Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata

Tian Tan* Yue Li * Jingling Xue

School of Computer Science and Engineering, UNSW, Australia

Abstract

Mainstream points-to analysis techniques for object-oriented languages rely predominantly on the allocation-site abstraction to model heap objects. We present MAHJONG, a novel heap abstraction that is specifically developed to address the needs of an important class of type-dependent clients, such as call graph construction, devirtualization and may-fail casting. 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.

MAHJONG is simple conceptually, efficient, and drops easily on any allocation-site-based points-to analysis. We demonstrate its effectiveness by discussing some insights on why it is a better alternative of the allocation-site abstraction for type-dependent clients and evaluating it extensively on 12 large real-world Java programs with five context-sensitive points-to analyses and three widely used type-dependent clients. MAHJONG is expected to provide significant benefits for many program analyses where call graphs are required.

CCS Concepts • **Theory of computation** → **Program analysis**

Keywords points-to analysis, heap abstraction

1. Introduction

Pointer Analyses should be designed to be appropriate in cost and precision for specific groups of client problems. We do not need a different pointer analysis per client problem, but rather we should look for classes of client problems with similar needs.

— Barbara Ryder [17]

Every points-to analysis, especially for object-oriented languages such as Java and C#, requires a heap abstraction for partitioning the infinitely-sized heap into a finite number of (abstract) objects. For object-oriented programs, context-sensitivity is important for achieving useful precision. Due to many years of research, context-sensitivity can be achieved by three main approaches with different efficiency and precision tradeoffs: call-site-sensitivity [15, 22, 36, 42, 51, 53], object-sensitivity [29, 40, 48] and type-sensitivity [39].

However, little progress has been made on developing heap abstractions for points-to analysis. Mainstream points-to analysis frameworks for Java, such as CHORD [10], DOOP [14], SOOT [49] and WALA [50], 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 [19, 38, 44]. The suitability of the allocation-site abstraction as a 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, WALA [50] and DOOP [14], 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 paper, 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 [18] with minor

* These authors contributed equally to this work

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062360

modifications. MAHJONG is simple conceptually and drops easily 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 [3, 5, 7, 16, 26, 31, 32, 43, 54, 55].

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 [20, 22, 39, 40, 42]. Take, *3obj*, a 3-object-sensitive points-to analysis [29], the most precise one used in our evaluation, as an example. For the four programs that can be analyzed scalably under *3obj*, our MAHJONG-based *3obj* runs 131X faster, on average, while achieving nearly the same precision for all the three clients. For the remaining eight, where *3obj* is unscalable in 5 hours each, our MAHJONG-based *3obj* can analyze five of them in an average of 33.42 minutes.

In summary, our paper 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 stand-alone open-source tool. MAHJONG is simple (with only 1500 LOC of Java in total) and drops easily on any allocation-site-based points-to analysis.
- We conduct extensive experiments to evaluate the effectiveness of MAHJONG in practice.

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 paper, 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.

```
1 A x = new A(); // o_1^A
2 A y = new A(); // o_2^A
3 A z = new A(); // o_3^A
4 x.f = new B(); // o_4^B
5 y.f = new C(); // o_5^C
6 z.f = new C(); // o_6^C
7 A a = z.f;
8 a.foo();
9 C c = (C) a;

10 class A {
11   A f;
12   void foo() {...}
13 }
14 class B extends A {
15   void foo() {...}
16 }
17 class C extends A {
18   void foo() {...}
19 }
```

Figure 1. An example program illustrating object merging.

In Section 2.1, we see that blindly merging objects of the same type is ineffective. In Section 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.

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 [19, 27, 38, 51].

Example 2.1. Consider Figure 1, where o_i^t represents the abstract object of type t created at the allocation site at line i. We will use this notation in the rest of the paper.

For the three type-dependent clients, call graph construction, devirtualization and may-fail casting, only lines 8 – 9 are relevant. According to an allocation-site-based Andersen's points-to analysis [4], 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 8 is a mono-call and can thus be devirtualized, and in addition, the cast (C) at line 9 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 poly-call and thus non-devirtualizable. In addition, the cast (C) is no longer considered safe. □

Consider `pmd`, a program analyzed by (1) *3obj*—a 3-object-sensitive points-to analysis [29] using the allocation-site abstraction, (2) *T-3obj*—*3obj* using the allocation-type abstraction, and (3) *M-3obj*—*3obj* using the MAHJONG heap abstraction introduced in this paper. For *3obj*, `pmd` is analyzed in 14469.3 seconds, allowing 44004 call graph edges to be discovered. *T-3obj* is the fastest (50.3 seconds), but is the most imprecise (50666 call graph edges). In contrast, *M-3obj* is as precise as *3obj* (44016 call graph edges) but is also nearly as fast as *T-3obj* (127.7 seconds).

