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

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

List of Figures

List of Tables

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

  `http://www.cse.unsw.edu.au/~corg/mahjong`.

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

Statement	**Kind**
`x = new T()`	New
`x = y`	Assign
`x = y.f`	Load
`x.f = y`	Store
`x = y.g(arg1, ..., argn)`	Call

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

class type	$t \in \mathbb{T}$
variable	$x, y \in \mathbb{V}$
heap object	$o_i, o_j \in \mathbb{H}$
field	$f \in \mathbb{F}$
method	$m \in \mathbb{M}$
method invocation	$l \in \mathbb{I}$
method context	$c \in \mathbb{C}$
heap context	$hc \in \mathbb{HC}$

$$\begin{array}{rcl}
pt & : & \mathbb{C} \times \mathbb{V} \to \mathcal{P}(\mathbb{HC} \times \mathbb{H}) \\
fpt & : & \mathbb{HC} \times \mathbb{H} \times \mathbb{F} \to \mathcal{P}(\mathbb{HC} \times \mathbb{H}) \\
mtdCtxSelector & : & \mathbb{C} \times \mathbb{I} \times \mathbb{HC} \times \mathbb{H} \to \mathbb{C} \\
heapCtxSelector & : & \mathbb{C} \times \mathbb{H} \to \mathbb{HC} \\
contextsOf & : & \mathbb{M} \to \mathcal{P}(\mathbb{C})
\end{array}$$

Figure 2.1: Notations.

for object allocations. Specifically, *mtdCtxSelector* uses the information available at a call site, i.e., the method context for the method containing the call site ($\mathbb{C}$), the label of the call site ($\mathbb{I}$), the receiver object of the call site ($\mathbb{H}$) and the heap context for the receiver object ($\mathbb{HC}$), 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 ($\mathbb{C}$) and a label of the call site ($\mathbb{I}$), to select a new context. But the selector of object-sensitivity only considers the last input elements to select contexts.

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 ($\mathbb{C}$) and the heap object ($\mathbb{H}$) allocated at the site) to select a new heap context for the heap object.

Finally, *contextsOf* maps a method to its method contexts.

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.

$$\boxed{m\text{: the containing method for each statement being analysed}}$$

$$\frac{\texttt{i: x = new T()} \quad c \in \mathit{contextsOf}(m) \quad hc = \mathit{heapCtxSelector}(c, o_i)}{\langle hc, o_i \rangle \in \mathit{pt}(c, x)} \; [\textsc{New}]$$

$$\frac{\texttt{x = y} \quad c \in \mathit{contextsOf}(m)}{\mathit{pt}(c, y) \subseteq \mathit{pt}(c, x)} \; [\textsc{Assign}]$$

$$\frac{\texttt{x = y.f} \quad c \in \mathit{contextsOf}(m) \quad \langle hc, o_i \rangle \in \mathit{pt}(c, y)}{\mathit{fpt}(hc, o_i, f) \subseteq \mathit{pt}(c, x)} \; [\textsc{Load}]$$

$$\frac{\texttt{x.f = y} \quad c \in \mathit{contextsOf}(m) \quad \langle hc, o_i \rangle \in \mathit{pt}(c, x)}{\mathit{pt}(c, y) \subseteq \mathit{fpt}(hc, o_i, f)} \; [\textsc{Store}]$$

$$\frac{\begin{array}{c}\texttt{l: x = y.g(arg1,...,argn)} \quad c \in \mathit{contextsOf}(m) \quad \langle hc, o_i \rangle \in \mathit{pt}(c, y) \\ m' = \mathit{dispatch}(o_i, g) \quad c' = \mathit{mtdCtxSelector}(c, l, hc, o_i)\end{array}}{\begin{array}{c} c' \in \mathit{contextsOf}(m') \quad \langle hc, o_i \rangle \in \mathit{pt}(c', m'_{this}) \\ \forall\, 1 \le k \le n : \mathit{pt}(c, arg_k) \subseteq \mathit{pt}(c', m'_{pk}) \quad \mathit{pt}(c', m'_{ret}) \subseteq \mathit{pt}(c, x)\end{array}} \; [\textsc{Call}]$$

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 `arr` as in [80, 42]. Hence, `x = y[i]` (`x[i] = y`) is handled as `x = y.arr` (`x.arr = y`) by [Load] ([Store]).

[Call] handles method invocation. It uses function $dispatch(o_i, g)$ to resolve the virtual dispatch of method g on the receiver object o_i to be m'. Here, *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'_{this} for the receiver object and $m'_{p1}, ..., m'_{pn}$ for the remaining parameters, and a pseudo-variable m'_{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'_{this} and $m'_{p1}, ..., m'_{pn}$ (under context c'). In addition, [Call] makes the points-to set of m'_{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 *mtdCtxSelector* and *heapCtxSelector*.

In each context-sensitivity, *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):

$$\mathit{heapCtxSelector}(c, \_) = c$$

It implies that the domains of method context ($\mathbb{C}$) and heap context ($\mathbb{HC}$) 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 $\mathbb{C}$ consists of sequences of call sites:

$$\mathbb{C} = \mathbb{I}^0 \cup \mathbb{I}^1 \cup \mathbb{I}^2 \ldots$$

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):

$$\mathit{mtdCtxSelector}(c, l, \_, \_) = c +\!\!+ l$$

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:

$$\mathbb{C} = \mathbb{H}^0 \cup \mathbb{H}^1 \cup \mathbb{H}^2 ...$$

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:

$$\mathbb{C} = \mathbb{T}^0 \cup \mathbb{T}^1 \cup \mathbb{T}^2 ...$$

To formalize type-sensitivity, we define a function $enclosingType : \mathbb{H} \rightarrow \mathbb{T}$ which maps a heap object to a class type enclosing the allocation site of the object. Now we can instantiate *mtdCtxSelector* for type-sensitivity:

$$mtdCtxSelector(\_, \_, hc, o_i) = hc \mathbin{++} enclosingType(o_i)$$

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 [25, 106], data race detection [57], information flow analysis [6, 27, 50], and program slicing [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* [34], provided in Doop [22], resulting in two Bean-directed points-to analyses, *B-2-obj* and *B-S-2-obj*, respectively. We have considered *may-alias* and *may-fail-cast*, two representative clients used elsewhere [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 [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.

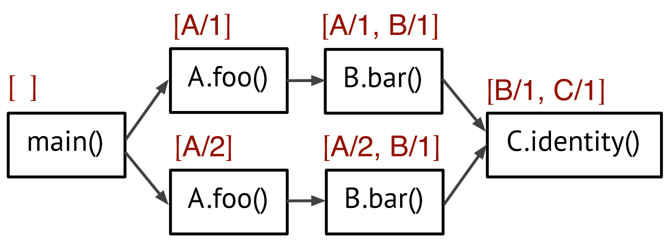
3.2.1 Redundant Elements in Method Contexts

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

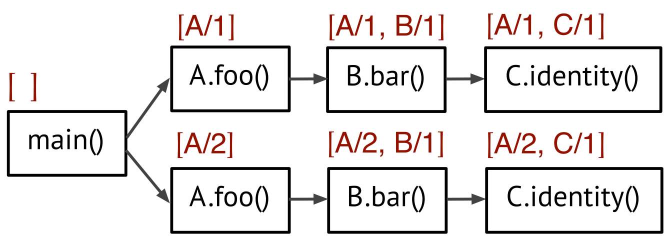
```
void main(Object[] args) {
    A a1 = new A(); // A/1
    Object v1 = a1.foo(new Object()); // O/1

    A a2 = new A(); // A/2
    Object v2 = a2.foo(new Object()); // O/2
}
class A {
    Object foo(Object v) {
        B b = new B(); // B/1
        return b.bar(v);
    }
}
class B {
    Object bar(Object v) {
        C c = new C(); // C/1
        return c.identity(v);
    }
}
class C {
    Object identity(Object v) { return v; }
}
```

(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 `v1` and `v2`.

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-obj*, where each method is analyzed separately for each different calling context, denoted by [. . .]. `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 `v1` and `v2` are made to point to both `O/1` and `O/2` at the same time, causing *may-alias* to report a spurious alias. During any program execution, `v1` and `v2` can only point to `O/1` and `O/2`, respectively.

In Figure 3.1(c), we give the context-sensitive call graph computed by Bean, where `C.identity()` is now analyzed separately under two different contexts, [`A/1,C/1`] and [`A/2,C/1`]. Due to the improved precision, `v1` (`v2`) now points to `O/1` (`O/2`) only, causing *may-alias* to conclude that both are no longer aliases.

3.2.2 Redundant Elements in Heap Contexts

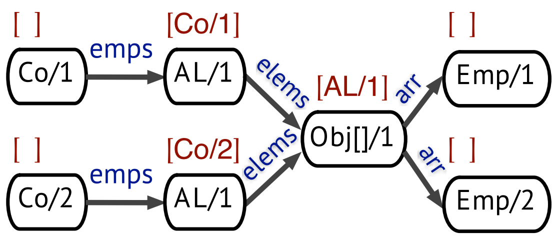
We now use an example in Figure 3.2 to illustrate how *2-obj* analyzes it imprecisely due to its use of a redundant element in heap contexts and how Bean avoids the imprecision by avoiding this redundancy. Our *may-alias* client now issues an alias query for variables `emp1` and `emp2`. In Figure 3.2(a), 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-obj*, where each node represents an abstract heap object created under the corresponding context, denoted [. . .], and each edge represents a field points-to relation with the corresponding field name being labeled on the edge. An array object is

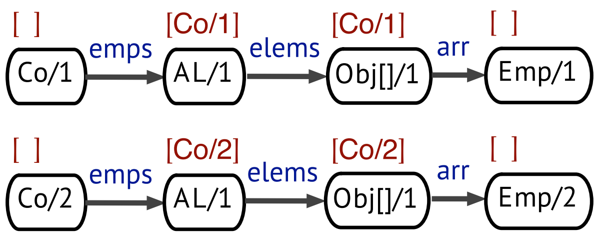
```java
void main(String[] args) {
    Company comp1 = new Company(); // Co/1
    comp1.addEmployee(new Employee()); // Emp/1
    Employee emp1 = comp1.getEmployee(0);

    Company comp2 = new Company(); // Co/2
    comp2.addEmployee(new Employee()); // Emp/2
    Employee emp2 = comp2.getEmployee(0);
}
class Employee {...}
class Company {
    private ArrayList emps;
    Company() { emps = new ArrayList(); } // AL/1
    void addEmployee(Employee emp) { emps.add(emp); }
    Employee getEmployee(int i) {
       return (Employee) emps.get(i);
   }
}
class ArrayList {
    private Object[] elems;
    private int size = 0;
    ArrayList() { elems = new Object[10]; } // Obj[]/1
    void add(Object e) { elems[size++] = e; }
    Object get(int i) { return elems[i]; }
}
```

(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 `arr`. Hence, `x[i] = y` (`y = x[i]`) is handled as `x.arr = y` (`y = x.arr`).

