NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES PROGRAM

MASTER THESIS

The Role of Exceptions in Static Program Analysis for Java

George Kastrinis

Supervisor: **Yannis Smaragdakis**, Associate Professor NKUA

ATHENS

JULY 2012

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Η σημασία των Εξαιρέσεων στη Στατική Ανάλυση Προγραμμάτων Java

Γιώργος Καστρίνης

Επιβλέπων: **Γιάννης Σμαραγδάκης**, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2012

MASTER THESIS

The Role of Exceptions in Static Program Analysis for Java

George Kastrinis

RN: M1054

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Η σημασία των Εξαιρέσεων στη Στατική Ανάλυση Προγραμμάτων Java

Γιώργος Καστρίνης

ΑΜ: Μ1054

ΕΠΙΒΛΕΠΩΝ :

Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:

Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ
Παναγιώτης Ροντογιάννης, Αναπληρωτής Καθηγητής ΕΚΠΑ

Περίληψη

Η στατική ανάλυση προγραμμάτων αποτελεί ένα είδος ανάλυσης λογισμικού, που ασχολείται με την εξέταση του πηγαίου κώδικα χωρίς να εκτελείται το παραγόμενο πρόγραμμα. Μια σημαντική κατηγορία στατικής ανάλυσης είναι η ανάλυση δεικτών, η οποία εκτιμά σε ποιά αντικείμενα μπορεί να 'δείξει' κάθε μεταβλητή του προγράμματος για *κάθε* πιθανή εκτέλεση του κώδικα. Τα αποτελέσματα αυτά αποτελούν θεμελιώδες στάδιο για περαιτέρω πιο σύνθετες αναλύσεις.

Για να είναι μια τέτοια ανάλυση ακριβής, πρέπει να προσομοιώνει κάθε πτυχή του προγράμματος καθώς και του συστήματος στο οποίο θα εκτελεστεί ο κώδικας, και που επηρεάζουν την ροή των αντικειμένων στις μεταβλητές του προγράμματος. Από την άλλη για να είναι μια τέτοια ανάλυση πρακτική χρειάζεται να παράγει αποτελέσματα σε λογικό χρονικό διάστημα. Σαν αποτέλεσμα είναι αρκετά κρίσιμο οι υπερεκτιμήσεις που γίνονται να είναι όσο το δυνατόν πιο 'σφικτές'.

Ένα σημαντικό χαρακτηριστικό των αντικειμενοστρεφών γλωσσών όπως η Java είναι οι εξαιρέσεις. Προηγούμενες μελέτες [2] έδειξαν ότι ο ακριβής χειρισμός των εξαιρέσεων επηρεάζει σημαντικά την ακρίβεια των αποτελεσμάτων. Παρουσιάζουμε στην εργασία αυτή τρεις εναλλακτικές για το χειρισμό των εξαιρέσεων καθώς και τις επιπτώσεις που έχουν στην ακρίβεια και στην απόδοση της ανάλυσης. Ένα εντυπωσιακό εύρημα αποτελεί το γεγονός ότι αντί να καταγράφεται κάθε αντικείμενο-εξαίρεση, μπορούν τα αντικείμενα με τον ίδιο τύπο να συγχωνευτούν σε ένα αντικείμενο-αντιπρόσωπο. Κάτι τέτοιο επιφέρει ελάχιστη αλλαγή στην ακρίβεια, αλλά σημαντική βελτίωση στην απόδοση (σε αρκετές αναλύσεις πάνω από 20% βελτίωση).

Η ανάλυσή μας είναι μέρος του Doop framework [3], το οποίο περιλαμβάνει μία ανάλυση δεικτών για ένα σύνολο από υποστηριζόμενους τύπους συμφραζομένων, γραμμένο αποκλειστικά σε Datalog.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική Ανάλυση Προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ : Datalog, ανάλυση πλήρους προγράμματος, Java, εξαιρέσεις, συμφραζόμενα

Abstract

Static program analysis is the analysis of computer software that focuses on the examination of the source code, without actually executing the program built from that code. An important subclass of static program analysis is that of Points-To Analysis, an analysis that reasons about which objects can flow into which variables, for *every* possible program execution. The points-to results, are fundamental for further, more complex analyses.

For an analysis like the above to be precise, it has to simulate every aspect of the source code and of the underlying system in which the program will be executed, that can influence the flow of objects into variables. On the other hand, the results need to be produced within a logical timespan for the analysis to be practical. Thus it is crucial that every overapproximation made by the analysis is as "tight" as possible.

One important feature of object-oriented languages like Java is that of exceptions. Previous work [2] has shown that accurate handling of exceptions can significantly affect the precision of the results. In this work, we present three alternative ways to handle exceptions in Java, as well as the effect each one has over the precision and the performance of the resulting analysis. An impressive find is the fact that, instead of recording each distinct exception object, we can collapse all exceptions of the same type, and use one representative object per type, with barely any loss in precision but at the same time with a significant boost in performance (in many analyses achieving more than 20% improvement).

Our analysis is part of the Doop framework [3], that provides a points-to analysis for a number of possible types of context, written entirely in Datalog.

SUBJECT AREA: Static Program Analysis

KEYWORDS: Datalog, whole-program analysis, Java, exceptions, context-sensitivity

Acknowledgements

I am grateful to my supervisor, Prof. Yannis Smaragdakis, whose expertise, guidance, and patience were decisive for this work. His valuable insights and vast knowledge on the field of static analysis were crucial throughout the research and writing of this thesis.

I would also like to thank my parents, to whom I am deeply indebted for their support.

Athens, July 2012

Contents

List of Figures

List of Tables

Preface

This report is my master thesis for the conclusion of my postgraduate studies at the Department of Informatics & Telecommunications, University of Athens. It was developed as a part of the PADECL Project for the University of Athens, while conducting research with Prof. Yannis Smaragdakis on advanced program analysis using declarative languages.

This work is built as part of the DOOP framework for pointer analysis using Datalog, and aims to demonstrate the capabilities of the Datalog language, the importance of how exceptions are handled in a points-to analysis and their effect on the precision and performance of the resulting analysis.

Athens, July 2012

Chapter 1

Introduction

Exceptions are the way modern programming languages offer for elegant error handling. The old-fashioned way of error handling comprised setting a flag variable to some value at the point where the error was encountered and afterwards passing that variable around (e.g., via function return) and checking at various points for that value in order to address the issue. In that way, error handling is interleaved with “normal” code and gets mixed up with the normal control flow intended by the programmer, making error handling a tedious task. In addition, error handling can become optional as the programmer might easily ignore or forget to check the flag variable or the return value of a function call, thus allowing code to continue to run normally while in an erroneous state.

In contrast, by using exceptions as the mechanism for error handling, the programmer can write normal code, as was originally intended without the obstruction introduced by old-fashioned error handling. Afterwards, he can address the error handling itself in a separate section, thus resulting in more readable code. Furthermore, error handling cannot be ignored. Once an exception has occurred, it has to be addressed at some point of the code.

In general when normal control flow encounters an exceptional situation that it cannot address with the information available in the current context, it creates a data structure referred to as *exception*, that stores information about the error, and *throws* it out of the current context to an enclosing context. The context that *catches* the exception, executes afterwards code to address the error. If the exception is not caught inside the function it occurred, it goes out of the function, to its caller. If it is not handled there, it continues to the next enclosing dynamic scope, and so on until it is either caught or it has reached the entry point of the program in which case, execution is terminated because there was not exception handler found to address the error. In that way, errors cease to be simply an indication that something went wrong, and rather become a first-class component that needs to be handled appropriately.

Accurate handling of exceptions thus results in an accurate analysis overall, as they constitute a significant factor affecting the general control flow. In the context of a points-to analysis, as the ones provided by the Doop framework, exceptions contribute to the set

```java
public Object pop() {
  Object obj;

  if (size == 0)
    throw new EmptyStackException();

  obj = objectAt(size - 1);
  setObjectAt(size - 1, null);
  size--;
  return obj;
}
```

Figure 1.1: Example of Java exceptions. The “normal” flow for a `pop` action on a stack, is to return the object at the top. If `pop` is performed on an empty stack, an exception is thrown to signify the erroneous condition.

of objects that can flow into a program variable, which in turn determines which methods are reachable, which in turn determines which object can flow to other variables, with the process continuing recursively until fixpoint is reached.

It is common practice in points-to analysis to use contexts [11] as a way to qualify program variables and possibly object abstractions, in order to differentiate information that would otherwise collapse and thus attain higher precision. Three main kinds of context-sensitivity have been explored: *call-site sensitivity* [18, 19], *object sensitivity* [14] and *type sensitivity* [4].[1]

Specifically as far as exceptions are concerned, we have found three alternatives in combining exceptions with context, each one with pros and cons:

- Don’t distinguish “throwable” heap objects (objects that can be used as an exception in a *throw* expression) from “normal” heap objects, but instead use the full context that the current analysis assumes for heap objects.

- Treat throwable heap objects context-insensitively. That means that each path that leads to the allocation site of a specific throwable heap object, is collapsed and only one context is recorder per heap allocation site for each of the above heap objects.

[1]The context is typically a sequence of the N top call-sites of the calling stack (*call-site sensitivity*), or of the static abstractions of the receiver objects for the N top calling stack methods (*object sensitivity*) or of the types of those receiver objects (*type sensitivity*).

- Instead of representing each throwable heap object with an abstraction of its allocation site, a representative heap object is used, one specific for each particular class type. That way all throwable heap objects of the same type are collapsed to the same heap abstraction. Though in theory this should result in a loss in precision, we observed that in practice that is not the case, as exceptions are usually not treated the way that "normal" heap objects are. In our experiments there was barely any loss in precision, but at the same time a huge boost in performance.

The rest of the thesis is organized as follows. In Chapter 2 we give a background of points-to analysis in Datalog using the DOOP framework. In Chapter 3, we present how DOOP implements a joint points-to and exception analysis. In Chapter 4 we present the three alternative methods for treating exceptions that we suggest, as well as the insight behind each one. In Chapter 5, we present the evaluation of each method by testing it on the DaCapo benchmark suite for a variety of context-sensitive analyses, and comparing the results in each case, in terms of precision and performance. We conclude in Chapter 6.

Chapter 2

Background

Our analysis uses the DOOP framework [3], which provides a collection of points-to analyses (e.g., context insensitive, call-site sensitive, object sensitive, type sensitive). However, each alternative that we present regarding the treatment of exceptions and their combination with context, can be entirely oblivious to the exact choice of context (which is specified at runtime) due to the modular way that context is represented in the framework.

2.1 Points-To Analysis in Datalog

DOOP's primary defining feature is the use of Datalog for its analyses. Architecturally, however, an important aspect of DOOP's performance is that it employs an *explicit* representation of relations (i.e., all tuples of a relation are represented as an explicit table, as in a database), instead of using Binary Decision Diagrams (BDDs), which have often been considered necessary for scalable points-to analysis [22, 21, 12, 11].

DOOP uses a commercial Datalog engine, developed by LogicBlox Inc. This version of Datalog allows "stratified negation", that is, negated clauses, as long as the negation is not part of a recursive cycle. It also allows specifying that some relations are functions, that is, the variable space is partitioned into domain and range variables, and there is only one range value for each unique combination of values in domain variables.