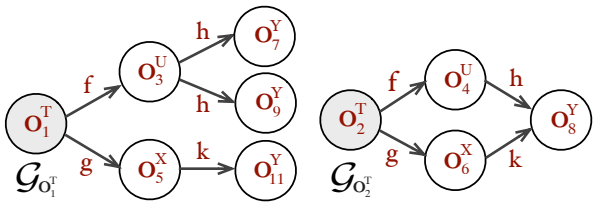

Figure 2. Field points-to graph rooted at o_1^T and o_2^T.

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 [4] 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 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 2.2.2).

2.2.1 Defining Type-Consistent Objects

After the pre-analysis, the field points-to graph (FPG) is available, representing the points-to information for the object fields. To facilitate a subsequent reduction of the problem of checking type-consistency as one of testing the equivalence of automata, we introduce the field points-to graph rooted at an object o as $\mathcal{G}_o = (\mathcal{H}, \mathcal{F}, \alpha, o, \mathcal{T}, \tau)$. $\mathcal{H}$ 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 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 1.

Example 2.2. Consider o_2^T first in Figure 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 $\bar{f} = f_1.f_2.\cdots.f_n$, where $n > 0$, 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 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 one single type.

Definition 2.1 (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 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. $\left|\{\tau[o] \mid o \in pts(o_i.\bar{f})\}\right| = 1$.

In Figure 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 2.3. Let us return to Figure 1, for which the allocation-type abstraction will merge o_1^A, o_2^A and o_3^A (Section 2.1). By Definition 2.1, 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 8 and may-fail casting at line 9 will not be affected. □

Let us examine Definition 2.1. 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. Enforcing Condition 2 maximally avoids precision loss, as shown below.

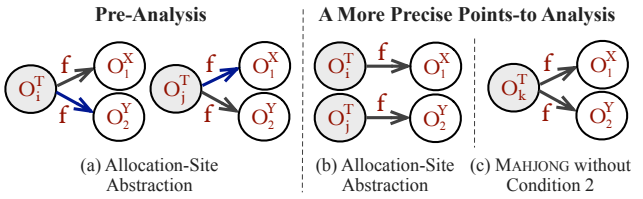

Figure 3. Illustrating Condition 2 in Definition 2.1.

Example 2.4. 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 3(a)) but o_1^X and o_2^Y, respectively, in a more precise allocation-site-based points-to analysis, $\mathcal{A}$ (Figure 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 abstraction will result in precision loss, as $o_i^T.f$ and $o_j^T.f$ now point to objects of types X and Y (Figure 3(c)). □

In Definition 2.1, the type-consistency relation $\equiv$ is an equivalence relation. It is straightforward to verify that $\equiv$ is reflexive, symmetric and transitive.

Let $\mathbb{H}$ be the set of all abstract objects in the program.

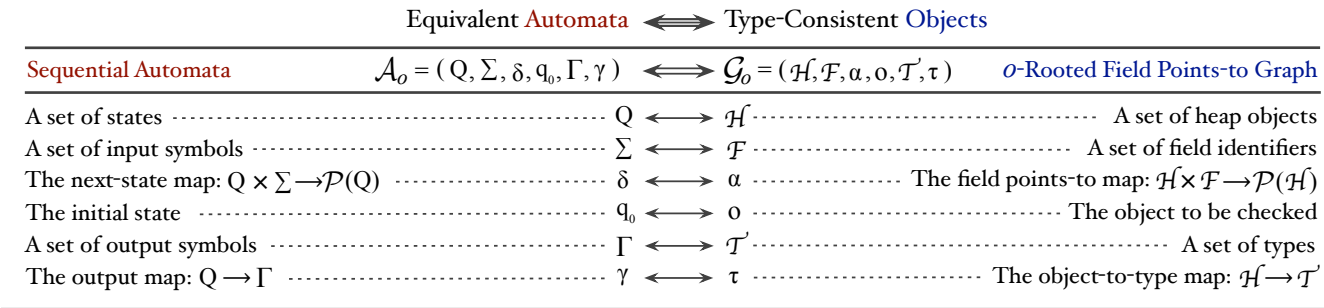

Figure 4. The mapping of a field points-to graph rooted at an object to a sequential automaton.

Definition 2.2 (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 2.1, especially in the presence of cycles, may be exponential in terms of the number of edges traversed [28, 34], causing the pre-analysis to be too inefficient or even unscalable. We describe a fast and elegant solution below.

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 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 2.5. Continuing from Example 2.2 (Figure 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. 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 $\mathcal{A}_o$ finally reaches the states, $s_1, s_2, \cdots, s_n$, after having read an input w in Σ^*, then $\beta_{\mathcal{A}_o}(w) = \cup_{i=1}^{n} \gamma[s_i]$.

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. 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 2.1) and (2) $|\beta_{\mathcal{A}_{o_1^T}}(w)| = 1$ (Condition 2 of Definition 2.1).

