, \( G = (N, E) \).

Figure 3.8 gives the rules for computing two context selectors, \( \text{heapCtxSelector} \)
and \( \text{mtdCtxSelector} \), introduced in Section 2.2.2 and used in refining an object-
sensitive points-to analysis. In \( \text{heapCtxSelector}(c, o_i) = hc \), \( c \) denotes an (abstract
calling) context of the method that made the allocation of object \( o_i \) and \( hc \) is the
heap context selected for \( o_i \) when \( o_i \) is allocated in the method with context \( c \).
In \( \text{mtdCtxSelector}(\_, \_, hc, o_i) = c \) (the first two arguments are unused here and
ignored), \( hc \) denotes a heap context of object \( o_i \), and \( c \) is the method context
selected for the method whose receiver object is \( o_i \) under its heap context \( hc \).

For \( k\text{-}obj \) \( [55, 75] \), both context selectors are simple. In the case of full-
Chapter 3. **BEAN: Precise Points-to Analysis via Object Allocation Graph**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o_i \in N ) ( o_{root} \to o_i \in E ) ( o_{i} \rightsquirl o_{t} )</td>
<td>If ( o_i = o_t ) then ( heapCtxSelector([], o_i) = [] ) [HCTX-INIT]</td>
</tr>
<tr>
<td>( o'<em>j : \langle rep', hc' \rangle ) ( o_j \to o_i \in E ) ( o</em>{i} \rightsquirl o_{t} ) ( o_j \neq o_t )</td>
<td>If ( o_j = o_t ) then ( heapCtxSelector(hc' \to o_j, o_i) = hc' ) [HCTX-DIV]</td>
</tr>
<tr>
<td>( o'<em>j : \langle rep, hc \rangle ) ( o_j \to o_i \in E ) ( o</em>{i} \rightsquirl o_{t} ) ( o_j = o_t )</td>
<td>If ( o_j = o_t ) then ( heapCtxSelector(hc' \to o_j, o_i) = hc' ) [HCTX-CYC]</td>
</tr>
<tr>
<td>( heapCtxSelector(_ , o_i) = hc ) ( c = hc' \to o_j, o_i )</td>
<td>( mtdCtxSelector(_ , hc, o_i) = c ) [MCTX]</td>
</tr>
</tbody>
</table>

Figure 3.8: Rules for context selection in an OAG, \( G = (N, E) \).

object-sensitivity, we have \( heapCtxSelector([o_1, ..., o_{n-1}], o_n) = [o_1, ..., o_{n-1}] \) and \( mtdCtxSelector(\_ , \langle o_1, ..., o_{n-1} \rangle, o_n) = [o_1, ..., o_n] \) for every path from \( o_{root} \) to a node \( o_n \) in the OAG, \( o_{root} \to o_1 \to ... \to o_{n-1} \to o_n \). For a \( k \)-object-sensitive analysis with a \((k-1)\)-context-sensitive heap, \( heapCtxSelector([o_{n-k}, ..., o_{n-1}], o_n) = [o_{n-k}, ..., o_{n-1}] \) and \( mtdCtxSelector(\_ , \langle o_{n-k+1}, ..., o_{n-1} \rangle, o_n) = [o_{n-k+1}, ..., o_n] \). Essentially, a suffix of length of \( k \) is selected from \( o_{root} \to o_1 \to ... \to o_{n-1} \to o_n \), resulting in potentially many redundant context elements to be used blindly.

Let us first use an OAG in Figure 3.9 to explain how we avoid redundant context elements selected by \( k \)-obj. The set of contexts for a given node, denoted \( o_i \), can be seen as the set of paths reaching \( o_i \) from \( o_{root} \). Instead of using all the nodes on a path to distinguish it from the other four, we use only the five representative nodes, labeled by 1 – 5, and identify the five paths uniquely as \( 1 \to 3, 1 \to 4 \),
Chapter 3. **BEAN: Precise Points-to Analysis via Object Allocation Graph**

The rules in Figure 3.8 are used to identify such representative nodes (on the paths from a divergence node to a confluence node) and compute the contexts for \( o_i \).

![Figure 3.9: An OAG.](image)

In Figure 3.8, the first three rules select heap contexts and the last rule selects method contexts based on the heap contexts selected. The first three rules traverse the OAG from \( o_{root} \) and select heap contexts for a node \( o_i \). Meanwhile, each rule also records at \( o_i \), which reaches \( o_i \), a set of pairs of the form \( o_i : \langle \text{rep}, \text{hc} \rangle \). For a pair \( o_i : \langle \text{rep}, \text{hc} \rangle \), \( \text{hc} \) is a heap context of \( o_i \) that uniquely represents a particular path from \( o_{root} \) to \( o_i \). In addition, \( \text{rep} \) is a boolean flag considered for determining the suitability of \( o_i \) as a representative node, i.e., context element for \( o_i \) *under* \( \text{hc} \) (i.e., for the path \( \text{hc} \) leading to \( o_i \)). There are two cases. If \( \text{rep} = \text{false} \), then \( o_i \) is redundant for \( o_i \). If \( \text{rep} = \text{true} \), then \( o_i \) is potentially a representative node (i.e., context element) for \( o_i \). \( \text{hc} \gg o \) returns the concatenation of \( \text{hc} \) and \( o \).

Specifically, for the first three rules on heap contexts, \([\text{Hctx-Init}]\) bootstraps heap context selection, \([\text{Hctx-Cyc}]\) handles the special case when \( o_i \) is in a cycle such that \( o_j = o_i \), and \([\text{Hctx-Div}]\) handles the remaining cases. In \([\text{Mctx}]\), the contexts
for a method are selected based on its receiver objects and the heap contexts of these receiver objects computed by the first three rules. Thus, removing redundant elements from heap contexts benefits method contexts directly.

Figure 3.10 illustrates the four non-trivial cases marked in Figure 3.8, i.e., ①, ② (split into two sub-cases), and ③. In ①, o_t appears on a divergent path from o_j leading to o_t, o_j’s rep’ is set to true to mark o_i as a potential context element for o_t. In ②, there are two sub-cases: ¬o_j ≺ o_t and o_j ≺ o_t. In both cases, o_j is in a branch (since o_j’s rep is true) and o_i is a confluence node (since ∀o_i holds). Thus, o_j is included as a context element for o_t. In the case of ¬o_j ≺ o_t, o_i is redundant for o_t under c. In the case of o_j ≺ o_t, the paths to o_t diverge at o_j. Thus, o_i can be potentially a context element to distinguish the paths from o_j to o_t via o_i. If o_i is ignored, the two paths o_j → o_k → o_t and o_j → o_i → o_k → o_t as shown cannot be distinguished. In [Hctx-Cyc], its two cases are identically handled as the last two cases in [Hctx-Cyc], except that [Hctx-Cyc] always sets o_j’s rep’ to true. If [Hctx-Cyc] is applicable, o_t must appear in a cycle such that o_j = o_t. Then, any successor of o_t may be a representative node to be used to distinguish the paths leading to o_t via the cycle. Thus, o_j’s rep’ is set to true. The first case in [Hctx-Cyc], marked as ③ in Figure 3.8, is illustrated in Figure 3.10.

To enforce k-limiting in the rules given in Fig 3.8, we simply make every method context hc ++ o_t k-bounded and every heap context hc ++ o_j (k - 1)-bounded.

**Example 2** For the two programs illustrated in Figures 3.1 and 3.2, Bean is more precise than 2-obj (with k = 2) in handling the method and heap contexts of o_4, shown in their isomorphic OAG in Figure 3.4(c). We give some relevant derivations for o_t, with t = 4, only. By [Hctx-Init], we obtain o_1^t : (true, []) and o_2^t : (true, []). By [Hctx-Div], we obtain o_3^t : (false, [o_1]), o_3^t : (false, [o_2]), o_4^t : (false, [o_1]) and o_4^t : (false, [o_2]). Thus, heapCtxSelector([o_1, o_3], o_4) = [o_1] and
Figure 3.10: Three Cases marked for \([HCTX-DIV]\) and \([HCTX-CYC]\) in Figure 3.8.
heapCtxSelector([o2, o3], o4) = [o2]. By [Mctx], mtdCtxSelector(\_, \_, [o1], o4) = [o1, o4], and mtdCtxSelector(\_, \_, [o2], o4) = [o2, o4]. For 2-obj, the contexts selected for o4 are heapCtxSelector([o1, o3], o4) = [o3], heapCtxSelector([o2, o3], o4) = [o3] and mtdCtxSelector(\_, \_, [o3], o4) = [o3, o4]. As result, Bean can successfully separate the two concrete calling contexts for o4 and the two o4 objects created in the two contexts but 2-obj fails to do this.

3.4.3 Bean-directed Object-Sensitive Points-to Analysis

To make use of Bean in object-sensitive points-to analysis, we just need to replace the context selectors mtdCtxSelector and heapCtxSelector of points-to analysis in Figure 2.2 by the selectors computed by Bean in Figure 3.8.

Compared to k-obj, Bean avoids its redundant context elements in [New] and [Call]. In [New], heapCtxSelector (by [Hctx-Init], [Hctx-Div] and [Hctx-Cyc]) is used to select the contexts for object allocation. In [Call], mtdCtxSelector (by [Mctx]) is used to select the contexts for method invocation.

3.4.4 Properties

Theorem 1 Under full-context-sensitivity (i.e., when k = \infty), Bean is as precise as the traditional k-object-sensitive points-to analysis (k-obj).

Proof Sketch. The set of contexts for any given abstract object, say o_t, is the set P_t of its paths reaching o_t from o_root in the OAG of the program. Let R_t be the set of representative nodes, i.e., context elements identified by Bean for o_t. We argue that R_t is sufficient to distinguish all the paths in P_t (as shown in Figure 3.9).

For the four rules given in Figure 3.8, we only need to consider the first three for selecting heap contexts as the last one for method contexts depends on the first three. [Hctx-Init] performs the initialisation for the successor nodes of o_root.
Chapter 3. **BEAN: Precise Points-to Analysis via Object Allocation Graph**

[HCTX-Div] handles all the situations except the special one when \( o_t \) is in a cycle such that \( o_t = o_j \). [HCTX-Div] has three cases. In the first case, marked ① (Figure 3.10), our graph reachability analysis concludes conservatively whether it has processed a divergence node or not during the graph traversal. In the second case, marked ② (Figure 3.10), \( o_t \) is a confluence node. By adding \( o_j \) to \( hc \) in \( hc \leftarrow o_j \), we ensure that for each path \( p \) from \( o_i \)’s corresponding divergence node to \( o_t \) traversed earlier, at least one representative node that is able to represent \( p \), i.e., \( o_j \), is always selected, i.e., to \( R_t \). In cases ① and ②, as all the paths from \( o_{\text{root}} \) to \( o_t \) are traversed, all divergence and confluence nodes are handled. The third case simply propagates the recorded information across the edge \( o_j \rightarrow o_t \).

[HCTX-Cyc] applies only when \( o_t \) is in a cycle such that \( o_t = o_j \). Its two cases are identical to the last two cases in [HCTX-Div] except \( o_j \)’s rep’ is always set to true. This ensures all the paths via the cycle can be distinguished correctly. In the case, marked ③ and illustrated in Figure 3.10, \( o_j \) is selected, i.e., added to \( R_t \).

Thus, \( R_t \) is sufficient to distinguish the paths in \( P_t \). Hence, the theorem. □

**Theorem 2** For any fixed context depth \( k \), **BEAN** is no less precise than the traditional \( k \)-object-sensitive points-to analysis (\( k \)-obj).

**Proof Sketch.** This follows from the fact that, for a fixed \( k \), based on Theorem 1, **BEAN** will eliminate some redundant context elements in a sequence of \( k \)-most-recent allocation sites in general or nothing at all in the worst case. Thus, **BEAN** may be more precise than (by distinguishing more contexts for a call or allocation site) or has the same precision as \( k \)-obj (by using the same contexts). □
3.5 Implementation

We have implemented BEAN as a standalone tool for performing OAG construction (Figure 3.6) and context selection (Figure 3.8), as shown in Figure 3.3, in Java. To demonstrate the relevance of BEAN to points-to analysis, we have integrated BEAN with DOOP [22], a state-of-the-art context-sensitive points-to analysis framework for Java. To apply BEAN to refine an existing object-sensitive points-to analysis written in Datalog from DOOP, it is only necessary to modify some Datalog rules in DOOP to adopt the contexts selected by `heapCtxSelector` and `mtdCtxSelector` in BEAN (Figure 3.8).

Our entire BEAN framework has been released as open-source software at http://www.cse.unsw.edu.au/~corg/bean.

3.6 Evaluation

In our evaluation, we attempt to answer the following two research questions:

**RQ1.** Can BEAN improve the precision of an object-sensitive points-to analysis at slightly increased cost to enable a client to answer its queries more precisely?

