DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Strong Update for Object-Oriented Flow-Sensitive Points-To Analysis

Citation
Chao, Ling-Ya Monica. 2015. Strong Update for Object-Oriented Flow-Sensitive Points-To Analysis. Bachelor's thesis, Harvard College.

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:14398535

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Strong Update for Object-Oriented Flow-Sensitive Points-To Analysis

by

Ling-Ya Monica Chao

A Thesis
Presented to the Faculty of the Committee on
Degrees in Computer Science
In partial fulfillment of the requirements for the
Degree of Bachelor of Arts with Honors

Harvard University
Cambridge, Massachusetts
April 2015

Contents

Acknowledgements **2**

1 Introduction **3**

2 Background and Related Work **5**
2.1 Points-To Analysis and Applications . . . 6
2.2 Context Sensitivity . . . 7
2.3 Flow Sensitivity . . . 12
2.4 Recency Abstraction . . . 14

3 Design of Analysis **15**
3.1 A Brief Introduction to Datalog . . . 16
3.2 Domains and Relations . . . 17
3.3 Inference Rules . . . 24

4 Implementation Overview **33**
4.1 Java Bytecode and the WALA Framework . . . 34
4.2 Representation of the Points-To Graph . . . 34
4.3 Program-Point Reachability Query . . . 35
4.4 Context-Sensitivity Choice . . . 36
4.5 Multi-Threaded Dependency-Driven Engine . . . 37

5 Evaluation **38**
5.1 Points-To Analysis Performance . . . 39
5.2 Client Analysis Framework : Data-Flow Analysis . . . 40
5.3 Client I : Non-Null Analysis . . . 41
5.4 Client II : Interval Analysis . . . 42
5.5 Client III : Cast Removal . . . 43

6 Conclusion **45**

Acknowledgements

I would like to extend my deep gratitude to Professor Stephen Chong, whose group I have been a part of for the past 1.5 years. I am continually impressed by Prof. Chong's involvement in my research in addition to his busy workload, and I cannot thank him enough for his guidance and encouragement. Under the supervision of Prof. Chong, I have been able to become involved in the preparation and submission of my first conference paper.

Furthermore, I am grateful for Andrew Johnson's guidance throughout my research experience. Both he and Prof. Chong took lead roles in the implementation of our analysis, and their work has been crucial for the success of my project. Andrew spent a considerable amount of time to ensure that I fully understood the conceptual details of my project. I also want to thank the members of the PL group for their endless support.

Finally, I would like to acknowledge those who have been instrumental in my education. Professor Greg Morrisett, thank you for your mentorship and advice. Professors Jenny Hoffman and Peter Park, thank you for your guidance in other research projects that I was part of. And of course, thank you mom and dad. Your undying support and encouragement from Taiwan have made it possible for me to experience a tremendously fulfilling time at Harvard.

Chapter 1

Introduction

Points-to analysis is a program analysis that seeks to statically infer the memory locations each program expression can refer to at runtime. Many client analyses incorporate the points-to information to inspect other program properties and reason about the security and correctness of a piece of code. The effectiveness of these client analyses strongly depends on the precision of the points-to analysis, and thus, how to design accurate yet scalable points-to analyses is an important and widely-researched issue.

There are many design choices faced when implementing a points-to analysis, and one of them is *flow sensitivity*. A flow-sensitive analysis computes points-to information for each program point (as opposed to a *flow-insensitive* analysis, which conservatively computes what each program expression may point to at any point in the execution). One means by which flow-sensitivity increases precision is through enabling *strong update*, which allows the overwriting of statically computed facts upon certain program instructions. By contrast, *weak update* only *adds* to known facts and will possibly end up with points-to relations that never actually take place.

For a *strong update* to be performed soundly, it is required that the abstract memory location under consideration represents only one concrete location; otherwise, we cannot be certain that all data corresponding to the abstract location is overwritten. We determine this information by utilizing *recency abstraction*: each allocation site is represented by two abstract objects, the most-recently-allocated one and the one that summarizes all other objects allocated at the site [1]. The former, by default, only corresponds to one concrete object, and thus its fields each represent one concrete location and can be strongly updated.

Interestingly, since object construction typically takes place on the most-recently-allocated object, recency abstraction enables strong update exactly where it is most useful for object-oriented analyses: during the establishment of object invariants. For instance, it is often useful to determine whether a field of an object is non-null. Since the initial value of reference-type fields are always null, strong update is required to replace the default points-to information and soundly reason that the field is non-null after the constructor is called.

In this project, we present a flow-sensitive points-to analysis for object-oriented programs that is specifically designed to enable strong update, both in the points-to analysis itself and other client analyses. More specifically, we hope to make the following contributions:

1. A novel flow-sensitive points-to analysis that enables strong update using the recency abstraction. To ensure scalability, flow-sensitivity is only used for a limited set of abstract locations that will likely gain precision from it [1, 10]. We will also parameterize the analysis using Kastrinis and Smaragdakis's framework [8], thus providing flow-sensitivity to a wide variety of known points-to analyses.

2. A multi-threaded implementation of our analysis. The computation time is much longer than flow-insensitive analyses as expected, but it is still scalable (130k lines of code can be analyzed in 92 seconds) and the precision gained may be considered a reasonable tradeoff. We also designed a novel representation of the points-to graph to enable efficient computation.

3. A demonstration that our points-to analysis effectively increases precision for some client analyses. We compared the performance between our analysis and a flow-insensitive analysis using a non-null analysis, an interval analysis and a cast removal analysis.

The following chapters are presented in the following order: Chapter 2 presents some background information and related work. Chapter 3 describes our analysis in details using Datalog rules. Chapter 4 briefly introduces the implementation. Lastly, we report the performance of our analysis and compare the results of three client analyses in Chapter 5.

Chapter 2

Background and Related Work

In this chapter, we will present some background information for our flow-sensitive points-to analysis framework. First, we provide a brief introduction of points-to analysis and its applications in other program analyses. Next, we discuss possible design choices that can potentially increase accuracy: 1) *Context sensitivity* distinguishes between different invocations of a method by qualifying reference variables and abstract objects with context information. Further details are provided regarding various types of context sensitivity and the framework proposed by Smaragdakis that abstracts the differences between them. 2) *Flow sensitivity* incorporates the program's control flow and generates a points-to graph for each program point. This enables more opportunities for strong update, which is used to replace points-to facts associated with a memory location if the abstract location corresponds to one concrete memory location. 3) We briefly touch upon the concept of *recency abstraction*, which is used to enable strong update in our analysis. In this abstraction, each abstract object is identified as either most-recent or non-most-recent. The former by default corresponds to only one concrete object, and thus its fields are subject to strong update.

2.1 Points-To Analysis and Applications

Points-to analysis is a static program analysis that computes all the memory locations that a program expression may refer to (or *point to*) during runtime. In object-oriented languages, the expressions include *reference variables* and *fields of object instances*. In Java, reference variables range from static fields to method parameters and local variables. The points-to targets are *abstract heap objects* (abbreviated *abstract objects*).

Client analyses use the results of a points-to analysis to reason about program correctness, security, performance, etc. The accuracy of these client analyses strongly depends on the precision of the points-to analysis.

Consider the following program:

```
class X {
  Object f;
}

X x1 = new X(); // a1 (allocation site 1)
x1.f = new Object(); // a2 (allocation site 2)
X x2 = x1;
```

Figure 2.1: Points-to analysis example

One possible *points-to graph* is $\{x1 \rightarrow \{a1\}, x1.f \rightarrow \{a2\}, x2 \rightarrow \{a1\}\}$. In other words, if we simply represent abstract objects with their allocation sites, we can see that the local variable $x1$ can point to the abstract object that is allocated at $a1$, and the field f of this same abstract object can point to the object allocated at $a2$. The last line (a move instruction) implies that the points-to set of $x1$ will always be included in the points-to set of $x2$, and thus $x2 \rightarrow \{a1\}$. $\{a1\}$ is called the *points-to set* of $x1$. It is important to note that the elements in a points-to set are *abstract objects* and not *concrete objects*. The former represent statically-distinguishable, abstract memory locations, and can correspond to many (or zero) of the latter at run-time. For instance, if the allocation site $a1$ is enclosed in a for-loop that repeats for 10 iterations, $a1$ will correspond to 10 concrete objects.

Points-to analysis is the core part of many data-flow analyses (details in Section 5.2). It provides a more accurate memory status for program expressions, and as a result, client analyses do not need to assume that pointers can reference all allocated objects. This greatly improves the precision of

many program-understanding tools. Some interesting client analyses include side-effect analysis, which formalizes how the execution of a statement modifies memory locations, and def-use analysis, which identifies instructions that set and use the value of a memory location. The development of interprocedural points-to analysis has also enabled whole-program optimization, greatly increasing the efficiency of compilers. In our evaluation, we experiment with interprocedural non-null analysis, interval analysis and cast removal analysis.

The challenge of points-to analysis is developing frameworks with reasonable approximations that provide satisfactory accuracy but do not incur unreasonable time and storage cost. One design decision often faced is context sensitivity. Context-sensitive analyses annotate reference variables and abstract objects with context information to avoid collapsing points-to information that can potentially be distinguished for higher accuracy; it is also subjected to extra computational cost. Moreover, choosing the type of context sensitivity used also poses further questions regarding the trade-off between precision and scalability. The other design choice entails flow sensitivity. Flow-sensitive analyses take into account the control flow and keep track of points-to information at various program points, as opposed to flow-insensitive analyses, which compute points-to information that may hold at any point in the program's execution. As we will see later in this chapter, flow sensitivity promises more precise results, but also adds an additional dimension to the time and space complexity.

One foundational algorithm for points-to analysis was proposed by Anderson in 1994. He developed a flow-insensitive, context-sensitive set-based points-to analysis framework, and implemented it via constraint solving. On a high level, the theoretical framework for the analysis is expressed as subset constraints that a sound points-to map should fulfill. The solving procedure involves iterating through the constraints until a fixed point is reached. Analogous to Anderson's algorithm, we designed and implemented a *flow-sensitive* set-based points-to analysis, incorporating a parameterizable abstraction for context sensitivity.