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 by the Hopcroft-Karp algorithm [18] with minor modifications. The worst-case time complexity is $O(|\Sigma| \times |Q_{\text{larger}}|)$, which is almost linear in terms of $|Q_{\text{larger}}|$, where Q_{larger} is the set of states of the larger automaton [18].

Example 2.6. Continuing from Example 2.5, we see easily that o_1^T and o_2^T are type-consistent (Figure 2) since their corresponding automata $\mathcal{A}_{o_1^T}$ and $\mathcal{A}_{o_2^T}$ are equivalent. □

3. MAHJONG

We first give an overview of MAHJONG that consists of four components (Section 3.1). We then describe each component in detail (Sections 3.2 – 3.5). Finally, we discuss MAHJONG-based points-to analysis (Section 3.6).

3.1 Overview

As shown in Figure 5, MAHJONG takes the field points-to graph (FPG) computed by a pre-analysis (Section 2.2.1) as input and builds a heap abstraction (Definition 2.2) to be used by a subsequent points-to analysis. The pre-analysis is fast but imprecise, by using Andersen's algorithm [4] with the allocation-site abstraction, context-insensitively. The subsequent points-to analysis will be more precise, usually performed context-sensitively, especially for object-oriented programs, based on the MAHJONG heap abstraction.

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

The detailed algorithms are given in Section 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 $\mathcal{A}_o =$

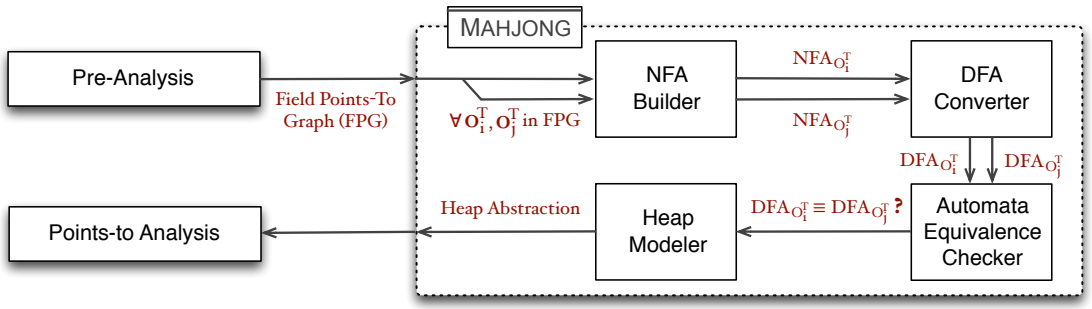

Figure 5. Overview of MAHJONG.

$(Q, \Sigma, \delta, q_0, \Gamma, \gamma)$ according to the mapping, as shown in Figure 4. In fact, $\mathcal{A}_o$ can be immediately read off from $\mathcal{G}_o$.

3.3 The DFA Converter

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.

3.4 The Automata Equivalence Checker

The Automata Equivalence Checker tests the equivalence of two DFAs by applying a classic Hopcroft-Karp algorithm [18] with minor modifications in almost linear time.

3.5 The Heap Modeler

After all type-consistent objects have been found, the type-consistency equivalence relation $\equiv$ given in Definition 2.1 becomes fully constructed. By Definition 2.2, 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.

3.6 MAHJONG-based Points-To Analysis

Let $\mathcal{A}$ be an allocation-site-based points-to analysis, which is either call-site-sensitive [15, 22, 36, 42, 51], object-sensitive [29, 40, 48] or type-sensitive [39]. We first discuss how to obtain *M-*$\mathcal{A}$, a MAHJONG-based points-to analysis, from $\mathcal{A}$ (Section 3.6.1). We then discuss briefly the soundness and precision of *M-*$\mathcal{A}$ relative to $\mathcal{A}$ for type-dependent clients.

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 otherwise 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 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 [37] 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 [20, 39, 48].

If $\mathcal{A}$ is k-call-site-sensitive [37], 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 [29]. 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 $[o_{i_{k-1}}, \ldots, o_{i_1}]$ (of length $k-1$), where i_j is the allocation site for the receiver object o_{i_j} of the method that contains i_{j-1} (with $i_0 = i$). If x points to an object o_i modeled under a context $[o_{i_{k-1}}, \ldots, o_{i_1}]$, then the k-object-sensitive calling context used for analyzing a callee of a method call $x.foo()$ is $[o_{i_{k-1}}, \ldots, o_{i_1}, o_i]$.

If $\mathcal{A}$ is a k-object-sensitive points-to analysis, *M-*$\mathcal{A}$ 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 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 [39].

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.

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 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 6, due to the imprecision of the pre-analysis, causing precision loss for all the three types of MAHJONG-based context-sensitivity.