**RQ2.** Does BEAN make any difference for a real-world application?

To address RQ1, we apply BEAN to refine two state-of-the-art whole-program object-sensitive points-to analyses, 2-obj and S-2-obj, the top two most precise yet scalable solutions provided in DOOP [22, 34], resulting in two BEAN-directed analyses, B-2-obj and B-S-2-obj, respectively. Altogether, we will compare the following five context-sensitive points-to analyses:

- **2-cs:** 2-call-site-sensitive analysis [22]
• 2-obj: 2-object-sensitive analysis with 1-context-sensitive heap [22]

• B-2-obj: the BEAN-directed version of 2-obj

• S-2-obj: selective hybrids of 2 object-sensitive analysis proposed in [22, 34]

• B-S-2-obj: the BEAN-directed version of S-2-obj

Note that 2-obj is discussed in Section 3.2. S-2-obj is a selective 2-object-sensitive with 1-context-sensitive heap hybrid analysis [34], which applies call-site-sensitivity to static call sites and 2-obj to virtual call sites. For S-2-obj, BEAN proceeds by refining its object-sensitive part of the analysis, demonstrating its generality in improving the precision of both pure and hybrid object-sensitive analyses. For comparison purposes, we have included 2-cs to demonstrate the superiority of object-sensitivity over call-site-sensitivity.

We have considered may-alias and may-fail-cast, two representative clients used elsewhere [39, 79, 75, 23] for measuring the precision of points-to analysis. The may-alias client queries whether two variables may point to the same object or not. The may-fail-cast client identifies the type casts that may fail at run time.

To address RQ2, we show how BEAN can enable may-alias and may-fail-cast to answer alias queries more precisely for java.util.HashSet. This container from the Java library is extensively used in real-world Java applications.

3.6.1 Experimental Setting

All the five points-to analyses evaluated are written in terms of Datalog rules in the DOOP framework (version r160113) [14]. In our experiments, the pre-analysis for a program is performed by using a context-insensitive Andersen’s points-to analysis provided in DOOP. Our evaluation setting uses the LogicBlox Datalog engine (v3.9.0), on an Xeon E5-2650 2GHz machine with 64GB of RAM.
We use all the Java programs in the DaCapo benchmark suite (2006-10-MR2) [8] except hsqldb and jython, because all the four object-sensitive analyses, cannot finish analyzing each of the two in a time budget of 5 hours. All these benchmarks are analyzed together with a large Java library, JDK 1.6.0_45.

DOOP handles native code (in terms of summaries) and (explicit and implicit) exceptions [14, 33]. As for reflection, we leverage SOLAR [42] by adopting its string inference to resolve reflective calls but turning off its other inference mechanisms that may require manual annotations. We have also enabled DOOP to merge some objects, e.g., reflection-irrelevant string constants, in order to speed up each analysis without affecting its precision noticeably, as in [22, 34].

When analyzing a program, by either a pre-analysis or any of the five points-to analyses evaluated, its native code, exceptions and reflective code are all handled in exactly the same way. Even if some parts of the program are unanalyzed, we can still speak of the soundness of all these analyses with respect to the part of the program visible to the pre-analysis. Thus, Theorems 1 and 2 still hold.

3.6.2  RQ1: Precision and Performance Measurements

Table 3.1 compare the precision and performance results for the five analyses.

Precision  We measure the precision of a points-to analysis in terms of the number of may-alias variable pairs reported by may-alias and the number of may-fail-casts reported by may-fail-cast. For the may-alias client, the obvious aliases (e.g., due to a direct assignment) have been filtered out, following [23]. The more precise a points-to analysis is, the smaller these two numbers will be.

Let us consider may-alias first. B-2-obj improves the precision of 2-obj for all the nine benchmarks, ranging from 6.2% for antlr to 16.9% for xalan, with an
### Table 3.1: Precision and performance results for all the five analyses.

The two precision metrics shown are the number of variable pairs that may be aliases generated by *may-alias* ("may-alias pairs") and the number of casts that cannot be statically proved to be safe by *may-fail-cast* ("may-fail casts"). In both cases, *smaller is better*. One performance metric used is the analysis time for a program.

<table>
<thead>
<tr>
<th></th>
<th>2-cs</th>
<th>2-obj</th>
<th>B-2-obj</th>
<th>S-2-obj</th>
<th>B-S-2-obj</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>xalan</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>25,245,307</td>
<td>6,196,945</td>
<td><strong>5,146,694</strong></td>
<td>5,652,610</td>
<td><strong>3,958,998</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>1154</td>
<td>711</td>
<td>653</td>
<td>608</td>
<td>550</td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>1400</td>
<td>8653</td>
<td>11450</td>
<td>1150</td>
<td>1376</td>
</tr>
<tr>
<td><strong>chart</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>43,124,320</td>
<td>4,189,805</td>
<td><strong>3,593,584</strong></td>
<td>3,485,082</td>
<td><strong>3,117,825</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>2026</td>
<td>1064</td>
<td><strong>979</strong></td>
<td>923</td>
<td><strong>844</strong></td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>3682</td>
<td>630</td>
<td>1322</td>
<td>1145</td>
<td>1814</td>
</tr>
<tr>
<td><strong>eclipse</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>20,979,544</td>
<td>5,029,492</td>
<td><strong>4,617,883</strong></td>
<td>4,636,675</td>
<td><strong>4,346,306</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>1096</td>
<td>722</td>
<td><strong>655</strong></td>
<td>615</td>
<td><strong>551</strong></td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>1076</td>
<td>119</td>
<td>175</td>
<td>119</td>
<td>188</td>
</tr>
<tr>
<td><strong>fop</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>38,496,078</td>
<td>10,548,491</td>
<td><strong>9,870,507</strong></td>
<td>9,613,363</td>
<td><strong>9,173,539</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>1618</td>
<td>1198</td>
<td><strong>1133</strong></td>
<td>1038</td>
<td><strong>973</strong></td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>3054</td>
<td>796</td>
<td>1478</td>
<td>961</td>
<td>1566</td>
</tr>
<tr>
<td><strong>luindex</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>10,486,363</td>
<td>2,190,854</td>
<td><strong>1,949,134</strong></td>
<td>1,820,992</td>
<td><strong>1,705,415</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>794</td>
<td>493</td>
<td><strong>438</strong></td>
<td>408</td>
<td><strong>353</strong></td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>650</td>
<td>90</td>
<td>140</td>
<td>88</td>
<td>145</td>
</tr>
<tr>
<td><strong>pmd</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>13,134,083</td>
<td>2,868,130</td>
<td><strong>2,598,100</strong></td>
<td>2,457,457</td>
<td><strong>2,328,304</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>1216</td>
<td>845</td>
<td><strong>787</strong></td>
<td>756</td>
<td><strong>698</strong></td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>816</td>
<td>131</td>
<td>191</td>
<td>132</td>
<td>193</td>
</tr>
<tr>
<td><strong>antlr</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>16,445,862</td>
<td>5,082,371</td>
<td><strong>4,768,233</strong></td>
<td>4,586,707</td>
<td><strong>4,419,166</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>995</td>
<td>610</td>
<td><strong>551</strong></td>
<td>525</td>
<td><strong>466</strong></td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>808</td>
<td>109</td>
<td>162</td>
<td>105</td>
<td>163</td>
</tr>
<tr>
<td><strong>lusearch</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>11,788,332</td>
<td>2,251,064</td>
<td><strong>2,010,780</strong></td>
<td>1,886,967</td>
<td><strong>1,771,280</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>874</td>
<td>504</td>
<td><strong>450</strong></td>
<td>412</td>
<td><strong>358</strong></td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>668</td>
<td>94</td>
<td>153</td>
<td>91</td>
<td>155</td>
</tr>
<tr>
<td><strong>bloat</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>may-alias pairs</td>
<td>43,408,294</td>
<td>12,532,334</td>
<td><strong>11,608,822</strong></td>
<td>12,155,175</td>
<td><strong>11,374,583</strong></td>
</tr>
<tr>
<td>may-fail casts</td>
<td>1944</td>
<td>1401</td>
<td><strong>1311</strong></td>
<td>1316</td>
<td><strong>1226</strong></td>
</tr>
<tr>
<td>analysis time (secs)</td>
<td>10679</td>
<td>4508</td>
<td>4770</td>
<td>4460</td>
<td>4724</td>
</tr>
</tbody>
</table>
average of 10.0\%. In addition, \textit{B-S-2-obj} is also more precise than \textit{S-2-obj} for all the nine benchmarks, ranging from 3.7\% for \texttt{antlr} to 30.0\% for \texttt{xalan}, with an average of 8.8\%. Note that the set of non-aliased variable pairs reported under \textit{2-obj} (\textit{S-2-obj}) is a strict subset of the set of non-aliased variable pairs reported under \textit{B-2-obj} (\textit{B-S-2-obj}), validating practically the validity of Theorem 2, i.e., the fact that \textbf{BEAN} is always no less precise than the object-sensitive analysis improved upon. Finally, \textit{2-obj}, \textit{S-2-obj}, \textit{B-2-obj} and \textit{B-S-2-obj} are all substantially more precise than \textit{2-cs}, indicating the superiority of object-sensitivity over call-site-sensitivity.

Let us now move to \textit{may-fail-cast}. Again, \textit{B-2-obj} improves the precision of \textit{2-obj} for all the nine benchmarks, ranging from 5.4\% for \texttt{fop} to 11.2\% for \texttt{luindex}, with an average of 8.4\%. In addition, \textit{B-S-2-obj} is also more precise than \textit{S-2-obj} for all the nine benchmarks, ranging from 6.7\% for \texttt{fop} to 15.6\% for \texttt{luindex}, with an average of 10.8\%. Note that the casts that are shown to be safe under \textit{2-obj} (\textit{S-2-obj}) are also shown to be safe by \textit{B-2-obj} (\textit{B-S-2-obj}), verifying Theorem 2 again. For this second client, \textit{2-obj}, \textit{S-2-obj}, \textit{B-2-obj} and \textit{B-S-2-obj} are also substantially more precise than \textit{2-cs}.

\textbf{Performance} \textbf{BEAN} improves the precision of an object-sensitive analysis at some small increase in cost, as shown in Table 3.1. As can be seen in Figures 3.1 and 3.2, \textbf{BEAN} may spend more time on processing more contexts introduced. \textit{B-2-obj} increases the analysis cost of \textit{2-obj} for all the nine benchmarks, ranging from 5.8\% for \texttt{bloat} to 109.8\% for \texttt{chart}, with an average of 54.8\%. In addition, \textit{B-S-2-obj} also increases the analysis cost of \textit{S-2-obj} for all the nine benchmarks, ranging from 5.9\% for \texttt{bloat} to 70.3\% for \texttt{lusearch}, with an average of 49.1\%.

Table 3.2 shows the pre-analysis times of \textbf{BEAN} for the nine benchmarks. The pre-analysis is fast, finishing within 2 minutes for the most of the benchmarks and in under 6 minutes in the worst case. In Table 3.1, the analysis times for \textit{B-2-obj}
Table 3.2: Pre-analysis times of BEAN (secs). For a program, its pre-analysis time comes from three components: (1) a context-insensitive points-to analysis (“CI”), (2) OAG construction per Figure 3.6 (OAG), and (3) object-sensitive context computation per Figure 3.8 (“CTX-COMP”).

and B-S-2-obj do not include their corresponding pre-analysis times. There are three reasons: (1) the points-to information produced by “CI” in Table 3.2 (for some other purposes) can be reused, (2) and the combined overhead for “OAG” and “CTX-COMP” is small, and (3) the same pre-analysis is often used to guide BEAN to refine many object-sensitive analyses (e.g., 2-obj and S-2-obj).

2-obj and S-2-obj are the top two most precise yet scalable object-sensitive analyses ever designed for Java programs [34]. BEAN is significant as it improves their precision further at only small increases in analysis cost.

3.6.3 RQ2: A Real-World Case Study

Let us use java.util.HashSet, a commonly used container from the Java library to illustrate how B-2-obj improves the precision of 2-obj by enabling may-alias and may-fail-cast to answer their queries more precisely. In Figure 3.11, the code in main() provides an abstraction of a real-world usage scenario for HashSet, with some code in HashSet and its related classes being extracted directly from JDK 1.6.0_45. In main(), X and Y do not have any subtype relation.

We consider two queries: (Q1) are v1 and v2 at lines 5 and 11 aliases (from may-alias)? and (Q2) may the casts at lines 6 and 12 fail (from may-fail-cast)?

Let us examine main(). In lines 2 – 6, we create a HashSet object, HS/1, insert
1 void main(String[] args) {
2     HashSet xSet = new HashSet(); // HS/1
3     xSet.add(new X()); // X/1
4     Iterator xIter = xSet.iterator();
5     Object v1 = xIter.next();
6     X x = (X) v1;
7     HashSet ySet = new HashSet(); // HS/2
8     ySet.add(new Y()); // Y/1
9     Iterator yIter = ySet.iterator();
10    Object v2 = yIter.next();
11    Y y = (Y) v2;
12 }
13 }
14 class HashSet ... {
15     HashMap map = new HashMap(); // HM/1
16     public boolean add(Object e) {
17         return map.put(e, ...) == null;
18     }
19     public Iterator iterator() {
20         return map.newKeyIterator();
21     }
22 }
23 }
24 class HashMap ... {
25     Entry[] table = new Entry[16]; // Entry[]/1
26     public Object put(Object key, ...) { ...}
27     table[bucketIndex] = new Entry(key, ...); // Entry/1
28     ... 
29 }
30 static class Entry {
31     final Object key;
32     Entry(Object k, ...) {
33         key = k;
34     }
35 }
36 private final class KeyIterator ... {
37     public Object next() { ...}
38     Entry e = table[index];
39     return e.key;
40 }
41 }
42 Iterator newKeyIterator() {
43     return new KeyIterator(); // KeyIter/1
44 }
45 ...
46 }

Figure 3.11: A real-world application for using java.util.HashSet.
Figure 3.12: Part of OAG related to HS/1 and HS/2.