2.2 Context Sensitivity

Context sensitivity distinguishes between different invocations of a method by qualifying reference variables and abstract objects with context information. In other words, two variables or two abstract objects associated with the same context will be considered as the same key in a points-to graph and their points-to sets will be collapsed; points-to sets for different contexts will be separated. In contrast, context-insensitive analyses do not distinguish nor keep track of contexts. Although context

insensitivity can be practical and efficient for large programs, they often incur significant imprecision.

For example, given a class definition like `X` in Figure 2.2. If the method `set` is called twice, with different receivers o_1, o_2 (these are abstract heap objects) and parameters n_1, n_2 (these are reference variables), a context-insensitive analysis will consider the variable `this` the same in both invocations, since no context is associated with it. As a result, the fields $o_1.f$ and $o_2.f$ will both have a points-to set of $s_1 \cup s_2$, where s_1 and s_2 are the points-to set of n_1 and n_2 respectively. In reality, a more precise points-to result should be $\{o1.f \rightarrow s_1, o2.f \rightarrow s_2\}$.

Different types of context sensitivity are distinguished by the information they use as contexts. In general, they can be grouped into two categories: *call-site sensitivity* and *object sensitivity*. Call-site sensitive analyses use the call-stack as the context. For reference variables, this is the sequence of invocation sites that lead to the current method call. For abstract objects, the sequence of invocations that lead to the object's allocation is used as the context. On the other hand, object-sensitive analyses use the stack of allocation sites as the context instead, and method calls are distinguished by the allocation site of the receiver.

```
class X {
  Object f;
  void set(Object n) { this.f = n; }
}

class Y {
  void foo(X x1, X x2, Object n1, Object n2) {
    x1.set(n1); // call-site 1
    x2.set(n2); // call-site 2
  }
}
```

Figure 2.2: Another context sensitivity example

Consider the same program in Figure 2.2: A call-site-sensitive analysis will separate the two calls to method `set` by the call-sites 1 and 2, and analyze them with the points-to set of $n1$ and the points-to set of $n2$ respectively [13,14]. Object-sensitive analyses, on the other hand, distinguish `set` as multiple cases depending on the allocation sites of the receiver objects that $x1$ and $x2$ can point to [11, 12, 15]. For example, if the points-to information of $x1$ and $x2$ are $\{x1 \rightarrow \{o_1, o_2\}, x2 \rightarrow \{o_3, o_4, o_5\}\}$, then `set` will be analyzed with 5 different contexts. In other words, if $x1$ and $x2$ can each point to more than one distinct objects, then `set` will be analyzed under more than two contexts. $x1$ and $x2$ can

also have the same points-to set with one single element $\{x1 \rightarrow \{o_1\}, x2 \rightarrow \{o_1\}\}$, and `set` will only be analyzed with one context. This shows that the precision of call-site-sensitive analyses and object-sensitive analyses are not always comparable. However, it is apparent that, object sensitivity and call-site sensitivity are always at least as precise as context insensitivity, in which the method `set` will always be analyzed as one instance despite different calls.

In order to achieve a better understanding of object sensitivity, Kastrinis and Smaragdakis proposed a parameterizable formalism that captures and abstracts the differences between various types of established context-sensitive analyses [8]. The semantics consists of several inference rules that specify how the points-to graph changes when different statements are encountered, including allocation ($var = new...$), move ($to = from$), static call ($Calls.meth(arg1, arg2, ...)$), etc. The differences between various kinds of context sensitivity are abstracted with three constructor functions RECORD, MERGE and MERGESTATIC, as listed in Figure 2.3.

RECORD: *allocation site* × *context* → *heap context*

MERGE: *allocation site* × *heap context* × *call site* × *context* → *context*

MERGESTATIC: *call site* × *context* → *context*

Figure 2.3: Parameterizable framework for context sensitivity

These constructors, as we will see formally in Chapter 3, formulate how the contexts associated with reference variables, abstract heap objects and method calls are computed. More specifically, RECORD(*alloc, ctx*) constructs a new heap context when processing allocation statements, using the allocation site and the context at the allocation site. MERGE(*alloc, heapCtx, invo, callerCtx*) and MERGESTATIC(*invo, callerCtx*) are used to create the context associated with method calls (virtual and static calls, respectively), and subsequently with all the reference variables in the callees. Both constructors take the input of the call site (invocation site) information and the caller context. MERGE also requires the receiver information (allocation site and heap context, the latter defined by the RECORD constructor upon the allocation of the receiver). Depending on the type of context sensitivity, not all inputs will be used in the three constructors.

The following are some examples of how the constructors are formulated for traditional context-sensitive analyses.

Context-insensitive. Context insensitivity implies that no context information is associated with method calls. Naturally, no context information needs to be recorded at allocation sites.

$\textsc{Record}(alloc, ctx) = \phi$

$\textsc{Merge}(alloc, heapCtx, invo, callerCtx) = \phi$

$\textsc{MergeStatic}(invo, callerCtx) = \phi$

1-call-site-sensitive. Call-site-sensitive analyses use the call-stack as context when analyzing the called method. The prefix 1 implies that the call-stack only has one element, thus the constructors MERGE and MERGESTATIC create contexts with only the current invocation site information. The context at the receiver's allocation site is not relevant in these analyses, thus no context needs to be recorded at allocation sites.

$\textsc{Record}(alloc, ctx) = \phi$

$\textsc{Merge}(alloc, heapCtx, invo, callerCtx) = invo$

$\textsc{MergeStatic}(invo, callerCtx) = invo$

1-object-sensitive. Object-sensitivity uses the receiver's allocation site as the method call context. For static calls, there is no receiver object, so the caller context continues to act as the method-call context.

$\textsc{Record}(alloc, ctx) = \phi$

$\textsc{Merge}(alloc, heapCtx, invo, callerCtx) = alloc$

$\textsc{MergeStatic}(invo, callerCtx) = callerCtx$

2-object-sensitive with a 1-context-sensitive heap. This is an object-sensitive analysis with a deeper context. The method-call context consists not only of the receiver object's allocation site, but also of its heap context (i.e. the first context element of the allocating method, which is the receiver object on which it was invoked). In other words, the context for a method call is the receiver object and the parent receiver object that allocates it.

RECORD(*alloc*, *ctx*) = first(*ctx*)

MERGE(*alloc*, *heapCtx*, *invo*, *callerCtx*) = pair(*alloc*, *heapCtx*)

MERGESTATIC(*invo*, *callerCtx*) = *callerCtx*

As the depth of context sensitivity increases, context tuples will contain more elements, and the size of the context domain increases exponentially. This is an apparent problem for object-sensitive analyses, as the number of allocation sites is usually large. Smaragdakis proposed a new category of context sensitivity – type sensitivity [15]. It is very similar to object-sensitivity, but seeks to reduce the number of allocation sites by collapsing them using type information. Directly using the type of the receiver object is not advisable, since the method under inspection already reveals much information about the receiver type. Instead, the dynamic type of the *allocator* object correlates less with the method called and is a better choice. More specifically, if an allocation site `new A()` is within class C, it is much better to use C instead of A as the type information for the allocation site.

As an example, this is a type-sensitive scheme:

2-type-sensitive with a 1-context-sensitive heap. This is analogous to 2-object-sensitive with a 1-context-sensitive heap. $\mathcal{T}$: *allocation site* $\rightarrow$ *type* is a dictionary between allocation sites and types: given an allocation site, $\mathcal{T}$ returns the type of the allocator.

RECORD(*alloc*, *ctx*) = first(*ctx*)

MERGE(*alloc*, *heapCtx*, *invo*, *callerCtx*) = pair($\mathcal{T}(alloc)$,*heapCtx*)

MERGESTATIC(*invo*, *callerCtx*) = *callerCtx*

In our flow-sensitive scheme, we continue to use this abstraction in the proposed inference rules. Although the choice of context sensitivity is not the main focus of our work, this parameterizable model provides a flexible framework for future experimentation on how the type and depth of context sensitivity influence precision and cost in flow-sensitive analyses. In our evaluation, we use a hybrid of full-object sensitivity and type sensitivity to balance the precision needed by clients and scalability (implementation details described in Chapter 4).

2.3 Flow Sensitivity

Designing a scalable flow-sensitive analysis has been an interesting topic of research. In contrast to a flow-insensitive analysis, which derives points-to information that may hold at any point in the program, a flow-sensitive analysis incorporates the program's control flow and generate a points-to mapping for each program point. The space and time scalability has always been a major concern: the fact that points-to sets depend on the specific location in the program greatly increases the number of constraints, the time iterating through them and the space required to store the points-to information at each program point.

A flow-sensitive analysis is usually based on a pre-computed *control flow graph* (CFG), a directed graph with program statements as nodes. The edges represent the control flow between statements. As we will see in the next chapter, the points-to analysis can be written in a set-based scheme, formalized with constraints. These constraints either represent how points-to graphs change between program points, across statements, or between calls. The changes across a statement are strongly dependent on the points-to information generated and killed by the statement.

One major precision benefit provided by flow sensitivity is the increased opportunity for *strong update* [5]. Strong update occurs when it is sound to *kill* a reference variable or a field (of an object instance) at a program point (i.e. replace the points-to facts associated with the memory location). This can only be done if we know that the instruction always overwrites the value, hence the points-to set, of the variable or field. By contrast, in a weak update, an assignment *adds* to the statically computed facts of a memory location instead of replacing it.

To see the benefits of flow sensitivity and strong update, consider the code in Figure 2.4 and a client analysis that attempts to identify whether the target of a field access or method call is always non-null (and `NullPointerException` will not be thrown). Note that the target of the call `toString()` at program point $p3$ is never null in runtime. However, in Java, all fields with reference type are

```
class A {
  Object f;
}

A a = new A(); // p1
a.f = new Object(); // p2
a.f.toString(); // p3
```

Figure 2.4: Strong update example

initialized to null so field $a.f$ will be null immediately after $p1$. Thus, a flow-insensitive points-to analysis must include null in the points-to set for $a.f$, and the client cannot conclude that the target of the method call will be non-null. Strong update can help precision here: by distinguishing the points-to set of $a.f$ immediately after $p1$ (where it might be null) and immediately after $p2$ (where it is definitely not null), and noticing that $p2$ overwrites the points-to information of $a.f$, we can assert that $a.f$ at $p3$ is non-null.