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

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

3.2.3 Discussion

As illustrated above, *k-obj* selects blindly a sequence of k-most-recent allocation sites as a context. To analyze large-scale software scalably, k is small, which is 2 for a method context and 1 for a heap context in *2-obj*. Therefore, redundant context elements, such as `B/1` in [`B/1,C/1`] in Figure 3.1(b) and `AL/1` in [`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 *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).

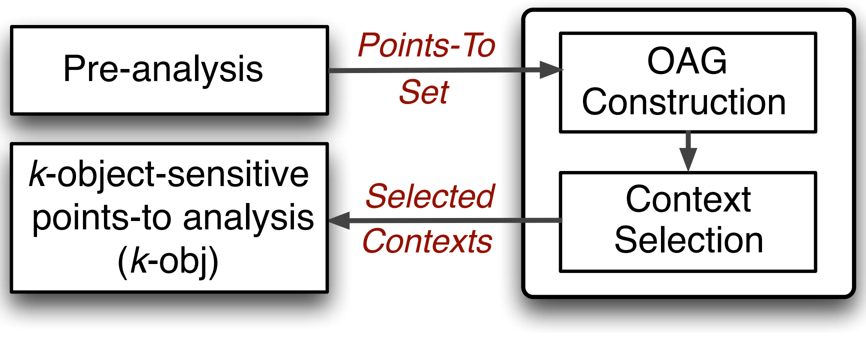
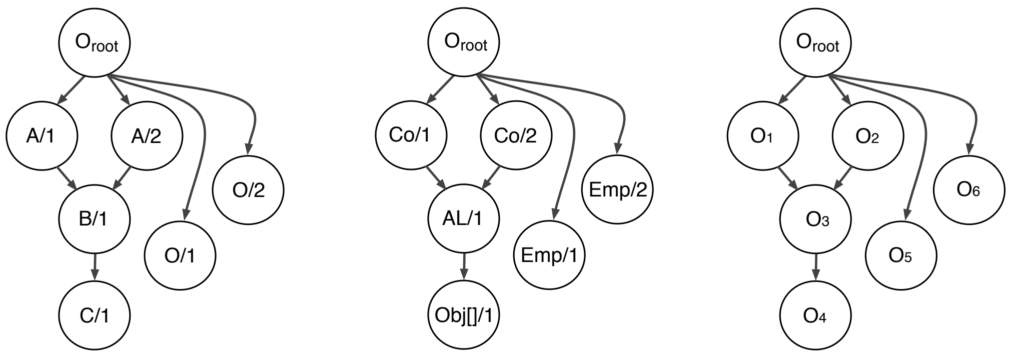


Figure 3.3: Overview of Bean.

3.3 Bean

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.



(a) Figure 3.1 (b) Figure 3.2 (c) Example 2

Figure 3.4: The OAGs for the two motivating programs in Figures 3.1 and 3.2.

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 $\ell_1 \to \ell_2 \in E$ 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 ℓ. Given this, $\ell_1 \to \ell_2$ signifies that ℓ_1 is the receiver object of the method that made the allocation of ℓ_2. Therefore, ℓ_1 is called an *allocator object* of ℓ_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, $\mathtt{O}_{root}$, 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 ℓ, i.e., an abstract object ℓ, consists of its allocator object (ℓ'), the allocator object of ℓ', and so on. The OAG provides a new perspective for object sensitivity, since a context for an object ℓ is simply a path from $\mathtt{O}_{root}$ to ℓ. As a result, the problem of selecting contexts for ℓ can be recast as one of solving a problem of distinguishing different paths from $\mathtt{O}_{root}$ to ℓ. Traditionally, a k-object-sensitive analysis selects blindly a suffix of a path from $\mathtt{O}_{root}$ to ℓ with length k.

3.3.2 Context Selection

Given an object ℓ in G, BEAN selects its contexts in G as sequences of its direct or indirect allocators that are useful to distinguish different paths from $\mathtt{O}_{root}$ to ℓ 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 `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 $\mathtt{O}_{root}$ to `C/1`: $\mathtt{O}_{root} \to \mathtt{A/1} \to \mathtt{B/1} \to \mathtt{C/1}$ and $\mathtt{O}_{root} \to \mathtt{A/2} \to \mathtt{B/1} \to \mathtt{C/1}$. Note that *2-obj* has selected a suffix of the two paths, $\mathtt{B/1} \to \mathtt{C/1}$, which happens to represent the same context [`B/1`,`C/1`] for `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, `C.identity()` is now analyzed under 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 `Obj[]/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 $\texttt{O}_{root}$ to `Obj[]/1`: $\texttt{O}_{root} \rightarrow$ `Co/1` $\rightarrow$ `AL/1` $\rightarrow$ `Obj[]/1` and $\texttt{O}_{root} \rightarrow$ `Co/2` $\rightarrow$ `AL/1` $\rightarrow$ `Obj[]/1`. Note that *2-obj* has selected a suffix of the two paths, `AL/1` $\rightarrow$ `Obj[]/1`, which happens to represent the same heap context [`AL/1`] for `Obj[]/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 `Obj[]/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-*obj*, making the resulting analysis even more precise.

- Bean is expected to be more precise than k-*obj* in general, with a precision that is guaranteed to be as good as k-*obj* in the worst case.

3.4 Formalism

Without loss of generality, we formalize 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.

OAG	$G = (N, E)$
node	$o_i, o_j \in N \subseteq \mathbb{H}$
edge	$o_i \rightarrow o_j \in E \subseteq N \times N$

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_{root} to N.

$$\frac{\langle \_, o_i \rangle \in pt(\_, \_)}{o_i \in N}\ \text{[OAG-NODE]} \qquad \frac{}{o_{root} \in N}\ \text{[OAG-DUMMYNODE]}$$

$$\frac{\langle \_, o_i \rangle \in pt(\_, m_{this}) \quad m \in \mathbb{M} \quad o_j \textbf{ is allocated in } m}{o_i \rightarrow o_j \in E}\ \text{[OAG-EDGE]}$$

$$\frac{o_i \in N \quad o_i \neq o_{root} \quad o_i \textbf{ does not have any incoming edge}}{o_{root} \rightarrow o_i \in E}\ \text{[OAG-DUMMYEDGE]}$$

Figure 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_{this}, where m is the name of a method, represents the `this` variable of method m, which points to the receiver object of method m. By [OAG-DUMMYEDGE], we add an edge from o_{root} to every object o_i without any incoming edge yet, to indicate that o_{root} is now a pseudo allocator object of o_i. Note that an object allocated in `main()` or a static initialiser does not have an allocator object. Due to o_{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:* $pt(\texttt{v1}) = pt(\texttt{v2}) = \{\texttt{O/1}, \texttt{O/2}\}$, $pt(\texttt{a1}) = \{\texttt{A/1}\}$, $pt(\texttt{a2}) = \{\texttt{A/2}\}$, $pt(\texttt{b}) = \{\texttt{B/1}\}$, *and* $pt(\texttt{c}) = \{\texttt{C/1}\}$. *By [OAG-NODE] and [OAG-DUMMYNODE],* $N = \{o_{root}, \texttt{A/1}, \texttt{A/2}, \texttt{B/1}, \texttt{C/1}, \texttt{O/1}, \texttt{O/2}\}$. *By [OAG-EDGE], we add* $\texttt{A/1} \rightarrow \texttt{B/1}$, $\texttt{A/2} \rightarrow \texttt{B/1}$ *and* $\texttt{B/1} \rightarrow \texttt{C/1}$, *since* `B/1` *is allocated in* `foo()` *with the receiver objects being* `A/1` *and* `A/2` *and* `C/1` *is allocated in* `bar()` *on the receiver object* `B/1`. *By [OAG-DUMMYEDGE], we add* $o_{root} \rightarrow \texttt{A/1}$, $o_{root} \rightarrow \texttt{A/2}$, $o_{root} \rightarrow \texttt{O/1}$ *and* $o_{root} \rightarrow \texttt{O/2}$. □

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 [Reach-Reflexive] and [Reach-Transitive], we speak of graph reachability in the standard manner. In [Confluence], $\curlyvee o_i$ identifies a conventional confluence point. In [Divergence], $o_i \prec o_t$ states that o_i is a divergence point, with at least two outgoing paths reaching o_t, implying that either o_t is a confluence point or at least one confluence point exists earlier on the two paths.

$$\frac{o_i \in N}{o_i \rightsquigarrow o_i}\ \text{[Reach-Reflexive]} \qquad \frac{o_i \rightarrow o_j \in E \quad o_j \rightsquigarrow o_k}{o_i \rightsquigarrow o_k}\ \text{[Reach-Transitive]}$$

$$\frac{o_j \rightarrow o_i \in E \quad o_k \rightarrow o_i \in E \quad o_j \neq o_k}{\curlyvee o_i}\ \text{[Confluence]}$$

$$\frac{o_i \rightarrow o_j \in E \quad o_i \rightarrow o_k \in E \quad o_j \neq o_k \quad o_j \rightsquigarrow o_t \quad o_k \rightsquigarrow o_t}{o_i \prec o_t}\ \text{[Divergence]}$$