Datalog is a great fit for the domain of program analysis and, as a consequence, has been extensively used both for low-level [17, 10, 22] and for high-level [6, 9] analyses. The essence of Datalog is its ability to define recursive relations. Mutual recursion is the source of all complexity in program analysis. For a standard example, the logic for computing a call-graph depends on having points-to information for pointer expressions, which, in turn, requires a call-graph. Such recursive definitions are common in points-to analysis.

Consider, for instance two relations, `AssignHeapAllocation(?heap, ?var)` and `Assign (?to, ?from)`[1] The former relation represents all occurrences in the Java program of an instruction "$a = newA();$" where a heap object is allocated and assigned to a variable. Those

[1]We follow the DOOP convention of capitalizing the first letter of relation names, while writing variable names in lower case and prefixing them with a question-mark.

relations are the result of a pre-processing [2] step that takes a Java program (in Doop this is in intermediate, bytecode, form) as input and produces the relation contents. That kind of relations that directly derive from the input Java program, are also known in Datalog semantics, as the *EDB* (Extensional Database) predicates. A static abstraction of the heap object is captured in variable `?heap`—it can be concretely represented as, for example, a fully qualified class name and the allocation's bytecode instruction index. Similarly, relation `Assign` contains an entry for each assignment between two Java program (reference) variables.

The mapping between the input Java program and the input relations is straightforward and purely syntactic. After this step, a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation:

```
1 VarPointsTo(?heap, ?var) <- AssignHeapAllocation(?heap, ?var).
2 VarPointsTo(?heap, ?to) <- Assign(?to, ?from), VarPointsTo(?heap, ?from).
```

The Datalog program consists of a series of *rules*, also known in Datalog semantics as the *IDB* (Intensional Database) rules, that are used to establish facts about derived relations (such as `VarPointsTo`, which is the points-to relation, i.e., it links every program variable, `?var`, with every heap object abstraction, `?heap`, it can point to) from a conjunction of previously established facts. In the LB-Datalog syntax, the left arrow symbol (<–) separates the inferred fact (i.e., the *head* of the rule) from the previously established facts (i.e., the *body* of the rule).

For instance, line 2 above says that if, for some values of `?from`, `?to`, and `?heap`, `Assign(?to,?from)` and `VarPointsTo(?heap,?from)` are both true, then it can be inferred that `VarPointsTo(?heap,?to)` is true. Note the base case of the computation above (line 1), as well as the recursion in the definition of `VarPointsTo` (line 2).

The declarativeness of Datalog makes it attractive for specifying complex program analysis algorithms. Particularly important is the ability to specify recursive definitions—program analysis is fundamentally an amalgam of mutually recursive tasks. For instance, Doop uses mutually recursive definitions of points-to analysis and call-graph construction.

The key for a precise points-to analysis is context-sensitivity, which consists of qualifying program variables and possibly object abstractions—in which case the context-sensitive analysis is said to also have a *context-sensitive heap*—with context information: the analysis collapses information (e.g., "what objects this method argument can point to") over all possible executions that result in the same context, while separating all information for different

[2]We use the SOOT framework for the pre-processing of the Java program and the generation of input facts.

```
1  class A {
2    void foo (Object o) { ... }
3  }
4
5  class B {
6    void bar (A a1, A a2) {
7      ...
8      a1.foo(someobj1);
9      ...
10     a2.foo(someobj2);
11   }
12 }
```

Figure 2.1: Simple Java Example for context-sensitivity

contexts. Call-site-sensitivity, object-sensitivity and type-sensitivity are the main flavors of context sensitivity in modern points-to analyses. They differ in the context's components and in how they are combined in order to create new context.

A call-site sensitive analysis uses method call-sites as context elements. For instance, in the code example in Figure 2.1, a 1-call-site sensitive analysis will distinguish the two call-sites of method `foo` on lines 7 and 9. In contrast, object-sensitivity uses object allocation sites as context elements. That is, when a method is called on an object, the analysis separates the inferred facts depending on the allocation site of the receiver object (i.e., the object on which the method was called), as well as other allocation sites used as context. Thus, in our example, a 1-object-sensitive analysis will analyze `foo` separately depending on the allocation sites of the objects that `a1` and `a2` may point to.

It is not apparent from the code fragment neither whether `a1` and `a2` may point to different objects, nor to how many objects. Similarly, it is not possible to compare the precision of an object-sensitive and a call-site sensitive analysis in principle. Nonetheless, ever since the introduction of object-sensitivity by Milanova et al. [14], there has been accumulating evidence [3, 11, 12, 13, 15] that it is a superior context abstraction for object-oriented languages, yielding high precision relative to cost.

Type-sensitivity was introduced [4] as an analysis directly analogous to an object-sensitive analysis, yet approximating (some) context elements using types instead of full allocation sites. It is a relatively new variant that combines scalability with good precision. In contrast to past uses of types in points-to analysis (e.g., [1, 16, 20]), types used as contexts should *not* be the types of the corresponding objects. Instead, the precision of type-sensitive analysis is due to replacing the allocation site of an objects `o` (which would be used as context in an object-sensitive analysis) with an upper-bound of the dynamic type of `o`'s allocator object.

Context-sensitive analysis in Doop is, to a large extent, similar to the above context-insensitive logic. The main changes are due to the introduction of Datalog variables representing contexts for variables (and, in the case of a context-sensitive heap, also objects), in the analyzed program. For an illustrative example, the following two rules handle method calls as implicit assignments from the actual parameters of a method to the formal parameters, in a 1-context-sensitive analysis with a context-*insensitive* heap.[3]

```
Assign(?calleeCtx, ?formal, ?callerCtx, ?actual) <-
  CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?method),
  FormalParam[?index, ?method] = ?formal,
  ActualParam[?index, ?invocation] = ?actual.

VarPointsTo(?heap, ?toCtx, ?to) <-
  Assign(?toCtx, ?to, ?fromCtx, ?from),
  VarPointsTo(?heap, ?fromCtx, ?from).
```

Note that some of the above relations are functions, in which case the functional notation "`Relation[?domainvar] = ?val`", is used instead of the traditional relational notation, "`Relation(?domainvar, ?val)`". Semantically the two are equivalent, with the only difference being that the execution engine enforces the functional constraint and produces an error if a computation causes a function to have multiple range values for the same domain value.

The example shows how a derived `Assign` relation (unlike the input relation `Assign` in the earlier basic example) is computed, based on the call-graph information, and then used in deriving a context-sensitive `VarPointsTo` relation.

For deeper context, the above rules need not to change at all (with the exception of adding a variable for the heap context as well). Doop makes use of the notion of *skolem functions* for creating new contexts. Skolem functions are the way that the Datalog engine provides in order to create new values from existing ones. The existing values, also known as keys, are combined and generate a new unique value. Each skolem predicate is 1:1, meaning that the underlying engine guarantees that the same key values will always generate the same value,

[3]This code is the same for either call-site-sensitivity, object-sensitivity or type-sensitivity.

and that two different key value sets, will generate different resulting values.

In Doop, skolem functions are used when we need to create a new context for a variable abstraction, simply referred to as *Context*, or a new context for a heap abstraction, simply referred to as *HContext*. We use the notion of *record* and *merge* functions [4], that manipulate contexts.

$$\begin{aligned} &\textit{record} : \text{Label} \times \text{Context} \rightarrow \text{HContext} \\ &\textit{merge} : \text{Label} \times \text{HContext} \times \text{Context} \rightarrow \text{Context} \end{aligned}$$

The record function is used every time an object is created, in order to store the creation context with the object. Its first argument is the current allocation statement label and its second one is the current (caller's) context. The merge function is used on every method invocation. Its first argument is the current call statement label, while the second and third arguments are the context of allocation of the method's receiver object and the current (caller's) context, respectively. The key for different flavors of context sensitivity is to specify different record and merge functions.

For instance, the rule for an object allocation in a 1-object-sensitive analysis with a context-sensitive heap can be as following: [4]

```
Record[?heap, ?ctx] = ?hCtx,
VarPointsTo(?hCtx, ?heap, ?ctx, ?var) <-
  AssignNormalHeapAllocation(?heap, ?ctx, ?var).
```

The above rule states that whenever a new heap object is allocated, its heap context will be uniquely identified by the current context. Each analysis has to define the *Record* and *Merge* predicates, and Doop employs a macro system to make the integration with the main part of the code, where most of the rules concerning the point-to analysis are defined, quite straight-forward.

Generally, the declarative nature of Doop often allows for very concise specifications of analyses. In [3], Bravenboer and Smaragdakis demonstrate a striking example of the logic for the Java cast checking—i.e., the answer to the question "can type A be cast to type B?". The Datalog rules are almost an exact transcription of the Java Language Specification. A small excerpt, with the Java Language Specification text included in comments, can be seen in Figure 2.2.

[4]Technical details about the usage of skolem functions have been omitted.

```
//  If S is an ordinary (nonarray) class, then:
//     o If T is a class type, then S must be the
//       same class as T, or a subclass of T.
CheckCast(?s, ?s) <- ClassType(?s).
CheckCast(?s, ?t) <- Subclass(?t, ?s).
...
//     o If T is an array type TC[], that is, an array of components
//       of type TC, then one of the following must be true:
//          + TC and SC are the same primitive type
CheckCast(?s, ?t) <-
  ArrayType(?s), ArrayType(?t),
  ComponentType(?s, ?sc), ComponentType(?t, ?sc), PrimitiveType(?sc).

//          + TC and SC are reference types (2.4.6), and type SC can be
//            cast to TC by recursive application of these rules.
CheckCast(?s, ?t) <-
  ComponentType(?s, ?sc), ComponentType(?t, ?tc),
  ReferenceType(?sc), ReferenceType(?tc), CheckCast(?sc, ?tc).
```

Figure 2.2: Excerpt of Datalog code for Java cast checking, together with Java Language Specification text in comments. The rules are quite faithful to the specification.

Chapter 3

Background of Points-to and Exception Analysis

In this chapter, we present previous work supporting the claim that exception analysis is better done in combination with points-to analysis and explaining how context-sensitivity can help in achieving higher precision and better performance.

3.1 Exception Analysis

Exception analysis and points-to analysis are typically done in complete separation. Past algorithms for precise exception analysis use precomputed points-to information. Past points-to analyses either unsoundly ignore exceptions altogether, or conservatively compute a crude approximation of exception throwing (e.g., considering an exception throw as an assignment to a global variable accessible from any catch clause). It was shown however [2] that this separation results in significant slowdowns or vast imprecision. This is because the two kinds of analyses are interdependent and neither can be performed accurately without the other.

3.2 Joint Points-To and Exception Analysis

The approach where exceptions are treated equally to other code features is the one followed by the Doop framework. The resulting exception analysis is "fully precise", as it models closely the Java exception handling semantics. The necessary approximation is provided only through whichever abstractions are used for contexts and objects in the base points-to analysis. This combined approach achieves similar precision relative to exceptions as the best past precise exceptions analysis, with a runtime of seconds instead of tens of minutes. At the same time, the precision of points-to information is much higher than that in points-to analyses that treat exceptions conservatively, all at a fraction of the execution time.