Both local variables and fields of object instances are subject to strong updates. It is well known that the strong updates for local variables can be easily achieved by converting the program to Static Single Assignment (SSA) form. In SSA form, each variable can only be assigned once and every use is associated with a unique definition. This prevents redefinition and overwriting, and as a result, strong update is not needed.

The case for field update is more complicated. To soundly perform strong update on an instruction in the form of `x.f = ...` where x is a reference variable and f a field, we need to know that $x.f$ corresponds to one concrete memory location (i.e., the points-to set of x consists of exactly one element at *run-time*), or else $x.f$ cannot be safely overwritten. In other words, x should only point to one abstract heap object, and the abstract object should only correspond to a single *concrete* object during execution of the program.

In Lhotak and Chung's paper, abstract memory locations that correspond to one single concrete memory location are called *singletons* [10]. Since strong update contributes to most of the precision benefit of flow-sensitive analyses, it is crucial to keep track of singletons with high precision. Tracking abstract locations that correspond to multiple concrete locations (i.e. abstract locations that are not singleton) is often not necessary, and can be very costly. In other words, scalability can be greatly increased without sacrificing much precision if only singletons are tracked flow-sensitively, and oth-

ers are not. The strong update algorithm proposed by Lhotak has a worst-case complexity the same as that of the flow-insensitive analysis, yet the precision is greatly increased from tracking singletons flow-sensitively.

2.4 Recency Abstraction

In order to determine whether an abstract memory location is a singleton, we incorporate the *recency abstraction* proposed by Balakrishnan and Reps [1]. Although it was presented for C programs, an analogy can be drawn for object-oriented languages. Each allocation site is represented with two abstract object: the most-recent allocation, and the non-most-recent allocation. The most-recent allocation, as suggested by the name, is the most-recently-allocated abstract object at the site. By definition, there will only be one concrete object corresponding to this abstract object, and as a result, we can soundly assume that its fields hold one concrete memory location each and are subject to strong update. On the other hand, the non-most-recent allocation can correspond to one or more concrete objects. The exact number of corresponding concrete objects is not known, so its fields are assumed to not be singletons and are more efficient to be tracked flow-insensitively. In summary, the only circumstance in which strong update can be performed upon field update $x.f = ...$ is when x points to exactly one abstract object, and the abstract object is most-recent.

By contrast, the *allocation-site abstraction* collapses all the abstract objects allocated at a given allocation into a single summary node. In this case, strong update can only be performed if we are certain that the allocation only occurs once (i.e. it occurs in a method-context pair that is only invoked once and does not occur in a loop). Much of the related work performs strong update using this abstraction.

It is interesting to note that Balakrishnan and Reps introduce the recency abstraction for a different purpose. They use it to resolve virtual function calls in executables. Since function dispatch tables are typically established immediately after object creation, gaining additional precision on the most recently created object enables precision exactly where it is most useful.

Chapter 3

Design of Analysis

In this section we describe our flow-sensitive analysis using Datalog syntax. We utilize recency-abstraction to model memory locations and in turn allow strong updates during the analysis itself and its client analyses. Datalog rules are monotonic, logical inferences that are iteratively applied until a fixpoint is reached. Each rule consists of multiple atomic predicates, which, when all true, implies that some additional relations are true.

We first discuss the domain of our analysis, representing the parameter space for the atomic relations. Then, we present the relations that model the input language. We also list the relations that represent the output and intermediates. Many of them are identifiers of whether an abstract object is in the points-to set of a reference variable or a field of an object. Finally we list the inference rules that define how we process each statement in the input language, as well as how the flow of points-to information is done between and across program points.

In the next chapter, we will continue to discuss our implementation of the analysis. The Datalog scheme presented in this section is a simplified version of the actual implemented analysis, as we hope to emphasize how strong update and recency abstraction are utilized in a flow-sensitive setting and leave out details that are not relevant to this discussion.

3.1 A Brief Introduction to Datalog

Our flow-sensitive points-to analysis will be presented as Datalog rules, in the style of Bravenboer and Smaragdakis [3]. Datalog rules are monotonic, logical inferences that are iteratively applied until a fixpoint is reached. A *rule* usually consists of multiple atomic predicates (conditions), written on the right side of the arrow (the *body* of the rule). When all predicates are true, the relations on the left (the *head* of the rule) are inferred to be true. The atomic relations are represented with a pre-defined set of functions that return true or false depending on the parameters given, thus this formalism is often called a *parametric Datalog program*. The following is a simple example:

$$\textsc{IsGrandfather}(X,Y) \leftarrow \textsc{IsFather}(X,Z), \textsc{IsMother}(Z,Y)$$

In English, this rule is translated as: *if* X is the father of Z and Z is the mother of Y, *then* X is the grandfather of Y. Although this formalism does not explicitly show subset relations, it can still represent subset-based constraints. For instance, we can define a relation $\textsc{IsElementInSet}(e,S)$ represeting that element e is in the set S. Then the rule

$$\textsc{IsElementInSet}(e,Y) \leftarrow \textsc{IsElementInSet}(e,X)$$

will imply that every element in X is also in Y, corresponding to the subset relation $X \subseteq Y$. Many of the relations we use in the analysis will be in this format: the element e will correspond to an abstract object and the sets X, Y will represent the points-to set of a reference variable or a field.

Although our implementation does not directly implement the Datalog rules (i.e., we do not use a Datalog engine), Datalog provides a simple and compact specification of the analysis. A few differences between our specification and the traditional Datalog includes: 1) We use *stratified negation*, allowing negations that are outside of recursive cycles and do not interfere with the termination of the analysis. The Logicblox engine that was used by Smaragdakis in his flow-insensitive points-to analysis also supports this functionality [15]. 2) We occasionally write relations using function notation (e.g., $\textsc{Record}(h,c)$ = hc instead of $\textsc{Record}(h,c,hc)$) for clarity, when the relation is restricted to being a function.

In the exposition and figures, we indicate Datalog relations with small caps (i.e., KILLFIELD). Variables are written italics with an initial lowercase letter (e.g., *c*), domains are written with italicized uppercase letters (e.g., *H*), and domain elements are written using a typewriter font (e.g., `NR`).

3.2 Domains and Relations

Domain. Figure 3.1 shows the domain of our analysis, representing the parameter space for the atomic relations. It is mostly identical to that of a standard flow-insensitive analysis.

T	: class types	C	: contexts
F	: fields	H	: allocation sites
M	: methods	HC	: heap contexts
P	: program points	R	: recency = $\{\texttt{R}, \texttt{NR}\}$
S	: method signatures	IP	: inter-program points
V	: reference variables	N	: natural numbers

Figure 3.1: Analysis domains

Domain T is the set of class types used by the program, F is the set of fields, M is the set of methods, and P is the set of program points. Domain S contains method signatures, and is used to resolve dynamic dispatch. A method signature can be thought of as consisting of a method name and the types of the arguments.

V is the set of *reference variables*, representing program variables with reference type. This includes method formal parameters and local variables. Additionally, static fields of reference type are also represented as elements in this set, as they correspond to one concrete memory location. In our analysis, the majority of reference variables are tracked flow-insensitively, because most of the benefit from flow-sensitivity can be achieved using SSA form for local variables, and tracking flow-sensitively is very costly. The only exception is static fields, which we choose to be flow-sensitive, because it is very hard to transform them into SSA form. The relations and rules we present here are more general. We provide the flexibility of categorizing whether a variable is flow-sensitive or not; the inference rules handle both cases. Only at one place, as we will specifically point out, we assume for simplicity that flow-sensitive reference variables can only be static fields (although the rules can certainly be modified to remove this assumption). C is the set of contexts, which are used in context-sensitive analyses. A pair $(v, c) \in V \times C$ is a *reference-variable replica* (i.e. a local variable v of a method analyzed in context c) [12].

Abstract heap objects are represented by a tuple $(h, hc, r) \in H \times HC \times R$. H is the set of allocation sites, HC includes all heap contexts, and $R = \{\texttt{R}, \texttt{NR}\}$ is a flag that indicates whether an object is most-recent (R) or non-most-recent (NR). If an abstract object is most-recent, it corresponds to a single concrete object, thus its fields can be strongly updated and should be tracked flow-sensitively. Non-most-recent abstract objects, on the other hand, summarize the history of an allocation site and may represent many concrete objects; they are tracked flow-insensitively for efficiency.

Object fields represent the fields of abstract objects, and are defined using a tuple (h, hc, r, fld). The first three arguments identify the object and *fld* specifies the field under consideration.

As described in Section 3.2, contexts and heap contexts are used in context-sensitive analyses to distinguish reference variables, abstract heap objects and object fields that are analyzed in different executions. In our analysis, they are parameterized by three functions, RECORD, MERGE and MERGESTATIC, as listed in Figure 3.2. RECORD(h, c) provides the heap context for an object created at allocation site h in context c. MERGE(h, hc, r, pp, c) provides the callee analysis context for a dynamic-dispatch method invocation at program point pp in a method analyzed in context c, where the receiver is the abstract heap object (h, hc, r). Note that the recency flag r is also included in the parameters to distinguish between two abstract objects (most-recent and non-most-recent), both of which can act as the receiver. MERGESTATIC(pp, c) provides the callee analysis context for a static method invocation at program point pp in a method analyzed in context c, without a receiver object.

RECORD$(h : H, c : C) = newHC : HC$

MERGE$(h : H, hc : HC, r : R, callSite : P, c : C) = newC : C$

MERGESTATIC$(callSite : P, c : C) = newC : C$

Figure 3.2: Context creation functions

Program points P and inter-program points IP are especially important for flow-sensitivity and are not normally needed in a flow-insensitive analysis. They allow the analysis to take the program's control flow into account. A *program point* can either correspond to a statement or serve as a point for method entry and exits. Prior to the points-to analysis, a program-point graph, G_{pp}, will be recorded and used as a reference to determine the control-flow edges along which points-to information flows. *Inter-program point*s, on the other hand, represent the locations right before or after a program point

Program statement $\boldsymbol{var} = \textbf{new } \boldsymbol{h}$ at program point pp.