An X object into it, retrieve the object from HS/1 through its iterator into v1, and finally, copy v1 to x via a type cast operation (X). In lines 8 – 12, we proceed as in lines 2 – 6 except that another HashSet object, HS/2, is created, and the object inserted into HS/2 is a Y object and thus cast back to Y.

Let us examine HashSet, which is implemented in terms of HashMap. Each HashSet object holds a backing HashMap object, with the elements in a HashSet being used as the keys in its backing HashMap object. In HashMap, each key and its value are stored in an Entry object pointed by its field table.

In main(), the elements in a HashSet object are accessed via its iterator, which is an instance of KeyIterator, an inner class of HashMap.

As before, we have labeled all the allocation sites in their end-of-line comments. Figure 3.12 gives the part of the OAG related to the two HashSet objects, HS/1 and HS/2, which are known to own their distinct HM/1, Entry/1, Entry[]/1 and KeyIter/1 objects during program execution.
To answer queries Q1 and Q2, we need to know the points-to sets of $v_1$ and $v_2$ found at lines 5 and 11, respectively. As revealed in Figure 3.12, 2-obj is able to distinguish the HashMap objects in HS/1 and HS/2 by using two different heap contexts, [HS/1] and [HS/2], respectively. However, the two iterator objects associated with HS/1 and HS/2 are still modeled under one context [HM/1] as one abstract object KeyIter/1, which is pointed to by $xIter$ at line 5 and $yIter$ at line 11. By pointing to $X/1$ and $Y/1$ at the same time, $v_1$ and $v_2$ are reported as aliases and the casts at lines 6 and 12 are also warned to be unsafe.

B-2-obj. By examining the part of the OAG given in Figure 3.12, B-2-obj recognizes that HM/1 is redundant in the single heap context [HM/1] used by 2-obj for representing Entry/1, Entry[]/1 and KeyIter/1. Thus, it will create two distinct sets of these three objects, one under [HS/1] and one under [HS/2], causing $v_1$ ($v_2$) to point to $X/1$ ($Y/1$) only. For query Q1, $v_1$ and $v_2$ are no longer aliases. For query Q2, the casts at lines 6 and 12 are declared to be safe.

3.7 Related Work

Object-sensitivity, introduced by Milanova et al. [53, 55], has now been widely used as an excellent context abstraction for points-to analysis in object-oriented languages [34, 40, 75]. By distinguishing the calling contexts of a method call in terms of its receiver object’s $k$-most-recent allocation sites (rather than $k$-most-recent call sites) leading to the method call, object-sensitivity enables object-oriented features and idioms to be better exploited. This design philosophy enables a $k$-object-sensitive analysis to yield usually significantly higher precision at usually much less cost than a $k$-CFA analysis [39, 23, 34]. The results from our evaluation have also validated this argument further. In Table 3.1, 2-obj is significantly more
precise than 2-cs in all the configurations considered and also significantly faster than 2-cs for all the benchmarks except xalan.

There once existed some confusion in the literature regarding which allocation sites should be used for context elements in a k-object-sensitive analysis [24, 25, 36, 39, 55, 79]. This has recently been clarified by Smaragdakis et al. [75], in which the authors demonstrate that the original statement of object-sensitivity given by Milanova et al. [55], i.e., full-object-sensitivity in [75], represents a right approach in designing a k-object-sensitive analysis while the other approaches (e.g., [36]) may result in substantial loss of precision. In this chapter, we have formalized and evaluated BEAN based on this original design [55, 75].

For Java programs, hybrid object-sensitivity [34] enables k-CFA (call-site-sensitivity) to be applied to static call sites and object/type-sensitivity to virtual call sites. The resulting hybrid analysis is often more precise than their corresponding non-hybrid analyses at sometimes less and sometimes more analysis cost (depending on the program). As a general approach, BEAN can also improve the precision of such a hybrid points-to analysis, as demonstrated in our evaluation.

Type-sensitivity [75], which is directly analogous to object-sensitivity, provides a new sweet spot in the precision-efficiency trade-off for analyzing Java programs. This context abstraction approximates the allocation sites in a context by the class types containing the allocation sites, making itself more scalable but less precise than object-sensitivity [34, 75]. In practice, type-sensitivity usually yields an acceptable precision efficiently [42, 43]. How to generalize BEAN to refine type-sensitive analysis is considered as future work.

Oh et al. [63] introduce a selective context-sensitive program analysis for C. The basic idea is to leverage a pre-impact analysis to guide a subsequent main analysis in applying context-sensitivity to where the precision is likely to be improved with
Chapter 3. **Bean**: Precise Points-to Analysis via Object Allocation Graph

respect to a given query. Although both the analysis and **Bean** select contexts to improve precision, there are two fundamental differences. First, our goals are different. Their approach is query-directed, i.e., the pre-impact analysis is used to estimate when and where applying context-sensitivity could help resolve a given query. However, **Bean** is designed to improve the precision of a whole-program points-to analysis [14, 36, 75, 34, 76], so that all the clients and queries depending on it may directly benefit from the improved points-to information obtained. Second, our methodologies are different. Their pre-analysis selects contexts via a full-context-sensitive analysis with simple query-related domain, while ours selects contexts by traversing a program-related graph (OAG).
Chapter 4

Mahjong: Efficient Points-to Analysis by Merging Equivalent Automata

4.1 Overview

Every points-to analysis, especially for object-oriented languages such as Java, requires a heap abstraction [32, 46] for partitioning the infinitely-sized heap into a finite number of (abstract) objects as discussed in Section 2.1. However, little progress has been made on developing heap abstractions for points-to analysis. Mainstream points-to analysis frameworks for Java, such as DOOP [22], SOOT [77], CHORD [16], and WALA [98], rely predominantly on the allocation-site abstraction to model heap objects. In this case, distinct allocation sites are represented by distinct (abstract) objects, with one object per site, which can be further separated context-sensitively in an orthogonal manner.

As programming languages become more heap-intensive, the need for effective
heap abstractions is greater \([72, 80, 32]\). The suitability of the allocation-site abstraction as an universal solution for all clients of points-to analysis needs to be revisited. While maximizing the precision for \textit{may-alias}, this abstraction often over-partitions the heap without improving the precision much for an important class of type-dependent clients such as \textit{call graph construction}, \textit{devirtualization} and \textit{may-fail casting}, causing often the underlying points-to analysis to be unscalable for large programs. For this reason, DOOP \cite{22} and WALA \cite{98}, provide an option for all objects of a certain class, such as \texttt{java.lang.String} or \texttt{java.lang.StringBuffer}, to be merged ad hocly.

In this chapter, we present MAHJONG, a novel heap abstraction that is specifically developed to address the needs of type-dependent clients. Given a program, we first create a lightweight alternative of the allocation-site abstraction by performing a fast but imprecise allocation-site-based points-to analysis as a pre-analysis and then use it to drive a subsequent points-to analysis. Based on the points-to information found during the pre-analysis, MAHJONG merges two objects if both are \textit{type-consistent}, i.e., if the objects reached from both along the same sequence of field accesses have a common type. We formulate the problem of checking the type-consistency of two objects as one of testing the equivalence of two sequential automata in almost linear time, by applying a classic Hopcroft-Karp algorithm \cite{30} with minor modifications. MAHJONG is simple conceptually and can be easily added on any allocation-site-based points-to analysis.

Compared to the allocation-site abstraction, MAHJONG allows a points-to analysis to run significantly faster while achieving nearly the same precision for type-dependent clients. Thus, MAHJONG makes it possible to accelerate a given points-to analysis or replace it with a more precise but usually more costly points-to analysis that is either inefficient or unscalable if the allocation-site abstraction is
used. MAHJONG is expected to provide significant benefits to many program analyses, such as bug detection, security analysis, program verification and program understanding, where call graphs are required [58, 59, 106, 10, 43, 6, 25, 81].

We demonstrate the effectiveness of MAHJONG by discussing some insights on why it is a better alternative of the allocation-site abstraction for type-dependent clients and conducting an evaluation extensively on 12 large Java programs with five widely used context-sensitive points-to analyses and three significant type-dependent clients, call graph construction, devirtualization and may-fail casting [76, 34, 75, 39, 79]. Take, 3-obj, a 3-object-sensitive points-to analysis [53], the most precise one used in our evaluation, as an example. For the four programs that can be analyzed scalably under 3-obj, our MAHJONG-based 3-obj runs 131X faster, on average, while achieving nearly the same precision for all the three clients. For the remaining eight, where 3-obj is unscalable in 5 hours each, our MAHJONG-based 3-obj can analyze five of them in an average of 33.42 minutes.

In summary, this chapter makes the following contributions:

- We present MAHJONG, a new heap abstraction that can significantly scale an allocation-site-based points-to analysis for object-oriented programs while achieving nearly the same precision for type-dependent clients.

- We formulate the problem of checking the type-consistency of two objects as one of testing the equivalence of two automata, which is solvable in almost linear time.

- We implement MAHJONG as a standalone open-source tool. MAHJONG is simple (with only 1500 LOC of Java in total) and can be easily added on any allocation-site-based points-to analysis.

- We conduct extensive experiments to evaluate MAHJONG by applying it to
five commonly used context-sensitive points-to analyses on 12 large Java programs. Our evaluation demonstrates the effectiveness of MAHJONG.

The rest of this chapter is organized as follows. We first describe the motivation of MAHJONG in Section 4.2. Then, in Section 4.3, we present the methodology of MAHJONG. In Section 4.4, we give the algorithms of MAHJONG. The implementation of MAHJONG is introduced in Section 4.5. In Section 4.6, we evaluate the effectiveness of MAHJONG in practice. In Section 4.7, we discuss the research work related to MAHJONG.

4.2 Motivation

For points-to analysis, type-dependent clients, such as call graph construction, de-virtualization and may-fail casting, share similar needs: their precision depends on the types of pointed-to objects rather than the pointed-to objects themselves. For such clients, the conventional allocation-site abstraction is often too fine-grained, contributing little to improving their precision but rendering the underlying points-to analysis unduly inefficient or eventually unscalable. In this chapter, we aim to improve this by looking for a lightweight alternative that satisfies the needs of type-dependent clients, but not necessarily others such as may-alias. To this end, we would like to avoid distinguishing two objects if merging them loses no or little precision for type-dependent clients.

In Section 4.2.1, we see that blindly merging objects of the same type is ineffective. In Section 4.2.2, we describe our solution that merges objects representing equivalent automata only. For object-oriented programs, merging objects amounts to merging their corresponding allocation sites.
void main(String[] args) {
  A x = new A(); // o_A^1
  A y = new A(); // o_A^2
  A z = new A(); // o_A^3
  x.f = new B(); // o_B^4
  y.f = new C(); // o_C^5
  z.f = new C(); // o_C^6
  A a = z.f;
  a.foo();
  C c = (C) a;
}
class A {
  A f;
  void foo() {...}
}
class B extends A {
  void foo() {...}
}
class C extends A {
  void foo() {...}
}

Figure 4.1: An example program illustrating object merging.

4.2.1 Allocation-Type Abstraction: A Naive Solution

In this so-called allocation-type abstraction, all objects with the same type are merged, with one object per type. As previously noted, this naive solution often gains efficiency but may incur a significant loss of precision [32, 45, 99, 72].

Example 3 Consider Figure 4.1, where o_t^i represents the abstract object of type t created at the allocation site with label i.

For the three type-dependent clients, call graph construction, devirtualization and may-fail casting, only lines 9 – 10 are relevant. According to an allocation-site-based Andersen’s points-to analysis [5], x, y and z point to o_A^1, o_A^2 and o_A^3, respectively. As x.f, y.f and z.f are not aliases, a points to o_C^6. Thus, a.foo() at line 9 has only one callee, C::foo(), and can thus be devirtualized as a monomorphic call, and in addition, the cast (C) at line 10 is safe.
However, if \( o_1^A, o_2^A \) and \( o_3^A \) are merged, then \( x.f, y.f \) and \( z.f \) will be aliases, causing \( a \) to also point to \( o_4^B \). As a result, \( a.foo() \) becomes a polymorphic call as it has two callee methods \( (B::foo() \) and \( C::foo() \)), and thus cannot be devirtualized. In addition, the cast \((C)\) is no longer considered safe.