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

Figure 3.8 gives the rules for computing two context selectors, *heapCtxSelector* and *mtdCtxSelector*, introduced in Section 2.2.2 and used in refining an object-sensitive points-to analysis. In $\mathit{heapCtxSelector}(c, o_i) = hc$, 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 $\mathit{mtdCtxSelector}(\_, \_, hc, o_i) = c$ (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-obj* [55, 75], both context selectors are simple. In the case of full-

$$\frac{o_t \in N \quad o_{root} \rightarrow o_i \in E \quad o_i \rightsquigarrow o_t}{o_i^t : \langle o_{root} \prec o_t,\ [\,]\rangle \quad \textbf{if } o_i = o_t \textbf{ then } \mathit{heapCtxSelector}([\,], o_i) = [\,]} \quad \text{[HCTX-INIT]}$$

$$\frac{o_j^t : \langle \mathit{rep}, \mathit{hc}\rangle \quad o_j \rightarrow o_i \in E \quad o_i \rightsquigarrow o_t \quad o_j \neq o_t}{o_i^t : \langle \mathit{rep}', \mathit{hc}'\rangle \quad \begin{cases} \mathit{rep}' = \textbf{true}, & \mathit{hc}' = \mathit{hc} & \textbf{if } \neg \mathit{rep} \wedge o_j \prec o_t \quad ① \\ \mathit{rep}' = o_j \prec o_t, & \mathit{hc}' = \mathit{hc} \mathbin{+\!\!+} o_j & \textbf{if } \mathit{rep} \wedge \curlyvee o_i \quad ② \\ \mathit{rep}' = \mathit{rep}, & \mathit{hc}' = \mathit{hc} & \textbf{otherwise} \end{cases} \atop \textbf{if } o_i = o_t \textbf{ then } \mathit{heapCtxSelector}(\mathit{hc} \mathbin{+\!\!+} o_j, o_i) = \mathit{hc}'} \quad \text{[HCTX-DIV]}$$

$$\frac{o_j^t : \langle \mathit{rep}, \mathit{hc}\rangle \quad o_j \rightarrow o_i \in E \quad o_i \rightsquigarrow o_t \quad o_j = o_t}{o_i^t : \langle \mathit{rep}', \mathit{hc}'\rangle \quad \begin{cases} \mathit{rep}' = \textbf{true}, & \mathit{hc}' = \mathit{hc} \mathbin{+\!\!+} o_j, & \textbf{if } \mathit{rep} \wedge \curlyvee o_i \quad ③ \\ \mathit{rep}' = \textbf{true}, & \mathit{hc}' = \mathit{hc}, & \textbf{otherwise} \end{cases} \atop \textbf{if } o_i = o_t \textbf{ then } \mathit{heapCtxSelector}(\mathit{hc} \mathbin{+\!\!+} o_j, o_i) = \mathit{hc}'} \quad \text{[HCTX-CYC]}$$

$$\frac{\mathit{heapCtxSelector}(\_, o_i) = \mathit{hc} \quad c = \mathit{hc} \mathbin{+\!\!+} o_i}{\mathit{mtdCtxSelector}(\_, \_, \mathit{hc}, o_i) = c} \quad \text{[MCTX]}$$

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*$(\_,\ \_,\ [o_1, ..., o_{n-1}],\ o_n) = [o_1, ..., o_n]$ for every path from o_{root} to a node o_n in the OAG, $o_{root} \rightarrow o_1 \rightarrow ... \rightarrow o_{n-1} \rightarrow o_n$. For a k-object-sensitive analysis with a $(k-1)$-context-sensitive heap, *heapCtxSelector*$([o_{n-k}, ..., o_{n-1}],\ o_n)$ $= [o_{n-k+1}, ..., o_{n-1}]$ and *mtdCtxSelector*$(\_,\ \_,\ [o_{n-k+1}, ..., o_{n-1}],\ o_n) = [o_{n-k+1}, ..., o_n]$. Essentially, a suffix of length of k is selected from $o_{root} \rightarrow o_1 \rightarrow ... \rightarrow o_{n-1} \rightarrow o_n$, 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_t, can be seen as the set of paths reaching o_t 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 → ③, ① → ④, ② → ③, ② → ④, and ⑤. The other six nodes are redundant with respect to o_t. 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_t.

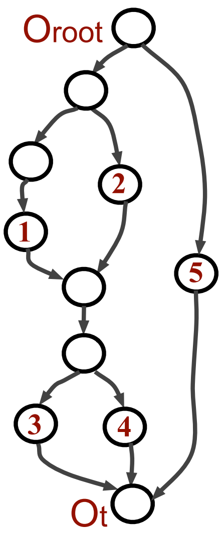


Figure 3.9: An OAG.

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_t. Meanwhile, each rule also records at o_i, which reaches o_t, a set of pairs of the form $o_i^t : \langle rep, hc\rangle$. For a pair $o_i^t : \langle rep, hc\rangle$, hc is a heap context of o_i that uniquely represents a particular path from o_{root} to o_i. In addition, rep is a boolean flag considered for determining the suitability of o_i as a representative node, i.e., context element for o_t *under* hc (i.e., for the path hc leading to o_i). There are two cases. If $rep =$ **false**, then o_i is redundant for o_t. If $rep =$ **true**, then o_i is potentially a representative node (i.e., context element) for o_t. $hc +\!\!+\, o$ returns the concatenation of hc and o.

Specifically, for the first three rules on heap contexts, [Hctx-Init] bootstraps heap context selection, [Hctx-Cyc] handles the special case when o_t is in a cycle such that $o_j = o_t$, and [Hctx-Div] handles the remaining cases. In [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_i appears on a divergent path from o_j leading to o_t, o_i^t'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: $\neg o_j \prec o_t$ and $o_j \prec o_t$. In both cases, o_j is in a branch (since o_j^t's *rep* is **true**) and o_i is a confluence node (since $\curlyvee o_i$ holds). Thus, o_j is included as a context element for o_t. In the case of $\neg o_j \prec o_t$, o_i is redundant for o_t under c. In the case of $o_j \prec o_t$, 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 $o_j \dashrightarrow o_k \dashrightarrow o_t$ and $o_j \dashrightarrow o_i \dashrightarrow o_k \dashrightarrow o_t$ 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_i^t'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_i^t'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 $hc \mathbin{+\!\!+} o_i$ k-bounded and every heap context $hc \mathbin{+\!\!+} o_j$ $(k-1)$-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_i^t, with $t = 4$, only. By* [Hctx-Init]*, we obtain $o_1^4 : (\textbf{\textit{true}}, [\,])$ and $o_2^4 : (\textbf{\textit{true}}, [\,])$. By* [Hctx-Div]*, we obtain $o_3^4 : (\textbf{\textit{false}}, [\texttt{o1}])$, $o_3^4 : (\textbf{\textit{false}}, [\texttt{o2}])$, $o_4^4 : (\textbf{\textit{false}}, [\texttt{o1}])$ and $o_4^4 : (\textbf{\textit{false}}, [\texttt{o2}])$. Thus, $heapCtxSelector([\texttt{o1,o3}], o4) = [\texttt{o1}]$ and*

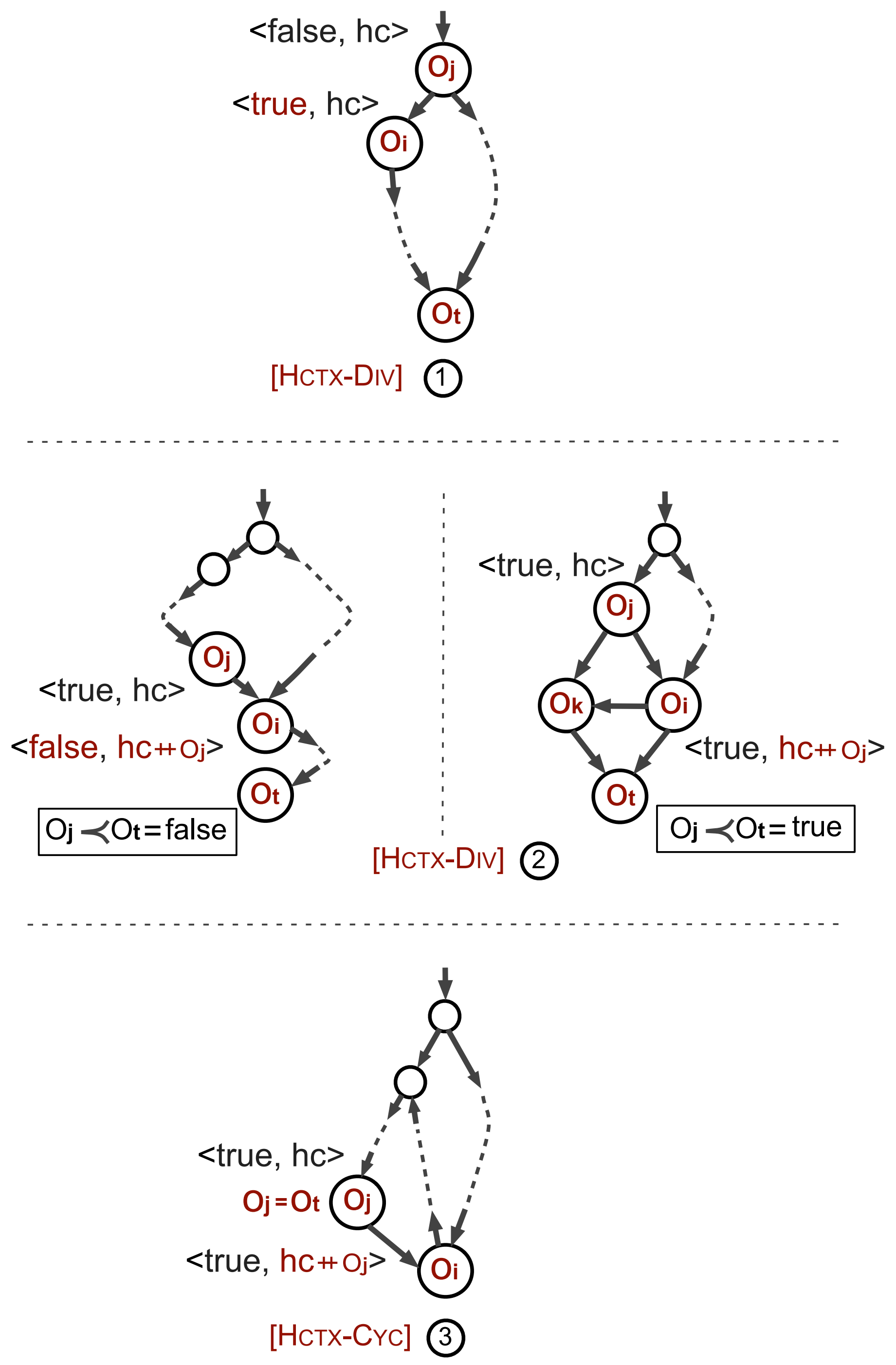


Figure 3.10: Three Cases marked for [Hctx-Div] and [Hctx-Cyc] in Figure 3.8.

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* o_4 *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* o_4 *and the two* o_4 *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}.

[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_i is a confluence node. By adding o_j to hc in $hc \mathbin{+\!\!+} o_j$, we ensure that for each path p from o_i's corresponding divergence node to o_i 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_{root} to o_t are traversed, all divergence and confluence nodes are handled. The third case simply propagates the recorded information across the edge $o_j \to o_i$.