Example 3.1. Suppose $o_i^T.f$ and $o_j^T.f$ both point to o_1^X during the pre-analysis (Figure 6(a)) but o_1^X and `null`, respectively, in $\mathcal{A}$ (Figure 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 6(c)). □

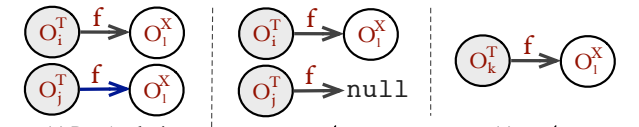

Figure 6. Illustrating *the null-field problem*.

If $\mathcal{A}$ is call-site-sensitive, *M-*$\mathcal{A}$ is as 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 2.1, 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 heap 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 [29] is to distinguish the side-effects of different receiver objects of an instance method $foo()$ by analyzing it under multiple calling contexts, one per receiver object. By merging a set of type-consistent receiver objects for $foo()$, we end up achieving a significant performance benefit at little precision loss by analyzing $foo()$ 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 $[o] = \{o_1, \ldots, o_n\} \in \mathbb{H} / \equiv$ (Definition 2.2) formed by the MAHJONG heap abstraction. In $\mathcal{A}$, every o_i that is used as a context element in a calling context is replaced by the class type that contains the allocation site for o_i. In *M-*$\mathcal{A}$, $o_1, \ldots, o_n$ are merged and replaced by the class type that contains the allocation site for a representative in $[o]$. Thus, the MAHJONG heap abstraction can be coarser than the allocation-site abstraction for some methods and finer for some others in partitioning their calling contexts, which depends on the representatives chosen.

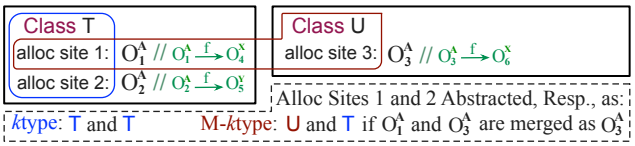

Figure 7. Precision of *M-ktype* over *ktype*.

Let us see how the choice of representative for an equivalence class affects the precision of *M-ktype*.

Example 3.2. In Figure 7, *ktype* (k-type-sensitive analysis) will represent the allocation sites 1 and 2 by `T`. Thus, the two allocation sites that are distinguished by *kobj* (k-object-sensitive analysis) 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-ktype* will be able to distinguish the allocation sites 1 and 2 by `U` and `T`, respectively. However, if o_1^A is selected as the representative (not shown in Figure 7), then *M-ktype* will merge the allocation sites 1, 2 and 3 by using `T` as the context, and become less precise than *ktype*.

However, the choice of representative for an equivalence class $[o] = \{o_1, \ldots, o_n\} \in \mathbb{H} / \equiv$ does not affect the soundness of *M-ktype*. Regardless of what object is selected, replacing o_i in a context used in the corresponding *kobj* by the containing type of a representative in $[o]$ in *M-ktype* always yields a context abstraction that is either identical or coarser, by the definition of type-sensitivity [39].

4. Algorithms

We present the algorithms used in MAHJONG. In Section 4.1, we give some domains used and then the main algorithm. In Sections 4.2 – 4.5, we describe the algorithms of its four components introduced in Sections 3.2 – 3.5.

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 used $\mathbb{H}$ earlier in Definition 2.2.

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 $o_i.f$ = `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, $\mathsf{MOM} \subseteq \mathbb{H} \rightarrow \mathbb{H}$, which relates an object in an equivalence class in $\mathbb{H} / \equiv$ to its representative object (as described in Section 3.5).

Algorithm 1: MAHJONG

```
   Input : FPG    (Field Points-to Graph)
   Output: MOM    (Merged Object Map)
1  Let W be a new set
2  foreach o ∈ H do
3      Add {o} to W
4  foreach o_i, o_j ∈ H s.t. W.FIND(o_i) ≠ 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 ∈ H do
16     MOM[o] = W.FIND(o)
17 return MOM
```

Algorithm 1 gives the main algorithm. To facilitate merging type-consistent objects, we make use of the concept of disjoint sets [11]. 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 the 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 resulting 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 : $\mathbb{H} \rightarrow \mathbb{T}$ 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 2.1 efficiently, we handle its two conditions separately, with Condition 2 in lines 6 – 7 and Condition 1 in lines 8 – 12. In lines 6 – 7, the function SINGLETYPE-CHECK : $\mathbb{H} \times \mathsf{FPG} \rightarrow \{\text{TRUE}, \text{FALSE}\}$ 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 4.2), convert the NFAs to their equivalent DFAs (Section 4.3), and finally, test their equivalence (Section 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 4.5).

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.