The relevant parts of exception handling in Java consist of *declaring* and *throwing* excep-

```
ThrowPointsTo(?hctx, ?heap, ?callerCtx, ?callerMethod) <-
  CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?tomethod),
  ThrowPointsTo(?hctx, ?heap, ?calleeCtx, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.

VarPointsTo(?hctx, ?heap, ?callerCtx, ?param) <-
  CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?tomethod),
  ThrowPointsTo(?hctx, ?heap, ?calleeCtx, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

Figure 3.1: Propagation of exceptions for method invocations. `ThrowPointsTo`: A method `?callerMethod` throws an exception `?heap` if there is a call-graph edge from an invocation in `?callerMethod` to some method `?tomethod` and `?tomethod` throws `?heap`. Also, the exception should not be caught immediately by an exception handler in `?tomethod`. `VarPointsTo`: If there is such an exception handler, then the exception `?heap` is assigned to the formal parameter `?param` of the exception handler.

tions, as well as *catching* them. An extra element, the Java `finally` clause, often causes headaches for static analysis purposes [5] but is a non-issue in our context. DOOP performs all analysis on bytecode. At that level, all uses of `finally` have already been translated away by the Java compiler.[1]

The logic in DOOP models the Java exception handling semantics. Exceptions introduce an interprocedural layer of assignments over the normal source code. At `throw` statements, normal objects flow in the exception-flow. At exception handlers, exception objects flow to normal Java variables. The propagation of exceptions is similar to the propagation of objects over assignments, except that assignment to an exception handler depends on the run-time type of an exception, somewhat similarly to dynamic dispatch. An excerpt of the code responsible for the exception handling is included in Appendix A.

An example of how the exception logic interacts with the main logic of the points-to analysis is given in Figure 3.1.

We can observe here that the declarative definition of our analysis is the main reason for its power with such conciseness. Defining the complex mutual interdependencies man-

[1]The finally block is copied appropriately and executed on any possible exit point, normal or exceptional, of the `try` block or any local `catch` block.

ually would have been very hard. Doop has multiple rules that cause the computation of `VarPointsTo` to depend on call-graph information. Similarly, there are many rules that cause the call-graph computation to depend on `VarPointsTo`. We add more complex mutual recursion to these rules, by making `VarPointsTo` depend on `ThrowPointsTo` (and vice versa) while `ThrowPointsTo` also depends on call-graph information (`CallGraphEdge`). The Datalog engine automatically incrementalizes all computation so that all iterations to fixpoint only need to operate on facts newly added to each relation.

Another element to note is that the exception handling logic is, in a sense, fully precise. This is not a formal statement, but we claim it informally based on inspection of the natural language text of the Java Virtual Machine Specification. We certainly have not consciously introduced any approximation in the exception handling logic. Of course, there is an approximation introduced in our exception analysis, but this comes directly from the abstraction of the host points-to analysis.

3.3 Context-Sensitive Exception Analysis

Adding context to our exception analysis requires propagating exceptions over the context-sensitive call-graph, and not over the conventional, user-visible context-insensitive one. Why is context-sensitivity important, however? The goal of a joint analysis is to employ the well-understood precision mechanisms of a standard points-to analysis in order to match the precision of (much more expensive) analyses specifically designed to track exception flow. Indeed, rich context abstractions allow our analysis to handle even complicated scenarios of the exception-flow analysis by Fu et al. [7] which are used to motivate improvements over the original DataReach algorithm [8].

A simple but illustrative example is shown in Figure 3.2 [7]. If we do not distinguish different calls to `BufferedInputStream.read`, exceptions resulting from the `read` invocation in `readFile` leak to `readNet` and vice versa. Our context-insensitive pointer analysis with precise exception analysis returns 9 exception-links between native methods throwing I/O exceptions and the exception handlers in `readFile` and `readNet` (Table 3.1). Even a 1-call-site-sensitive analysis is ineffective. A single call-site is not sufficient context to distinguish the different calling contexts of the native methods—a very long call string would be required for that. However, a 1-object-sensitive analysis is sufficient and yields the required precision (Table 3.2). This example also shows that a context-sensitive representation of `ThrowPointsTo` is crucial: both `readFile` and `readNet` share some methods on their call-graph paths to potential exception throwing code. Therefore, although the call-graph is context-sensitive, a context-insensitive `ThrowPointsTo` would merge the exception information of those distinct paths.

```java
public void readFile(String filename) {
  byte[] buffer = new byte[256];
  try {
    InputStream f = new FileInputStream(filename);
    InputStream fin = new BufferedInputStream(f);
    int c = fin.read(buffer);
  } catch(IOException exc) { ... }
}
public void readNet(Socket socket) {
  byte[] buffer = new byte[256];
  try {
    InputStream s = socket.getInputStream();
    InputStream sin = new BufferedInputStream(s);
    int c = sin.read(buffer);
  } catch(IOException exc) { ... }
}
```

Figure 3.2: Exception-flow analysis example

readFile: catch IOException	
FileInputStream:	void open(java.lang.String)
FileInputStream:	int readBytes(byte[],int,int)
FileInputStream:	int available()
PlainSocketImpl:	int socketAvailable()
SocketInputStream:	int socketRead(byte[],int,int)
readNet: catch IOException	
FileInputStream:	int readBytes(byte[],int,int)
FileInputStream:	int available()
PlainSocketImpl:	int socketAvailable()
SocketInputStream:	int socketRead(byte[],int,int)

Table 3.1: Exception-catch links for Fig. 3.2 using context-insensitive analysis, focusing on native methods.

readFile: catch IOException	
FileInputStream:	void open(java.lang.String)
FileInputStream:	int readBytes(byte[],int,int)
FileInputStream:	int available()
readNet: catch IOException	
PlainSocketImpl:	int socketAvailable()
SocketInputStream:	int socketRead(byte[],int,int)

Table 3.2: Exception-catch links for Fig. 3.2 using 1-object-sensitive analysis, focusing on native methods.

3.4 Precise Versus Imprecise Exception Analysis

We include here experimental results from previous work on the Doop framework [2] that support the claim that the separation of exception analysis from pointer analysis results in either significant slowdowns or vast imprecision. The comparison is between a precise joint exception and points-to analysis, and an imprecise one. The imprecise exception analysis assigns all exceptions thrown in reachable methods to a single variable. This variable is assigned to all reachable exception handlers. Type filtering removes exceptions that are not assignment-compatible with the type of a specific exception handler. Four analyses are evaluated: context-insensitive (insens), 1-call-site sensitive (1-call), 1-call-site sensitive with a context-sensitive heap abstraction (1-call+H), and 1-object-sensitive (1-obj). The results are for a subset of the DaCapo benchmarks programs. For context-sensitive analysis, there are two sets of statistics. One corresponds to end-user visible results and the other to primary internal complexity metrics. The first group ("*after dropping contexts*") drops all contexts after a context-sensitive analysis. The second group of results ("*before dropping contexts*") is context-sensitive, although it does drop the context of the heap abstraction for the 1-call+H analysis, so that it is comparable with the rest. The statistics for the imprecise analyses are relative to the corresponding statistics of the precise ones (e.g., in 1-call for antlr, the $\times 1.7$ in vars means that the `VarPointsTo` relation is 1.7 times larger compared to the precise 1-call analysis). The imprecise exception analysis does not compute the exceptions potentially thrown by each method, therefore the corresponding cells of Table 3.3 are empty (-).

The primary benefit from a precise analysis, is evident in the precision of the points-to results. Mainly the *context-insensitive* var points-to relation (the primary user-visible end result of a points-to analysis) is substantially smaller compared to imprecise exception handling. The *context-sensitive* var points-to relation is comparatively even smaller when using precise exception handling.

Another point of interest is that of call-graph precision. Precise exception analysis does not substantially reduce the number of nodes and edges of the *context-insensitive* call-graph. This is not surprising, since earlier studies demonstrated that improvements in precision barely influence the context-insensitive call-graph [12]. The *context-sensitive* call-graph is also not significantly affected, except in the case of the 1-obj analysis. This is quite expected, since the analysis uses objects as contexts.[2]

A major metric of the precision of the exception analysis is that of throw points-to relation. Comparing insensitive analyses and the context-sensitive ones, the *context-insensitive* throw points-to relation is generally two times smaller. This confirms that using a context-sensitive pointer analysis is useful for determining which exceptions may be thrown by a method. In addition, similarly to var points-to, the 1-obj analysis is the most precise analysis for the throw points-to. This corresponds to conclusions in related work that object sensitivity is the most useful context abstraction for object-oriented programs.

Finally, besides the precision improvements, the most striking result is the performance improvement of using precise exception analysis with the 1-obj pointer analysis. The analysis time comparison is a trade-off in most cases. Compared to an imprecise exception analysis, a precise one computes more information: the throw points-to relation. This relation is big, usually comparable to the var points-to relation. The performance of the benchmarks is correlated with the sum of the var points-to, throw points-to and call-graph edge relations. If introducing the throw points-to calculation substantially reduces the others, then the performance improves substantially.

[2]If a variable points to more abstract objects, then methods invoked on this variable will be invoked in more contexts under a 1-obj analysis.

			after dropping contexts						before dropping contexts						
prog	analysis		nodes	edges	vars		throws		nodes	edges	vars		throws		time (sec)
antlr	precise	insens	5K	38K	10M	181	1.3M	252	5K	38K	10M	181	1.3M	252	74
		1-call	5K	38K	767K	13	757K	150	38K	159K	4.3M	17	4.5M	119	91
		1-call+H	5K	38K	759K	13	756K	150	38K	154K	4.1M	16	4.5M	119	464
		1-obj	5K	38K	598K	10	579K	115	60K	1.0M	3.4M	9	1.9M	31	191
	imprecise	insens	×1.0	×1.0	×1.0	+9	-	-	×1.0	×1.0	×1.0	+9	-	-	53
		1-call	×1.0	×1.0	×1.7	+10	-	-	×1.0	×1.0	×1.4	+7	-	-	59
		1-call+H	×1.0	×1.0	×1.7	+10	-	-	×1.0	×1.0	×1.4	+7	-	-	467
		1-obj	×1.0	×1.0	×2.0	+11	-	-	×1.1	×6.1	×4.6	+28	-	-	2680
chart	precise	insens	8K	40K	5.7M	82	1.8M	230	8K	40K	5.7M	82	1.8M	230	65
		1-call	8K	39K	2.5M	36	956K	123	40K	170K	17M	64	3.7M	94	138
		1-call+H	8K	39K	2.5M	35	955K	123	40K	169K	17M	64	3.7M	94	650
		1-obj	8K	39K	2.3M	33	792K	103	95K	1.9M	18M	27	4.5M	47	447
	imprecise	insens	×1.0	×1.0	×1.2	+13	-	-	×1.0	×1.0	×1.2	+13	-	-	44
		1-call	×1.0	×1.0	×1.4	+14	-	-	×1.0	×1.0	×1.2	+11	-	-	113
		1-call+H	×1.0	×1.0	×1.4	+14	-	-	×1.0	×1.0	×1.2	+11	-	-	671
		1-obj	×1.0	×1.0	×1.4	+15	-	-	×1.1	×5.4	×2.3	+31	-	-	5429
eclipse	precise	insens	5K	26K	3.1M	69	1.4M	286	5K	26K	3.1M	69	1.4M	286	41
		1-call	5K	25K	841K	19	728K	151	25K	125K	4.5M	26	2.9M	114	69
		1-call+H	5K	25K	838K	19	727K	151	25K	125K	4.5M	26	2.9M	114	316
		1-obj	5K	25K	672K	15	592K	123	55K	2.3M	5.3M	14	3.1M	56	480
	imprecise	insens	×1.0	×1.0	×1.2	+11	-	-	×1.0	×1.0	×1.2	+11	-	-	27
		1-call	×1.0	×1.0	×1.7	+14	-	-	×1.0	×1.0	×1.3	+9	-	-	50
		1-call+H	×1.0	×1.0	×1.7	+13	-	-	×1.0	×1.0	×1.3	+9	-	-	287
		1-obj	×1.0	×1.0	×2.0	+15	-	-	×1.1	×4.1	×3.8	+33	-	-	3794
jython	precise	insens	6K	33K	5.1M	93	1.9M	322	6K	33K	5.1M	93	1.9M	322	66
		1-call	6K	33K	2.3M	41	1.2M	198	33K	150K	14M	58	5.1M	154	141
		1-call+H	6K	33K	2.3M	41	1.2M	198	33K	150K	14M	58	5.1M	154	1358
		1-obj	6K	33K	2.0M	36	1.1M	189	95K	2.6M	17M	25	14M	146	914
	imprecise	insens	×1.0	×1.0	×1.2	+18	-	-	×1.0	×1.0	×1.2	+18	-	-	41
		1-call	×1.0	×1.0	×1.5	+22	-	-	×1.0	×1.0	×1.4	+24	-	-	117
		1-call+H	×1.0	×1.0	×1.5	+22	-	-	×1.0	×1.0	×1.4	+24	-	-	1436
		1-obj	×1.0	×1.0	×1.6	+22	-	-	×1.1	×3.2	×2.2	+25	-	-	3037
xalan	precise	insens	4K	18K	1.7M	51	855K	229	4K	18K	1.7M	51	855K	229	35
		1-call	4K	18K	464K	14	459K	123	18K	70K	2.4M	20	1.7M	93	51
		1-call+H	4K	17K	459K	14	458K	123	18K	65K	2.3M	20	1.6M	93	154
		1-obj	4K	18K	394K	12	358K	97	37K	885K	2.6M	11	1.5M	39	174
	imprecise	insens	×1.0	×1.0	×1.2	+8	-	-	×1.0	×1.0	×1.2	+8	-	-	18
		1-call	×1.0	×1.0	×1.7	+10	-	-	×1.0	×1.0	×1.3	+7	-	-	33
		1-call+H	×1.0	×1.0	×1.7	+10	-	-	×1.0	×1.0	×1.4	+7	-	-	141
		1-obj	×1.0	×1.0	×1.9	+11	-	-	×1.1	×3.2	×3.0	+18	-	-	834

legend
nodes, edges = call-graph nodes, call-graph edges
vars = *total* and *mean (per variable)* entries in var points-to relation
throws = *total* and *mean (per method)* entries in throw points-to relation

Table 3.3: Precise versus imprecise exception analysis.

Chapter 4

Combining Exceptions with Context

In this chapter, we present three alternative methods for treating exceptions in the presence of a context-sensitive points-to analysis, and their effect on the resulting precision and performance. Each method differentiates on the way that exception objects are combined with context. Exception objects are identified by their type, which must be a subtype of `java.lang.Throwable` in order for them to appear in a `throw` statement.

4.1 "Full" Context Sensitive Exceptions

The most straight-forward way of handling exception objects, in the presence of a context-sensitive points-to analysis, is to do nothing special. This means that for exception objects, the full context of the points-to analysis is used the same way that it's used for "normal" heap objects. From a theoretical point of view, this method should attain the highest precision for a specific context depth, as it fully utilizes the context that is available. In practice, however, this method suffers (heavily in many cases) in terms of performance. In many analyses (e.g. 2-object sensitive with a 2-context sensitive heap abstraction) the size of the `ThrowPointsTo` relation is comparable to that of the `VarPointsTo` relation, or even bigger.

The size of `ThrowPointsTo` relation (as well as that of the `VarPointsTo` and the `CallGraphEdge` relations) can significantly affect the computation time needed for the analysis. Consequently, the remaining methods try to ameliorate the impact that exceptions have on performance and to minimize the size of the resulting `ThrowPointsTo` relation, while trying to keep precision as high as possible.

One interesting thing that we should note here is that the precise analyses in the experiments in Chapter 3.4, were *not* using the above method. Instead exception objects were allocated context-insensitively (as described in more detail, in the next section).