ALLOC(*var : V, h : H, pp : P*)

Program statement $\boldsymbol{to} = \boldsymbol{from}$ at program point pp.

MOVE(*to : V, from : V, pp : P*)

Program statement $\boldsymbol{to} = \boldsymbol{null}$ at program point pp.

MOVENULL(*to : V, pp : P*)

Program statement $\boldsymbol{to} = \boldsymbol{base.fld}$ at program point pp.

LOAD(*to : V, base : V, fld : F, pp : P*)

Program statement $\boldsymbol{base.fld} = \boldsymbol{from}$ at program point pp.

STORE(*base : V, fld : F, from : V, pp : P*)

Virtual call statement $\boldsymbol{base.sig}$ at program point $callSite$.

VCALL(*base : V, sig : S, callSite : P*)

Static call statement to method *meth* at program point $callSite$.

SCALL(*meth : M, callSite : P*)

Figure 3.3: Input relations (core instructions)

(i.e *prePP* or *postPP*). Because a program point usually alters the points-to information, the points-to sets computed for a flow-sensitive variable or object field will be specific to an inter-program point, not a program point. In addtion, because program points occur in methods, and methods may be analyzed in multiple contexts, *program point replica* $(pp, c) \in P \times C$ represents program point *pp* of method *m* being analyzed in context *c*. Similarly, *inter-program point replica* $(ipp, c) \in IP \times C$ represents inter-program point *ipp* of method *m* being analyzed in context *c*.

Input Relations. The input language is a simplified intermediate language which captures the pointer-related effects of Java bytecode, as well as many other high-level languages that does not contain the address-of operator (the techniques used for this operator are fairly different, although the approach is equally applicable). As shown in Figure 3.3, the language consists of the following seven core instructions: ALLOC instruction indicates allocation sites. MOVE copies objects between variables, while MOVENULL is generated by an assignment of `null` to a program variable. LOAD

and STORE read from and write to the heap memory locations, which are identified as object fields. Calls are either virtual or static (VIRTUALCALL and STATICCALL). The former represents a dynamic dispatch method call, which looks up the method identifier dynamically based on the signature and the class of the receiver object; the later simply calls a statically known target.

# Variable *arg* is the *i*th formal argument of method *meth*.
FORMALARG(*meth : M, i :* $\mathbb{N}$, *arg : V*)
# Variable *arg* is the *i*th actual argument at call site *callSite*.
ACTUALARG(*callSite : P, i :* $\mathbb{N}$, *arg : V*)
# Variable *ret* represents the return value in method *meth*.
FORMALRETURN(*meth : M, ret : V*)
# Variable *var* receives the return value at call site *callSite*.
ACTUALRETURN(*callSite : P, var : V*)
# *this* represents the receiver object when calling method *meth*.
THISVAR(*meth : M, this : V*)
# Objects created at allocation site *h* is of type *t*.
HEAPTYPE(*h : H, t : T*)
# Objects of type *t* have a field *fld*.
LOOKUPFLD(*t : T, fld : F*)
# A dynamic method invocation with signature *sig* on a receiver with run-time type *t* will resolve to method *meth*.
LOOKUPMETH(*t : T, sig : S, meth : M*)

Figure 3.4: Input relations (interprocedural data-passing)

In addition to these core instructions, there are also a few relations that encode relevant interprocedural data-passing information. They are listed in Figure 3.4. Facts ACTUALARG and ACTUALRETURN determine the variables passed into a method and the variable in which the returned value is stored. FORMALARG and FORMALRETURN represent the formal arguments and formal result for a given method. THISVAR indicates the receiver in a method. HEAPTYPE is used to look up the type of an allocation site. LOOKUPMETHOD dynamically resolves non-static methods to a given type, and

LOOKUPFIELD identifies the fields of a heap type.

Figure 3.5 shows a few additional relations regarding flow-sensitivity and program points: METHENTRY and METHEXIT determine the entry and exit points of a method. VARISFS shows if the given variable is tracked flow-sensitively (in our implementation, this relation holds only when *var* is a static field). Lastly, the relations between program points and inter-program points are represented by PRED, BEFORE and AFTER facts.

# Program point *ppEntry* is the entry point of method *meth*.
  METHENTRY(*meth : M, ppEntry : P*)
# Program point *ppExit* is an exit point of method *meth*.
  METHEXIT(*meth : M, ppExit : P*)
# Program point *pp* is in method *meth*.
  INMETH(*pp : P, meth : M*)
# Program point *pp*1 is an immediate predecessor of *pp*2.
  PRED(*pp1 : P, pp2 : P*)
# Inter-program point *ipp* is immediately before program point *pp*.
  BEFORE(*pp : P, ipp : IP*)
# Inter-program point *ipp* is immediately after program point *pp*.
  AFTER(*pp : P, ipp : IP*)
# *var* is tracked flow-sensitively.
  VARISFS(*var : V*)

Figure 3.5: Input relations (flow-sensitivity and program points)

Output Relations. Five relations are used to represent the output (points-to graph and call-graph), as listed in Figure 3.6. A points-to graph consists of a mapping from each reference variable replica and object field to their respective points-to set. VARPOINTSTOFS and VARPOINTSTOFI identify whether a variable replica can point to an abstract object; the former is for flow-sensitive variables, and the latter for flow-insensitive variables. For instance, VARPOINTSTOFI(*var*, *c*, *h*, *hc*, *r*) indicates that reference variable replica (*var*, *c*) may point to abstract object (*h*, *hc*, *r*) at any point in the program

execution. VARPOINTSTOFS includes two additional parameters, *ipp* and *ippc*, to show that the given points-to relation specifically holds at inter-program point replica (*ipp, ippc*). A similar format is used for object fields: FLDPOINTSTOFS(*bh, bhc, br, fld, h, hc, r, ipp, ippc*) indicates that at inter-program point replica (*ipp, ippc*), field *fld* of abstract object (*bh, bhc, br*) may point to abstract object (*h, hc, r*). Since we are only tracking most-recently-allocated objects flow-sensitively to maintain scalability, *br* should, by definition, always be `R`. FLDPOINTSTOFI is similar, but inter-program point is not needed and *br* = `NR`, since non-most-recent abstract objects are tracked flow-insensitively. Lastly, CALLGRAPH encodes calling relationships between methods. The methods are also qualified with context information, which will be used when analyzing the statements in the callee.

# Variable replica (var, c) may point to object (h, hc, r) at any program point.
VARPOINTSTOFI(*var : V, c : C, h : H, hc : HC, r : R*)
# Variable replica (var, c) points to object (h, hc, r) at inter-program point replica $(ipp, ippc)$.
VARPOINTSTOFS(*var : V, c : C, h : H, hc : HC, r : R, ipp : IP, ippc : C*)
# Object field (bh, bhc, br, fld) points to object (h, hc, r). br should always be `NR`.
FLDPOINTSTOFI(*bh : H, bhc : HC, br : R, fld : F, h : H, hc : HC, r : R*)
# Object field (bh, bhc, br, fld) points to object (h, hc, r) at inter-program point replica $(ipp, ippc)$. br should always be `R`.
FLDPOINTSTOFS(*bh : H, bhc : HC, br : R, fld : F, h : H, hc : HC, r : R, ipp : IP, ippc : C*)
# Method *meth* with context *calleeC* is called at site *callSite* with context *callerC*.
CALLGRAPH(*callSite : P, callerC : C, meth : M, calleeC : C*)

Figure 3.6: Output relations

To simplify the Datalog, we also informally define a universal VARPOINTSTO relation, as shown in Figure 3.7, that serves both flow-sensitive and flow-insensitive variables.

Intermediate Relations. Figure 3.8 lists a few intermediate relations that are computed on-the-fly to facilitate the computation of points-to results. REACHABLE identifies whether a method-context pair is reachable from the program entry point. OBJECTJUSTALLOCATED records the abstract objects that are allocated at an allocation site (specified as a program point replica). Recency is not recorded

VarPointsTo(*var, c, h, hc, r, ipp, ippc*) =

VarPointsToFS(*var, c, h, hc, r, ipp, ippc*) if VarIsFS(*var*),

VarPointsToFI(*var, c, h, hc, r*) otherwise.

Figure 3.7: Simplification for points-to information for reference variables

here, as it should be R by default. Fact KillField is especially important for strong updates, as it specifies whether an object field can be killed (i.e., strongly updated) at a given program point replica. More specifically, upon a field-store *base.fld* = *from* at program point replica (*pp*, *ppc*), an object field (*h*, *hc*, *r*, *fld*) can be strongly updated only if 1) (*base*, *ppc*) points to one single object (*h*, *hc*, *r*) and 2) r = R (i.e. the abstract object pointed to is most-recent, and corresponds to one concrete object). Similarly, KillVar identifies whether the points-to information of a flow-sensitive reference variable v is propagated across *pp*. Since flow-sensitive variables each corresponds to one concrete memory location, they will always be killed upon Move, MoveNull, Load and Alloc.

Method *meth* is reachable at context c.

Reachable(*meth : M, c : C*)

Abstract object (h, hc) is allocated at program point replica (pp, ppc).

ObjectJustAllocated(*h : H, hc : HC, pp : P, ppc : C*)

Object field (h, hc, r, fld) can be strongly updated across program point replica (pp, ppc).

KillField(*h : H, hc : HC, r : R, fld : F, pp : P, ppc : C*)

Variable v can be strongly updated across program point pp.

KillVar(*var : V, pp : P*)

Figure 3.8: Intermediate relations

3.3 Inference Rules

We can now examine the inference rules that are used to derive points-to relationships. The predicates and inferred heads are all represented using the above relations. We will first introduce the rules that deal with the seven core instructions. Then we consider how the points-to information flow between sequential statements and between calls. Lastly, we discuss the rules that propagate points-to sets across statements. This applies to both reference-variable and object-field points-to sets, and strong updates are performed when applicable.