Algorithm 2: NFA-BUILDER

```
   Input : o                          (Input object)
           FPG = (N, E)               (Field Points-to Graph)
   Output: NFA = (Q, Σ, δ, q_0, Γ, γ)
1  q_0 = o
2  Let Q be a set of objects reachable from o in FPG
3  Let Σ and Γ be two new sets
4  Let γ and δ be two new maps
5  foreach o_i ∈ Q do
6      Σ = Σ ∪ FIELDSOF(o_i)
7      Γ = Γ ∪ { TYPEOF(o_i) }
8      γ[o_i] = TYPEOF(o_i)
9  foreach (o_i, f, o_j) ∈ E do
10     if o_i ∈ Q then
11         Add o_j to δ[o_i, f]
12 return NFA = (Q, Σ, δ, q_0, Γ, γ)
```

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 FIELDSOF : $\mathbb{H} \rightarrow \mathcal{P}(\mathbb{F})$ 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).

4.3 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. 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 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,

Algorithm 3: DFA-CONVERTER

Input : $\mathsf{NFA} = (Q, \Sigma, \delta, q_0, \Gamma, \gamma)$
Output: $\mathsf{DFA} = (Q', \Sigma', \delta', q_0', \Gamma', \gamma')$

1 $q_0' = \{q_0\}$
2 $\Sigma' = \Sigma$
3 Let Q' and Γ' be two new sets
4 Let δ' and γ' 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** $\mathsf{DFA} = (Q', \Sigma', \delta', q_0', \Gamma', \gamma')$

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 [18] with minor modifications at line 19 on testing whether all states in $s \in V$ have the same type. As discussed in Section 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}.

4.5 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 2.2, as discussed in Section 3.5.

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

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 DFA_1 and 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** $\begin{cases} \text{TRUE} & \textbf{if } \forall s \in V : \forall p, q \in s : \gamma[p] = \gamma[q] \\ \text{FALSE} & \textbf{otherwise} \end{cases}$

to work with mainstream allocation-site-based points-to analysis frameworks such as CHORD [10], WALA [50], SOOT [49] and DOOP [14]. To demonstrate its effectiveness, we have integrated MAHJONG with DOOP [9, 14], a state-of-the-art whole-program points-to analysis framework for Java. MAHJONG is released as an open-source tool at `http://www.cse.unsw.edu.au/~corg/mahjong`. Below we discuss three major optimizations.

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* [11]. As a result, the average execution time of each UNION/FIND operation over a disjoint-set forest can be reduced to nearly $O(1)$ [11].

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 beforehand and concurrently read only.

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?

Type-Dependent Clients We consider three representative type-dependent clients, call graph construction, devirtualization and may-fail casting, provided by DOOP [14].

Context-Sensitive Points-to Analyses We consider five context-sensitive points-to analyses also from DOOP as baselines. These cover the three main types of mainstream context-sensitivity: call-site-sensitivity [15, 22, 36, 42, 51], object-sensitivity [29, 40, 48] and type-sensitivity [39]. We also provide experimental evidence on why context-insensitivity is inadequate for type-dependent clients.

Benchmarks We consider 12 large Java programs including 3 popular applications `findbugs`, `checkstyle` and `JPC` and all standard DaCapo benchmarks [12] 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 the efficiency and precision of points-to analysis [24, 25, 38], we adopt the same resolution results generated by a dynamic reflection analysis tool, TAMIFLEX [8], in both the five baselines and their corresponding MAHJONG-based points-to analyses.

Computing Platform We have done our experiments on a Xeon E5-1620 3.7GHz machine with 128GB of RAM. The analysis time of a 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 [14]. Different pair-wise type-consistency tests are performed in parallel, as discussed in Section 5, with 8 threads on 4 cores.

Table 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.

6.1 RQ1: MAHJONG's Effectiveness as a Pre-Analysis

6.1.1 Efficiency

The overall pre-analysis phase is fast, as shown in Column 2 of Table 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, a FPG consists of 10073 objects of 1559 types with 2411 fields. MAHJONG builds an NFA for each object in the FPG, with its size measured in terms of its number of states. The average sizes of NFAs range from 356 in `luindex` to 3789 in `eclipse`, with an average of 992. For each program, the smallest NFA always has one state only. Across all the programs, the sizes of their largest NFAs range from 1935 in `luindex` to 10034 in `eclipse`. This costs MAHJONG only an average of 3.8 seconds for each program. Such good performance is due to both our design (by merging objects in terms of merging equivalent automata) and several effective optimizations performed (see Section 5).

6.1.2 Heap Partitioning

Figure 8 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%.

Let us examine `checkstyle` in detail. As shown in Figure 8, a total of 10888 objects are created by the allocation-site abstraction but only 4028 objects by MAHJONG.

Given the heap partitioned as $\mathbb{H} \ / \equiv$ for `checkstyle`, Figure 9 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.