```
HeapAllocationContextInsensitive(?heap) <-
  HeapAllocationType[?heap] = ?heaptype,
  Type[?throw] = "java.lang.Throwable",
  AssignCompatible(?throw, ?heaptype).

AssignContextInsensitiveHeapAllocation(?heap, ?var, ?inmethod) <-
  AssignHeapAllocation(?heap, ?var, ?inmethod),
  HeapAllocationContextInsensitive(?heap).

RecordImmutable[] = ?immCtx,
VarPointsTo(?immCtx, ?heap, ?ctx, ?var) <-
  AssignContextInsensitiveHeapAllocation(?heap, ?var, ?inmethod),
  ReachableContext(?ctx, ?inmethod).
```

Figure 4.1: Code for the context-insensitive treatment of exception objects.

4.2 Context Insensitive Exceptions

The next method for the treatment of exception objects suggests that each exception object should use a context-insensitive heap abstraction. The pointer analysis is not affected in any way and continues to use the type and amount of contexts that is implied. Only exception objects are treated in a special way during their allocation where a context-insensitive heap abstraction is produced. Otherwise, the flow of the analysis remains the same.

The code responsible for the context-insensitive treatment of exceptions, is given in Figure 4.1. Every heap object whose type is a subtype of `java.lang.Throwable` is identified as an exception object. The `HeapAllocationContextInsenstive` relation is used in order to store heap objects deemed by the analysis for context-insensitive treatment. This relation is not new in the context of the Doop framework and its usage is not confined to exception objects. It is used in general when some group of heap objects should be treated context-insensitively. A different example from exception objects is that of Class-name string constants, which are also treated context-insensitively.

The `AssignContextInsensitiveHeapAllocation` relation is used to store heap objects that are selected for context-insensitive treatment. Those objects are subsequently used to derive facts for the `VarPointsTo` relation. Since those objects are allocated context-insensitively but a heap context is needed for the `VarPointsTo` relation, `RecordImmutable` is used to generates a single unique context. In practice, heap objects that should be treated context-insensitively, are allocated context-*sensitively* in a sense, but with a unique heap context that is shared amongst them.

This method manages to extensively reduce the size of the `ThrowPointsTo` relation, with barely any loss in terms of precision but with a significant improvement in terms of performance.

We should also note here, how easily we were able to differentiate the context-sensitivity for a subset of the heap objects with only a few lines of code. This should be attributed to the declarative programming style of Datalog.

4.3 Type Representatives for Exceptions

In comparison to the context-sensitive treatment of exceptions, the context-insensitive one improves the performance of the resulting analysis because the size of the `ThrowPointsTo` relation is significantly reduced. However, the size of the `ThrowPointsTo` relation is still comparable to that of the `VarPointsTo` relation (in many cases as high as half of the `VarPointsTo` relation), and thus highly affecting the performance of the analysis.

With that as our initial motivation, we tried to pinpoint the source of the problem more accurately. Exceptions are one possible way for heap objects to leak out of a method's boundaries. Method arguments and the return value are another. Thus, we examined and compared the two sets. The findings further supported our initial insight. With the context-insensitive treatment of exceptions, the `ThrowPointsTo` relation is in some cases still comparable to the `VarPointsTo` relation, although smaller. But strikingly enough, when compared to the amount of heap objects leaking out of a method's boundaries through "normal" control flow (i.e., arguments and return value), the `ThrowPointsTo` relation can be almost as large (or in some cases even larger). For instance, in a 2-object sensitive analysis with a context sensitive heap abstraction, the amount of heap objects on "normal" method boundaries is roughly 2.5 millions. At the same time, the size of the `ThrowPointsTo` relation is roughly 5.5 millions! In general, our experiments showed an average ratio of 1.81 and a median of 1.26 for the `ThrowPointsTo` relation over the number of heaps on method boundaries.

A new insight, in order to address the issue, is that instead of using every exception object in our analysis, we can use a unique representative heap object per class type. This means that, for every group of exception objects with the same type, one object is uniquely selected as the group representative. All exception objects of the same type are merged, and this representative object is subsequently used.

```
TypeToHeap(?heap, ?heaptype) <-
  HeapAllocationType[?heap] = ?heaptype,
  Type[?throw] = "java.lang.Throwable",
  AssignCompatible(?throw, ?heaptype).

HeapRepresentative[?heap] = ?representativeHeap <-
  agg<<?representativeHeap = min(?otherHeap)>>
    TypeToHeap(?otherHeap, HeapAllocationType[?heap]).

HeapAllocationMerge[?heap] = ?mergeHeap <-
  HeapRepresentative[?heap] = ?mergeHeap.