Consider \texttt{pmd}, a program analyzed by (1) \texttt{3-obj} — a 3-object-sensitive points-to analysis [53] using the allocation-site abstraction, (2) \texttt{T-3-obj} — \texttt{3-obj} using the allocation-type abstraction, and (3) \texttt{M-3-obj} — \texttt{3-obj} using our MAHJONG heap abstraction. For \texttt{3-obj}, \texttt{pmd} is analyzed in 14469.3 seconds, allowing 44004 call graph edges to be discovered. \texttt{T-3-obj} is the fastest (50.3 seconds), but is the most imprecise (50666 call graph edges). In contrast, \texttt{M-3-obj} is as precise as \texttt{3-obj} (44016 call graph edges) but is also nearly as fast as \texttt{T-3-obj} (127.7 seconds).

4.2.2 MAHJONG: Our Solution

To address the needs of type-dependent clients, MAHJONG is designed to maximally preserve the precision of the allocation-site abstraction while reaping the efficiency of the allocation-type abstraction as much as possible. For a given program, we first build a heap abstraction by performing a pre-analysis, i.e., a fast but imprecise allocation-site-based Andersen’s points-to analysis [5] and then use it to guide a subsequent points-to analysis. Based on the pre-analysis, we define type-consistent objects that can be merged (Section 4.2.2.1) and formulate the problem of checking the type-consistency of two objects as one of testing the equivalence of two automata in almost linear time (Section 4.2.2.2).

4.2.2.1 Defining Type-Consistent Objects

After the pre-analysis, the field points-to graph (FPG) is available, representing the points-to information for the object fields. To facilitate a subsequent reduction
of the problem of checking type-consistency as one of testing the equivalence of automata, we introduce the field points-to graph rooted at an object $o$ as $G_o = (\mathcal{H}, \mathcal{F}, \alpha, o, \mathcal{T}, \tau)$. $\mathcal{H}$ is the set of objects reachable from $o$. $\mathcal{F}$ is the set of field names traversed along the way. The points-to relations for the object fields are defined by a field points-to map $\alpha : \mathcal{H} \times \mathcal{F} \mapsto \mathcal{P}(\mathcal{H})$. $\mathcal{T}$ is the set of types of the objects in $\mathcal{H}$. The object-to-type map $\tau : \mathcal{H} \mapsto \mathcal{T}$ reveals the type of an object.

Figure 4.2 gives the field points-to graphs rooted at $o_1^T$ and $o_2^T$, by using the same notation for objects in Figure 4.1.

**Example 4** Consider $o_2^T$ first in Figure 4.2. $G_{o_2^T} = (\mathcal{H}, \mathcal{F}, \alpha, o_2^T, \mathcal{T}, \tau)$. $\mathcal{H} = \{o_2^T, o_4^T, o_6^X, o_8^Y\}$; $\mathcal{F} = \{f, g, h, k\}$; $\alpha[o_2^T, f] = \{o_4^T\}$, $\alpha[o_4^T, h] = \{o_8^Y\}$, $\alpha[o_2^T, g] = \{o_6^X\}$, and $\alpha[o_6^X, k] = \{o_8^Y\}$; $\mathcal{T} = \{T, U, X, Y\}$; and $\tau[o_2^T] = T$, $\tau[o_4^T] = U$, $\tau[o_6^X] = X$, and $\tau[o_8^Y] = Y$. Similarly, $G_{o_1^T}$ can be constructed. □

Unlike the allocation-type abstraction, where all the objects with the same type are merged blindly, we will merge so-called type-consistent objects, thereby avoiding the imprecision introduced by the allocation-type abstraction.

Let $\bar{f} = f_1, f_2, \ldots, f_n$ be a sequence of field names. For the field points-to graph $G_o$ rooted at an object $o$, we write $\text{pts}(o, \bar{f})$ to represent the set of objects that can be reached from $o$ along any path of points-to edges labeled by $f_1$, $f_2$, $\ldots$, $f_n$ in $G_o$ in that order. In Figure 4.2, $\text{pts}(o_1^T, f) = \{o_3^U\}$ and $\text{pts}(o_1^T, f, h) = \{o_7^Y, o_9^Y\}$. 
Two objects with the same type are type-consistent if traversing from the two objects along the same sequence of field names always lead to objects of the same single type.

**Definition 1 (Type-Consistent Objects)** Two objects, \( o_i \) and \( o_j \), with the same type are said to be type-consistent, denoted \( o_i \equiv o_j \), if for every non-empty sequence of field names, \( \vec{f} = f_1.f_2.\cdots.f_n \), the following two conditions hold:

1. \( \{ \tau[o] \mid o \in \text{pts}(o_i, \vec{f}) \} = \{ \tau[o] \mid o \in \text{pts}(o_j, \vec{f}) \} \), and

2. \( \left| \{ \tau[o] \mid o \in \text{pts}(o_i, \vec{f}) \} \right| = 1. \)

In Figure 4.2, \( o^T_1 \) and \( o^T_2 \) are type-consistent. For the objects reached from \( o^T_1 \) and \( o^T_2 \), along \( f, f.h, g \) and \( g.k \), their sets of types are \( \{U\} \), \( \{Y\} \), \( \{X\} \) and \( \{Y\} \), respectively.

We illustrate the intuition behind the notion of type-consistency with an example discussed below.

**Example 5** Let us return to Figure 4.1, for which the allocation-type abstraction will merge \( o^A_1 \), \( o^A_2 \) and \( o^A_3 \) (Section 4.2.1). By Definition 1, \( o^A_2 \) and \( o^A_3 \) are type-consistent (as \( o^A_2.f \) points to \( o^C_5 \) and \( o^A_3.f \) points to \( o^C_6 \)) but \( o^A_1 \) is not type-consistent with any (as \( o^A_1.f \) points to \( o^B_4 \)). After \( o^A_2 \) and \( o^A_3 \) are merged, \( y.f \) and \( z.f \) are regarded as aliases. Thus, \( a \) will point to not only \( o^C_6 \) as before but also \( o^C_5 \) spuriously. However, as \( o^C_5 \) and \( o^C_6 \) have the same type \( C \), the precision of call graph construction and devirtualization at line 9 as well as may-fail casting at line 10 is not affected. \( \square \)

Let us examine Definition 1. Condition 1 is self-explanatory in order to maximally preserve precision for type-dependent clients. What is the rationale behind Condition 2? The pre-analysis is fast but imprecise. Its enforcement maximally avoids precision loss, as illustrated below.
Chapter 4. Mahjong: Efficient Points-to Analysis by Merging Equivalent Automata

Pre-Analysis

A More Precise Points-to Analysis

Example 6 Suppose \( o^T_i.f \) and \( o^T_j.f \) point to both \( o_1^X \) and \( o_2^Y \) during the pre-analysis (Figure 4.3(a)) but \( o_1^X \) and \( o_2^Y \), respectively, in a more precise allocation-site-based points-to analysis, \( A \) (Figure 4.3(b)). If Condition 2 is ignored, \( o^T_i \) and \( o^T_j \) will become type-consistent according to the pre-analysis and thus merged into, say, \( o^T_k \) (represented by \( o^T_i \) or \( o^T_j \)). Running \( A \) with this new heap abstraction will result in precision loss, as \( o^T_i.f \) and \( o^T_j.f \) now point to objects of both types \( X \) and \( Y \) (Figure 4.3(c)).

In Definition 1, the type-consistency relation \( \equiv \) is an equivalence relation. It is straightforward to verify that \( \equiv \) is reflexive, symmetric and transitive.

Let \( H \) be the set of all abstract objects in the program as in formulation for points-to analysis in Section 2.2.

Definition 2 (Mahjong’s Heap Abstraction) Given the quotient set, \( H / \equiv \), Mahjong will merge all the objects in the same equivalence class into one object.
Therefore, the key insight behind our new heap abstraction is not to distinguish two (container) objects of the same type if both containers store the objects of the same type at all their corresponding nested sub-containers.

How do we check the type-consistency of two objects efficiently, especially for large programs with a large number of heap objects, field names and class types? Enumerating all the possible field access paths $\bar{f}$ as required in Definition 1, especially in the presence of cycles, may be exponential in terms of the number of edges traversed [65, 51], causing the pre-analysis to be too inefficient or even unscalable. We describe a fast and elegant solution below.

### 4.2.2.2 Merging Equivalent Automata

We transform the problem of checking the type-consistency of two objects into one of testing the equivalence of two automata. Figure 4.4 relates the field points-to graph rooted at an object $o$, $G_o = (\mathcal{H}, \mathcal{F}, \alpha, o, \mathcal{T}, \tau)$, to a 6-tuple sequential automaton $A_o = (Q, \Sigma, \delta, q_o, \Gamma, \gamma)$ [1], which is more general than a traditional (5-tuple) automaton. In fact, a 5-tuple automaton can be turned into a 6-tuple automaton, if its accepting (acc) and non-accepting (non-acc) states are distinguished by $\gamma : Q \rightarrow \Gamma$, where $\Gamma = \{\text{acc, non-acc}\}$.

**Example 7** Continuing from Example 4 (Figure 4.2), the automaton $A_o^T$ for $G_o^T = (\mathcal{H}, \mathcal{F}, \alpha, o^T, \mathcal{T}, \tau)$ is obtained according to Figure 4.4. Similarly, $A_o^T$ is constructed.

The behavior of $A_o$, which can be an NFA (consisting of multiple edges with the same label leaving a state), is:

$$\beta_{A_o} : \Sigma^* \rightarrow \mathcal{P}(\Gamma)$$

If $A_o$ reaches the states, $s_1, s_2, \cdots, s_n$, after having read an input $w$ in $\Sigma^*$, then:
Figure 4.4: The mapping of a field points-to graph rooted at an object to a sequential automaton.
Chapter 4. MAHJONG: Efficient Points-to Analysis by Merging Equivalent Automata

\[ \beta_{A_o}(w) = \bigcup_{i=1}^{n} \gamma[s_i] \]

Let \( o_{T_1} \) and \( o_{T_2} \) be two objects with the same type \( T \). Let their automata \( A_{o_{T_1}} \) and \( A_{o_{T_2}} \) be built as shown in Figure 4.4. \( o_{T_1} \) and \( o_{T_2} \) are type-consistent if, for every input \( w \) in \( \Sigma^* \), (1) \( \beta_{A_{o_{T_1}}}(w) = \beta_{A_{o_{T_2}}}(w) \) (Condition 1 of Definition 1) and (2) \( |\beta_{A_{o_{T_1}}}(w)| = 1 \) (Condition 2 of Definition 1).

Therefore, we have reduced the problem of checking the type-consistency of \( o_{T_1} \) and \( o_{T_2} \) to one of testing the equivalence of their corresponding automata \( A_{o_{T_1}} \) and \( A_{o_{T_2}} \), which is solvable in almost linear time by a classic Hopcroft-Karp algorithm \[30\] with minor modifications. The worst-case time complexity is \( O(|\Sigma| \times |Q_{\text{larger}}|) \), where \( Q_{\text{larger}} \) is the set of states of the larger automaton.

Example 8 Continuing from Example 7, we see easily that \( o_{T_1} \) and \( o_{T_2} \) are type-consistent (Figure 4.2) since their corresponding automata \( A_{o_{T_1}} \) and \( A_{o_{T_2}} \) are equivalent.

\[ \square \]

4.3 MAHJONG

We first give an overview of MAHJONG that consists of four components (Section 4.3.1). We then describe each component in detail (Sections 4.3.2 – 4.3.5). Finally, we discuss MAHJONG-based points-to analysis (Section 4.3.6).

4.3.1 Overview of MAHJONG

As shown in Figure 4.5, MAHJONG takes the field points-to graph (FPG) computed by a pre-analysis (Section 4.2.2.1) as input and builds a heap abstraction (Definition 2) to be used by a subsequent points-to analysis. The pre-analysis is fast but imprecise, by using Andersen’s algorithm \[5\] with the allocation-site abstraction,
context-insensitively. The subsequent points-to analysis will be more precise, usually performed context-sensitively, especially for object-oriented programs, based on the Mahjong heap abstraction.

Mahjong iteratively picks a pair of objects $o_i^T$ and $o_j^T$ with the same type $T$ and merges them if they are type-consistent, until no such pair can be found. Given $o_i^T$ and $o_j^T$, their corresponding NFAs, $NFA_{o_i^T}$ and $NFA_{o_j^T}$, are first built by using the NFA Builder. Then the two NFAs are converted into their equivalent DFAs, $DFA_{o_i^T}$ and $DFA_{o_j^T}$, by using the DFA Converter. Next, the Automata Equivalence Checker determines whether $DFA_{o_i^T}$ and $DFA_{o_j^T}$ are equivalent or not. Finally, the Heap Modeler outputs a new heap abstraction.

4.3.2 The NFA Builder

The NFA builder takes an object $o$, with the field points-to graph $G_o$ rooted at $o$, and constructs a 6-tuple NFA $A_o = (Q, \Sigma, \delta, q_0, \Gamma, \gamma)$ according to the mapping, as shown in Figure 4.4. In fact, $A_o$ can be immediately read off from $G_o$.

4.3.3 The DFA Converter

To ease the test of automata equivalence, we convert NFA to DFA. The DFA Converter converts an NFA to an equivalent DFA based on the subset construc-
tion algorithm [2] with minor modifications. The resulting DFA is still a 6-tuple sequential automaton except that it is deterministic.