Let us examine some equivalence classes, given in Table 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

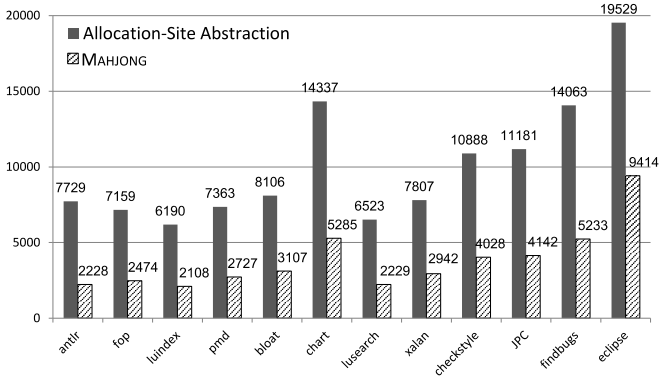

Figure 8. Number of abstract objects created by the allocation-site abstraction and MAHJONG.

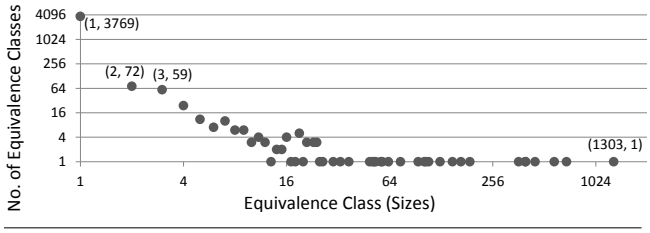

Figure 9. Object merging in `checkstyle`.

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 1. Some equivalence classes in `checkstyle`.

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 9.

For some other types like `Object[]` (Rows 2, 4 and 5), blindly merging all its objects would be imprecise (Section 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 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).

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 [20–22, 39, 40, 42, 48]. 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.

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 [14] are selected as baselines: *2cs* (2-call-site-sensitive), *2obj* (2-object-sensitive), *3obj* (3-object-sensitive), *2type* (2-type-sensitive), and *3type* (3-type-sensitive). In principle, *2cs* is not compatible with the others, $3\mathcal{A}$ is no less precise than $2\mathcal{A}$, and *kobj* is no less precise than *ktype*. As for $1\mathcal{A}$, it has been demonstrated that its precision is significantly less than that of $k\mathcal{A}$, where $k > 1$ [20, 39]. As a result, $1\mathcal{A}$ is not used in the recent points-to analysis literature [15, 40, 48] and is thus omitted in our baselines.

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 [20, 22, 39, 40, 48]. 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.

6.2.2 Efficiency and Precision

Table 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 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

Program	Pre-analysis	Metrics	2cs	M-2cs	2type	M-2type	3type	M-3type	2obj	M-2obj	3obj	M-3obj
antlr	ci: 44.1s	analysis time (sec.)	2790.7	373.6	63.6	45.5	459.3	61.0	116.2	36.7	8302.0	69.9
	FPG: 1.3s	speedup	7.5X		1.4X		7.5X		3.2X		118.8X	
	MAHJONG: 1.3s	#may-fail casts	888	888	648	649	599	600	524	524	463	463
		#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
fop	ci: 34.7s	analysis time (sec.)	1510.3	430.5	66.1	46.6	526.9	67.8	73.8	36.7	8647.0	70.0
	FPG: 0.7s	speedup	3.5X		1.4X		7.8X		2.0X		123.5X	
	MAHJONG: 1.1s	#may-fail casts	682	682	527	517	479	469	428	428	375	375
		#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
luindex	ci: 26.2s	analysis time (sec.)	1480.2	301.9	45.4	30.1	526.4	42.8	72.9	28.0	10651.9	63.1
	FPG: 0.8s	speedup	4.9X		1.5X		12.3X		2.6X		168.8X	
	MAHJONG: 1.1s	#may-fail casts	701	701	522	513	473	464	413	413	358	358
		#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
pmd	ci: 44.8s	analysis time (sec.)	2099.4	547.6	92.2	62.2	906.1	82.9	145.1	82.3	14469.3	127.7
	FPG: 1.4s	speedup	3.8X		1.5X		10.9X		1.8X		113.3X	
	MAHJONG: 1.5s	#may-fail casts	1319	1319	1082	1072	1014	1004	930	930	871	871
		#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
bloat	ci: 37.7s	analysis time (sec.)	7769.3	5350.9	87.2	67.3	533.6	124.5	3611.9	3501.5	>5h	>5h
	FPG: 2.4s	speedup	1.5X		1.3X		4.3X		1.03X		–	
	MAHJONG: 1.9s	#may-fail casts	1840	1840	1614	1608	1521	1515	1302	1302	–	–
		#poly call sites	2005	2005	1811	1814	1673	1676	1567	1571	–	–
		#call graph edges	64102	64102	57619	57625	57136	57142	56364	56374	–	–
chart	ci: 89.6s	analysis time (sec.)	5476.2	1665.9	174.0	86.8	2967.8	518.5	997.9	279.8	>5h	>5h
	FPG: 2.3s	speedup	3.3X		2.0X		5.7X		3.6X		–	
	MAHJONG: 4.0s	#may-fail casts	2093	2093	1708	1699	1621	1612	1349	1349	–	–
		#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 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.