RECORD(*h*, *c*) = *hc*,
OBJECTJUSTALLOCATED(*h*, *hc*, *pp*, *c*),
VARPOINTSTO(*var*, *c*, *h*, *hc*, `R`, *postPP*, *c*) ←
  ALLOC(*var*, *h*, *pp*), INMETH(*pp*, *meth*),
  REACHABLE(*meth*, *c*), AFTER(*pp*, *postPP*).

RECORD(*h*, *c*) = *hc*,
FLDPOINTSTOFS(*h*, *hc*, `R`, *fld*, `HNULL`, `HCNULL`, `R`, *postPP*, *c*) ←
  ALLOC(*var*, *h*, *pp*), INMETH(*pp*, *meth*),
  REACHABLE(*meth*, *c*), AFTER(*pp*, *postPP*), HEAPTYPE(*h*, *t*), LOOKUPFIELD(*t*, *fld*).

Figure 3.9: Rules for allocation

Alloc. Figure 3.9 shows the rules that process allocations. If method *meth* is reachable in context *c*, the newly created abstract heap object (*h*, *hc*, `R`) should be added to the points-to set of the target variable replica (*var*, *c*). Heap context *hctx* is defined by the RECORD function. The recency of this object is always most-recent (`R`), as it is the most-recently-allocated object by definition. All fields of the abstract object should be initialized to point to null immediately after the allocation site. We use a special abstract object (`HNULL`, `HCNULL`) to represent `null`. Note that, in the first rule, VARPOINTSTO, as defined in Figure 3.7, is an abbreviated notation for either a flow-sensitive or flow-insensitive points-to relation. Formally, the rule should be split into two cases to handle both cases.

Move, MoveNull, Load and Store. Figure 3.10 shows the processing of these instructions. The first rule simply states that given assignment *to* = *from*, if reference variable replica (*from, c*) points to heap object (*h, hc, r*) before the statement, then so does reference variable replica (*to, c*) after the statement. The second states that, after an assignment *to* = *null* in a method reachable in context *c*, reference variable replica (*to, c*) points to the null abstract object, (`HNull`, `HCNull`). The next two rules handle field loads *to* = *base.fld* for both flow-sensitive and flow-insensitive cases. In the flow-sensitive case, we ensure that if reference variable replica (*base, c*) points to heap object (*bh, bhc,* `R`) and object field (*bh, bhc,* `R`*, fld*) points to heap object (*h, hc, r*) immediately before the field load, then reference variable replica (*to, c*) also points to (*h, hc, r*). Field stores are very similar, and are handled by the last two rules.

Virtual and Static Calls. When processing call statements, the call graph and reachability graph are updated with the new call. The rules are listed in Figure 3.11. For virtual calls, the receivers are determined from the known points-to information, and the appropriate methods dispatched to are looked up. Then, we mark the callee reachable in the context specified by function MERGE, and record that the `this` variable of the callee method points to the receiver of the method invocation. The rule for static method invocation is similar, but uses function MERGESTATIC, and is not concerned with a receiver object. The last two rules ensure that, after the callee's entry point, the formal arguments of the callee point to everything that the actual arguments point to prior to the call, and that, after the call, the actual return variable points to everything that the formal return variable points to before exiting the called method.

```
VarPointsTo(to, c, h, hc, r, postPP, c) ←
  Move(to, from, pp), VarPointsTo(from, c, h, hc, r, prePP, c),
  Before(pp, prePP), After(pp, postPP).

VarPointsTo(to, c, HNULL, HCNULL, R, postPP, c) ←
  MoveNull(to, pp), InMeth(pp, meth),
  Reachable(meth, c), After(pp, postPP).

VarPointsTo(to, c, h, hc, r, postPP, c) ←
  Load(to, base, fld, pp),
  VarPointsTo(base, c, bh, bhc, NR, prePP, c), FldPointsToFI(bh, bhc, NR, fld, h, hc, r),
  After(pp, postPP).

VarPointsTo(to, c, h, hc, r, postPP, c) ←
  Load(to, base, fld, pp),
  VarPointsTo(base, c, bh, bhc, R, prePP, c),
  FldPointsToFS(bh, bhc, R, fld, h, hc, r, prePP, c),
  Before(pp, prePP), After(pp, postPP).

FldPointsToFI(bh, bhc, NR, fld, h, hc, r) ←
  Store(base, fld, from, pp),
  VarPointsTo(from, c, h, hc, r, prePP, c), VarPointsTo(base, c, bh, bhc, NR, prePP, c),
  Before(pp, prePP).

FldPointsToFS(bh, bhc, R, fld, h, hc, r, postPP, c) ←
  Store(base, fld, from, pp),
  VarPointsTo(from, c, h, hc, r, prePP, c), VarPointsTo(base, c, bh, bhc, R, prePP, c),
  Before(pp, prePP), After(pp, postPP).
```

Figure 3.10: Rules for move, move-null, field load and field store

MERGE(*h*, *hc*, *r*, *callSite*, *callerC*) = *calleeC*,
VARPOINTSTO(*this*, *calleeC*, *h*, *hc*, *r*, *postEntry*, *calleeC*),
REACHABLE(*toMeth*, *calleeC*),
CALLGRAPH(*callSite*, *callerC*, *toMeth*, *calleeC*) ←
  VCALL(*base*, *sig*, *callSite*),
  INMETH(*callSite*, *meth*), REACHABLE(*meth*, *callerC*),
  VARPOINTSTO(*base*, *callerC*, *h*, *hc*, *r*, *preCallSite*, *callerC*),
  HEAPTYPE(*h*, *t*), LOOKUPMETH(*t*, *sig*, *toMeth*), THISVAR(*toMeth*, *this*),
  METHENTRY(*toMeth*, *ppEntry*), AFTER(*ppEntry*, *postEntry*), BEFORE(*preCallSite*, *callSite*).

MERGESTATIC(*callSite*, *callerC*) = *calleeC*,
REACHABLE(*toMeth*, *calleeC*),
CALLGRAPH(*callSite*, *callerC*, *toMeth*, *calleeC*) ←
  SCALL(*toMeth*, *callSite*), INMETH(*callSite*, *meth*),
  REACHABLE(*meth*, *callerC*).

VARPOINTSTO(*to*, *calleeC*, *h*, *hc*, *r*, *postEntry*, *calleeC*) ←
  CALLGRAPH(*callSite*, *callerC*, *meth*, *calleeC*),
  FORMALARG(*meth*, *i*, *to*), ACTUALARG(*callSite*, *i*, *from*), BEFORE(*callSite*, *preCallSite*),
  VARPOINTSTO(*from*, *callerC*, *h*, *hc*, *r*, *preCallSite*, *callerC*),
  METHENTRY(*toMeth*, *ppEntry*), AFTER(*ppEntry*, *postEntry*).

VARPOINTSTO(*to*, *callerC*, *h*, *hc*, *r*, *postCallSite*, *callerC*) ←
  CALLGRAPH(*callSite*, *callerC*, *meth*, *calleeC*),
  FORMALRETURN(*meth*, *from*), ACTUALRETURN(*callSite*, *to*), AFTER(*callSite*, *postCallSite*),
  VARPOINTSTO(*from*, *calleeC*, *h*, *hc*, *r*, *preExit*, *calleeC*),
  METHEXIT(*toMeth*, *ppExit*), BEFORE(*ppExit*, *preExit*).

Figure 3.11: Rules for virtual and static calls

Control Flow. In Figure 3.12, the two rules propagate the points-to information between two consecutive program points. More specifically, the first rule states that if program point *pp2* is an immediate successor in the control flow graph to program point *pp1*, and if object field (*bh, bhc, r, fld*) points to something immediately after program point replica (*pp1, c*), then object field (*bh, bhc, r, fld*) points to the same thing immediately before program point replica (*pp2, c*). The same holds for reference variable replica (*var, c*).

FLDPOINTSTOFS(*bh, bhc, br, fld, h, hc, hr, prePP2, c*) ←
  FLDPOINTSTOFS(*bh, bhc, br, fld, h, hc, hr, postPP1, c*),
  PRED(*pp1, pp2*), BEFORE(*pp2, prePP2*), AFTER(*pp1, postPP1*).

VARPOINTSTOFS(*var, c, h, hc, r, prePP2, c*) ←
  VARPOINTSTOFS(*var, c, h, hc, r, postPP1, c*),
  PRED(*pp1, pp2*), BEFORE(*pp2, prePP2*), AFTER(*pp1, postPP1*).

Figure 3.12: Rules for control flow

In Figure 3.13, the first two rules state that, flow-sensitive points-to sets are preserved over method calls and returns. That is, points-to facts that hold immediately before a method invocation hold immediately after entry to the method, and points-to facts that hold immediately before method exit also hold immediately after the method invocation. The same applies to static fields, which, in our implementation, are the only source for flow-sensitive reference variables. Thus, for simplicity, our rules state that the points-to information of all flow-sensitive variables are preserved over calls. Note that, this preservation might not be necessary if other kinds of reference type variables are tracked flow-sensitively. In that case, a predicate ISSTATICFIELD might be helpful in distinguishing whether propagation into callees in necessary.

Flow-insensitivity and Recency. It is important to note how the recency abstraction interacts with flow-insensitive points-to sets. In particular, as stated in the last rule in Figure 3.13, whenever a flow-insensitive object field points to a most-recent object, it should also point to the non-most recent version of the same abstraction. The reason is because, for any flow-insensitive object field, its

FLDPOINTSTOFS(*bh, bhc, br, fld, h, hc, hr, postEntry, calleeC*) ←
FLDPOINTSTOFS(*bh, bhc, br, fld, h, hc, hr, preCallSite, callerC*),
CALLGRAPH(*callSite, callerC, toMeth, calleeC*),
METHENTRY(*toMeth, ppEntry*),
BEFORE(*callSite, preCallSite*), AFTER(*ppEntry, postEntry*).

FLDPOINTSTOFS(*bh, bhc, br, fld, h, hc, hr, postCallSite, calleeC*) ←
FLDPOINTSTOFS(*bh, bhc, br, fld, h, hc, hr, preExit, calleeC*),
CALLGRAPH(*callSite, callerC, toMeth, calleeC*),
METHEXIT(*toMeth, ppExit*),
BEFORE(*ppExit, preExit*), AFTER(*callSite, postCallSite*).