4.3.4 The Automata Equivalence Checker


4.3.5 The Heap Modeler

After all type-consistent objects have been found, the type-consistency equivalence relation $\equiv$ given in Definition 1 becomes fully constructed. By Definition 2, the new heap abstraction found is simply given by $H / \equiv$. For every equivalent class $[o^T_i] \in H / \equiv$, a representative object $o^T_j$ is arbitrarily picked to substitute for the other objects in the class. Essentially, the allocation sites for all objects in $[o^T_i]$ are merged and represented by the allocation site of $o^T_j$ only.

To enable a points-to analysis to use our new heap abstraction, we only need to change its rule for handling allocation sites. Given $i : x = \text{new } T()$ in a Java program, where $o^T_j$ is a representative for $[o^T_i]$, $x$ is made to point to $o^T_j$.

4.3.6 Mahjong-based Points-to Analysis

Let $\mathcal{A}$ be an allocation-site-based points-to analysis introduced in Section 2.2, which is either call-site-sensitive [99, 39, 79, 23], object-sensitive [53, 95, 76] or type-sensitive [75]. We first discuss how to obtain $M-\mathcal{A}$, a Mahjong-based points-to analysis, from $\mathcal{A}$ (Section 4.3.6.1). We then discuss briefly the soundness and precision of $M-\mathcal{A}$ relative to $\mathcal{A}$ for type-dependent clients.
4.3.6.1 Obtaining \( M-A \) from \( A \)

In a context-sensitive points-to analysis, local variables are analyzed context-sensitively by distinguishing the calling contexts for a method. Heap objects are modeled context-sensitively by distinguishing the calling contexts for allocation sites. Different context-sensitivity are distinguished by different kinds of context elements used, as discussed below.

We obtain \( M-A \) from \( A \) by first replacing \( A \)'s allocation-site abstraction with the MAHJONG heap abstraction. We then need to make minor modifications to \( A \) to enable \( M-A \) to handle merged objects effectively.

Regardless of whether \( A \) is call-site-, object- or type-sensitive, \( M-A \) will always model a merged object \( o \) context-insensitively. There would be of little benefit in modeling \( o \) context-sensitively, since the objects accessed by \( o.f_1.f_2.\ldots.f_n \) for any \( f_1.f_2.\ldots.f_n \) under different contexts are expected to have the same type, in practice. Below we briefly review the three kinds of context-sensitivity presented in Section 2.2.3 and discuss how the calling contexts for methods are modified, if needed, when they are related to merged objects.

**Call-Site-Sensitivity** A \( k \)-call-site-sensitive points-to analysis, i.e., a \( k \)-CFA \([71]\) separates information on local variables per call-stack (i.e., sequence of \( k \) call-sites) of method invocations that lead to the current method. By convention, a sequence of \( k-1 \) call-sites is used as a calling context for an allocation site \([75, 34, 95]\).

If \( A \) is \( k \)-call-site-sensitive \([71]\), then \( M-A \) behaves identically as \( A \) in handling methods. For the reason mentioned above, \( M-A \) models the merged objects context-insensitively but everything else context-sensitively as in \( A \).

**Object-Sensitivity** \( k \)-object-sensitivity is similar to \( k \)-call-site-sensitivity except that allocation sites rather than call sites are used as context elements \([53]\). Let
Chapter 4. MAHJONG: Efficient Points-to Analysis by Merging Equivalent Automata

$o_i$ be an abstract object identified by its allocation site $i$. In $k$-object-sensitivity, the object $o_i$ at allocation site $i$ is modeled context-sensitively by a calling context $[o_{i_{k-1}}, \ldots, o_{i_1}]$ (of length $k-1$), where $i_j$ is the allocation site for the receiver object $o_{i_j}$ of the method that contains allocation site $i_{j-1}$ (with $i_0 = i$). If $x$ points to an object $o_i$ modeled under a context $[o_{i_{k-1}}, \ldots, o_{i_1}]$, then the $k$-object-sensitive calling context used for analyzing the callee of $x.foo()$ is $[o_{i_{k-1}}, \ldots, o_{i_1}, o_i]$.

If $\mathcal{A}$ is a $k$-object-sensitive points-to analysis, $M\cdot\mathcal{A}$ models merged objects context-insensitively, i.e., object-insensitively but everything else objective-sensitively as in $\mathcal{A}$. As a result, calling contexts that contain merged objects as context elements are modified accordingly. For an object $o$ that is used in a calling context under $\mathcal{A}$, $o$ is replaced by a representative of $[o] \in \mathbb{H} / \equiv$ (Section 4.3.5) under $M\cdot\mathcal{A}$. In other words, if $o$ is merged with some type-consistent objects, then its representative is used, instead.

**Type-Sensitivity** To trade precision for efficiency, $k$-type-sensitivity is derived from $k$-object-sensitivity by replacing every object in a calling context with the class type that contains the corresponding allocation site for the object [75].

If $\mathcal{A}$ is a $k$-type-sensitive analysis obtained from its corresponding $k$-object-sensitive analysis $\mathcal{A}'$, then $M\cdot\mathcal{A}$ is simply obtained from $M\cdot\mathcal{A}'$ in the same type-sensitive manner.

4.3.6.2 Soundness and Precision of $M\cdot\mathcal{A}$ over $\mathcal{A}$

The soundness of $M\cdot\mathcal{A}$ is easy to establish. If $\mathcal{A}$ is sound, then $M\cdot\mathcal{A}$ is also sound as the MAHJONG heap abstraction is coarser than the allocation-site abstraction used in $\mathcal{A}$.

We discuss some insights below on why merging type-consistent objects enables $M\cdot\mathcal{A}$ to maximally preserve the precision of $\mathcal{A}$ for type-dependent clients. This is
true for all three types of context-sensitivity as validated later.

We first describe a rarely occurring subtle case, the null-field problem, illustrated in Figure 4.6, due to the imprecision of the pre-analysis, causing precision loss for all the three types of MAHJONG-based context-sensitivity.

**Example 9** Suppose \( o^T_i \cdot f \) and \( o^T_j \cdot f \) both point to \( o^X_1 \) during the pre-analysis (Figure 4.6(a)) but \( o^X_1 \) and \text{null} respectively, in \( A \) (Figure 4.6(b)). In \( M\cdot A \), \( o^T_i \) and \( o^T_j \) are type-consistent and thus merged into \( o^T_k \) (represented by either \( o^T_i \) or \( o^T_j \)), \( M\cdot A \) is less precise, as \( o^T_j \cdot f \), which points to \text{null} in \( A \), now points to an object of type \( X \) (Figure 4.6(c)).

![Figure 4.6: Illustrating the null-field problem.](image)

If \( A \) is call-site-sensitive, \( M\cdot A \) is equally precise as \( A \) for a type-dependent client if the null-field problem never occurs in a program analyzed by \( A \). Recall that the pre-analysis is no more precise than \( A \). By Definition 1, the objects reached from \( o \) along the same sequence of field accesses must have exactly the same type when \( o \) is modeled both context-sensitively under \( A \) and context-insensitively under \( M\cdot A \), resulting in the same precision in both cases. In general, \( M\cdot A \) is no more precise than \( A \) due to the null-field problem but very close to \( A \) as the null-fields are rare.

If \( A \) is object-sensitive, then \( M\cdot A \) is no more precise than \( A \) for type-dependent clients, as some objects that are used in distinguishing different contexts in \( A \) are merged by MAHJONG if they are type-consistent. However, this hardly hurts the precision, making \( M\cdot A \) nearly as precise as \( A \) for type-dependent clients, in practice.
The key insight behind object-sensitivity [53, 55] is to distinguish the side-effects of different receiver objects of an instance method \( T::\text{foo}(\cdot) \) by analyzing it under multiple calling contexts, one per receiver object. By merging the type-consistent receiver objects for \( T::\text{foo}(\cdot) \), we end up achieving a significant performance benefit at little precision loss by analyzing \( T::\text{foo}(\cdot) \) under the same context by \( M-\mathcal{A} \) rather than separately but unnecessarily by \( \mathcal{A} \) for these receiver objects. For type-dependent clients, this represents a generalization of object-sensitivity.

If \( \mathcal{A} \) is type-sensitive, then \( M-\mathcal{A} \) is nearly as precise as (sometimes slightly better or worse than) \( \mathcal{A} \) for type-dependent clients, in practice. Consider an equivalence class \([o] = \{o_1, \ldots, o_n\} \in \mathbb{H} / \equiv \) (Definition 2) formed by the MAHJONG heap abstraction. In \( \mathcal{A} \), every \( o_i \) that is used as a context element in a calling context is replaced by the class type that contains the allocation site for \( o_i \). In \( M-\mathcal{A} \), \( o_1, \ldots, o_n \) are merged and replaced by the class type that contains the allocation site for a representative in \([o]\). Thus, the MAHJONG heap abstraction can be coarser than the allocation-site abstraction for some methods and finer for some others in partitioning their calling contexts.

### 4.4 Algorithms

We present the algorithms used in MAHJONG. In Section 4.4.1, we give some domains used and then the main algorithm. In Section 4.4.2-4.4.5, we describe the algorithms of its four components, introduced in Sections 4.3.2 – 4.3.5.

#### 4.4.1 MAHJONG

For a program, we use the three domains: (1) \( \mathbb{H} \) is the set of all abstract heap objects (i.e., allocation sites), (2) \( \mathbb{F} \) is the set of all field names, and (3) \( \mathbb{T} \) is the
set of all types. Note that we have introduced these domains in Section 2.2.1 and also have used $\mathbb{H}$ earlier in Definition 2.

Now, we can formally define the input and output of MAHJONG. MAHJONG takes a field points-to graph, $\text{FPG} = (N, E)$, which is a directed weighted graph, as input. A node $o_i \in N = \mathbb{H}$ represents a heap object in the program. An edge $(o_i, f, o_j) \in E \subseteq N \times F \times N$ indicates that $o_i.f$ points to $o_j$. We assume that the FPG contains a dummy node $o_{\text{null}}$ to represent null. If $o_i.f = \text{null}$, then $(o_i, f, o_{\text{null}}) \in E$. We also assume $(o_{\text{null}}, f, o_{\text{null}}) \in E$ for every field $f \in F$.

The output of MAHJONG is a new heap abstraction, represented by a merged object map, $\text{MOM} \subseteq \mathbb{H} \to \mathbb{H}$, which relates an object in an equivalence class in $\mathbb{H} / \equiv$ to its representative object (as described in Section 4.3.5).

Algorithm 1 gives the main algorithm. To facilitate merging type-consistent objects, we make use of the concept of disjoint sets [17]. In a set $S$ of disjoint sets, each disjoint set is identified by a representative, which is some member of the disjoint set. We make use of two classic operations over disjoint sets, \textsc{Union} and \textsc{Find}. $S.\text{Union}(x, y)$ unites two disjoint sets in $S$ that contain $x$ and $y$, say $S_x$ and $S_y$, into a new disjoint set that is the union of the two, adds it to $S$, and destroys $S_x$ and $S_y$ in $S$. The representative of the new set is any member of $S_x \cup S_y$. $S.\text{Find}(x)$ returns the representative of the disjoint set in $S$ that contains $x$.

MAHJONG first initializes $W$ by adding to it a singleton set for each object (lines 1 – 3). Then it iterates over every pair of objects, $o_i$ and $o_j$ in $\mathbb{H}$, that are not yet merged, and merges the pair if both are type-consistent (lines 4 – 13). According to line 5, $o_i$ and $o_j$ are mergeable only if both have the same type. The function $\text{TypeOf} : \mathbb{H} \to T$ returns the type of a given object and a special type for $o_{\text{null}}$.

To check the type consistency of $o_i$ and $o_j$ by Definition 1 efficiently, we handle its two conditions separately, with Condition 2 in lines 6 – 7 and Condition 1 in
Algorithm 1: MAHJONG

Input : FPG (Field Points-to Graph)
Output: MOM (Merged Object Map)

1 Let $W$ be a new set
2 foreach $o \in H$ do
3 Add $\{o\}$ to $W$
4 foreach $o_i, o_j \in H$ s.t. $W.FIND(o_i) \neq W.FIND(o_j)$ do
5 if $\text{TYPEOF}(o_i) == \text{TYPEOF}(o_j)$ and
6 $\text{SINGLETYPE-CHECK}(o_i, \text{FPG})$ and
7 $\text{SINGLETYPE-CHECK}(o_j, \text{FPG})$ then
8 $NFA_{o_i} = \text{NFA-BUILDER}(o_i, \text{FPG})$
9 $NFA_{o_j} = \text{NFA-BUILDER}(o_j, \text{FPG})$
10 $DFA_{o_i} = \text{DFA-CONVERTER}(NFA_{o_i})$
11 $DFA_{o_j} = \text{DFA-CONVERTER}(NFA_{o_j})$
12 if $\text{EQUIV-CHECKER}(DFA_{o_i}, DFA_{o_j})$ then
13 $W.UNION(o_i, o_j)$
14 Let MOM be a new map
15 foreach $o \in H$ do
16 $\text{MOM}[o] = W.FIND(o)$
17 return MOM

lines 8 – 12. In lines 6 – 7, the function $\text{SINGLETYPE-CHECK} : H \times \text{FPG} \rightarrow \{\text{TRUE, FALSE}\}$ is applied to see if Condition 2 holds for both $o_i$ and $o_j$. If so, MAHJONG then proceeds to build the NFAs for the two objects (Section 4.4.2), convert the NFAs to their equivalent DFAs (Section 4.4.3), and finally, test their equivalence (Section 4.4.4). If the two DFAs are equivalent, then MAHJONG calls $W.UNION(o_i, o_j)$ to merge $o_i$ and $o_j$ at line 13. Finally, in lines 14 – 16, MAHJONG builds a new heap abstraction as desired (Section 4.4.5).