AssignContextInsensitiveHeapAllocation(?mergeHeap, ?var, ?inmethod) <-
  AssignHeapAllocation(?heap, ?var, ?inmethod),
  HeapAllocationMerge[?heap] = ?mergeHeap.
```

Figure 4.2: Code for the usage of type representatives for exception objects.

Consequently, points-to results for throwable objects are not accurate and this method is not preferable in case of an analysis that focuses on exception objects. However, interestingly enough, the precision for the rest of the points-to metrics is barely affected, while at the same time a boost in performance is achieved. The improvement in performance can be explained by the significantly smaller `ThrowPointsTo` relation, due to the usage of representative objects.

It is not hard to also explain why the general precision of the analysis is barely affected by the usage of representatives. The common usage of exceptions is to indicate that some kind of erroneous condition has been encountered, and the exception object itself is used to alter the control-flow in order for the error to be addressed. As such, exception objects are used in lieu of indications, have only few fields (that mostly keep some indication of the error that occurred) and rarely have any method invoked on them (mainly getters in order to access the information about the error). As far as control-flow is concerned, what is important in most cases is not the exception object itself, but its type, in order to transfer control-flow to the correct exception handler.

The code responsible for this method of exception treatment can be found in Figure 4.2. Each throwable object is associated with its corresponding type, and one representative heap object is uniquely selected for each type. One way to uniquely select a representative object for a specific type, is to choose the object with the smaller internal id. The notation "`agg<<?minVal = min(?val)>>Relation(?val)`" means that an aggregator is used to calculate the minimum value `?minVal` of the collection identified by `?val`. Finally, each exception object that is not selected as a representative is merged and the representative object is subsequently used in its place. The `AssignContextInsensitiveHeapAllocation` relation is used once more to store the heap objects that are selected for context-insensitive treatment, as discussed previously in section 4.2.

Chapter 5

Experimental Results

In this chapter, we present the evaluation of the three alternative methods suggested in Chapter 4, on a well-known benchmark suite and comment on the experimental results.

5.1 Setup

We use a 64-bit machine with a quad-core Xeon E5530 2.4GHz CPU (only one thread was active at a time). The machine has 24GB of RAM.

We analyzed the DaCapo benchmark programs, v.2006-10-MR2, with JDK 1.4. These benchmarks are the largest in the literature on context-sensitive points-to analysis. We concentrated on a subset of the DaCapo benchmarks, namely *antlr*, *bloat*, *chart*, *eclipse*, *xalan*, all of which can be successfully analyzed by the Doop framework with reflection-analysis enabled.

There was an upper bound for the execution time of each analysis. A time limit of 5400 seconds (one hour and a half) was used. Analyses that did not finish within that timespan were terminated and the corresponding table cells are empty (-).

5.2 Evaluation

Figure 5.1 presents the execution time needed for each alternative method of exception treatment that we proposed in chapter 4, on the Eclipse benchmark. It is clear that the full context-sensitive treatment ("*no merge + sensitive*") has, almost always, a significant execution overhead compared to the other alternatives. The context-insensitive treatment ("*no merge*") shows a clear improvement over the full context-sensitive, with the time needed almost cut in half in most cases. Finally, the type representative alternative ("*merge*") presents a noticeable improvement over the context-insensitive one.

In general, for the set of benchmarks that we tested, "no merge" has a median of 26% (and an average of 40%) improvement over "no merge + sensitive" (with many analyses showing over 50% improvement). Similarly, "merge" has a median of 26% (and an average of 30%)

improvement over “no merge”.

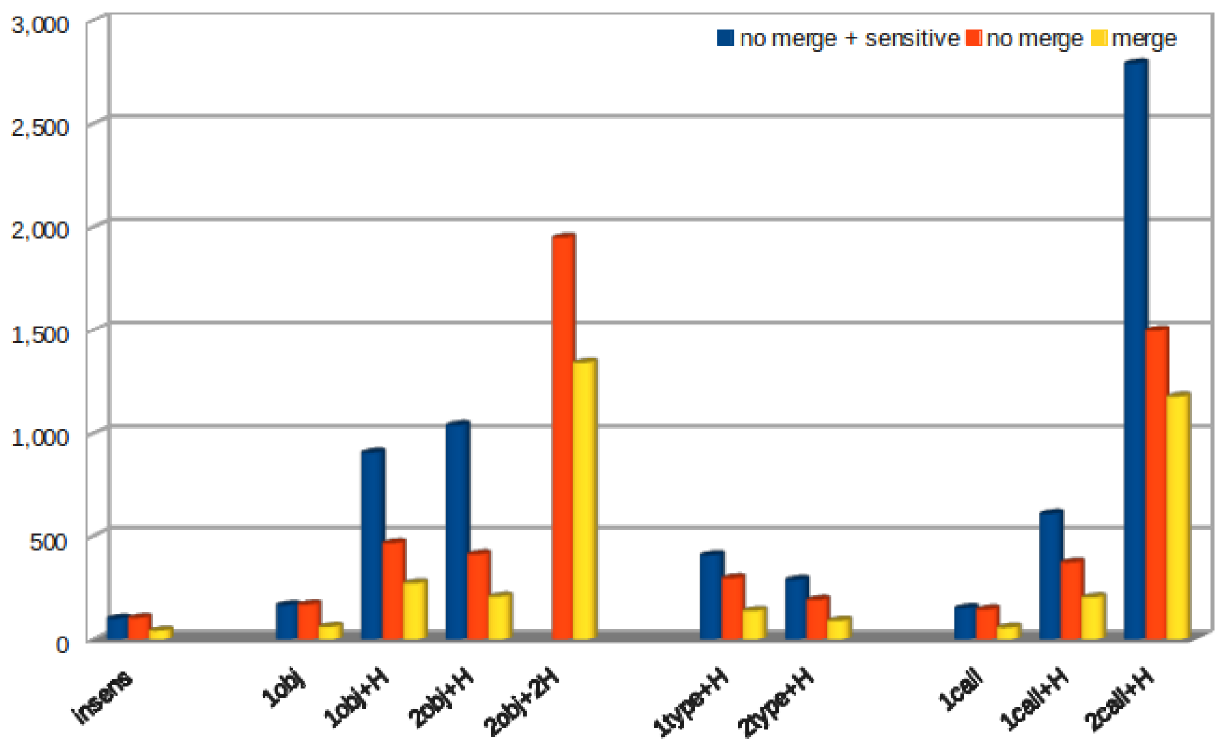

Figure 5.1: Execution time (seconds) of the Eclipse Benchmark. 2-obj+2H did not finish within the time allotted for the “no merge + sensitive” alternative.

Figure 5.2 presents the disk footprint (in KB) of the database for each alternative method. The results are in accordance with the execution time needed for each analysis. Analyses that have a larger disk footprint (and thus larger relations), need more time to finish their computations (and vice versa).

Table 5.1 and table 5.3, present the execution time and disk footprint respectively, for a variety of analyses on various benchmarks. The behavior of each benchmark is similar to that of the Eclipse benchmark which was analyzed above. Table 5.2 and table 5.4, present the ratios of the “no merge” alternative to the “no merge + sensitive” one, and of the “merge” alternative to the “no merge” one, for tables 5.1 and 5.3 respectively.

Figures 5.3, 5.4 and 5.5 depict the difference on the number of heap objects on method boundaries (i.e., arguments and return value) when compared to the `ThrowPointsTo` relation, for the 2-obj+H analysis. The comparisons are done context-sensitively—this means that the contexts associated with the results are still present. This is helpful when analyzing

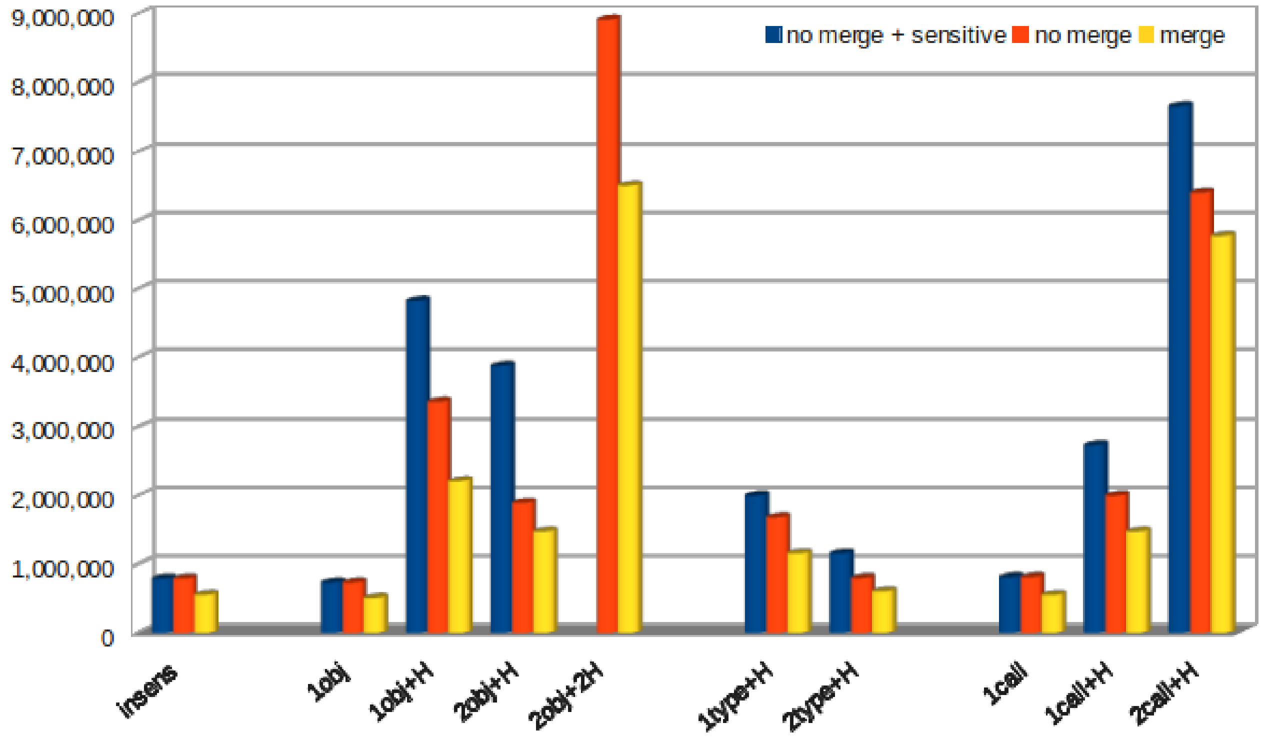

Figure 5.2: Disk footprint (KB) of the Eclipse Benchmark. 2-obj+2H did not finish within the time allotted for the “no merge + sensitive” alternative.

		analysis										
prog	alt.	insens	1-obj	1-obj+H	2-obj+H	2-obj+2H	1-type+H	2-type+H	1-call	1-call+H	2-call+H	2-call+2H
antlr	sens	88	114	506	372	1684	178	126	116	532	1289	-
	insens	88	115	298	130	232	128	95	109	295	896	5956
	merge	76	82	250	92	176	109	87	71	230	778	5098
bloat	sens	87	518	3272	-	-	347	650	265	2120	-	-
	insens	83	521	1343	-	-	216	255	247	1352	-	-
	merge	55	220	1099	4977	-	147	113	157	1255	-	-
chart	sens	79	312	1216	1852	-	345	210	145	577	2789	-
	insens	78	320	770	470	1838	249	133	146	347	1786	-
	merge	58	242	688	233	1356	209	104	106	313	1723	-
eclipse	sens	100	166	905	1038	-	408	288	151	609	2792	-
	insens	103	170	466	411	1983	296	191	144	372	1497	-
	merge	41	60	271	206	1340	137	89	56	203	1176	5639
xalan	sens	103	319	1844	-	-	466	479	176	700	1554	-
	insens	104	332	934	-	-	326	307	177	446	977	-
	merge	76	187	773	5282	-	230	228	109	380	788	-

legend
sens = context-sensitive treatment of exceptions
insens = context-insensitive treatment of exceptions
merge = using type representatives for exceptions

Table 5.1: Execution time (seconds) for a variety of analyses on various benchmarks.

		analysis											
prog	ratio	insens	1obj	1obj+H	2obj+H	2obj+2H	1type+H	2type+H	1call	1call+H	2call+H	2call+2H	avg (med)
	A1 / A2	1.00	1.00	1.70	2.86	7.26	1.39	1.33	1.06	1.80	1.44	-	2.08 (1.41)
antlr	A2 / A3	1.16	1.39	1.19	1.41	1.32	1.17	1.09	1.54	1.28	1.15	1.17	1.26 (1.19)
	A1 / A2	1.06	0.99	2.44	-	-	1.61	2.55	1.07	1.57	-	-	1.61 (1.57)
bloat	A2 / A3	1.52	2.37	1.22	1.09	-	1.47	2.26	1.09	1.08	-	-	1.51 (1.35)
	A1 / A2	1.00	0.98	1.58	3.94	2.94	1.39	1.58	0.99	1.66	1.35	-	1.74 (1.48)
chart	A2 / A3	1.34	1.32	1.12	2.02	1.36	1.19	1.28	1.38	1.11	1.18	-	1.33 (1.30)
	A1 / A2	0.97	0.98	1.94	2.52	2.77	1.38	1.51	1.05	1.64	1.87	-	1.54 (1.51)
eclipse	A2 / A3	2.51	2.82	1.72	2.00	1.45	2.16	2.15	2.57	1.83	1.27	-	2.05 (2.07)
	A1 / A2	0.99	0.96	1.97	-	-	1.43	1.56	0.99	1.57	1.59	-	1.38 (1.49)
xalan	A2 / A3	1.37	1.78	1.21	-	-	1.42	1.35	1.62	1.17	1.24	-	1.39 (1.36)

legend

A1 / A2 = ratio of alternative 1 (“no merge + sensitive”) to alternative 2 (“no merge”)
A2 / A3 = ratio of alternative 2 (“no merge”) to alternative 3 (“merge”)

Table 5.2: Execution time ratios for the analyses in Table 5.1.

		analyss										
prog	alt.	insens	1-obj	1-obj+H	2-obj+H	2-obj+2H	1-type+H	2-type+H	1-call	1-call+H	2-call+H	2-call+2H
	sens	1433	860	3584	2867	11264	1228	691	1126	4812	13312	-
antlr	insens	1433	860	2662	1126	2150	1126	581	1126	2457	11264	24576
	merge	1228	723	2969	790	1536	867	494	833	2048	11264	23552
	sens	1126	1433	6860	-	-	1536	1536	2150	13312	-	-
bloat	insens	1126	1433	4915	-	-	1331	1126	2150	12288	-	-
	merge	931	1228	5017	12288	-	1022	761	1843	9728	-	-
	sens	1126	1638	6758	5939	-	1945	1009	1638	4505	19456	-
chart	insens	1126	1638	5324	2457	14336	1740	794	1638	3276	11264	-
	merge	797	1433	4915	1843	11264	1331	608	1331	2560	11264	-
	sens	768	713	4710	3788	-	1945	1126	789	2662	7475	-