[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_i^t'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

		2-cs	*2-obj*	*B-2-obj*	*S-2-obj*	*B-S-2-obj*
xalan	may-alias pairs	25,245,307	6,196,945	**5,146,694**	5,652,610	**3,958,998**
	may-fail casts	1154	711	**653**	608	**550**
	analysis time (secs)	1400	8653	11450	1150	1376
chart	may-alias pairs	43,124,320	4,189,805	**3,593,584**	3,485,082	**3,117,825**
	may-fail casts	2026	1064	**979**	923	**844**
	analysis time (secs)	3682	630	1322	1145	1814
eclipse	may-alias pairs	20,979,544	5,029,492	**4,617,883**	4,636,675	**4,346,306**
	may-fail casts	1096	722	**655**	615	**551**
	analysis time (secs)	1076	119	175	119	188
fop	may-alias pairs	38,496,078	10,548,491	**9,870,507**	9,613,363	**9,173,539**
	may-fail casts	1618	1198	**1133**	1038	**973**
	analysis time (secs)	3054	796	1478	961	1566
luindex	may-alias pairs	10,486,363	2,190,854	**1,949,134**	1,820,992	**1,705,415**
	may-fail casts	794	493	**438**	408	**353**
	analysis time (secs)	650	90	140	88	145
pmd	may-alias pairs	13,134,083	2,868,130	**2,598,100**	2,457,457	**2,328,304**
	may-fail casts	1216	845	**787**	756	**698**
	analysis time (secs)	816	131	191	132	193
antlr	may-alias pairs	16,445,862	5,082,371	**4,768,233**	4,586,707	**4,419,166**
	may-fail casts	995	610	**551**	525	**466**
	analysis time (secs)	808	109	162	105	163
lusearch	may-alias pairs	11,788,332	2,251,064	**2,010,780**	1,886,967	**1,771,280**
	may-fail casts	874	504	**450**	412	**358**
	analysis time (secs)	668	94	153	91	155
bloat	may-alias pairs	43,408,294	12,532,334	**11,608,822**	12,155,175	**11,374,583**
	may-fail casts	1944	1401	**1311**	1316	**1226**
	analysis time (secs)	10679	4508	4770	4460	4724

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.

average of 10.0%. In addition, *B-S-2-obj* is also more precise than *S-2-obj* for all the nine benchmarks, ranging from 3.7% for `antlr` to 30.0% for `xalan`, with an average of 8.8%. Note that the set of non-aliased variable pairs reported under *2-obj* (*S-2-obj*) is a strict subset of the set of non-aliased variable pairs reported under *B-2-obj* (*B-S-2-obj*), validating practically the validity of Theorem 2, i.e., the fact that BEAN is always no less precise than the object-sensitive analysis improved upon. Finally, *2-obj*, *S-2-obj*, *B-2-obj* and *B-S-2-obj* are all substantially more precise than *2-cs*, indicating the superiority of object-sensitivity over call-site-sensitivity.

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

Performance 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, BEAN may spend more time on processing more contexts introduced. *B-2-obj* increases the analysis cost of *2-obj* for all the nine benchmarks, ranging from 5.8% for `bloat` to 109.8% for `chart`, with an average of 54.8%. In addition, *B-S-2-obj* also increases the analysis cost of *S-2-obj* for all the nine benchmarks, ranging from 5.9% for `bloat` to 70.3% for `lusearch`, with an average of 49.1%.

Table 3.2 shows the pre-analysis times of 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 *B-2-obj*

Benchmark	xalan	chart	eclipse	fop	luindex	pmd	antlr	lusearch	bloat
CI	82.6	112.2	49.6	105.5	39.0	65.3	56.9	39.1	52.5
OAG	0.2	0.2	0.1	0.2	0.2	0.1	0.2	0.1	0.1
CTX-COMP	83.0	168.0	32.1	236.5	11.7	13.9	13.9	18.3	13.3
Total	165.8	280.4	81.8	342.2	50.9	79.3	71.0	57.5	65.9

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

```java
void main(String[] args) {
    HashSet xSet = new HashSet(); // HS/1
    xSet.add(new X()); // X/1
    Iterator xIter = xSet.iterator();
    Object v1 = xIter.next();
    X x = (X) v1;

    HashSet ySet = new HashSet(); // HS/2
    ySet.add(new Y()); // Y/1
    Iterator yIter = ySet.iterator();
    Object v2 = yIter.next();
    Y y = (Y) v2;
}
class HashSet ... {
    HashMap map = new HashMap(); // HM/1
    public boolean add(Object e) {
        return map.put(e, ...) == null;
    }
    public Iterator iterator() {
        return map.newKeyIterator();
    }
    ...
}
class HashMap ... {
    Entry[] table = new Entry[16]; // Entry[]/1
    public Object put(Object key, ...) { ...
        table[bucketIndex] = new Entry(key, ...); // Entry/1
        ...
    }
    static class Entry {
        final Object key;
        Entry(Object k, ...) {
            key = k;
        }
    }
    private final class KeyIterator ... {
        public Object next() { ...
            Entry e = table[index];
            return e.key;
        }
    }
    Iterator newKeyIterator() {
        return new KeyIterator(); // KeyIter/1
    }
    ...
}
```

Figure 3.11: A real-world application for using `java.util.HaseSet`.

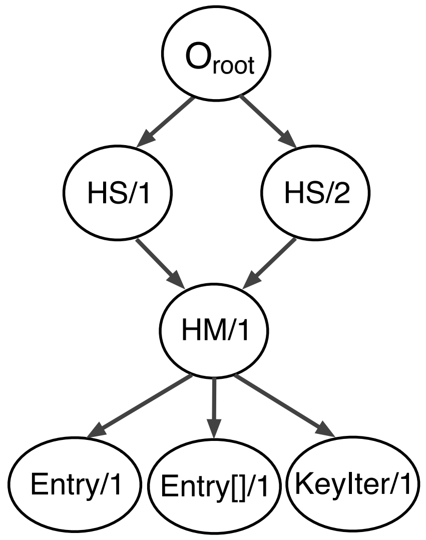


Figure 3.12: Part of OAG related to `HS/1` and `HS/2`.

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.

***2-obj*.** To answer queries Q1 and Q2, we need to know the points-to sets of `v1` and `v2` 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, `v1` and `v2` 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 `v1` (`v2`) to point to `X/1` (`Y/1`) only. For query Q1, `v1` and v2 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 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 *may-alias*, this 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*, causing often the underlying points-to analysis to be unscalable for large programs. For this reason, DOOP [22] and WALA [98], provide an option for all objects of a certain class, such as `java.lang.String` or `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 *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 [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, devirtualization 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.

```
 1 void main(String[] args) {
 2     A x = new A(); // $o_1^A$
 3     A y = new A(); // $o_2^A$
 4     A z = new A(); // $o_3^A$
 5     x.f = new B(); // $o_4^B$
 6     y.f = new C(); // $o_5^C$
 7     z.f = new C(); // $o_6^C$
 8     A a = z.f;
 9     a.foo();
10     C c = (C) a;
11 }
12 class A {
13     A f;
14     void foo() {...}
15 }
16 class B extends A {
17     void foo() {...}
18 }
19 class C extends A {
20     void foo() {...}
21 }
```

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_i^t 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_1^A, o_2^A and o_3^A, respectively. As `x.f`, `y.f` and `z.f` are not aliases, `a` points to o_6^C. 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 `pmd`, a program analyzed by (1) *3-obj* — a 3-object-sensitive points-to analysis [53] using the allocation-site abstraction, (2) *T-3-obj* — *3-obj* using the allocation-type abstraction, and (3) *M-3-obj* — *3-obj* using our Mahjong heap abstraction. For *3-obj*, `pmd` is analyzed in 14469.3 seconds, allowing 44004 call graph edges to be discovered. *T-3-obj* is the fastest (50.3 seconds), but is the most imprecise (50666 call graph edges). In contrast, *M-3-obj* is as precise as *3-obj* (44016 call graph edges) but is also nearly as fast as *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

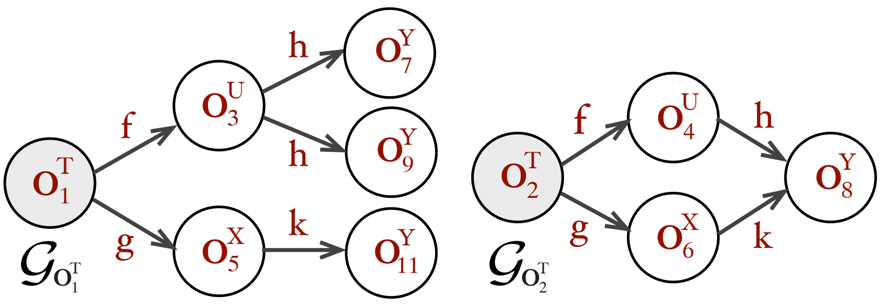


Figure 4.2: Field points-to graph rooted at o_1^T and o_2^T.

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 $\mathcal{G}_o = (\mathcal{H}, \mathcal{F}, \alpha, o, \mathcal{T}, \tau)$. $\mathcal{H}$ 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. $\mathcal{G}_{o_2^T} = (\mathcal{H}, \mathcal{F}, \alpha, o_2^T, \mathcal{T}, \tau)$. $\mathcal{H} = \{o_2^T, o_4^U, o_6^X, o_8^Y\}$; $\mathcal{F} = \{f, g, h, k\}$; $\alpha[o_2^T, f] = \{o_4^U\}$, $\alpha[o_4^U, h] = \{o_8^Y\}$, $\alpha[o_2^T, g] = \{o_6^X\}$, and $\alpha[o_6^X, k] = \{o_8^Y\}$; $\mathcal{T} = \{T, U, X, Y\}$; and $\tau[o_2^T] = T$, $\tau[o_4^U] = U$, $\tau[o_6^X] = X$, and $\tau[o_8^Y] = Y$. Similarly, $\mathcal{G}_{o_1^T}$ 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 $\bar{f} = f_1.f_2.\cdots.f_n$ be a sequence of field names. For the field points-to graph $\mathcal{G}_o$ rooted at an object o, we write $pts(o.\bar{f})$ 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, ..., f_n in $\mathcal{G}_o$ in that order. In Figure 4.2, $pts(o_1^T.f) = \{o_3^U\}$ and $pts(o_1^T.f.h) = \{o_7^Y, o_9^Y\}$.

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, $\bar{f} = f_1.f_2.\cdots.f_n$, the following two conditions hold:*

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

2. $\Big|\{\tau[o] \mid o \in pts(o_i.\bar{f})\}\Big| = 1$.

In Figure 4.2, o_1^T and o_2^T are type-consistent. For the objects reached from o_1^T and o_2^T, 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_1^A, o_2^A and o_3^A (Section 4.2.1). By Definition 1, o_2^A and o_3^A are type-consistent (as $o_2^A.f$ points to o_5^C and $o_3^A.f$ points to o_6^C) but o_1^A is not type-consistent with any (as $o_1^A.f$ points to o_4^B). After o_2^A and o_3^A are merged, `y.f` and `z.f` are regarded as aliases. Thus, `a` will point to not only o_6^C as before but also o_5^C spuriously. However, as o_5^C and o_6^C 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.* □

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.