"ci" in Table 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 *M-k*$\mathcal{A}$ is scalable, a speedup over $k\mathcal{A}$ is obtained.

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-2obj*/*2obj* for `bloat` to 168.8X by *M-3obj*/*3obj* for `luindex`). Table 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-3obj*/*3obj*. For every program in the bottom six, MAHJONG enables using a more precise points-to analysis that is not scalable if the allocation-site abstraction is used instead.

Preserved Precision For every program, as shown in Table 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 3.6 and validated here.

Call-Site-Sensitivity *M-2cs* is no more precise than *2cs* in principle (Section 3.6) but nearly as precise in practice. For devirtualization, M-*2cs* is equally as precise as *2cs*. For may-fail casting, M-*2cs* is negligibly worse than *2cs* (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-*2cs* 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-kobj* is also no more precise than *kobj* in principle (Section 3.6) but nearly as precise in practice. For call graph construction, devirtualization and may-fail casting, *M-2obj* experiences a small loss of precision of 0.02%, 0.23% and 0.04% over *2obj*, respectively, on average. For *M-3obj* over *3obj*, these percentages are 0.02%, 0.29% and 0.00%, respectively. For may-fail casting, *M-2obj* is on a par with *2obj* if `checkstyle` is ignored, and *M-3obj* is equally as precise as *3obj*.

Type-Sensitivity *M-ktype* may lose or gain precision compared with *ktype*, as discussed in Section 3.6. For may-fail casting, *M-ktype* is slightly more precise than *ktype* in all the programs except `antlr`. The average precision gains for M-*2type*/*2type* and M-*3type*/*3type* are 0.91% and 1.11%, respectively. For the other two clients, *M-ktype* is slightly less precise than *ktype* in every program. For call graph construction and devirtualization, *M-2type* experiences a small loss of precision of 0.02% and 0.18% over *2type*, respectively. In the case of *M-3type*/*3type*, these percentages are 0.02% and 0.22%, respectively.

Importance of Context-Sensitivity Context-sensitivity is significant for improving the precision of type-dependent clients, measured by #may-fail casts, #poly call sites and #call graph edges, in Table 2. Without context-sensitivity, #may-fail casts, #poly call sites and #call graph edges will be 2027, 3122 and 75162, respectively, on average, across all the programs. With context-sensitivity (by using the most precise MAHJONG-based points-to analysis for each program, e.g., *M-3obj* for `antlr` and *M-2obj* for `chart`), these numbers become substantially smaller: 1101, 2530 and 63994. This demonstrates convincingly the necessity of embracing context-sensitivity even for type-dependent clients.

6.2.3 Discussion

We discuss two observations about some results in Table 2.

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

Why does *M-3obj*/*3obj* deliver significantly better speedups than *M-2obj*/*2obj*? By using one extra level of context elements than *2obj*, *3obj* 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 *3obj*, *M-3obj* can drastically reduce the number of contexts used and thus accelerate the analysis. Consider `luindex`, where the speedup achieved by *M-3obj*/3obj is the highest obtained. The number of context-sensitive points-to relations produced under *2obj* is 9,255,034 but grows to 191,160,483 under *3obj*, which are both reduced significantly to 4,256,310 under *M-3obj*.

Unscalability of* MAHJONG*-based Points-to Analyses As shown in Table 2, *M-2cs* is unscalable for `eclipse` and *M-3obj* is unscalable for `bloat`, `chart` and `eclipse`. Why is *M-3obj* scalable for some large programs such as `findbugs` but unscalable for some small ones such as `bloat`? As shown in Figure 8, MAHJONG creates 5233 objects for `findbugs` but only 3107 objects for `bloat`.

M-3obj is unscalable for `bloat` possibly due to its object structure used. Some methods are both invoked on many (abstract) receiver objects and allocate many objects. Thus, the number of contexts becomes extremely large. To alleviate this problem, one solution is to use a coarser relation than $\equiv$ given in Definition 2.1 so that more objects can be merged together. Another solution is to apply *3obj* only selectively to parts of the program when moving from *2obj* to *3obj*.

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 [22, 23, 38, 41, 44]. There are three main flavors: call-site-sensitivity, object-sensitivity, and type-sensitivity.

Call-site-sensitivity [15, 22, 36, 42, 51], i.e., k-CFA [37] is often used to analyze C programs [6, 33, 45, 46, 52]. To better exploit the object-oriented features in Java, object-sensitivity is proposed [29, 30], yielding significantly higher precision at usually less cost [15, 20, 22, 48]. However, for large Java programs, object-sensitivity is often unscalable despite its good precision. To trade precision for efficiency, type-sensitivity is thus introduced [39].