4.4.2 The NFA Builder

Given an object $o$, Algorithm 2 (NFA-BUILDER) builds an NFA, $\mathcal{A}_o = (Q, \Sigma, \delta, q_0, \Gamma, \gamma)$, according to the mapping from the field points-to graph rooted at $o$ to $\mathcal{A}_o$ in Fig-
Algorithm 2: NFA-BUILDER

Input: 
- \( o \) (Input object)
- \( \text{FPG} = (N, E) \) (Field Points-to Graph)

Output: \( \text{NFA} = (Q, \Sigma, \delta, q_0, \Gamma, \gamma) \)

1. \( q_0 = o \)
2. Let \( Q \) be a set of objects reachable from \( o \) in \( \text{FPG} \)
3. Let \( \Sigma \) and \( \Gamma \) be two new sets
4. Let \( \gamma \) and \( \delta \) be two new maps
5. foreach \( o_i \in Q \) do
6. \( \Sigma = \Sigma \cup \text{FIELDSOF}(o_i) \)
7. \( \Gamma = \Gamma \cup \{\text{TYPEOF}(o_i)\} \)
8. \( \gamma[o_i] = \text{TYPEOF}(o_i) \)
9. foreach \((o_i, f, o_j) \in E\) do
10. if \( o_i \in Q \) then
11. Add \( o_j \) to \( \delta[o_i, f] \)
12. return \( \text{NFA} = (Q, \Sigma, \delta, q_0, \Gamma, \gamma) \)

NFA-BUILDER constructs all the six components for \( \mathcal{A}_o \). Its initial state \( q_0 \) is simply \( o \) (line 1). \( Q \) is the set of objects reachable from \( o \) in \( \text{FPG} \) (line 2). The objects in \( Q \) are iterated over to build \( \Sigma \) (set of input symbols), \( \Gamma \) (set of output symbols), and \( \gamma \) (output map) at lines 5 – 8. The function \( \text{FIELDSOF} : \mathbb{H} \rightarrow \mathcal{P}(\mathbb{F}) \) returns the fields of a given object. Finally, the relevant edges in \( \text{FPG} \) are traversed to build the state transition map \( \delta \) (lines 9 – 11).

4.4.3 The DFA Converter

Algorithm 3 (DFA-CONVERTER) converts an NFA to its equivalent DFA by using the subset construction [2].

There are three minor differences between Algorithm 3 and the standard subset construction [2]. First, we do not need to handle (non-existent) \( \epsilon \)-transitions. Second, we can find the next states of a DFA state \( q \) more efficiently. In the general
Chapter 4. Mahjong: Efficient Points-to Analysis by Merging Equivalent Automata

Algorithm 3: DFA-CONVERTER

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q_0' = {q_0}$</td>
</tr>
<tr>
<td>2</td>
<td>$\Sigma' = \Sigma$</td>
</tr>
<tr>
<td>3</td>
<td>Let $Q'$ and $\Gamma'$ be two new sets</td>
</tr>
<tr>
<td>4</td>
<td>Let $\delta'$ and $\gamma'$ be two new maps</td>
</tr>
<tr>
<td>5</td>
<td>Add $q_0'$ as an unmarked state to $Q'$</td>
</tr>
<tr>
<td>6</td>
<td>while there is an unmarked state $q \in Q'$ do</td>
</tr>
<tr>
<td></td>
<td>Mark $q$</td>
</tr>
<tr>
<td>7</td>
<td>Pick any $o_i$ from $q$</td>
</tr>
<tr>
<td>8</td>
<td>foreach $f \in \text{FIELDSOF}(o_i)$ do</td>
</tr>
<tr>
<td>9</td>
<td>$q' = {\delta[o_j, f] \mid o_j \in q}$</td>
</tr>
<tr>
<td>10</td>
<td>if $q' \notin Q'$ then</td>
</tr>
<tr>
<td>11</td>
<td>Add $q'$ as an unmarked state to $Q'$</td>
</tr>
<tr>
<td>12</td>
<td>$\delta'[q, f] = q'$</td>
</tr>
<tr>
<td>13</td>
<td>foreach $q \in Q'$ do</td>
</tr>
<tr>
<td>14</td>
<td>$\gamma'[q] = {\text{TYPEOF}(o_i) \mid o_i \in q}$</td>
</tr>
<tr>
<td>15</td>
<td>$\Gamma' = \Gamma' \cup \gamma'[q]$</td>
</tr>
<tr>
<td>16</td>
<td>return DFA = $(Q', \Sigma', \delta', q_0', \Gamma', \gamma')$</td>
</tr>
</tbody>
</table>

case, all input symbols must be examined. In our case (lines 7 – 9), we only need to iterate over the fields (input symbols) of an arbitrarily picked object (an NFA state) in $q$ to find its next states. Due to SINGLETYPE-CHECK in lines 6 – 7 of Algorithm 1, the objects grouped in a DFA state $q$ must have the same type. Finally, we need to compute $\Gamma'$ (set of output symbols) and $\gamma'$ (output map) at lines 14 – 16, which are absent in 5-tuple automata.

4.4.4 The Automata Equivalence Checker

Algorithm 4 (EQUIV-CHECKER) tests the equivalence of two 6-tuple DFAs, by applying a Hopcroft-Karp algorithm that was proposed for two 5-tuple DFAs [30] with minor modifications at line 19 on the condition for testing the equivalence of two DFAs. At line 19, two DFAs are equivalent if for each disjoint set $s \in V$, all
Algorithm 4: EQUIV-CHECKER

Input: \( \text{DFA}_1 = (Q_1, \Sigma_1, \delta_1, q_1, \Gamma_1, \gamma_1) \)
\( \text{DFA}_2 = (Q_2, \Sigma_2, \delta_2, q_2, \Gamma_2, \gamma_2) \)

Output: TRUE or FALSE (Are \( \text{DFA}_1 \) and \( \text{DFA}_2 \) equivalent?)

1 \( Q = Q_1 \cup Q_2 \)
2 \( \Sigma = \Sigma_1 \cup \Sigma_2 \)
3 \( \delta[q, f] = \begin{cases} 
\delta_1[q, f] & \text{if } q \in Q_1 \\
\delta_2[q, f] & \text{if } q \in Q_2 
\end{cases} \)
4 \( \Gamma = \Gamma_1 \cup \Gamma_2 \)
5 \( \gamma[q] = \begin{cases} 
\gamma_1[q] & \text{if } q \in Q_1 \\
\gamma_2[q] & \text{if } q \in Q_2 
\end{cases} \)
6 \( \text{DFA} = (Q, \Sigma, \delta, q_1, \Gamma, \gamma) \)
7 Let \( V \) be a new set
8 foreach \( q \in Q \) do
9 \( \text{Add } \{q\} \text{ to } V \)
10 \( V.\text{Union}(q_1, q_2) \)
11 Push \( (q_1, q_2) \) to a new stack, \( \text{STACK} \)
12 while \( \text{STACK is not empty} \) do
13 Pop \( (p_1, p_2) \) from \( \text{STACK} \)
14 foreach \( f \in \Sigma \) do
15 \( r_1 = V.\text{Find}(\delta[p_1, f]), r_2 = V.\text{Find}(\delta[p_2, f]) \)
16 if \( r_1 \neq r_2 \) then
17 \( V.\text{Union}(r_1, r_2) \)
18 Push \( (r_1, r_2) \) to \( \text{STACK} \)
19 return \( \begin{cases} 
\text{TRUE} & \text{if } \forall s \in V : \forall p, q \in s : \gamma[p] = \gamma[q] \\
\text{FALSE} & \text{otherwise} 
\end{cases} \)

objects in every state in \( s \) have the same type. As discussed in Section 4.2.2.2, a
5-tuple DFA can be modeled as a special case of a 6-tuple DFA.

EQUIV-CHECKER iterates over all fields \( f \in \Sigma \) (line 14) and queries the transition
map \( \delta \) to obtain the next states (line 15). By convention, if \( \delta[q, f] \) is not defined,
since the objects in \( q \) do not have the field \( f \), we assume that \( \delta[q, f] = q_{\text{error}} \). In
addition, \( \gamma[q_{\text{error}}] \) returns a special type for \( q_{\text{error}} \).
4.4.5 The Heap Modeler

After Algorithm 1 has terminated, we have $W = \mathbb{H} / \equiv$ in its line 16. Then MOM specifies the new heap abstraction given in Definition 2, as discussed in Section 4.3.5.

4.5 Implementation

We have implemented MAHJONG as a standalone tool in a total of only 1500 LOC in Java to build a new heap abstraction by merging equivalent automata. MAHJONG is designed to work with mainstream allocation-site-based points-to analysis frameworks such as DOOP [22], SOOT [77], CHORD [16] and WALA [98]. To demonstrate its effectiveness, we have integrated MAHJONG with DOOP [22], a state-of-the-art whole-program points-to analysis framework for Java. Our entire MAHJONG framework has been released as open-source software at http://www.cse.unsw.edu.au/~corg/mahjong. Below we discuss three major optimizations in our implementation.

Disjoint-Set Forest In Algorithms 1 and 4, disjoint sets are used. For efficiency, we have implemented a set of disjoint sets as a disjoint-set forest, by representing each disjoint set as a tree with its root being its representative. Thus, UNION amounts to linking the roots of different trees while FIND returns the root of a tree. To improve the efficiency further, we have also implemented two heuristics, \textit{union by rank} and \textit{path compression} [17]. As a result, the average execution time of each UNION/FIND operation over a disjoint-set forest can be reduced to nearly $O(1)$ [17].
Shared Sequential Automata  In Algorithms 2 and 3, new automata are frequently created. However, different automata can be partly identical, since their common parts correspond to the same objects. Instead of always creating new automata, we allow different automata to share their common parts. This optimization reduces significantly both the time and space costs of the overall algorithm.

Parallel Type-Consistency Checks  A synchronization-free parallelization scheme is used. This is achieved by requiring different threads to merge objects of different types (with every thread executing lines 6 – 13 of Algorithm 1). To avoid synchronizations, object merging takes place only at line 13 of Algorithm 1, and in addition, all shared automata are constructed before type-consistency checks and concurrently read only during the checks.

4.6 Evaluation

We show that MAHjong is effective in significantly scaling context-sensitive points-to analyses for large Java programs while achieving nearly the same precision for type-dependent clients. We address two major research questions:

RQ1. Is MAHjong effective as a pre-analysis?

(a) Is MAHjong lightweight for large programs?

(b) Can MAHjong avoid the allocation-site abstraction’s heap over-partitioning for type-dependent clients?

RQ2. Is MAHjong-based points-to analysis effective?

(a) Can MAHjong accelerate different types of mainstream context-sensitive points-to analyses?
(b) Can MAHJONG achieve comparable precision as the allocation-site abstraction for type-dependent clients?

**Context-Sensitive Points-to Analyses** We consider five context-sensitive points-to analyses from the DOOP framework (version r160113) as baselines. These cover the three main types of mainstream context-sensitivity, call-site-sensitivity [99, 39, 79, 23], object-sensitivity [53, 95, 76] and type-sensitivity [75].

**Benchmarks** We consider 12 large Java programs including 3 popular applications findbugs, checkstyle and JPC and all standard DaCapo benchmarks [8] except jython and hsqldb as they are not scalable under 3 out of the 5 baseline analyses with and without MAHJONG. These programs are all analyzed with a large Java library, JDK1.6.0_45.

As a static reflection analysis may affect both the efficiency and precision of points-to analysis [41, 72], we adopt the same resolution results generated by a dynamic reflection analysis tool, TAMIFLEX [13], in both the five baselines and their corresponding MAHJONG-based points-to analyses.

**Type-Dependent Clients** We consider three type-dependent clients, call graph construction, devirtualization and may-fail casting, also provided by DOOP [22].

**Computing Platform** We have done our experiments on a Xeon E5-1620 3.7GHz machine with 128GB of RAM. The elapsed time of each analysis for each program is the average of 3 runs.

**Pre-Analysis** For this, we use the fast context-insensitive points-to analysis, denoted ci, provided by DOOP [22]. Different pair-wise type-consistency tests are performed in parallel, as discussed in Section 4.5, with 8 threads on 4 cores.
Table 4.2 presents the main results, which will be analyzed when our research questions are discussed below. For a program, we consider the abstract objects reachable from `main()` in both the application and library code.