eclipse	insens	768	713	3276	1843	8704	1638	775	789	1945	6246	-
	merge	535	491	2150	1433	6348	1126	585	533	1433	5632	9830
	sens	815	1638	7270	-	-	2252	1843	1126	4403	8704	-
xalan	insens	815	1638	6758	-	-	1945	1433	1126	3379	7168	-
	merge	600	1228	4505	15360	-	1331	1126	794	2355	4505	-

Table 5.3: Disk footprint (MB) for a variety of analyses on various benchmarks.

		analysis											
prog	ratio	insens	1obj	1obj+H	2obj+H	2obj+2H	1type+H	2type+H	1call	1call+H	2call+H	2call+2H	avg (med)
	A1 / A2	1.00	1.00	1.35	2.55	5.24	1.09	1.19	1.00	1.96	1.18	-	1.76 (1.19)
antlr	A2 / A3	1.17	1.19	0.90	1.43	1.40	1.30	1.18	1.35	1.20	1.00	1.04	1.20 (1.19)
	A1 / A2	1.00	1.00	1.40	-	-	1.15	1.36	1.00	1.08	-	-	1.14 (1.08)
bloat	A2 / A3	1.21	1.17	0.98	-	-	1.30	1.48	1.17	1.26	-	-	1.22 (1.21)
	A1 / A2	1.00	1.00	1.27	2.42	-	1.12	1.27	1.00	1.37	1.73	-	1.35 (1.27)
chart	A2 / A3	1.41	1.14	1.08	1.33	1.27	1.31	1.31	1.23	1.28	1.00	-	1.24 (1.28)
	A1 / A2	1.00	1.00	1.44	2.06	-	1.19	1.45	1.00	1.37	1.20	-	1.30 (1.20)
eclipse	A2 / A3	1.44	1.45	1.52	1.29	1.37	1.45	1.32	1.48	1.36	1.11	-	1.38 (1.40)
	A1 / A2	1.00	1.00	1.08	-	-	1.16	1.29	1.00	1.30	1.21	-	1.13 (1.12)
xalan	A2 / A3	1.36	1.33	1.50	-	-	1.46	1.27	1.42	1.43	1.59	-	1.42 (1.43)

legend

A1 / A2 = ratio of alternative 1 (“no merge + sensitive”) to alternative 2 (“no merge”)
A2 / A3 = ratio of alternative 2 (“no merge”) to alternative 3 (“merge”)

Table 5.4: Disk footprint ratios for the analyses in Table 5.3.

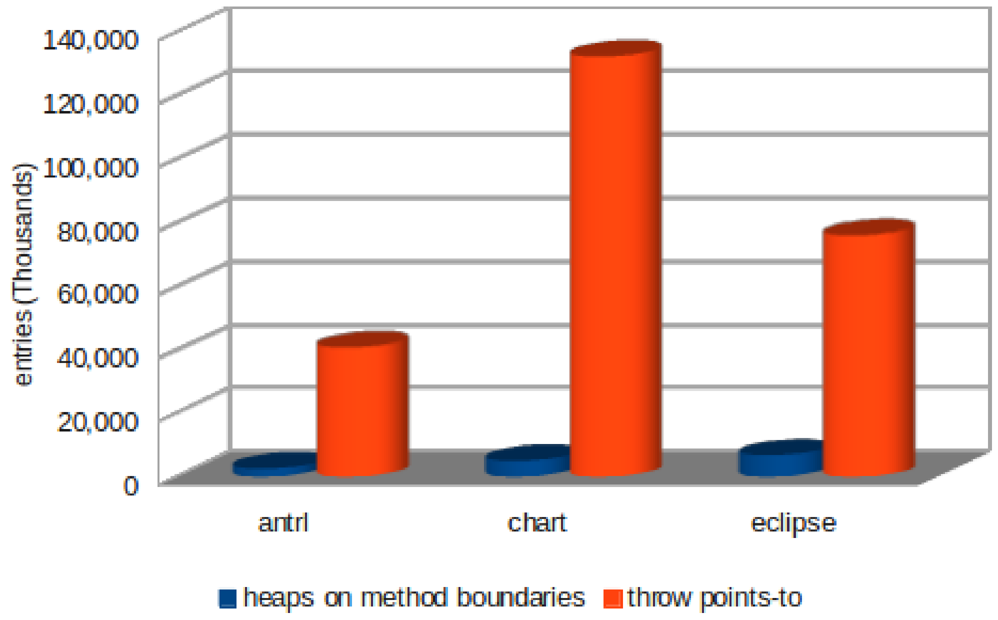

Figure 5.3: “no merge + sensitive” for the 2-obj+H analysis

the results, as the context-*sensitive* relations are those that affect the performance of the analysis.

The results support what we claim in Chapter 4. For the “no merge + sensitive” and “no merge” alternatives, the `ThrowPointsTo` relation is not only comparable to but even larger than the number of heap objects on method boundaries (although the difference is smaller in the “no merge” alternative). This can explain why these alternatives have a significant time overhead (and also a larger disk footprint), when compared to the “merge” alternative. Table 5.5 presents these metrics in more detail for a variety of analyses.

Table 5.6 presents the ratios of the “no merge” alternative to the “no merge + sensitive” one, and of the “merge” alternative to the “no merge” one, for table 5.5. We are interested in the following metrics: context-insensitive / context-sensitive var points-to entries (*vars* / *c-s vars*), context-insensitive / context-sensitive throw points-to entries (*throws* / *c-s throws*), and context-insensitive / context-sensitive heaps on method boundaries (*heaps on bounds* / *c-s heaps on bounds*).

One final important point of interest is the precision of the analyses under each alternative treatment of exceptions. Table 5.7 presents a few representative metrics for the precision of each analysis. We are interested in the following metrics: context-insensitive / context-sensitive var points-to entries to non-throwable objects (*vars* / *c-s vars*), context-insensitive

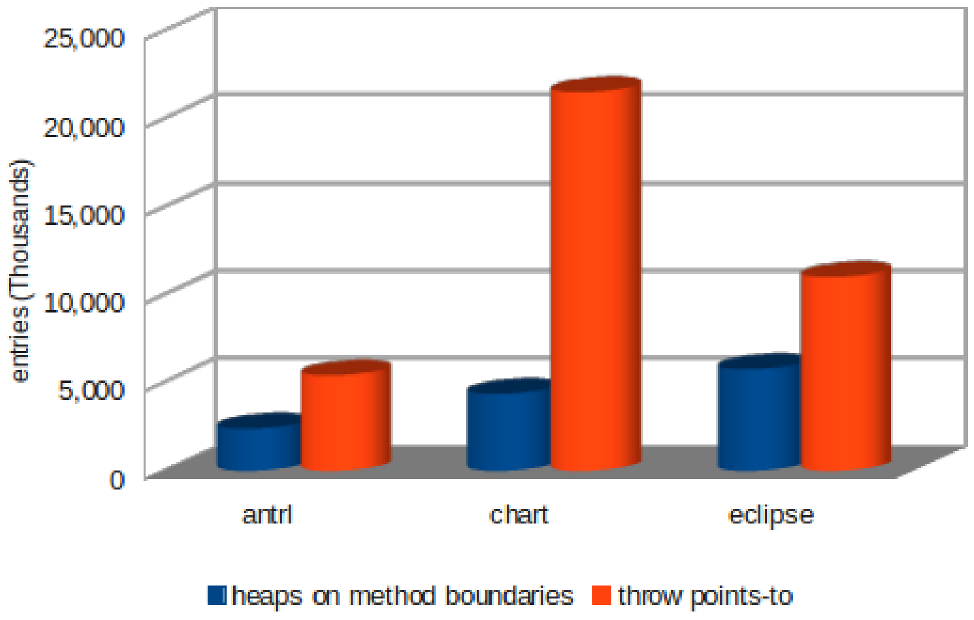

Figure 5.4: “no merge” for the 2-obj+H analysis

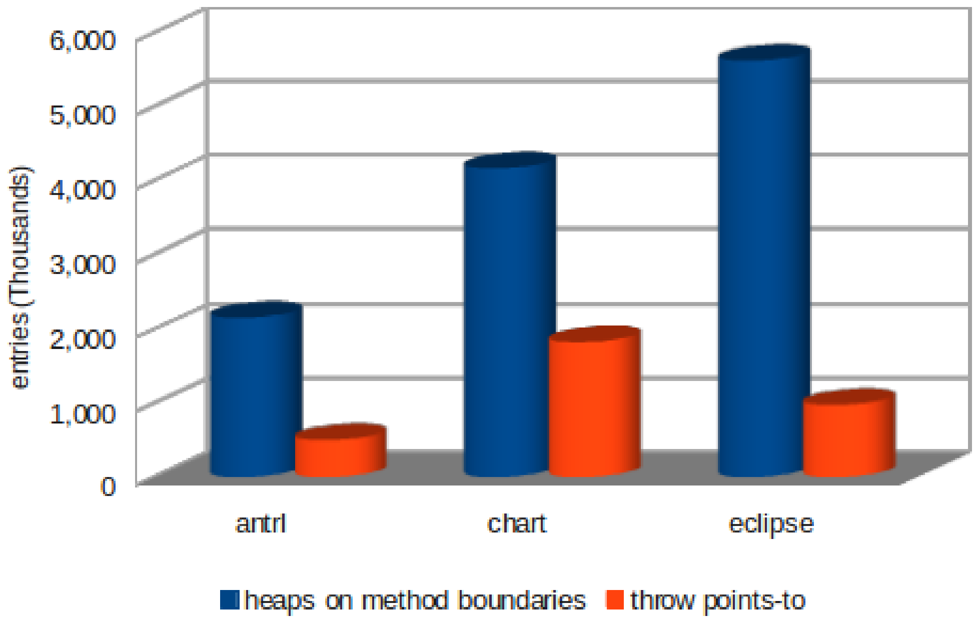

Figure 5.5: “merge” for the 2-obj+H analysis

			metrics					
prog	analysis	alt.	vars	c-s vars	throws	c-s throws	heaps on bounds	c-s heaps on bounds
antlr	1call+H	sens	1,755,928	39,629,645	896,483	18,795,136	306,228	8,232,824
		insens	1,755,928	37,881,574	896,483	5,366,432	306,228	8,174,846
		merge	1,577,404	37,286,572	71,291	425,031	297,540	8,157,714
	1obj+H	sens	904,331	53,533,098	716,341	30,242,473	143,813	15,883,155
		insens	904,331	48,498,480	716,341	3,032,188	143,813	15,733,578
		merge	774,915	42,562,162	61,311	260,241	137,198	14,958,784
	2obj+H	sens	477,786	18,588,395	380,418	40,718,205	58,664	2,572,927
		insens	477,825	8,947,303	380,418	5,444,711	58,673	2,417,767
		merge	383,193	7,065,213	38,653	501,018	54,424	2,152,874
	1type+H	sens	2,334,491	13,659,755	769,490	5,898,836	283,196	2,901,472
		insens	2,334,491	12,721,441	769,490	1,771,874	283,196	2,857,404
		merge	2,197,368	10,729,720	64,710	149,554	275,915	2,827,657
bloat	1call+H	sens	3,905,574	156,807,857	1,359,933	39,227,903	462,365	35,772,800
		insens	3,905,574	153,498,003	1,359,933	9,796,553	462,365	35,701,340
		merge	3,713,558	152,512,350	97,765	700,863	453,655	35,683,216
	1obj+H	sens	2,079,529	77,255,392	1,120,476	65,958,285	357,346	20,331,118
		insens	2,079,529	72,930,652	1,120,476	8,181,213	357,346	20,183,675
		merge	1,935,110	69,580,689	85,708	612,314	350,076	20,606,911
	1type+H	sens	2,535,716	19,367,073	1,329,596	13,440,987	420,278	3,974,687
		insens	2,535,716	17,842,592	1,329,596	4,241,840	420,278	3,925,346
		merge	2,360,828	15,139,472	97,670	295,463	412,429	3,928,446
chart	1call+H	sens	2,726,367	48,301,383	1,127,053	16,086,699	542,653	13,524,605
		insens	2,726,367	45,963,590	1,127,053	4,636,173	542,653	13,428,365
		merge	2,463,640	45,148,039	90,840	377,593	530,418	13,397,145
	1obj+H	sens	1,405,157	90,289,730	912,614	45,932,283	290,939	23,477,713
		insens	1,405,157	81,824,616	912,614	5,481,361	290,939	23,248,271
		merge	1,198,856	72,143,078	77,596	465,779	280,777	23,894,934
	2obj+H	sens	458,517	48,421,275	580,626	132,291,526	63,392	5,043,060
		insens	458,548	22,510,540	580,626	21,547,138	63,409	4,398,163
		merge	306,103	17,456,833	50,395	1,824,970	55,760	4,170,122
	1type+H	sens	1,983,416	27,292,650	1,016,952	9,963,411	420,682	5,687,124
		insens	1,983,416	25,442,532	1,016,952	2,926,732	420,682	5,601,568
		merge	1,756,258	18,978,391	84,806	242,385	409,358	5,576,654
eclipse	1call+H	sens	1,353,112	29,421,842	1,183,006	17,813,923	218,146	6,277,964
		insens	1,353,112	26,558,283	1,183,006	5,035,878	218,146	6,137,420
		merge	996,806	25,559,841	92,105	395,024	197,784	6,091,697
	1obj+H	sens	831,393	60,969,897	941,669	51,169,580	137,873	12,340,156
		insens	831,393	50,872,415	941,669	5,647,537	137,873	11,915,605
		merge	549,563	38,518,800	77,959	471,221	119,429	12,384,597
	2obj+H	sens	470,677	42,025,451	615,330	75,972,306	63,224	6,761,134
		insens	470,725	21,365,532	615,330	11,030,920	63,230	5,801,705
		merge	262,816	17,869,654	55,677	975,545	48,017	5,622,990
	1type+H	sens	1,117,879	32,491,607	1,139,517	13,031,671	190,948	4,555,635
		insens	1,117,879	29,985,596	1,139,517	3,816,446	190,948	4,440,201
		merge	797,712	16,137,100	91,655	307,049	171,593	4,390,787
xalan	1call+H	sens	1,755,952	43,642,644	1,292,735	18,321,006	325,275	11,124,313
		insens	1,755,952	40,723,505	1,292,735	5,259,843	325,275	10,949,465
		merge	1,376,105	39,698,133	99,246	406,130	299,453	10,892,320
	1obj+H	sens	848,229	124,527,709	1,088,295	89,407,699	138,145	30,526,014
		insens	848,229	94,706,464	1,088,295	10,501,818	138,145	29,778,686
		merge	532,856	81,777,325	88,097	831,449	114,679	29,811,380
	1type+H	sens	1,094,606	37,292,663	1,154,557	14,826,654	192,046	6,594,554
		insens	1,094,606	33,373,130	1,154,557	4,247,554	192,046	6,418,714
		merge	767,534	19,391,341	92,390	334,071	167,661	6,168,573

Table 5.5: Metrics concerning performance for a variety of analyses.

			metrics					
prog	analysis	alt.	vars	c-s vars	throws	c-s throws	heaps on bounds	c-s heaps on bounds
antlr	1call+H	A2 / A1	1.00	0.96	1.00	0.29	1.00	0.99
		A3 / A2	0.90	0.98	0.08	0.08	0.97	1.00
	1obj+H	A2 / A1	1.00	0.91	1.00	0.10	1.00	0.99
		A3 / A2	0.86	0.88	0.09	0.09	0.95	0.95
	2obj+H	A2 / A1	1.00	0.48	1.00	0.13	1.00	0.94
		A3 / A2	0.80	0.79	0.10	0.09	0.93	0.89
	1type+H	A2 / A1	1.00	0.93	1.00	0.30	1.00	0.98
		A3 / A2	0.94	0.84	0.08	0.08	0.97	0.99
bloat	1call+H	A2 / A1	1.00	0.98	1.00	0.25	1.00	1.00
		A3 / A2	0.95	0.99	0.07	0.07	0.98	1.00
	1obj+H	A2 / A1	1.00	0.94	1.00	0.12	1.00	0.99
		A3 / A2	0.93	0.95	0.08	0.07	0.98	1.02
	1type+H	A2 / A1	1.00	0.92	1.00	0.32	1.00	0.99
		A3 / A2	0.93	0.85	0.07	0.07	0.98	1.00
chart	1call+H	A2 / A1	1.00	0.95	1.00	0.29	1.00	0.99
		A3 / A2	0.90	0.98	0.08	0.08	0.98	1.00
	1obj+H	A2 / A1	1.00	0.91	1.00	0.12	1.00	0.99
		A3 / A2	0.85	0.88	0.09	0.08	0.97	1.03
	2obj+H	A2 / A1	1.00	0.46	1.00	0.16	1.00	0.87
		A3 / A2	0.67	0.78	0.09	0.08	0.88	0.95
	1type+H	A2 / A1	1.00	0.93	1.00	0.29	1.00	0.98
		A3 / A2	0.89	0.75	0.08	0.08	0.97	1.00
eclipse	1call+H	A2 / A1	1.00	0.90	1.00	0.28	1.00	0.98
		A3 / A2	0.74	0.96	0.08	0.08	0.91	0.99
	1obj+H	A2 / A1	1.00	0.83	1.00	0.11	1.00	0.97
		A3 / A2	0.66	0.76	0.08	0.08	0.87	1.04
	2obj+H	A2 / A1	1.00	0.51	1.00	0.15	1.00	0.86
		A3 / A2	0.56	0.84	0.09	0.09	0.76	0.97
	1type+H	A2 / A1	1.00	0.92	1.00	0.29	1.00	0.97
		A3 / A2	0.71	0.54	0.08	0.08	0.90	0.99
xalan	1call+H	A2 / A1	1.00	0.93	1.00	0.29	1.00	0.98
		A3 / A2	0.78	0.97	0.08	0.08	0.92	0.99
	1obj+H	A2 / A1	1.00	0.76	1.00	0.12	1.00	0.98
		A3 / A2	0.63	0.86	0.08	0.08	0.83	1.00
	1type+H	A2 / A1	1.00	0.89	1.00	0.29	1.00	0.97