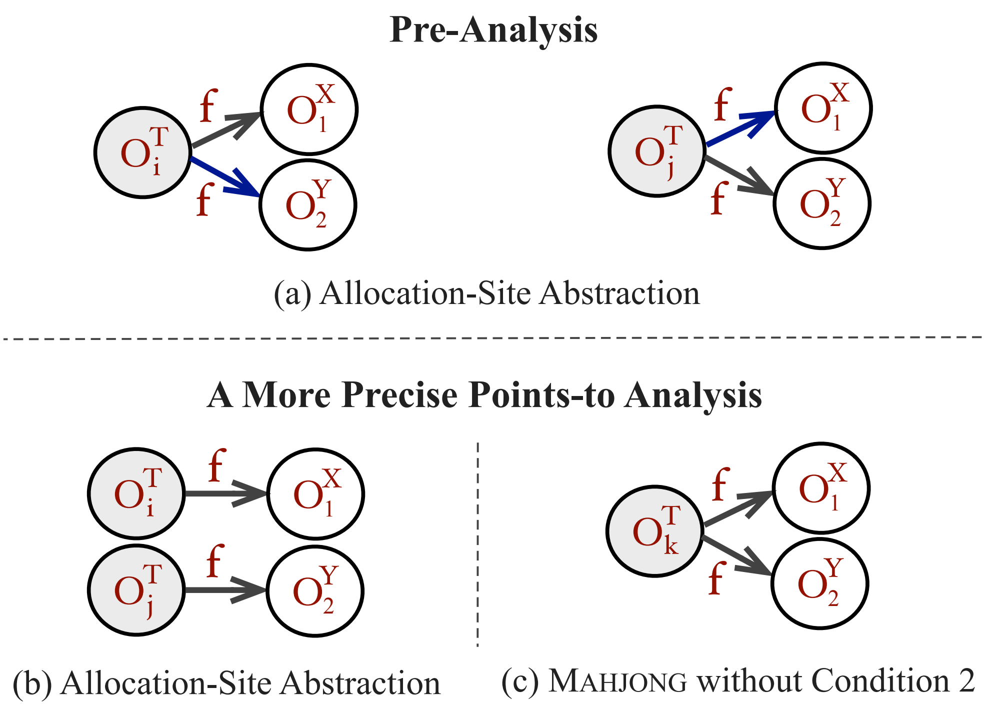

Pre-Analysis
O_i^T
f
f
O_1^X
O_2^Y
O_j^T
f
f
O_1^X
O_2^Y
(a) Allocation-Site Abstraction
A More Precise Points-to Analysis
O_i^T
f
O_1^X
O_j^T
f
O_2^Y
O_k^T
f
f
O_1^X
O_2^Y
(b) Allocation-Site Abstraction
(c) MAHJONG without Condition 2

Figure 4.3: Illustrating Condition 2 in Definition 1.

Example 6 *Suppose $o_i^T.f$ and $o_j^T.f$ 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, $\mathcal{A}$ (Figure 4.3(b)). If Condition 2 is ignored, o_i^T and o_j^T will become type-consistent according to the pre-analysis and thus merged into, say, o_k^T (represented by o_i^T or o_j^T). Running $\mathcal{A}$ with this new heap abstraction will result in precision loss, as $o_i^T.f$ and $o_j^T.f$ now point to objects of both types X and Y (Figure 4.3(c)).* □

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

Let $\mathbb{H}$ 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, $\mathbb{H} / \equiv$, 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, $\mathcal{G}_o = (\mathcal{H}, \mathcal{F}, \alpha, o, \mathcal{T}, \tau)$, to a 6-tuple sequential automaton $\mathcal{A}_o = (Q, \Sigma, \delta, q_o, \Gamma, \gamma)$ [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 $\gamma : Q \mapsto \Gamma$, where $\Gamma = \{\text{acc}, \text{non-acc}\}$.

Example 7 *Continuing from Example 4 (Figure 4.2), the automaton $\mathcal{A}_{o_2^T}$ for $\mathcal{G}_{o_2^T} = (\mathcal{H}, \mathcal{F}, \alpha, o_2^T, \mathcal{T}, \tau)$ is obtained according to Figure 4.4. Similarly, $\mathcal{A}_{o_1^T}$ is constructed.* □

The behavior of $\mathcal{A}_o$, which can be an NFA (consisting of multiple edges with the same label leaving a state), is:

$$\beta_{\mathcal{A}_o} : \Sigma^* \to \mathcal{P}(\Gamma)$$

If $\mathcal{A}_o$ reaches the states, $s_1, s_2, \cdots, s_n$, after having read an input w in Σ^*, then:

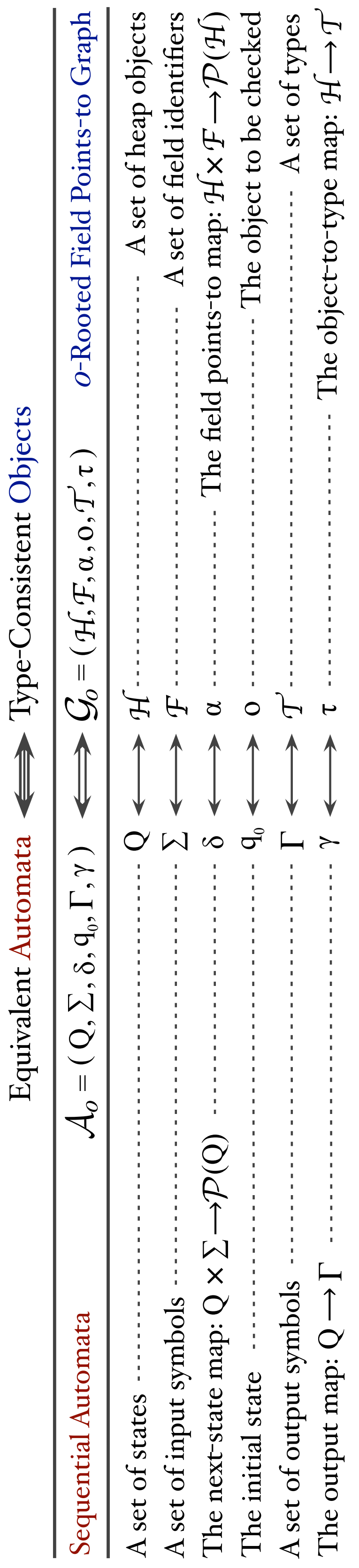

Equivalent Automata ⟺ Type-Consistent Objects
Sequential Automata
$\mathcal{A}_o = (Q, \Sigma, \delta, q_0, \Gamma, \gamma)$
$\mathcal{G}_o = (\mathcal{H}, \mathcal{F}, \alpha, o, \mathcal{T}, \tau)$
o-Rooted Field Points-to Graph
A set of states $Q \longleftrightarrow \mathcal{H}$ A set of heap objects
A set of input symbols $\Sigma \longleftrightarrow \mathcal{F}$ A set of field identifiers
The next-state map: $Q \times \Sigma \longrightarrow \mathcal{P}(Q)$ $\delta \longleftrightarrow \alpha$ The field points-to map: $\mathcal{H} \times \mathcal{F} \longrightarrow \mathcal{P}(\mathcal{H})$
The initial state $q_0 \longleftrightarrow o$ The object to be checked
A set of output symbols $\Gamma \longleftrightarrow \mathcal{T}$ A set of types
The output map: $Q \longrightarrow \Gamma$ $\gamma \longleftrightarrow \tau$ The object-to-type map: $\mathcal{H} \longrightarrow \mathcal{T}$

Figure 4.4: The mapping of a field points-to graph rooted at an object to a sequential automaton.

$$\beta_{\mathcal{A}_o}(w) = \bigcup_{i=1}^{n} \gamma[s_i]$$

Let o_1^T and o_2^T be two objects with the same type T. Let their automata $\mathcal{A}_{o_1^T}$ and $\mathcal{A}_{o_2^T}$ be built as shown in Figure 4.4. o_1^T and o_2^T are type-consistent if, for every input w in Σ^*, (1) $\beta_{\mathcal{A}_{o_1^T}}(w) = \beta_{\mathcal{A}_{o_2^T}}(w)$ (Condition 1 of Definition 1) and (2) $\left|\beta_{\mathcal{A}_{o_1^T}}(w)\right| = 1$ (Condition 2 of Definition 1).

Therefore, we have reduced the problem of checking the type-consistency of o_1^T and o_2^T to one of testing the equivalence of their corresponding automata $\mathcal{A}_{o_1^T}$ and $\mathcal{A}_{o_2^T}$, 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_{larger} is the set of states of the larger automaton.

Example 8 *Continuing from Example 7, we see easily that o_1^T and o_2^T are type-consistent (Figure 4.2) since their corresponding automata $\mathcal{A}_{o_1^T}$ and $\mathcal{A}_{o_2^T}$ are equivalent.* □

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,

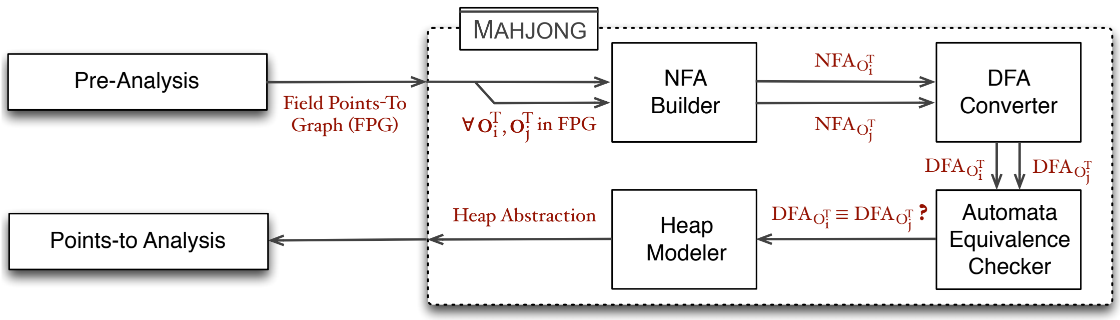


Figure 4.5: Overview of MAHJONG.

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 $\mathcal{G}_o$ rooted at o, and constructs a 6-tuple NFA $\mathcal{A}_o = (Q, \Sigma, \delta, q_0, \Gamma, \gamma)$ according to the mapping, as shown in Figure 4.4. In fact, $\mathcal{A}_o$ can be immediately read off from $\mathcal{G}_o$.

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 construction 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

The Automata Equivalence Checker tests the equivalence of two DFAs by applying a classic Hopcroft-Karp algorithm [30] with minor modifications in almost linear time.