### 4.6.1 RQ1: MAHJONG’s Effectiveness as a Pre-Analysis

#### 4.6.1.1 Efficiency

The overall pre-analysis phase is fast, as shown in Column 2 of Table 4.2. For a program, its analysis time is broken down into three components, taken by `ci` (the context-insensitive points-to analysis), `FPG` (a module for building its FPG), `MAHJONG` (for creating a new heap abstraction). For all the 12 programs, the average analysis time for `ci` is 62.3 seconds. The runtime overheads for the other two are negligible.

The efficiency of MAHJONG cannot be over-emphasized, as it could not otherwise be used as an enabling technology for a subsequent points-to analysis. On average, an FPG consists of 10073 objects of 1559 types with 2411 fields. This costs MAHJONG only an average of 3.8 seconds. Such good performance is due to both our design (by merging objects in terms of merging equivalent automata) and several effective optimizations performed (as described in Section 4.5).

#### 4.6.1.2 Heap Partitioning

Figure 4.7 shows that MAHJONG can alleviate the heap over-partitioning problem suffered by the allocation-site abstraction effectively for type-dependent clients. The allocation-site abstraction creates an average of 10073 objects per program, ranging from 6190 in `luindex` to 19529 in `eclipse`. In contrast, MAHJONG creates an average of 3826 objects per program, ranging from 2108 in `luindex` to 9414 in `eclipse`. This represents an average reduction of 62%.
Chapter 4. **Mahjong**: Efficient Points-to Analysis by Merging Equivalent Automata

Let us examine `checkstyle` in detail. As shown in Figure 4.7, a total of 10888 objects are created by the allocation-site abstraction but only 4028 objects by `Mahjong`.

![Figure 4.7: Number of abstract objects created by the allocation-site abstraction and Mahjong.](image)

Given the heap partitioned as $\mathbb{H} / \equiv$ for `checkstyle`, Figure 4.8 relates the number of equivalence classes with a particular equivalence class size. In the leftmost point marked by $(1, 3769)$, for example, there are 3769 equivalence classes.

![Figure 4.8: Object merging in checkstyle.](image)
containing one object each. Thus, neither object is merged with any other objects.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Type</th>
<th>Equiv. Class Size</th>
<th>Total No. of Objects</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>java.lang.StringBuilder</td>
<td>1303</td>
<td>1303</td>
<td>char[]</td>
</tr>
<tr>
<td>2</td>
<td>java.lang.Object[]</td>
<td>690</td>
<td>1353</td>
<td>String</td>
</tr>
<tr>
<td>12</td>
<td>antlr.ASTPair</td>
<td>108</td>
<td>109</td>
<td>DetailAST</td>
</tr>
<tr>
<td>55</td>
<td>java.lang.Object[]</td>
<td>12</td>
<td>1353</td>
<td>Integer</td>
</tr>
<tr>
<td>65</td>
<td>java.lang.Object[]</td>
<td>9</td>
<td>1353</td>
<td>QName</td>
</tr>
<tr>
<td>260</td>
<td>antlr.ASTPair</td>
<td>1</td>
<td>109</td>
<td>null</td>
</tr>
</tbody>
</table>

Table 4.1: Some equivalence classes in checkstyle.

Let us examine some equivalence classes, given in Table 4.1, with their ranks (measured in decreasing order of their sizes) shown as well. For StringBuilder (Row 1), all their objects are type-consistent (reaching only char[] objects along any field access path) and thus merged. This is the largest equivalence class, corresponding to the right-most point marked by (1303, 1) in Figure 4.8.

For some other types like Object[] (Rows 2, 4 and 5), blindly merging all its objects would be imprecise (Section 4.2.1). In contrast, MAHJONG merges only type-consistent objects in order to maximally preserve precision for type-dependent clients. Thus, MAHJONG ends up with different equivalent classes containing objects of type Object[] for storing objects of different types, such as String (Row 2), Integer (Row 4), and QName (Row 5).

Finally, we show that MAHJONG can also distinguish null from other objects, because null may affect precision as explained in Section 4.3.6. MAHJONG partitions 109 objects of ASTPair into two equivalence classes, with one containing 108 objects whose fields point to objects of type DetailAST (Row 3) and the other that contains one single object with null fields (Row 6).
4.6.2 RQ2: MAHjong-based Points-to Analysis

Mainstream points-to analyses for Java programs rely on the allocation-site-based abstraction to model the heap [38, 75, 39, 79, 34, 76, 95]. We demonstrate experimentally that MAHjong is a better alternative for type-dependent clients.

Concretely, we show that MAHjong can achieve the following goal in the real world. Suppose a software developer intends to apply a points-to analysis to a program under a given time budget. MAHjong opens up new opportunities for the developer to either accelerate the chosen points-to analysis or replace it with a more precise but more expensive points-to analysis under still the same budget.

4.6.2.1 Baselines and Metrics

We consider three types of context-sensitive points-to analyses: call-site-sensitivity ($cs$), object-sensitivity ($obj$) and type-sensitivity ($type$). Specifically, five points-to analyses in DOOP [22] are selected as baselines: 2-$cs$ (2-call-site-sensitive), 2-$obj$ (2-object-sensitive), 3-$obj$ (3-object-sensitive), 2-$type$ (2-type-sensitive), and 3-$type$ (3-type-sensitive). In principle, 2-$cs$ is not compatible with the others, 3-$\mathcal{A}$ is no less precise than 2-$\mathcal{A}$, and $k$-$obj$ is no less precise than $k$-$type$.

Currently, each baseline $k$-$\mathcal{A}$ uses the allocation-site abstraction. $M$-$k$-$\mathcal{A}$ denotes the version of $k$-$\mathcal{A}$ that uses the heap abstraction provided by MAHjong. Thus, there are also five MAHjong-based points-to analyses altogether.

The three type-dependent clients, call graph construction, devirtualization and may-fail casting, are widely used in the literature [39, 34, 76, 75, 95]. We consider the following metrics: the number of call graph edges (#call graph edges), the number of casting operations that may fail (#may-fail casts), and the number of virtual call sites that cannot be disambiguated into mono-calls (#poly call sites).

The time budget for each analysis is set to 5 hours.
4.6.2.2 Efficiency and Precision

Table 4.2 presents our results, showing clearly the effectiveness of MAHJONG in boosting existing points-to analyses while maintaining their precision for type-dependent clients.

For each program, five metrics are considered: “analysis time”, “speedup”, “#may-fail casts”, “#poly call sites” and “#call graph edges”. In all cases except “speedup”, smaller is better. With “speedup” ignored, Table 4.2 contains 480 concrete results (= 4 metrics $\times$ 12 programs $\times$ 10 points-to analyses (including the 5 baselines and 5 MAHJONG variants)).

In computing the speedup of $M_k-A$ over $k-A$ for a program, the pre-analysis time on the program is ignored. There are three reasons: (1) the points-to information produced by “ci” in Table 4.2 may already exist and can be reused, (2) the pre-analysis time is relatively small (compared to the analysis time of a subsequent $M_k-A$), and (3) the pre-analysis will be used to drive many points-to analyses.

**Improved Efficiency** MAHJONG is versatile enough in accelerating all the five points-to analyses with three different types of context-sensitivity. For every program where $k-A$ is scalable, $M_k-A$ is also scalable and faster than $k-A$.

MAHJONG is highly effective in boosting performance. For the programs where both $k-A$ and $M_k-A$ are scalable, MAHJONG achieves an average speedup of 15.4X (ranging from 1.03X by $M_2$-obj/$2$-obj for bloat to 168.8X by $M_3$-obj/$3$-obj for luindex). Table 4.2 divides visually the 12 programs into two groups. For the top six, $k-A$ scales whenever $M_k-A$ scales. However, $M_k-A$ is faster than $k-A$, achieving an average speedup of 22.2X. This is especially significantly for the most-precise configuration $M_3$-obj/$3$-obj. For every program in the bottom six, MAHJONG enables using a more precise points-to analysis that is not scalable if
<table>
<thead>
<tr>
<th>Program</th>
<th>Pre-analysis</th>
<th>Metrics</th>
<th>2-es</th>
<th>M-2-es</th>
<th>2-type</th>
<th>M-2-type</th>
<th>3-type</th>
<th>M-3-type</th>
<th>2-obj</th>
<th>M-2-obj</th>
<th>3-obj</th>
<th>M-3-obj</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>analysis time (sec.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>speedup</td>
<td>7.5X</td>
<td>1.4X</td>
<td>7.5X</td>
<td>3.2X</td>
<td>118.8X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>antlr</td>
<td>ci: 44.1s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 1.3s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 1.3s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>analysis time (sec.)</td>
<td>1510.3</td>
<td>1510.3</td>
<td>526.9</td>
<td>67.8</td>
<td>38.7</td>
<td>16.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>speedup</td>
<td>3.5X</td>
<td>1.4X</td>
<td>7.8X</td>
<td>2.0X</td>
<td>123.5X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pchip</td>
<td>ci: 34.7s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 0.7s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 1.1s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>analysis time (sec.)</td>
<td>1480.2</td>
<td>1480.2</td>
<td>526.4</td>
<td>42.8</td>
<td>72.9</td>
<td>28.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>speedup</td>
<td>4.9X</td>
<td>1.5X</td>
<td>12.3X</td>
<td>2.6X</td>
<td>168.8X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>luindex</td>
<td>ci: 26.2s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 0.8s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 1.1s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>analysis time (sec.)</td>
<td>2099.4</td>
<td>2099.4</td>
<td>906.1</td>
<td>82.9</td>
<td>145.1</td>
<td>82.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>speedup</td>
<td>3.8X</td>
<td>1.5X</td>
<td>10.9X</td>
<td>1.8X</td>
<td>113.3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pnd</td>
<td>ci: 44.8s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 1.4s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 1.5s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>analysis time (sec.)</td>
<td>7769.3</td>
<td>7769.3</td>
<td>533.6</td>
<td>124.5</td>
<td>3611.9</td>
<td>3501.5</td>
<td>&gt;5h</td>
<td></td>
<td>&gt;5h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>speedup</td>
<td>1.0X</td>
<td>1.3X</td>
<td>4.3X</td>
<td>1.2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>ci: 37.7s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 2.4s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 1.9s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>analysis time (sec.)</td>
<td>5476.2</td>
<td>5476.2</td>
<td>2967.8</td>
<td>518.5</td>
<td>997.9</td>
<td>279.8</td>
<td>&gt;5h</td>
<td></td>
<td>&gt;5h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>speedup</td>
<td>3.3X</td>
<td>2.0X</td>
<td>5.7X</td>
<td>3.6X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program</td>
<td>checkstyle</td>
<td>analysis time (sec.)</td>
<td>speedup</td>
<td>ci: 66.6s</td>
<td>FPG: 3.0s</td>
<td>MAHJONG: 3.1s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
<td>----------------------</td>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7644.8</td>
<td>3186.7</td>
<td>187.8</td>
<td>92.3</td>
<td>5120.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1923.0</td>
<td>1123.8</td>
<td>1123.5</td>
<td>1135</td>
<td>1140</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5120.6</td>
<td>379.8</td>
<td>1946.6</td>
<td>277.1</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3037.7</td>
<td>1123.5</td>
<td>1123.5</td>
<td>1135</td>
<td>1140</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>ci: 38.7s</td>
<td>1996.1</td>
<td>464.4</td>
<td>99.0</td>
<td>57.7</td>
<td>1122.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 1.2s</td>
<td>1996.1</td>
<td>464.4</td>
<td>99.0</td>
<td>57.7</td>
<td>1122.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 1.7s</td>
<td>1996.1</td>
<td>464.4</td>
<td>99.0</td>
<td>57.7</td>
<td>1122.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1996.1</td>
<td>464.4</td>
<td>99.0</td>
<td>57.7</td>
<td>1122.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1996.1</td>
<td>464.4</td>
<td>99.0</td>
<td>57.7</td>
<td>1122.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>ci: 41.4s</td>
<td>1444.7</td>
<td>309.4</td>
<td>46.4</td>
<td>29.6</td>
<td>780.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 0.8s</td>
<td>1444.7</td>
<td>309.4</td>
<td>46.4</td>
<td>29.6</td>
<td>780.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 1.0s</td>
<td>1444.7</td>
<td>309.4</td>
<td>46.4</td>
<td>29.6</td>
<td>780.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1444.7</td>
<td>309.4</td>
<td>46.4</td>
<td>29.6</td>
<td>780.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1444.7</td>
<td>309.4</td>
<td>46.4</td>
<td>29.6</td>
<td>780.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPC</td>
<td>ci: 58.9s</td>
<td>3464.1</td>
<td>1155.1</td>
<td>147.1</td>
<td>90.6</td>
<td>1599.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 2.1s</td>
<td>3464.1</td>
<td>1155.1</td>
<td>147.1</td>
<td>90.6</td>
<td>1599.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 4.5s</td>
<td>3464.1</td>
<td>1155.1</td>
<td>147.1</td>
<td>90.6</td>
<td>1599.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3464.1</td>
<td>1155.1</td>
<td>147.1</td>
<td>90.6</td>
<td>1599.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3464.1</td>
<td>1155.1</td>
<td>147.1</td>
<td>90.6</td>
<td>1599.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findbugs</td>
<td>ci: 90.6s</td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 4.6s</td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 3.2s</td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eclipse</td>
<td>ci: 174.1s</td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPG: 15.5s</td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAHJONG: 21.4s</td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1492.8</td>
<td>5646.6</td>
<td>1229.3</td>
<td>107.4</td>
<td>&gt;5h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Efficiency and precision metrics for all programs and analyses with and without MAHJONG. In all cases (except speedup), lower is better. Symbol ∞ is used in speedup when a baseline analysis is not scalable but MAHJONG is scalable.
the allocation-site abstraction is used instead.