		A3 / A2	0.70	0.58	0.08	0.08	0.87	0.96

	AVG	A2 / A1	1.00	0.84	1.00	0.22	1.00	0.97
		A3 / A2	0.80	0.84	0.08	0.08	0.92	0.99
	MEDIAN	A2 / A1	1.00	0.91	1.00	0.27	1.00	0.98
		A3 / A2	0.83	0.86	0.08	0.08	0.94	1.00

legend
A2 / A1 = ratio of alternative 2 (“no merge”) to alternative 1 (“no merge + sensitive”)
A3 / A2 = ratio of alternative 3 (“merge”) to alternative 2 (“no merge”)

Table 5.6: Ratios of metrics concerning performance for a variety of analyses.

/ context-sensitive call graph edges (*edges* / *c-s edges*), reachable methods (*meths*), reachable virtual call sites (*v-calls*), polymorphic virtual call sites (*poly v-calls*), reachable casts (*casts*), and reachable casts that may fail (*fail casts*).

The metrics that are significant for precision are the context-*insensitive* ones (the end-user visible ones). Their context-sensitive variations are internal metrics of the analysis, but are included here for comparison among different alternatives, for an insight on their impact on performance. For instance, the context-sensitive call-graph edge relation is larger in the full context-sensitive alternative when compared to the one with the representatives (i.e., merged exceptions), but the end result (the context-*insensitive* metric) is the same in both cases.

Thus, it is apparent that the alternative with the representatives is more efficient. It is clear that all the alternative methods for exception treatment achieve the same levels of precision (while having different execution costs). One metric that has in some cases a noticeable, but small, loss of precision is the `VarPointsTo` relation (to non-throwable objects). This is within acceptable levels (the three most significant digits remain the same), and does not affect other significant metrics (i.e., the number of reachable virtual call sites).

			metrics									
prog	analysis	alt.	vars	c-s vars	edges	c-s edges	meths	c-s meths	v-calls	poly v-calls	casts	fail casts
antlr	1call+H	sens	1,560,753	37,225,255	42,351	187,937	5,728	43,373	27,761	1,280	1,033	690
		insens	1,560,753	37,225,255	42,351	187,937	5,728	43,373	27,761	1,280	1,033	690
		merge	1,560,753	37,225,255	42,351	187,937	5,728	43,373	27,761	1,280	1,033	690
	1obj+H	sens	760,457	47,896,816	41,254	946,597	5,692	81,949	27,599	1,253	1,009	612
		insens	760,457	47,896,816	41,254	946,597	5,692	81,949	27,599	1,253	1,009	612
		merge	760,724	42,501,811	41,254	882,198	5,692	67,759	27,599	1,253	1,009	612
	2obj+H	sens	371,627	7,920,770	39,757	2,746,197	5,609	354,112	27,188	1,147	970	404
		insens	371,659	7,644,418	39,757	2,313,888	5,609	265,687	27,188	1,147	970	404
		merge	372,014	6,921,879	39,757	2,065,857	5,609	196,585	27,188	1,147	970	404
	1type+H	sens	2,183,345	12,318,820	41,755	198,555	5,701	23,743	27,610	1,264	1,011	684
		insens	2,183,345	12,318,820	41,755	198,555	5,701	23,743	27,610	1,264	1,011	684
		merge	2,182,913	10,691,398	41,755	182,900	5,701	20,467	27,610	1,264	1,011	679
bloat	1call+H	sens	3,697,206	152,427,121	51,006	301,356	6,988	51,892	25,367	1,432	2,083	1,659
		insens	3,697,206	152,427,121	51,006	301,356	6,988	51,892	25,367	1,432	2,083	1,659
		merge	3,697,206	152,427,121	51,006	301,356	6,988	51,892	25,367	1,432	2,083	1,659
	1obj+H	sens	1,921,000	72,254,097	47,792	1,770,781	6,945	94,878	25,220	1,406	2,062	1,544
		insens	1,921,000	72,254,097	47,792	1,770,781	6,945	94,878	25,220	1,406	2,062	1,544
		merge	1,921,064	69,519,850	47,792	1,708,956	6,945	82,554	25,220	1,406	2,062	1,544
	1type+H	sens	2,345,182	17,148,041	48,610	310,309	6,956	30,405	25,230	1,463	2,064	1,663
		insens	2,345,182	17,148,041	48,610	310,309	6,956	30,405	25,230	1,463	2,064	1,663
		merge	2,345,022	15,094,373	48,610	292,721	6,956	26,840	25,230	1,463	2,064	1,663
chart	1call+H	sens	2,439,946	45,066,857	43,965	206,396	8,432	45,049	23,777	1,150	1,714	1,236
		insens	2,439,946	45,066,857	43,965	206,396	8,432	45,049	23,777	1,150	1,714	1,236
		merge	2,439,946	45,066,857	43,965	206,399	8,432	45,049	23,777	1,150	1,714	1,236
	1obj+H	sens	1,178,269	80,561,690	41,628	1,064,777	8,339	106,799	23,384	1,104	1,668	1,021
		insens	1,178,269	80,561,690	41,628	1,064,777	8,339	106,799	23,384	1,104	1,668	1,021
		merge	1,178,275	72,033,464	41,628	928,207	8,339	86,921	23,384	1,104	1,668	1,021
	2obj+H	sens	290,102	17,698,766	38,576	8,337,762	8,171	616,870	22,844	902	1,603	737
		insens	290,133	17,398,268	38,576	7,319,011	8,171	512,984	22,844	902	1,603	737
		merge	290,497	16,971,013	38,576	7,016,882	8,171	435,350	22,844	902	1,603	737
	1type+H	sens	1,734,811	24,663,696	42,757	306,991	8,380	37,003	23,511	1,131	1,690	1,234
		insens	1,734,811	24,663,696	42,757	306,991	8,380	37,003	23,511	1,131	1,690	1,234
		merge	1,734,680	18,914,653	42,757	272,848	8,380	31,034	23,511	1,131	1,690	1,229
eclipse	1call+H	sens	965,085	25,460,908	34,945	198,463	6,483	36,367	18,231	815	1,261	751
		insens	965,085	25,460,908	34,945	198,463	6,483	36,367	18,231	815	1,261	751
		merge	965,194	25,464,554	34,945	198,473	6,483	36,367	18,231	815	1,261	751
	1obj+H	sens	521,033	49,472,319	32,103	775,329	6,318	89,597	17,840	745	1,228	682
		insens	521,033	49,472,319	32,103	775,329	6,318	89,597	17,840	745	1,228	682
		merge	521,790	38,394,794	32,106	632,852	6,318	71,126	17,840	745	1,228	682
	2obj+H	sens	239,983	19,644,130	29,513	5,945,598	6,189	441,164	17,259	625	1,177	498
		insens	239,999	18,193,814	29,513	4,957,980	6,189	334,504	17,259	625	1,177	498
		merge	240,727	17,563,264	29,516	4,665,960	6,189	236,259	17,259	625	1,177	498
	1type+H	sens	766,827	28,964,394	33,087	310,708	6,431	33,398	18,047	778	1,236	789
		insens	766,827	28,964,394	33,087	310,708	6,431	33,398	18,047	778	1,236	789
		merge	767,461	16,047,991	33,088	266,833	6,431	28,152	18,047	778	1,236	784
xalan	1call+H	sens	1,342,717	39,599,466	37,969	177,150	7,313	39,370	20,046	1,249	1,296	851
		insens	1,342,717	39,599,466	37,969	177,150	7,313	39,370	20,046	1,249	1,296	851
		merge	1,342,907	39,601,828	37,976	177,194	7,313	39,377	20,046	1,250	1,296	851
	1obj+H	sens	502,294	90,538,691	35,908	870,727	7,237	102,393	19,828	1,175	1,264	666
		insens	502,294	90,538,691	35,908	870,727	7,237	102,393	19,828	1,175	1,264	666
		merge	502,805	81,438,319	35,911	710,588	7,237	84,497	19,828	1,176	1,264	666
	1type+H	sens	736,952	31,829,242	36,731	315,005	7,268	36,627	19,888	1,194	1,271	836
		insens	736,952	31,829,242	36,731	315,005	7,268	36,627	19,888	1,194	1,271	836
		merge	737,178	19,268,160	36,733	265,394	7,268	30,892	19,888	1,195	1,271	831

Table 5.7: Metrics concerning precision for a variety of analyses.

Chapter 6

Conclusions

By using Datalog we were able to succinctly express a declarative joint exception and points-to analysis for Java. We were also able to express three alternative methods for exception treatment, by modifying only a few lines of code each time. Our analysis was built on top of the Doop framework [3] which allowed the convenient decoupling of the choice of context from the exception analysis code.

As following from previous work [2], we chose to do a joint exception and points-to analysis because of its benefits both in precision and performance. In order to further reduce the execution time of the analysis, we investigated how the treatment of exceptions affected both precision and performance, and we proposed three alternative methods for their handling.

The new insight was that the type of each exception suffices for an accurate points-to analysis (although losing precision in exception-specific metrics). Thus, we proposed a method for exception handling, where each exception object of the same type is merged and one representative object is used instead. This results in significant improvements in performance, while barely having any loss in precision.

We evaluated the methods we proposed, on the DaCapo benchmark suite. The experimental results confirmed our initial insights and supported our claim that the usage of type representatives for exception objects is an effective way of reducing the execution time of an analysis.

Acronyms and Abbreviations

Abbreviation	Full Name
insens	context-insensitive analysis
1obj	1-object-sensitive analysis
1obj+H	1-object-sensitive+heap analysis
2obj+H	2-object-sensitive+heap analysis
2obj+2H	2-object-sensitive+2-heap analysis
1type+H	1-type-sensitive+heap analysis
2type+H	2-type-sensitive+heap analysis
1call	1-call-site-sensitive analysis
1call+H	1-call-site-sensitive+heap analysis
2call+H	2-call-site-sensitive+heap analysis
2call+2H	2-call-site-sensitive+2-heap analysis
no-merge+sens	full context-sensitive exceptions
no-merge	context-insensitive exceptions
merge	type representatives for exceptions
LB	LogicBlox Inc.

Appendix A

Exception Analysis Code