VARPOINTSTOFS(*var, c, h, hc, hr, postEntry, calleeC*) ←
VARPOINTSTOFS(*var, c, h, hc, hr, preCallSite, callerC*),
CALLGRAPH(*callSite, callerC, toMeth, calleeC*),
METHENTRY(*toMeth, ppEntry*),
BEFORE(*callSite, preCallSite*), AFTER(*ppEntry, postEntry*).

VARPOINTSTOFS(*var, c, h, hc, hr, postCallSite, calleeC*) ←
VARPOINTSTOFS(*var, c, h, hc, hr, preExit, calleeC*),
CALLGRAPH(*callSite, callerC, toMeth, calleeC*),
METHEXIT(*toMeth, ppExit*),
BEFORE(*ppExit, preExit*), AFTER(*callSite, postCallSite*).

FLDPOINTSTOFI(*bh, bhc,* `NR`*, fld, h, hc,* `NR`) ←
FLDPOINTSTOFI(*bh, bhc,* `NR`*, fld, h, hc,* `R`).

Figure 3.13: Additional rules for control flow

points-to set should serve as a summary for the entire program (i.e., points-to relations may hold at any program point): if the object field points to a most-recent object *o* at any inter-program point,

including the one right before the allocation of *o*, it should then point to the non-most-recent version of *o* after the allocation.

Field Preserve. Field preserve is one of the more interesting cases in a flow-sensitive analysis. The inference rules listed in Figure 3.14 aim to propagate the points-to set of an object field *across* a statement. Rather than directly copying the entire points-to set, it is important to note two possible changes: 1) When the base only points to one abstract object, and the object is most-recent, strong update can be performed and no propagation is needed. In the following inference rules, this is carried out by identifying the kill-fields at a program point, and avoid propagating the points-to set of these fields across the program point. More specifically, if KILLFIELD(*bh*, *bhc*, R, *fld*, *pp*, *ppc*) then the field store at program point replica (*pp*, *ppc*) strongly updates object field (*bh*, *bhc*, R, *fld*). 2) The recency of an object can change from most-recent to non-most-recent across a statement that allocates the same abstract object. This applies to both the base as well as any object in the points-to set, thus there are in total four cases to consider.

If no relevant allocations occur, then object field (*bh*, *bhc*, R, *fld*) continues to point to (*h*, *hc*, *r*) immediately after the program point replica. If the program point replica allocates an object with allocation site *h* and heap context *hc*, then what was the most-recently allocated object for *h* and *hc* is no longer the most recent, and so the object field points to (*h*, *hc*, NR) after the program point replica instead. If the program point replica allocates an object with allocation site *bh* and heap context *bhc* then *h*, *hc*, *r* is in the points-to set of object field (*bh*, *bhc*, NR, *fld*), which we track flow insensitively. The last rule is only applicable when the above two special cases hold simultaneously.

Reference Variable Preserve. The rules we present in Figure 3.15 are specific to a setting in which flow-sensitive reference variables all correspond to one concrete memory location. This allows strong updates to happen upon any LOAD, MOVE, MOVENULL, or ALLOC instruction. Analogous to field preserve, a most-recent abstract object (*h*, *hc*, R) needs to be converted to its corresponding non-most-recent version during propagation if the statement under consideration is an allocation site *h* with heap context *hc*.

KillField(*h*, *hc*, R, *fld*, *pp*, *c*) ←
  Store(*base*, *fld*, *from*, *pp*), VarPointsTo(*base*, *c*, *h*, *hc*, R),
  count(VarPointsTo(*base*, *c*, _, _, _)) ≤ 1.

FldPointsToFS(*bh*, *bhc*, R, *fld*, *h*, *hc*, *r*, *postPP*, *c*) ←
  FldPointsToFS(*bh*, *bhc*, R, *fld*, *h*, *hc*, *r*, *prePP*, *c*),
  not KillField(*bh*, *bhc*, R, *fld*, *pp*, *c*),
  not ObjectJustAllocated(*h*, *hc*, *pp*, *c*), not ObjectJustAllocated(*bh*, *bhc*, *pp*, *c*),
  Before(*pp*, *prePP*), After(*pp*, *postPP*) .

FldPointsToFS(*bh*, *bhc*, R, *fld*, *h*, *hc*, NR, *postPP*, *c*) ←
  FldPointsToFS(*bh*, *bhc*, R, *fld*, *h*, *hc*, *r*, *prePP*, *c*),
  not KillField(*bh*, *bhc*, R, *fld*, *pp*, *c*),
  ObjectJustAllocated(*h*, *hc*, *pp*, *c*), not ObjectJustAllocated(*bh*, *bhc*, *pp*, *c*),
  Before(*pp*, *prePP*), After(*pp*, *postPP*).

FldPointsToFI(*bh*, *bhc*, NR, *fld*, *h*, *hc*, *r*) ←
  FldPointsToFS(*bh*, *bhc*, R, *fld*, *h*, *hc*, *r*, *prePP*, *c*),
  not KillField(*bh*, *bhc*, R, *fld*, *pp*, *c*),
  not ObjectJustAllocated(*h*, *hc*, *pp*, *c*), ObjectJustAllocated(*bh*, *bhc*, *pp*, *c*),
  Before(*pp*, *prePP*).

FldPointsToFI(*bh*, *bhc*, NR, *fld*, *h*, *hc*, NR) ←
  FldPointsToFS(*bh*, *bhc*, R, *fld*, *h*, *hc*, *r*, *prePP*, *c*),
  not KillField(*bh*, *bhc*, R, *fld*, *pp*, *c*),
  ObjectJustAllocated(*h*, *hc*, *pp*, *c*), ObjectJustAllocated(*bh*, *bhCTX*, *pp*, *c*),
  Before(*pp*, *prePP*).

Figure 3.14: Rules for preservation of fields' points-to information

KillVar(*to*, *pp*) ← Load(*to*, *base*, *fld*, *pp*).

KillVar(*to*, *pp*) ← Move(*to*, *from*, *pp*).

KillVar(*to*, *pp*) ← MoveNull(*to*, *pp*).

KillVar(*var*, *pp*) ← Alloc(*var*, *h*, *pp*).

VarPointsToFS(*var*, *c*, *h*, *hc*, *r*, *postPP*, *c*) ←
  VarPointsToFS(*var*, *c*, *h*, *hc*, *r*, *prePP*, *c*),
  not KillVar(*var*, *pp*),
  not ObjectJustAllocated(*h*, *hc*, *pp*, *c*),
  Before(*pp*, *prePP*), After(*pp*, *postPP*).

VarPointsToFS(*var*, *c*, *h*, *hc*, `NR`, *postPP*, *c*) ←
  VarPointsToFS(*var*, *c*, *h*, *hc*, *r*, *prePP*, *c*),
  not KillVar(*var*, *pp*),
  ObjectJustAllocated(*h*, *hc*, *pp*, *c*),
  Before(*pp*, *prePP*), After(*pp*, *postPP*).

Figure 3.15: Rules for preservation of reference variables' points-to information

Chapter 4

Implementation Overview

The analysis presented in Chapter 3 is implemented for Java bytecode, and is built upon the Watson Libraries for Analysis (WALA) framework [4]. First, we briefly introduce the WALA framework and Java bytecode. Next, the representation of the points-to graph is discussed. Specifically, we focus on the data structure used to record the points-to sets for flow-sensitive reference variables and object fields. The representation strongly depends on reachability queries, which are used to determine whether a program-point is reachable from one of the sources of the object under consideration. Lastly, we introduce the multi-threaded engine and discuss how dependencies are recorded and utilized to speed up the solver.

It is important to note that we handle all features of Java except for concurrency and reflection. Java signatures are provided for some important native methods (e.g., `System.arrayCopy`), as well as a handful of standard library classes (e.g., `java.lang.StringBuilder`) for efficiency and precision. For native methods without signatures, we assume that the method allocates an instance of the return type. These are potential sources of unsoundness in our analysis.

4.1 Java Bytecode and the WALA Framework

Our implementation of the flow-sensitive points-to analysis is built upon the existing T. J. Watson Libraries for Analysis (WALA) framework. This open-sourced library parses and translates the stack-based Java bytecode into a register-based intermediate language in Static Single Assignment (SSA) form [6]. Java bytecode is the machine language for Java virtual machine (JVM); it is often used as the base language for program analysis because it is simpler than normal Java code, but is also a full-feature language.

The WALA *IR* (intermediate representation) class provides a SSA-based representation of each method. An IR consists of a control flow graph of basic blocks, along with the set of instructions in each basic block. Each instruction is of class *SSAInstruction*, and can be one of the many predefined types, including *SSANewInstruction*, *SSAFieldAccessInstruction*, etc. Each IR also provides a type-inference tool, which is used to retrieve the inferred type of variables in the IR.

4.2 Representation of the Points-To Graph

The core storage structure of the points-to graph consists of two maps: *pointsToFI* and *pointsToFS*, representing the correspondence between *points-to graph nodes* (abbreviated *nodes*) and their respective points-to set. A points-to graph node can either be a reference variable replica or an object field. The former encapsulates a reference variable and a context information, and the latter identifies a specific field of an abstract object. The elements in the points-to set are abstract objects, which contain three pieces of information: the allocation site, the heap context, and the recency information.

The map *pointsToFI* tracks flow-insensitive nodes. More specifically, this includes the reference variables replicas that can be treated flow-insensitively, and also the fields of non-most-recent abstract objects. For any given node in this map, the corresponding value is simply the set of abstract objects the node can possibly point to at any point in the program. The map *pointsToFS* is more complicated; it tracks flow-sensitive nodes, which include reference variable replicas for static fields and the fields of most-recent abstract objects. The points-to information now needs to also account for program points, because the points-to set depends on the location in the control flow graph. This can be accomplished by associating with each flow-sensitive node n and abstract object o a set of inter-program point replicas $s_{n,o}$, such that if inter-program point replica (ipp, c) is in $s_{n,o}$, then n points to o at (ipp, c).