**Preserved precision** For every program, as shown in Table 4.2, MAHJONG achieves nearly the same precision for every client under every configuration $M-k.A/k.A$. Thus, merging type-consistent objects can maximally preserve precision as discussed in Section 4.3.6 and validated here.

**Call-Site-Sensitivity** $M-2-cs$ is no more precise than $2-cs$ in principle (Section 4.3.6) but nearly as precise in practice. For devirtualization, $M-2-cs$ is equally as precise as $2-cs$. For may-fail casting, $M-2-cs$ is negligibly worse than $2-cs$ (with an average precision loss of 0.04%), by reporting only 5 more may-fail casts each in checkstyle and findbugs. For call graph construction, $M-2-cs$ is also marginally worse (with an average precision loss of 0.006%), by including only a few extra edges in pmd (3), chart (14), checkstyle (20), and findbugs (17).

**Object-Sensitivity** $M-k.obj$ is also no more precise than $k.obj$ in principle (Section 4.3.6) but nearly as precise in practice. For call graph construction, devirtualization and may-fail casting, $M-2.obj$ experiences a small loss of precision of 0.02%, 0.23% and 0.04% over $2.obj$, respectively, on average. For $M-3.obj$ over $3.obj$, these percentages are 0.02%, 0.29% and 0.00%, respectively. For may-fail casting, $M-2.obj$ is on a par with $2.obj$ if checkstyle is ignored, and $M-3.obj$ is equally as precise as $3.obj$.

**Type-Sensitivity** $M-k.type$ may lose or even gain precision compared with $k.type$, as discussed in Section 4.3.6 and illustrated by an example in Section 4.6.2.3. For may-fail casting, $M-k.type$ is slightly more precise than $k.type$ in all the programs except antlr. The average precision gains for $M-2.type/2.type$
and M-3-type/3-type are 0.91% and 1.11%, respectively. For the other two clients, M-k-type is slightly less precise than k-type in every program. For call graph construction and devirtualization, M-2-type experiences a small loss of precision of 0.02% and 0.18% over 2-type, respectively. In the case of M-3-type/3-type, these percentages are 0.02% and 0.22%, respectively.

4.6.2.3 Discussion

We discuss three observations about some results in Table 4.2.

**Speedups of M-3-obj over 3-obj** MAHJONG is most impressive in scaling 3-obj, the most precise baseline used. For the four programs, antlr, fop, luindex and pmd, where 3-obj is scalable, M-3-obj is 131X faster, on average, while achieving nearly the same precision for all the three clients. For the remaining eight, where 3-obj is unscalable, M-3-obj is scalable for checkstyle, xalan, lusearch, JPC and fingbugs, by spending an average of 33.42 minutes only.

Why does M-3-obj/3-obj deliver significantly better speedups than M-2-obj/2-obj? By using one extra level of context elements than 2-obj, 3-obj often incurs an exponential growth in the number of contexts used. By merging type-consistent objects, which happen to be used as context elements at this extra level in 3-obj, M-3-obj can drastically reduce the number of contexts used and thus accelerate the analysis. Consider luindex, where the speedup achieved by M-3-obj/3-obj is the highest obtained. The number of context-sensitive points-to relations produced under 2-obj is 9255034 but grows to 191160483 under 3-obj, which are reduced significantly to 4256310 under M-3-obj.

**Precision Gains of M-k-type over k-type** For simplicity, we explain this by using an (abstract) example given in Figure 4.9.
Chapter 4. **Mahjong**: Efficient Points-to Analysis by Merging Equivalent Automata

In type-sensitivity [75], an allocation site $i$ in a context is approximated by the class that contains $i$. In this example, $k$-type will represent the allocation sites 1 and 2 by $T$ in contexts. Thus, the two allocation sites that are distinguished by $k$-obj are merged.

According to MAHJONG, $o_1^A$ and $o_3^A$ are type-consistent, falling into the same equivalence class. If $o_3^A$ happens to be selected as a representative, then $M$-$k$-type will be able to distinguish the allocation sites 1 and 2 by $U$ and $T$ respectively, and obtain better precision for this case.

**Unscalability of MAHJONG-based Points-to Analyses** As shown in Table 4.2, $M$-2-cs is unscalable for **eclipse** and $M$-3-obj is unscalable for **bloat**, **chart** and **eclipse**. Why is $M$-3-obj scalable for some large programs such as **findbugs** but unscalable for some small ones such as **bloat**? As shown in Figure 4.7, MAHJONG creates 5233 objects for **findbugs** but only 3107 objects for **bloat**.

$M$-3-obj is unscalable for **bloat** possibly due to its object structure used. Some methods are both invoked on many (abstract) receiver objects and allocate many objects. Thus, the number of contexts becomes extremely large. To alleviate this problem, one solution is to use a coarser relation than $\equiv$ given in Definition 1 so that more objects can be merged together. Another solution is to apply $3$-obj only selectively to parts of the program when moving from $2$-obj to $3$-obj.
4.7 Related Work

We review only the work most closely related to (whole-program) points-to analysis for object-oriented programs.

Points-to Analysis  Context-sensitivity is essential in achieving good efficiency and precision trade-offs for Java programs [39, 40, 72, 80]. There are three main flavors: call-site-sensitivity, object-sensitivity, and type-sensitivity.

Call-site-sensitivity [99, 39, 79, 23], i.e., k-CFA [71] is often used to analyze C programs [105, 63, 9, 64]. To better exploit the object-oriented features in Java, object-sensitivity is proposed [53, 55]. It distinguishes the calling contexts of a method call in terms of the allocation sites of its receiver object rather than call-sites (like what call-site-sensitivity does). Such design enables object-sensitivity to yield significantly higher precision at usually less cost than call-site-sensitivity [95, 23, 39, 34]. However, for large Java programs like the findbugs and eclipse in our experiment, object-sensitivity is hardly scalable despite its good precision. To trade precision for efficiency, type-sensitivity is thus introduced [75]. It approximates the allocation site $i$ in the context of object-sensitivity by the type containing $i$, which makes itself more scalable but less precise than object-sensitivity [34, 75, 95].

In this chapter, we have explained (Section 4.3.6) and demonstrated (Section 4.6.2) the effectiveness of MAHJONG on improving the efficiency of all the above three main flavors of context-sensitivity: call-site-sensitivity, object-sensitivity and type-sensitivity while preserving their precision for the type-critical clients. Hence, for type-dependent clients, MAHJONG represents a better alternative than the allocation-site abstraction.

In addition, the benefit of MAHJONG is expected to generalize to other variations of context-sensitivity [34, 95] as they build on existing context-sensitivity
techniques. Specifically, Hybrid context-sensitive points-to analysis [34] applies call-site-sensitivity to static call sites and object/type-sensitivity to virtual call sites. More precise object-sensitive points-to analysis [95] is achieved by avoiding automatically discovered redundant context elements which are useless in improving the precision.

There are other ways to improve the efficiency of points-to analysis. In [73], a pre-analysis is performed to identify and eliminate the redundant statements which will not affect the points-to facts in a flow-insensitive points-to analysis, if they are removed. As a result, it is able to speedup a points-to analysis by giving it an input program with less statements. This approach and MAHJONG can be complementary to each other. In [76], empirical heuristics are used to make efficiency and precision trade-offs. As a result, some parts of the program are analyzed context-sensitively and some other parts are analyzed context-insensitively. It estimates heuristically which program elements’ cost is vastly disproportionate (under context-sensitivity) and applies context-insensitivity to it. Such elements are estimated in a context-insensitive pre-analysis by observing different heuristic parameters values such as the cumulative size of points-to sets over all local variables in every method.

Heap Abstraction There are mainly two types of heap models in static analysis: store-based, e.g., the allocation-site abstraction and storeless, e.g., access paths [32]. The former is usually adopted in points-to analysis and the latter in alias analysis [72]. We focus on store-based models for Java below.

Due to its good precision, the allocation-site abstraction is adopted by (whole-program) points-to analysis techniques in both the literature [38, 55, 79, 75, 34, 76, 95] and open-source tools, such as DOOP [22], SOOT [77], CHORD [16] and WALA [98]. To achieve better precision, the allocation-site abstraction is usually further refined by the context-sensitive heap abstraction or heap cloning techniques
The allocation-type abstraction (with one abstract object per type) was used earlier to resolve virtual calls [94, 67]. It is reasonably precise, compared with 0-CFA [71] and CHA [20], which are fast but imprecise. Currently, points-to analysis no longer relies on the allocation-type abstraction to model the heap, as it is imprecise [99, 32, 72].

Liang and Naik [45] introduce a sophisticated allocation-type-based abstraction in a pre-pruning analysis to scale a subsequent refinement analysis to answer some queries effectively. An allocation site $h$ is represented by its dynamic type and the type containing $h$. Unlike MAHJONG, however, such an abstraction is still not precise for points-to analysis.
Chapter 5

Conclusions and Future Work

In this chapter, we conclude the two points-to analysis techniques for Java presented in this thesis, and then discuss some potential future work.

5.1 Conclusions

Points-to analysis, as a fundamental program analysis, is required by plenty of client applications, such as bug detection, security analysis and compiler optimization, as well as other program analyses, such as program slicing, reflection analysis and escape analysis. A practically useful points-to analysis should make a good trade-off between precision and efficiency. In this thesis, we introduce two techniques, BEAN and MAHJONG, to improve the precision and efficiency of points-to analysis for Java respectively without compromising the other much.

In Chapter 3, we have presented BEAN to make points-to analysis more precise. In the past decade, object-sensitivity has been recognized as an excellent context abstraction for designing precise context-sensitive points-to analysis for Java and thus adopted widely in practice. However, how to make a $k$-object-sensitive analysis more precise becomes rather challenging. Simply increasing the limit of context
length, $k$, is able to improve the precision but it will also cause a dramatic slowdown in the analysis. We provide a general approach, BEAN, to addressing this problem. By reasoning about an object allocation graph (OAG) built based on a pre-analysis on the program, we can identify and thus avoid redundant context elements (in improving the precision) that are otherwise used in a traditional $k$-object-sensitive analysis, thereby improving its precision at a small increase in cost.

In Chapter 4, we have presented MAHJONG, a novel technique for abstracting the heap for points-to analysis for object-oriented programs. By exploiting the type-consistency property of (abstract) heap objects based on a pre-analysis on the program, MAHJONG merges the type-consistent objects which are distinguished in the traditional allocation-site-based points-to analysis. As a result, MAHJONG is able to significantly accelerate existing points-to analyses while maximally preserving their precision for an important class of type-dependent clients, including call graph construction. MAHJONG is expected to provide significant benefits to many program analyses, such as bug detection, security analysis, program verification and program understanding, where call graphs are required.

5.2 Future Work

We have developed and presented BEAN in Chapter 3, which improves the precision of points-to analysis by removing redundant context elements from the contexts in traditional object-sensitivity. However, the problem of redundant context elements exists in not only object-sensitivity but also other forms of context-sensitivity, such as call-site-sensitivity ($k$-CFA) and type-sensitivity. Their redundant context elements can be identified and avoided in an OAG-like graph in a similar way.

The MAHJONG framework that we have developed and introduced in Chapter 4
Chapter 5. Conclusions and Future Work

opens up a number of research directions on providing suitable heap abstractions for points-to analysis for large codebases and addressing their interplay. First, our notion of type-consistency may be overly restrictive for some other clients and can be relaxed. Second, as there are little benefits to analyze merged objects context-sensitively for type-dependent clients, it may be worthwhile investigating how to enforce selective context-sensitivity systematically by exploiting this insight. Third, how do we adaptively refine a MAHJONG-like heap abstraction to support demand queries? Finally, it will be interesting to combine MAHJONG and a storeless heap abstraction to support points-to analysis.
Bibliography


grams using static class hierarchy analysis. In *Proceedings of the 9th Euro-

analysis for java in under a minute. In *Proceedings of the 2015 ACM SIG-
PLAN International Conference on Object-Oriented Programming, Systems,

program-analysis.org.

pointer analysis for Java. In *Proceedings of the 13rd Asian Conference on

estate verification in the presence of aliasing. In *Proceedings of the 2006 In-
ternational Symposium on Software Testing and Analysis*, ISSTA ’06, pages
133–144, 2006.


program understanding. *Journal of Systems and Software*, 44(3):213 – 227,
1999.


[34] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In Proceedings of the 34th ACM SIGPLAN Conference on Pro-


refinement for program analyses in Datalog. In Proceedings of the 35th ACM
SIGPLAN Conference on Programming Language Design and Implementa-

[107] Y. Zhang, T. Tan, Y. Li, and J. Xue. Ripple: Reflection analysis for android
apps in incomplete information environments. In Proceedings of the 7th ACM
Conference on Data and Application Security and Privacy, CODASPY ’17,
2017.