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 $\mathbb{H} / \equiv$. For every equivalent class $[o_i^T] \in \mathbb{H} / \equiv$, a representative object o_j^T is arbitrarily picked to substitute for the other objects in the class. Essentially, the allocation sites for all objects in $[o_i^T]$ are merged and represented by the allocation site of o_j^T 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 = new T()` in a Java program, where o_j^T is a representative for $[o_i^T]$, `x` is made to point to o_j^T.

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-$\mathcal{A}$ from $\mathcal{A}$

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-$\mathcal{A}$ from $\mathcal{A}$ by first replacing $\mathcal{A}$'s allocation-site abstraction with the Mahjong heap abstraction. We then need to make minor modifications to $\mathcal{A}$ to enable M-$\mathcal{A}$ to handle merged objects effectively.

Regardless of whether $\mathcal{A}$ is call-site-, object- or type-sensitive, M-$\mathcal{A}$ 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 $o.f_1.f_2.\cdots.f_n$ for any $f_1.f_2.\cdots.f_n$ 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 $\mathcal{A}$ is k-call-site-sensitive [71], then M-$\mathcal{A}$ behaves identically as $\mathcal{A}$ in handling methods. For the reason mentioned above, M-$\mathcal{A}$ models the merged objects context-insensitively but everything else context-sensitively as in $\mathcal{A}$.

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

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 $[o_{i_{k-1}}, \dots, o_{i_1}]$ (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 $[o_{i_{k-1}}, \dots, o_{i_1}]$, then the k-object-sensitive calling context used for analyzing the callee of `x.foo()` is $[o_{i_{k-1}}, \dots, o_{i_1}, o_i]$.

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

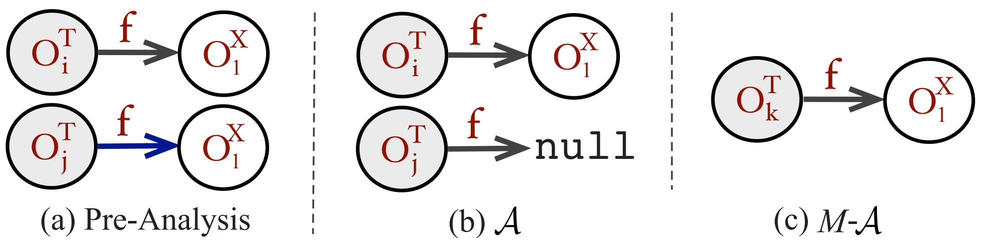
4.3.6.2 Soundness and Precision of M-$\mathcal{A}$ over $\mathcal{A}$

The soundness of M-$\mathcal{A}$ is easy to establish. If $\mathcal{A}$ is sound, then M-$\mathcal{A}$ 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-$\mathcal{A}$ 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 $o_i^T.f$ and $o_j^T.f$ both point to o_1^X during the pre-analysis (Figure 4.6(a)) but o_1^X and* **`null`***, respectively, in $\mathcal{A}$ (Figure 4.6(b)). In M-$\mathcal{A}$, o_i^T and o_j^T are type-consistent and thus merged into o_k^T (represented by either o_i^T or o_j^T), M-$\mathcal{A}$ is less precise, as $o_j^T.f$, which points to* **`null`** *in $\mathcal{A}$, now points to an object of type X (Figure 4.6(c)).* □



$O_i^T \xrightarrow{f} O_1^X$
$O_j^T \xrightarrow{f} O_1^X$
(a) Pre-Analysis
$O_i^T \xrightarrow{f} O_1^X$
$O_j^T \xrightarrow{f}$ null
(b) $\mathcal{A}$
$O_k^T \xrightarrow{f} O_1^X$
(c) M-$\mathcal{A}$

Figure 4.6: Illustrating *the null-field problem.*

If $\mathcal{A}$ is call-site-sensitive, M-$\mathcal{A}$ is equally precise as $\mathcal{A}$ for a type-dependent client if the null-field problem never occurs in a program analyzed by $\mathcal{A}$. Recall that the pre-analysis is no more precise than $\mathcal{A}$. 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 $\mathcal{A}$ and context-insensitively under M-$\mathcal{A}$, resulting in the same precision in both cases. In general, M-$\mathcal{A}$ is no more precise than $\mathcal{A}$ due to the null-field problem but very close to $\mathcal{A}$ as the null-fields are rare.

If $\mathcal{A}$ is object-sensitive, then M-$\mathcal{A}$ is no more precise than $\mathcal{A}$ for type-dependent clients, as some objects that are used in distinguishing different contexts in $\mathcal{A}$ are merged by Mahjong if they are type-consistent. However, this hardly hurts the precision, making M-$\mathcal{A}$ nearly as precise as $\mathcal{A}$ 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::foo()` by analyzing it under multiple calling contexts, one per receiver object. By merging the type-consistent receiver objects for `T::foo()`, we end up achieving a significant performance benefit at little precision loss by analyzing `T::foo()` 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 $[o] = \{o_1, \dots, o_n\} \in \mathbb{H} \,/\, \equiv$ (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-$\mathcal{A}$, $o_1, \dots, o_n$ 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, $\mathsf{FPG} = (\mathsf{N}, \mathsf{E})$, which is a directed weighted graph, as input. A node $o_i \in \mathsf{N} = \mathbb{H}$ represents a heap object in the program. An edge $(o_i, f, o_j) \in \mathsf{E} \subseteq \mathsf{N} \times \mathbb{F} \times \mathsf{N}$ indicates that $o_i.f$ points to o_j. We assume that the FPG contains a dummy node $o_{\texttt{null}}$ to represent `null`. If $o_i.f = \texttt{null}$, then $(o_i, f, o_{\texttt{null}}) \in \mathsf{E}$. We also assume $(o_{\texttt{null}}, f, o_{\texttt{null}}) \in \mathsf{E}$ for every field $f \in \mathbb{F}$.

The output of MAHJONG is a new heap abstraction, represented by a merged object map, $\mathsf{MOM} \subseteq \mathbb{H} \to \mathbb{H}$, 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, UNION and FIND. $S.$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 $S_x \cup S_y$. $S.$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 TYPEOF : $\mathbb{H} \to \mathbb{T}$ returns the type of a given object and a special type for $o_{\texttt{null}}$.

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

lines 8 – 12. In lines 6 – 7, the function SINGLETYPE-CHECK : $\mathbb{H}\times$FPG $\rightarrow$ {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 Figure 4.4.

Algorithm 2: Nfa-Builder

```
Input  :  o                (Input object)
          FPG = (N, E)     (Field Points-to Graph)
Output:   NFA = $(Q, \Sigma, \delta, q_0, \Gamma, \gamma)$
1  $q_0 = o$
2  Let $Q$ be a set of objects reachable from o in 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      $\Sigma = \Sigma \cup \textsc{FieldsOf}(o_i)$
7      $\Gamma = \Gamma \cup \{ \textsc{TypeOf}(o_i) \}$
8      $\gamma[o_i] = \textsc{TypeOf}(o_i)$
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 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 FPG (line 2). The objects in Q are iterated over to build Σ (set of input symbols), Γ (set of output symbols), and γ (output map) at lines 5 – 8. The function $\textsc{FieldsOf} : \mathbb{H} \to \mathcal{P}(\mathbb{F})$ returns the fields of a given object. Finally, the relevant edges in FPG are traversed to build the state transition map δ (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) ϵ-transitions. Second, we can find the next states of a DFA state q more efficiently. In the general

Algorithm 3: DFA-CONVERTER

```
   Input  :  NFA = $(Q, \Sigma, \delta, q_0, \Gamma, \gamma)$
   Output:  DFA = $(Q', \Sigma', \delta', q'_0, \Gamma', \gamma')$
 1 $q'_0 = \{q_0\}$
 2 $\Sigma' = \Sigma$
 3 Let $Q'$ and $\Gamma'$ be two new sets
 4 Let $\delta'$ and $\gamma'$ be two new maps
 5 Add $q'_0$ as an unmarked state to $Q'$
 6 while there is an unmarked state $q \in Q'$ do
 7     Mark $q$
 8     Pick any $o_i$ from $q$
 9     foreach $f \in$ FIELDSOF($o_i$) do
10         $q' = \{ \delta[o_j, f] \mid o_j \in q \}$
11         if $q' \notin Q'$ then
12             Add $q'$ as an unmarked state to $Q'$
13         $\delta'[q, f] = q'$
14 foreach $q \in Q'$ do
15     $\gamma'[q] = \{ \text{TYPEOF}(o_i) \mid o_i \in q \}$
16     $\Gamma' = \Gamma' \cup \gamma'[q]$
17 return DFA = $(Q', \Sigma', \delta', q'_0, \Gamma', \gamma')$
```

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 Γ' (set of output symbols) and γ' (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  :  $\mathsf{DFA}_1 = (Q_1, \Sigma_1, \delta_1, q_1, \Gamma_1, \gamma_1)$
          $\mathsf{DFA}_2 = (Q_2, \Sigma_2, \delta_2, q_2, \Gamma_2, \gamma_2)$
Output:  TRUE or FALSE  (Are $\mathsf{DFA}_1$ and $\mathsf{DFA}_2$ equivalent?)
1  $Q = Q_1 \cup Q_2$
2  $\Sigma = \Sigma_1 \cup \Sigma_2$
3  $\delta[q,f] = \begin{cases} \delta_1[q,f] & \textbf{if } q \in Q_1 \\ \delta_2[q,f] & \textbf{if } q \in Q_2 \end{cases}$
4  $\Gamma = \Gamma_1 \cup \Gamma_2$
5  $\gamma[q] = \begin{cases} \gamma_1[q] & \textbf{if } q \in Q_1 \\ \gamma_2[q] & \textbf{if } q \in Q_2 \end{cases}$
6  $DFA = (Q, \Sigma, \delta, q_1, \Gamma, \gamma)$
7  Let $V$ be a new set
8  foreach $q \in Q$ do
9      Add $\{q\}$ to $V$
10 $V$.UNION($q_1$, $q_2$)
11 Push $(q_1, q_2)$ to a new stack, $STACK$
12 while $STACK$ is not empty do
13     Pop $(p_1, p_2)$ from $STACK$
14     foreach $f \in \Sigma$ do
15         $r_1$ =V.FIND($\delta[p_1, f]$), $r_2$ =V.FIND($\delta[p_2, f]$)
16         if $r_1 \neq r_2$ then
17             $V$.UNION($r_1$, $r_2$)
18             Push $(r_1, r_2)$ to $STACK$
19 return $\begin{cases} \text{TRUE} & \textbf{if } \forall s \in V : \forall p, q \in s : \gamma[p] = \gamma[q] \\ \text{FALSE} & \textbf{otherwise} \end{cases}$
```

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 δ 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_{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, *union by rank* and *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%.