A program can have thousands of inter-program points (and thus inter-program point replicas),

and it is very space inefficient to enumerate them for each points-to relation between nodes and abstract objects. Even though a points-to relation won't be valid at all possible inter-program point replicas, the object-field points-to sets are usually propagated between different methods and can accumulate to a huge set of inter-program points replicas. To reduce the space needed to store them while not losing accuracy, we designed a novel representation for inter-program point replica sets called *program-point closure*. A program-point closure contains an explicit set of *sources*: inter-program point replicas at which the points-to relation between n and o is established (i.e., immediately following field stores that add o to the points-to set of n). An inter-program point replica is *implicitly* included in the set if there is a path from a source to the inter-program point replica that does not go through a program point replica that either allocates o, allocates n, or performs a strong update on n. At these implicitly-defined inter-program point replicas, the points-to relation between n and o should also hold.

Because many operations in the implementation involve the creation and usage of maps and sets, we use integer substitutions to increase the processing speed. In other words, there are dictionaries that map nodes to and from integers, and the same for abstract objects. The above maps can now be simplified to integer maps (the subscript indicates the actual type substituted by integer):

$$\begin{aligned}
&\mathit{pointsToFI} : \mathbf{int}_{node} \rightarrow \mathbf{set} < \mathbf{int}_{abstract\ object} > \\
&\mathit{pointsToFS} : \mathbf{int}_{node} \rightarrow \mathbf{int}_{abstract\ object} \rightarrow \mathit{program\ point\ set\ closure}
\end{aligned}$$

4.3 Program-Point Reachability Query

As described in the previous section, a *program-point colusre* is a representation of a set of inter-program point replicas at which a node can point to a specific abstract object. Instead of enumerating all inter-program point replicas at which the points-to relation is valid, this closure only records several *sources*. When the points-to set for a node is requested at a particular inter-program point replica, an iterator is returned. When the iterator computes the next element it skips abstract objects in the underlying set that fails the *program-point reachability query*. A reachability query determines whether the inter-program point replica under consideration is reachable from one of the sources of

the object without going through a program point replica that allocates a relevant abstract object or kills a relevant object field.

To efficiently answer these reachability queries, we compute a *method summary* for each method m that is analyzed in context c. Specifically, we compute and memoize the abstract objects allocated and the object fields killed on all paths from the entry program point replica to the exit program point replica for each method-context pair. That is, an abstract object o will only be in the summary if it is allocated on all possible paths from the entry to exit; the same applies for killed object fields. Since the method may invoke other methods, method summaries of the callees are required for the computation. In addition, we also compute the method-context pairs that are reachable from m in context c, and the allocated abstract objects and killed fields on all paths from the entry of m to the entry of each of the reachable methods.

Method summaries are very helpful in determining whether an inter-program point replica is reachable from another without going through relevant allocations and kills. In particular, whenever a method call is encountered during the reachability query, we can efficiently retrieve the callee's potential influence on the points-to information. Note that the computation of method summaries is iterated until a fixed point is reached, and is constantly updated as more call relations are discovered in the call graph. However, our experiments show that reachability queries greatly utilize these summaries, thus can be considered a reasonable tradeoff.

4.4 Context-Sensitivity Choice

As described in Chapter 3, we utilize a flexible context-sensitivity framework that abstracts the context using three constructors MERGE, MERGESTATIC and RECORD [8, 15]. We implemented several abstractions introduced, and concluded that type sensitivity provides the highest scalability. However, we realized that a type-sensitive analysis does not fully support strong update, which is how we gain the most precision in a flow-sensitive analysis. More specifically, in order to perform strong update upon a field store $base.f = from$, we need to soundly conclude that *base* points to one single most-recent abstract object. Unfortunately, this is generally not the case for a type-sensitive analysis.

To solve this issue, we designed a novel abstraction that provides full-object sensitivity for constructor calls and type sensitivity for everything else. The former ensures that the receiver of the constructor call is always a single most-recent abstract object, and the latter contributes to the scalability of the analysis.

4.5 Multi-Threaded Dependency-Driven Engine

Our points-to analysis engine is implemented as a work queue algorithm. The queue is managed using Java's fork/join framework, which is implemented based on a work-stealing algorithm [9]. Work-stealing is a scheduling strategy for multi-threaded computer systems. Each processor has its own queue of work items, which can spawn more work items during execution. When a processor runs out of items in its own queue, it *steals* items from other processors. This avoids overloading a few processors and leaving the others hanging. Compared to a work-sharing framework, where each task is assigned to a processor when it is spawned, work-stealing reduces the amount of task migration between processors, as no migration is needed when all processors are occupied.

There are two strategies we employ to reduce the number of times each input statement is processed. Firstly, subset constraints are produced when statements are processed the first time, and these constraints are then used to propagate changes of a points-to set to its known supersets. For instance, upon a MOVE(*to, from, pp*) statement, we record a relation where the points-to set of *to* after *pp* is a superset of the points-to set of *from* prior to *pp*. In the following iterations, whenever the points-to set of *to* at post-*pp* changes, the additional elements are copied to the relevant supersets. It is important to note that flow-sensitive statements produce subset constraints that are conditional on program point reachability queries. Essentially, these constraints, along with the conditions, represent the transitive closure of CONTROLFLOW and PRESERVE.

Secondly, the engine also performs dependency tracking. If processing a statement requires reading the points-to set of a reference variable or object field, a dependency between the statement and the variable or field is recorded. In future iterations, if the points-to set of the variable or field changes, the change will be propagated via reprocessing of the statement. To avoid repeatedly evaluating the same information, we incorporate *difference propagation*, in which only the changes will be evaluated when revisiting the statement [7]. For example, processing the constraints for LOAD and STORE depends on the points-to set of the receiver, and thus creates dependency between the receiver node and the statement. When there are additional elements in the points-to set of the receiver, the statements will be reprocessed and new subset constraints might be generated, but only the new elements are involved.

Chapter 5

Evaluation

In this chapter, we evaluate the computation time of our points-to analysis and precision improvement that it makes on client analyses. The programs that we apply the analysis to are in the DaCapo Benchmark suite [2], and all evaluations are done on a 32-core Amazon EC2 instance with 244GB of RAM. In Section 5.1, we present and discuss the performance of the points-to analysis. In the rest of the chapter, we introduce three commonly used client analyses: a non-null analysis, a dynamic cast removal analysis, and an interval analysis. We run these analyses on the benchmark suite programs, and the precision of each is compared between when a flow-sensitive points-to analysis is used and when a flow-insensitive analysis is used. To demonstrate the power of strong updates, we also show the precision data for cases where strong update is temporarily disabled in the points-to analysis.

5.1 Points-To Analysis Performance

To test the correctness and evaluate the performance of our points-to analysis implementation, we apply our analysis to several applications in the DaCapo benchmark suite (version 2006-10-MR2). The suite is intended for Java benchmarking, and consists of many open-source applications with non-trivial memory usage. Each program is analyzed with all the library code including JDK version 1.6. The number of lines of code and WALA instructions are estimates and include the relevant library code. All evaluations are performed on a 32-core Amazon EC2 instance with 244GB of RAM, and each program is analyzed 10 times.

Program	Lines of code	SSA Instructions	Mean (s)	Std Dev
antlr	62,750	79,395	55.11	4.08
eclipse	130,236	123,860	91.67	3.21
fop	74,621	88,348	62.13	3.44
hsqldb	46,562	46,375	10.25	0.72
luindex	104,306	104,774	19.22	2.64
lusearch	49,885	50,121	9.61	1.31
pmd	127,351	145,389	25.17	1.10
xalan	64,754	71,482	14.76	1.11

Figure 5.1: Flow-sensitive points-to analysis performance.

Figure 5.1 shows the mean performance of our analysis. The amount of time needed to compute the analysis, as expected, is significantly worse than a flow-insensitive points-to analysis. However, they are still very reasonable: the slowest program (*eclipse*) takes around 1.5 minutes. The precision gained may be a desirable tradeoff.

We also evaluated how our analysis scales with the number of processing threads. We continued to use the 16-core machine, but limited the number of working threads. We can see from Figure 5.2 that the speedup (the ratio between the time needed for a single-threaded analysis to a multi-threaded analysis) shows an upward trend. However, for some smaller programs, the speedup actually decreases with increasing thread count. This is most likely due to low CPU utilization: not all threads are occupied and thread contention is more apparent.

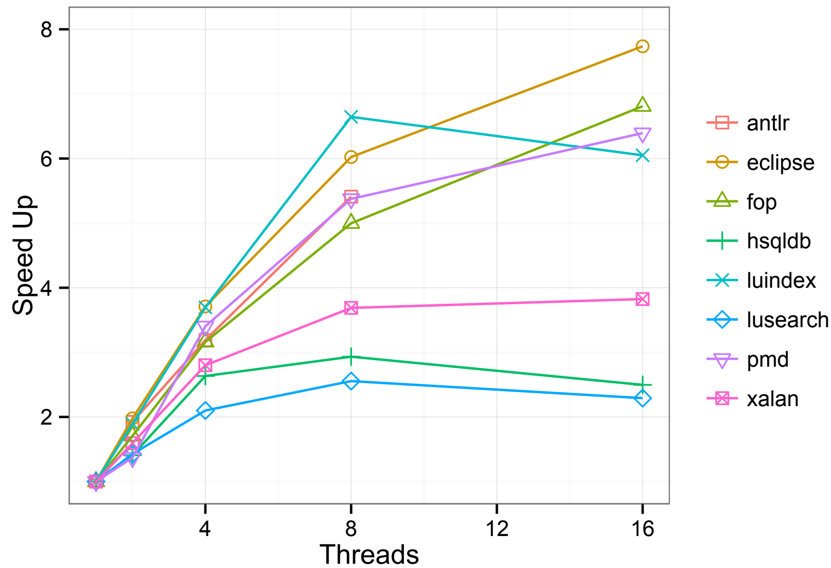

Figure 5.2: Performance of the flow-sensitive points-to analysis vs. number of processing threads.