For type-dependent clients, MAHJONG represents a better alternative than the allocation-site abstraction for the three types of context-sensitivity. This benefit is expected to generalize to other variations of context-sensitivity [20, 48].

There are other ways to improve the efficiency of points-to analysis. In [40], 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.

Heap Abstraction There are mainly two types of models in static analysis: store-based, e.g., the allocation-site abstraction and storeless, e.g., access paths [19]. The former is usually adopted in points-to analysis and the latter in alias analysis [38]. We focus on store-based models for Java here.

Due to its good precision, the allocation-site abstraction is adopted by (whole-program) points-to analysis techniques in the literature [20, 21, 30, 39, 40, 42, 48] and tools, such as CHORD [10], DOOP [14], SOOT [49] and WALA [50].

The allocation-type abstraction (with one abstract object per type) was used earlier to resolve virtual calls [35, 47]. It is reasonably precise, compared with *0*-CFA [37] and CHA [13], 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 [19, 38, 51].

Liang and Naik [27] 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.

8. Conclusion and Future Work

We have introduced MAHJONG, a novel technique for abstracting the heap to scale significantly points-to analyses for object-oriented programs 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.

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

Acknowledgments

We would like to thank our shepherd, Prof. Jeff Foster, and the anonymous reviewers for their valuable feedback on an earlier draft of this paper. This research has been supported by ARC grants, DP150102109 and DP170103956.

References

[1] J. Adamek and V. Trnkova. *Automata and Algebras in Categories*. Kluwer Academic Publishers, 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. Averroes: Whole-program analysis without the whole program. ECOOP, pages 378–400, 2013.

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

[5] 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. PLDI, pages 259–269, 2014.

[6] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Selective control-flow abstraction via jumping. OOPSLA, pages 163–182, 2015.

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

[8] 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. ICSE, pages 241–250, 2011.

[9] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. OOPSLA, pages 243–262, 2009.

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

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

[12] DaCapo. Java benchmark. `http://www.dacapobench.org`.

[13] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. ECOOP, pages 77–101, 1995.

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

[15] Y. Feng, X. Wang, I. Dillig, and T. Dillig. Bottom-up context-sensitive pointer analysis for Java. APLAS, pages 465–484, 2015.

[16] 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.

[17] M. Hind. Pointer analysis: Haven't we solved this problem yet? PASTE, pages 54–61, 2001.

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

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

[20] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. PLDI, pages 423–434, 2013.

[21] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. CC, pages 153–169, 2003.

[22] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? CC, pages 47–64, 2006.

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

[24] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. ECOOP, pages 27–53, 2014.

[25] Y. Li, T. Tan, and J. Xue. Effective soundness-guided reflection analysis. SAS, pages 162–180, 2015.

[26] Y. Li, T. Tan, Y. Zhang, and J. Xue. Program tailoring: Slicing by sequential criteria. ECOOP, pages 15:1–15:27, 2016.

[27] P. Liang and M. Naik. Scaling abstraction refinement via pruning. PLDI, pages 590–601, 2011.

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

[29] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. ISSTA, pages 1–11, 2002.

[30] 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.

[31] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. PLDI, pages 308–319, 2006.

[32] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. ICSE, pages 386–396, 2009.

[33] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. PLDI, pages 475–484, 2014.

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

[35] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. CC, pages 126–137, 2003.

[36] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

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

[38] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, pages 1–69, 2015.

[39] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. POPL, pages 17–30, 2011.

[40] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. PLDI, pages 485–495, 2014.

[41] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. ECOOP, pages 22:1–22:26, 2016.

[42] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. PLDI, pages 387–400, 2006.

[43] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. PLDI, pages 112–122, 2007.

[44] 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. 2013.

[45] Y. Sui and J. Xue. On-demand strong update analysis via value-flow refinement. In *FSE*, pages 460–473, 2016.

[46] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. CGO, pages 1–11, 2013.

[47] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. OOPSLA, pages 264–280, 2000.

[48] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. SAS, pages 489–510, 2016.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. CASCON, pages 1–13, 1999.

[50] WALA. Watson libraries for analysis. `wala.sf.net`.

[51] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. PLDI, pages 131–144, 2004.

[52] 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. CGO, pages 218–229, 2010.

[53] Q. Zhang and Z. Su. Context-sensitive data-dependence analysis via linear conjunctive language reachability. POPL, pages 344–358, 2017.

[54] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang. On abstraction refinement for program analyses in Datalog. PLDI, pages 239–248, 2014.

[55] Y. Zhang, T. Tan, Y. Li, and J. Xue. Ripple: Reflection analysis for android apps in incomplete information environments. 2017.