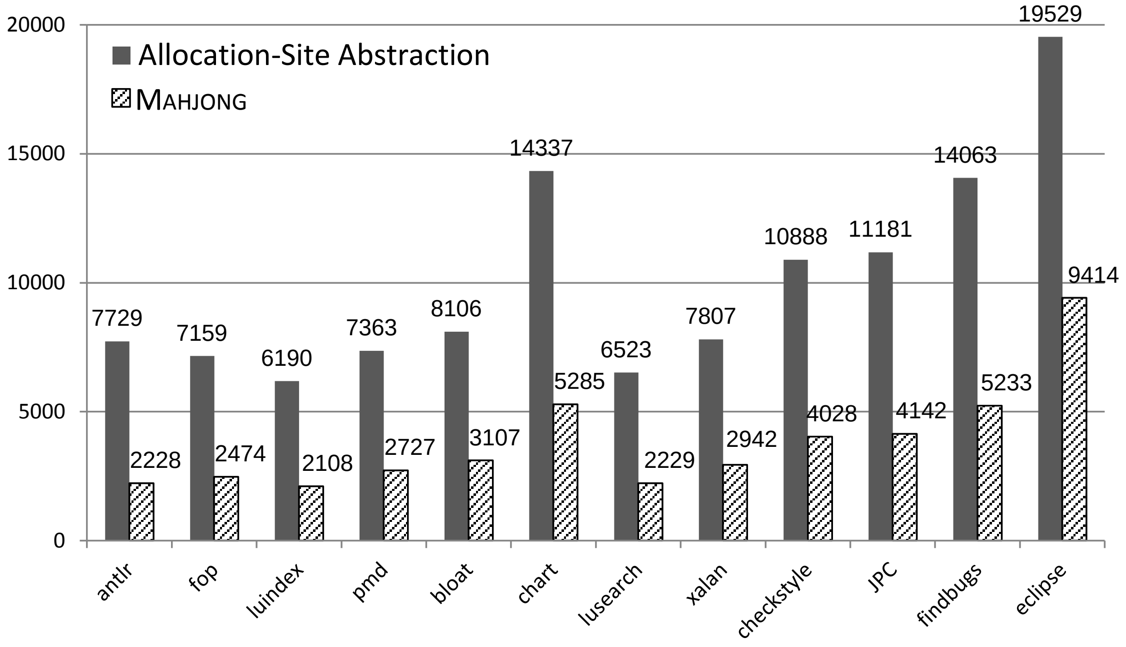


Figure 4.7: Number of abstract objects created by the allocation-site abstraction and Mahjong.

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.

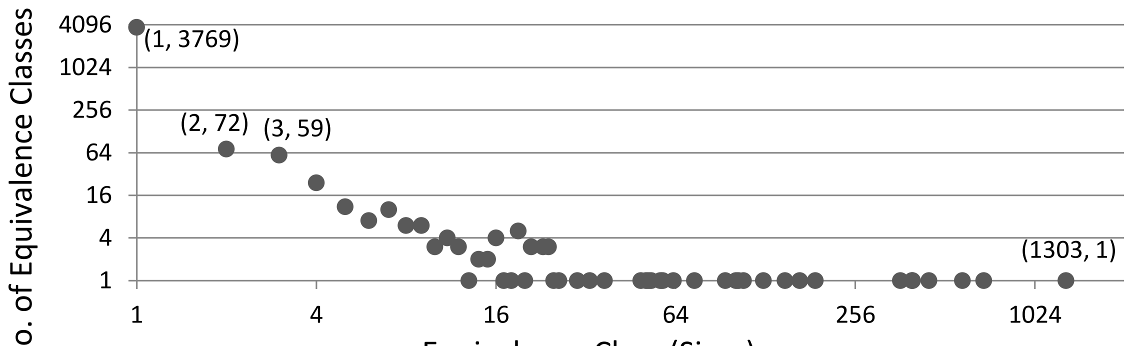


Figure 4.8: Object merging in `checkstyle`.

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 left-most point marked by (1, 3769), for example, there are 3769 equivalence classes containing one object each. Thus, neither object is merged with any other objects.

Rank	Type	Equiv. Class Size	Total No. of Objects	Remarks
1	java.lang.StringBuilder	1303	1303	char[]
2	java.lang.Object[]	690	1353	String
12	antlr.ASTPair	108	109	DetailAST
55	java.lang.Object[]	12	1353	Integer
65	java.lang.Object[]	9	1353	QName
260	antlr.ASTPair	1	109	`null`

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-$\mathcal{A}$ over k-$\mathcal{A}$ 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-$\mathcal{A}$), 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-$\mathcal{A}$ is scalable, M-k-$\mathcal{A}$ is also scalable and faster than k-$\mathcal{A}$.

Mahjong is highly effective in boosting performance. For the programs where both k-$\mathcal{A}$ and M-k-$\mathcal{A}$ 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-$\mathcal{A}$ scales whenever M-k-$\mathcal{A}$ scales. However, M-k-$\mathcal{A}$ is faster than k-$\mathcal{A}$, 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

Program	Pre-analysis	Metrics	2-cs	M-2-cs	2-type	M-2-type	3-type	M-3-type	2-obj	M-2-obj	3-obj	M-3-obj
		analysis time (sec.)	2790.7	373.6	63.6	45.5	459.3	61.0	116.2	36.7	8302.0	69.9
	ci: 44.1s	speedup	7.5X		1.4X		7.5X		3.2X		118.8X	
antlr	FPG: 1.3s	#may-fail casts	888	888	648	649	599	600	524	524	463	463
	MAHJONG: 1.3s	#poly call sites	1862	1862	1682	1685	1651	1654	1630	1633	1623	1626
		#call graph edges	55153	55153	51427	51435	51168	51176	51062	51070	51035	51043
		analysis time (sec.)	1510.3	430.5	66.1	46.6	526.9	67.8	73.8	36.7	8647.0	70.0
	ci: 34.7s	speedup	3.5X		1.4X		7.8X		2.0X		123.5X	
fop	FPG: 0.7s	#may-fail casts	682	682	527	517	479	469	428	428	375	375
	MAHJONG: 1.1s	#poly call sites	1068	1068	872	875	841	844	821	824	814	817
		#call graph edges	38154	38154	34580	34588	34321	34329	34211	34219	34184	34192
		analysis time (sec.)	1480.2	301.9	45.4	30.1	526.4	42.8	72.9	28.0	10651.9	63.1
	ci: 26.2s	speedup	4.9X		1.5X		12.3X		2.6X		168.8X	
luindex	FPG: 0.8s	#may-fail casts	701	701	522	513	473	464	413	413	358	358
	MAHJONG: 1.1s	#poly call sites	1157	1157	981	984	946	949	922	925	915	918
		#call graph edges	37445	37445	33760	33769	33496	33505	33383	33392	33356	33365
		analysis time (sec.)	2099.4	547.6	92.2	62.2	906.1	82.9	145.1	82.3	14469.3	127.7
	ci: 44.8s	speedup	3.8X		1.5X		10.9X		1.8X		113.3X	
pmd	FPG: 1.4s	#may-fail casts	1319	1319	1082	1072	1014	1004	930	930	871	871
	MAHJONG: 1.5s	#poly call sites	1424	1424	1210	1213	1175	1179	1137	1140	1130	1133
		#call graph edges	49731	49734	44768	44779	44419	44433	44070	44081	44004	44016
		analysis time (sec.)	7769.3	5350.9	87.2	67.3	533.6	124.5	3611.9	3501.5	>5h	>5h
	ci: 37.7s	speedup	1.5X		1.3X		4.3X		1.03X		–	
bloat	FPG: 2.4s	#may-fail casts	1840	1840	1614	1608	1521	1515	1302	1302	–	–
	MAHJONG: 1.9s	#poly call sites	2005	2005	1811	1814	1673	1676	1567	1571	–	–
		#call graph edges	64102	64102	57619	57625	57136	57142	56364	56374	–	–
		analysis time (sec.)	5476.2	1665.9	174.0	86.8	2967.8	518.5	997.9	279.8	>5h	>5h
	ci: 89.6s	speedup	3.3X		2.0X		5.7X		3.6X		–	
chart	FPG: 2.3s	#may-fail casts	2093	2093	1708	1699	1621	1612	1349	1349	–	–
	MAHJONG: 4.0s	#poly call sites	2475	2475	2093	2096	2036	2039	2017	2020	–	–
		#call graph edges	81224	81238	72968	72974	72321	72327	72297	72317	–	–

checkstyle	ci: 66.6s	analysis time (sec.)	7644.8	3186.7	187.8	92.3	5120.6	379.8	1946.6	277.1	>5h	3103.7
	FPG: 3.0s	speedup	2.4X		2.0X		13.5X		7.0X		∞	
	Mahjong: 3.1s	#may-fail casts	1596	1601	1345	1334	1243	1231	1135	1140	–	1022
		#poly call sites	2558	2558	2307	2311	2239	2243	2211	2215	–	2168
		#call graph edges	75802	75822	67390	67419	66550	66572	66718	66751	–	65943
xalan	ci: 38.7s	analysis time (sec.)	1996.1	464.4	99.0	57.7	1122.5	101.8	1816.8	247.3	>5h	1274.9
	FPG: 1.2s	speedup	4.3X		1.7X		11.0X		7.3X		∞	
	Mahjong: 1.7s	#may-fail casts	982	982	794	784	740	730	589	589	–	535
		#poly call sites	1879	1879	1651	1654	1620	1623	1595	1598	–	1591
		#call graph edges	50825	50825	46399	46407	46139	46147	45974	45982	–	45950
lusearch	ci: 41.4s	analysis time (sec.)	1444.7	309.4	46.4	29.6	780.9	44.5	110.2	27.8	>5h	65.0
	FPG: 0.8s	speedup	4.7X		1.6X		17.5X		4.0X		∞	
	Mahjong: 1.0s	#may-fail casts	779	779	561	552	514	505	424	424	–	372
		#poly call sites	1361	1361	1178	1181	1147	1150	1120	1123	–	1116
		#call graph edges	40724	40724	36631	36640	36372	36381	36255	36264	–	36237
JPC	ci: 58.9s	analysis time (sec.)	3464.1	1155.1	147.1	90.6	1509.8	340.5	477.2	306.0	>5h	5056.8
	FPG: 2.1s	speedup	3.0X		1.6X		4.4X		1.6X		∞	
	Mahjong: 4.5s	#may-fail casts	1828	1828	1595	1579	1507	1490	1381	1381	–	1226
		#poly call sites	4749	4749	4379	4382	4321	4324	4275	4279	–	4139
		#call graph edges	90111	90111	81723	81729	81251	81251	81031	81045	–	79370
findbugs	ci: 90.6s	analysis time (sec.)	14923.8	5646.6	1229.3	107.4	>5h	171.7	>5h	174.2	>5h	524.1
	FPG: 4.6s	speedup	2.6X		11.4X		∞		∞		∞	
	Mahjong: 3.2s	#may-fail casts	2923	2928	2469	2458	–	2143	–	2074	–	1671
		#poly call sites	4136	4136	3753	3756	–	3574	–	3565	–	3534
		#call graph edges	100046	100063	89036	89054	–	87581	–	87929	–	86985
eclipse	ci: 174.1s	analysis time (sec.)	>5h	>5h	2453.0	863.1	>5h	11316.5	>5h	15738.0	>5h	>5h
	FPG: 15.5s	speedup	–		2.8X		∞		∞		–	
	Mahjong: 21.4s	#may-fail casts	–	–	4236	4223	–	3994	–	3662	–	–
		#poly call sites	–	–	9906	9910	–	9740	–	9724	–	–
		#call graph edges	–	–	163760	163768	–	161448	–	162137	–	–

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-$\mathcal{A}$/k-$\mathcal{A}$*. 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.