5.2 Client Analysis Framework : Data-Flow Analysis

The first two client analyses that we present, a non-null analysis and an interval analysis, are based on a data-flow framework. *Data-flow analysis* gathers information about a pre-defined set of value at various points in a program. It analyzes the effect of each basic block, and composes effects of multiple basic blocks to derive information at the boundaries. How the data propagates between blocks is determined by the program's control flow graph (CFG), which represents all the paths that might be traversed through during execution. Many properties of a program can be computed using a data-flow analysis; existing analyses include the reaching definition analysis, live variable analysis, non-null analysis and interval analysis, etc. We will be implemented the latter two as clients of our points-to analysis.

A data-flow problem is usually solved using an iterative approach. The transfer function of each node in the control flow graph, i.e. the equation that relates the output data of a basic block to the input, is first determined. Then, all the equations are repeatedly examined until no new information is derived and a fixpoint is reached. In order for convergence to be guaranteed, a few constraints are usually imposed: 1) The value domain should be ordered and of finite height to avoid infinite

ascending or descending through the values. 2) The transfer functions and the join operation (the operation that compiles information at phi nodes) should compositely result in a monotonic function. This prevent oscillations of values at some program points.

5.3 Client I : Non-Null Analysis

Many programming languages allow the storage of null values in reference variables or fields. This causes possible `NullPointerException`s upon dereferencing. In other words, operations that work on receivers, including field accesses and method calls, etc., require the receiver to be non-null for the dereferencing to be safe; otherwise, a runtime exception occurs and the JVM exits if the exception is not handled. To avoid encountering these errors in run-time, it would be very helpful if they can be caught during compilation. This prompts the design of non-null analyses, which determine whether each reference variable and object field can be null at various program points. There are more benefits if this data can be computed statically: 1) JVM dynamically checks operations that involve dereferencing pointers. Knowing the nullness of the receiver can potentially lead to the removal of these checks and increase the efficiency of the program. 2) The control flow graph can be greatly simplified if some exceptional edges are known to be dead (impossible to reach). This in turn improves the accuracy of other static program analyses.

Many static non-null analyses have been developed in the past. One complexity that often arises is that, in some modern object-oriented languages like Java, the fields of object instances are initialized to null. Unless flow sensitivity and strong update are enabled, these object fields, by default, may be null, which often defeats the purpose of the analysis, because what we are interested in is whether a variable or field is non-null when it is dereferenced. Recall the example in Figure 2.4, strong update allows us conclude that $a.f$ is non-null at program point $p3$, which is a more precise and useful result.

We evaluate the precision of non-null analyses by comparing the percentage of null-pointer exceptions that can be proven to be impossible. In Figure 5.3, the first column shows this percentage when the result of our flow-sensitive points-to analysis is used. The second column still uses a flow-sensitive points-to analysis but does not carry out strong update, i.e. all object fields are weakly updated in the points-to analysis. The third column simply uses the result of a flow-insensitive points-to analysis.

We can see that, although the percentage of exceptions proved impossible are on very high in each configuration, there is still a substantial difference. The flow-sensitive analysis without strong update

Program	Percent of possible NPE proved impossible		
	Flow-sensitive w/ Strong Update	Flow-sensitive no Strong Update	Flow-insensitive
antlr	91.95%	84.93%	84.85%
eclipse	91.00%	86.96%	83.51%
fop	93.39%	88.21%	86.90%
hsqldb	94.24%	89.97%	89.15%
luindex	92.25%	87.36%	83.39%
lusearch	95.16%	90.47%	89.18%
pmd	92.82%	88.24%	85.63%
xalan	94.55%	88.24%	88.12%
TOTAL	92.88%	87.95%	85.72%

Figure 5.3: The precision of the non-null analysis as measured by the percent of possible `NullPointerException`s (e.g. non-static method calls) that can be proved impossible by the analysis.

enables the removal of an additional 2% of exceptions, compared to using a flow-insensitive analysis. This mainly results from a more precise points-to graph. Using the results of our flow-sensitive points-to analysis with strong update shows an even larger precision improvement: compared to the flow-insensitive analysis, it leaves less than half as many locations at which a manual check of nullness is needed. The difference between the flow-sensitive analysis with and without strong update shows the benefit of the recency abstraction, which is what enables client analyses to usefully employ strong update.

5.4 Client II : Interval Analysis

Interval analysis computes the possible value range of numeric variables. More specifically, the goal of the analysis is to create, at each program point, a map from numeric-type reference variables or object fields to their conservative boundaries. Similar to a non-null analysis, the result of an interval analysis can be used in many other compile-time optimizations. For instance, when operating with conditional branches that depend on numeric comparisons, interval information can help determine if a branch is unreachable. In terms of the safety of programs, interval analysis can lead to the finding of potential buffer-overflow vulnerabilities.

In many optimizations that utilize the interval analysis result, the most useful piece of information is whether a variable or object-field can be zero at a given program point. This raises a similar concern as the non-null analysis: all the primitive variables are automatically set to zero prior to initialization,

and thus strong update is needed to avoid the inclusion of zero by default.

Many interval analyses have been designed to serve a wide range of purposes. Here we will discuss a few design choices for our implementation. First, in order to account for all numeric data types, we use floating point values as the boundaries. We also ignore the inclusivity/exclusivity of the boundaries. However, we understand that this will sacrifice the accuracy of some more constrained data types, like integer and booleans. (For instance, if *integer* x has range $[1,3]$ prior to a conditional branch $x \neq 3$, a more precise analysis should conclude that $x \in [1,2]$ in the branch. With floating-point boundaries and when inclusivity is not under consideration, we are not able to make the same conclusion.) Thus, we include a *containsZero* flag, allowing the removal of zero in an interval that spans across zero. We hope that this reduces the number of zero-containing intervals in our analysis result.

Secondly, although the maximum and minimum doubles limit the value domain to a finite lattice, the program might still take an infeasible amount of time to reach the boundaries. For example, given a while loop that indefinitely adds 1 to a variable that starts with value zero, it will take at least 2^{1023} iterations for the interval to reach the maximum double and stabilize. To avoid this, we limit the value domain to subintervals of $[-10,10]$, and treat bounds above and below this range as positive infinity and negative infinity, respectively. There are more rigorous ways of approaching this problem. One of them is defining discrete and exponentially incrementing bounds beyond the precisely-tracked range. For example, when the upper boundary is above 10, the possible values are limited to $2^4, 2^5, 2^6$... (the upper bound is still tracked accurately when between $[-10,10]$), and the same applies to the lower boundary. This greatly increases the convergence speed and avoids loops that repeatedly make small increments to the intervals.

To compare the precision of our interval analysis, we record the percentage of intervals that the analysis determines cannot contain zero. Results are shown in Figure 5.4. An additional 5% of intervals that cannot contain zero is determined using a flow-sensitive points-to analysis with strong update. The performance of the interval analysis using a flow-sensitive points-to analysis with and without strong update is essentially equivalent.

5.5 Client III : Cast Removal

Cast removal is the simplest of all as it is not based on a data-flow analysis. It consists of a single pass over the code, which checks if each casting is necessary. More specifically, given an instruction

Program	Percent of intervals that do not contain zero		
	Flow-sensitive w/ Strong Update	Flow-sensitive no Strong Update	Flow-insensitive
antlr	62.19%	62.02%	59.15%
eclipse	62.98%	62.45%	58.11%
fop	58.83%	58.08%	57.40%
hsqldb	63.22%	63.14%	55.40%
luindex	58.51%	57.94%	55.20%
lusearch	61.55%	60.89%	55.62%
pmd	64.81%	64.14%	53.70%
xalan	57.06%	54.75%	53.88%
TOTAL	61.06%	60.64%	56.06%

Figure 5.4: The precision of the interval analysis as measured by the percent of intervals that contain zero.

Program	Percent of dynamic casts that can be proven unnecessary		
	Flow-sensitive w/ Strong Update	Flow-sensitive no Strong Update	Flow-insensitive
antlr	58.55%	52.71%	47.27%
eclipse	60.39%	47.99%	45.51%
fop	57.03%	49.81%	50.61%
hsqldb	59.69%	49.64%	52.26%
luindex	61.57%	53.72%	48.69%
lusearch	62.20%	54.01%	50.00%
pmd	66.54%	52.88%	35.44%
xalan	58.79%	58.12%	50.23%
TOTAL	60.72%	51.65%	44.73%

Figure 5.5: The precision of a cast removal analysis measured by the percent of dynamic casts that are always allowed.

that casts variable v to type T, if every abstract object in the the points-to set of v are subtypes of T, then the cast will never fail and it is safe to remove the `checkcast` instruction that is generated. We evaluate the precision of the cast-removal analysis with the percentage of casts that can be removed using the points-to information generated. Figure 5.5 shows that our analysis removes 60.72% of the casts across all programs, which is over 35% more than the flow-insensitive analysis and 17% more than the flow-sensitive analysis without strong update. This is the most significant precision improvement among the three client analyses. Note that, for two of the programs, the flow-sensitive analysis without strong update performs slightly worse than the flow-insensitive analysis. This is due to a more precise call graph, which implies that there are fewer class casts that could potentially be removed.

Chapter 6

Conclusion

We developed a flow-sensitive, context-sensitive points-to analysis for object-oriented programs. We incorporated the recency abstraction to enable strong update, both in the points-to analysis and client analyses. We showed that our analysis is scalable and significantly improves the precision of non-null analysis, interval analysis and cast-removal analysis, compared a flow-insensitive analysis. Since our analysis enables strong update during object construction, it is particularly useful for client analyses that benefit from precise reasoning about object invariants established upon object construction.

Bibliography

[1] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis*, pages 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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 Programing, Systems, Languages, and Applications*. ACM Press, 2006.

[3] Martin Bravenboer and Yannis 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*. ACM, 2009.

[4] IBM T. J. Watson Research Center. T. J. Watson Library for Analysis (WALA). http://wala.sf.net, 2006-2015. Accessed: 2015-02-20.

[5] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 296–310, New York, NY, USA, 1990. ACM.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[7] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nordic Journal of Computing*, 1998.

[8] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[9] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[10] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, 2011.

[11] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):3:1–3:53, October 2008.

[12] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[13] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[14] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[15] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, New York, NY, USA, 2011. ACM.