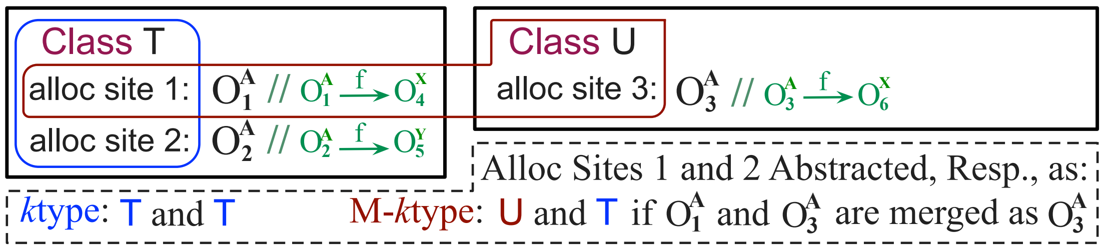

Class T
alloc site 1: O_1^A // $O_1^A \xrightarrow{f} O_4^X$
alloc site 2: O_2^A // $O_2^A \xrightarrow{f} O_5^Y$
Class U
alloc site 3: O_3^A // $O_3^A \xrightarrow{f} O_6^X$
Alloc Sites 1 and 2 Abstracted, Resp., as:
ktype: T and T
M-ktype: U and T if O_1^A and O_3^A are merged as O_3^A

Figure 4.9: Precision gains of *M-k-type* over *k-type*.

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 ≡ 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

[61, 39, 35, 102, 75, 88].

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 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

[1] J. Adamek and V. Trnkova. *Automata and Algebras in Categories.* Kluwer Academic Publishers, Norwell, MA, USA, 1990.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley, Boston, MA, USA, 2006.

[3] K. Ali and O. Lhoták. Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP '12, pages 688–712, 2012.

[4] N. Allen, B. Scholz, and P. Krishnan. Staged points-to analysis for large code bases. In *Proceedings of the 24th International Conference on Compiler Construction*, CC '15, pages 131–150, 2015.

[5] L. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

[7] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '03, pages 103–114, 2003.

[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, 2006.

[9] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Selective control-flow abstraction via jumping. In *Proceedings of the 30th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, pages 163–182, 2015.

[10] S. Blackshear, A. Gendreau, and B.-Y. E. Chang. Droidel: A general approach to Android framework modeling. SOAP '15, pages 19–25, 2015.

[11] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10, pages 5–14, 2010.

[12] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP '07, pages 525–549, 2007.

[13] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, 2011.

[14] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, pages 243–262, 2009.

[15] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 1–19, 1999.

[16] Chord. A program analysis platform for Java. `http://www.cis.upenn.edu/~mhnaik/chord.html`.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[18] J. Da Silva and J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 416–425, 2006.

[19] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 260–278, 2001.

[20] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, 1995.

[21] J. Dietrich, N. Hollingum, and B. Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, pages 535–551, 2015.

[22] DOOP. A sophisticated framework for Java pointer analysis. http://doop.program-analysis.org.

[23] Y. Feng, X. Wang, I. Dillig, and T. Dillig. Bottom-up context-sensitive pointer analysis for Java. In *Proceedings of the 13rd Asian Conference on Programming Languages and Systems*, APLAS '15, pages 465–484, 2015.

[24] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 133–144, 2006.

[25] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[26] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. Points-to analysis for program understanding. *Journal of Systems and Software*, 44(3):213 – 227, 1999.

[27] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS '15, 2015.

[28] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, 2011.

[29] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, 2001.

[30] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, Cornell University, 1971.

[31] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA '15, pages 106–117, 2015.

[32] V. Kanvar and U. P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2):29:1–29:47, 2016.

[33] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in java points-to analysis. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC '13, pages 41–60, 2013.

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

gramming Language Design and Implementation, PLDI '13, pages 423–434, 2013.

[35] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, 2007.

[36] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, 2006.

[37] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, 2011.

[38] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the 12nd International Conference on Compiler Construction*, CC '03, pages 153–169, 2003.

[39] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Proceedings of the 15th International Conference on Compiler Construction*, CC '06, pages 47–64, 2006.

[40] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, 2008.

[41] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *Proceedings of the 28th European Conference on Object-Oriented Programming*, ECOOP '14, pages 27–53, 2014.

[42] Y. Li, T. Tan, and J. Xue. Effective soundness-guided reflection analysis. In *Proceedings of the 22nd International Symposium on Static Analysis*, SAS '15, pages 162–180, 2015.

[43] Y. Li, T. Tan, Y. Zhang, and J. Xue. Program Tailoring: Slicing by Sequential Criteria. In *Proceedings of 30th European Conference on Object-Oriented Programming*, ECOOP '16, pages 15:1–15:27, 2016.

[44] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '99, pages 199–215, 1999.

[45] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 590–601, 2011.

[46] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 411–427, 2010.

[47] B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, USENIX Security '05, pages 271–286, 2005.

[48] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *Proceedings of the 22nd International Conference on Compiler Construction*, CC '13, pages 61–81, 2013.

[49] M. Madsen and A. Møller. Sparse dataflow analysis with pointers and reachability. In *Proceedings of the 21st International Symposium on Static Analysis*, SAS '14, pages 201–218, 2014.

[50] R. Mangal, X. Zhang, A. V. Nori, and M. Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, pages 462–473, 2015.

[51] A. Marino. *Analysis and Enumeration: Algorithms for Biological Graphs.* Atlantis Publishing Corporation, 2015.

[52] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 305–315, 2010.

[53] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 1–11, 2002.

[54] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graph construction in the presence of function pointers. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, pages 155–162, 2002.

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

[56] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '08, pages 347–366, 2008.

[57] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, 2006.

[58] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, 2006.

[59] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 386–396, 2009.

[60] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010.

[61] E. M. Nystrom, H.-S. Kim, and W.-m. W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 43–48, 2004.

[62] R. W. O'Callahan. *Generalized Aliasing As a Basis for Program Analysis Tools.* PhD thesis, 2001.

[63] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '14, pages 475–484, 2014.

[64] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective x-sensitive analysis guided by impact pre-analysis. *ACM Trans. Program. Lang. Syst.*, 38(2):6:1–6:45, 2015.

[65] R. C. Read and R. E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

[66] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 77–90, 1999.

[67] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12nd International Conference on Compiler Construction*, CC '03, pages 126–137, 2003.

[68] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 12–23, 2001.

[69] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274, 2012.

[70] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[71] O. G. Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, 1991.

[72] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[73] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis. Set-based pre-processing for points-to analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '13, pages 253–270, 2013.

[74] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer. More sound static handling of Java reflection. In *Proceedings of the 13rd Asian Conference on Programming Languages and Systems*, APLAS '15, pages 485–503, 2015.

[75] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, 2011.

[76] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, 2014.

[77] Soot. A framework for analyzing and transforming Java and Android applications. `https://sable.github.io/soot/`.

[78] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *Proceedings of 30th European Conference on Object-Oriented Programming*, ECOOP '16, pages 22:1–22:26, 2016.

[79] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, 2006.

[80] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Aliasing in object-oriented programming. chapter Alias Analysis for Object-oriented Programs, pages 196–232. Springer-Verlag, 2013.

[81] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122, 2007.

[82] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, 2005.

[83] Y. Su, D. Ye, and J. Xue. Accelerating inclusion-based pointer analysis on heterogeneous cpu-gpu systems. In *20th Annual International Conference on High Performance Computing*, HiPC '13, pages 149–158, 2013.

[84] Y. Su, D. Ye, and J. Xue. Parallel pointer analysis with CFL-reachability. In *Proceedings of the 43rd International Conference on Parallel Processing*, ICPP '14, pages 451–460, 2014.

[85] Y. Su, D. Ye, J. Xue, and X. Liao. An efficient gpu implementation of inclusion-based pointer analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):353–366, 2016.

[86] Y. Sui, P. Di, and J. Xue. Sparse flow-sensitive pointer analysis for multi-threaded programs. In *Proceedings of the 14th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '16, pages 160–170, 2016.

[87] Y. Sui, X. Fan, H. Zhou, and J. Xue. Loop-oriented array- and field-sensitive pointer analysis for automatic simd vectorization. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES '2016, pages 41–51, 2016.

[88] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the 11st Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, pages 1–11, 2013.

[89] Y. Sui and J. Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 460–473, 2016.

[90] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA '12, pages 254–264, 2012.

[91] Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Trans. Softw. Eng.*, 40(2), 2014.

[92] Y. Sui, S. Ye, J. Xue, and P. Yew. SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, APLAS '11, pages 155–171, 2011.

[93] Y. Sui, S. Ye, J. Xue, and J. Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Software: Practice and Experience*, 44(12):1485–1510, 2014.

[94] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 264–280, 2000.

[95] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *Proceedings of the 23rd International Symposium on Static Analysis*, SAS '16, pages 489–510, 2016.

[96] T. Tan, Y. Li, and J. Xue. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, 2017.

[97] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 35–46, 2001.

[98] WALA. T.J. Watson libraries for analysis. `http://wala.sf.net`.

[99] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, 2004.

[100] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, 1999.

[101] X. Xiao and C. Zhang. Geometric encoding: Forging the high performance context sensitive points-to analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 188–198, 2011.

[102] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 225–236, 2008.

[103] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *Proceedings of 12nd Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 154:154–154:164, 2014.

[104] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. In *Proceedings of the 21st International Symposium on Static Analysis*, SAS '14, pages 319–336, 2014.

[105] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 218–229, 2010.

[106] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang. On abstraction refinement for program analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 239–248, 2014.

[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.