```
/**
 * Represents the heap abstractions of the exceptions a method can
 * throw.
 */
ThrowPointsTo(?heapCtx, ?heap, ?ctx, ?method) ->
  HContext(?heapCtx), HeapAllocationRef(?heap),
  Context(?ctx), MethodSignatureRef(?method).

/*****************************************************************************
 *
 * Throw statements
 *
 *****************************************************************************/

/**
 * A method throws an exception in a context if there is a throw
 * statement in the method, and the thrown variable points to an
 * object in this context, but this object is not immediately caught
 * by an exception handler (ThrowPointsTo rule).
 *
 * If the object is a caught, then it is assigned to the formal
 * parameter of the exeception handler (VarPointsTo rule).
 */

ThrowPointsTo(?heapCtx, ?heap, ?ctx, ?method) <-
  Throw(?ref, ?var),
  VarPointsTo(?heapCtx, ?heap, ?ctx, ?var),
  HeapAllocation:Type[?heap] = ?heaptype,
  !(ExceptionHandler[?heaptype, ?ref]=_),
  Instruction:Method[?ref] = ?method.

VarPointsTo(?heapCtx, ?heap, ?ctx, ?param) <-
```

```
  Throw(?ref, ?var),
  VarPointsTo(?heapCtx, ?heap, ?ctx, ?var),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?ref] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.

/*****************************************************************************
 *
 * Method invocations
 *
 *****************************************************************************/

/**
 * A method M1 throws an exception in a context if there is a call
 * graph edge from an invocation in M1 to some method M2 and the
 * method M2 throws a an exception for this specific
 * (context-sensitive) call graph edge. Also, the exception should not
 * be caught immediately by an exception handler in M1 (ThrowPointsTo
 * rule).
 *
 * If there is such an exception handler, then the exception object is
 * assigned to the formal parameter of the exception handler
 * (VarPointsTo rule).
 */

ThrowPointsTo(?heapCtx, ?heap, ?callerCtx, ?callerMethod) <-
  CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?tomethod),
  ThrowPointsTo(?heapCtx, ?heap, ?calleeCtx, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  !(ExceptionHandler[?heaptype,?invocation]=_),
  Instruction:Method[?invocation] = ?callerMethod.

VarPointsTo(?heapCtx, ?heap, ?callerCtx, ?param) <-
  CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?tomethod),
  ThrowPointsTo(?heapCtx, ?heap, ?calleeCtx, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype,?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.


/*****************************************************************************
```

```
 *
 * Compute for an instruction which exception handlers handle which
 * exception types.
 *
 ******************************************************************************/

// Note how this logic is superlinear. We keep
// relations that link every exception handler to every relevant (i.e., throw
// or methcall) instruction under its range, and to every type that the
// exception handler can handle, including all subtypes of the declared type.
// It is not easy to change this, nor perhaps too valuable. But it is certainly
// a spot where bottom-up evaluation with an explicit representation hurts us.
// We have very large ExceptionHandler, PossibleExceptionHandler, etc. relations.
// Note: currently exception objects are allocated context-insensitively.
/**
 * An exception of a specific type, thrown at an instruction, is
 * handled by an exception handler.
 */
ExceptionHandler[?type, ?instruction] = ?handler ->
  ExceptionHandlerRef(?handler), Type(?type), InstructionRef(?instruction).

ExceptionHandler[?type, ?instruction] = ?handler <-
  PossibleExceptionHandler(?handler, ?type, ?instruction),
  ! ImpossibleExceptionHandler(?handler, ?type, ?instruction).

/**
 * An exception type that is caught by an earlier exception handler
 * (not ?handler).
 */

ImpossibleExceptionHandler(?handler, ?type, ?instruction) ->
  ExceptionHandlerRef(?handler), Type(?type), InstructionRef(?instruction).

ImpossibleExceptionHandler(?handler, ?type, ?instruction) <-
  PossibleExceptionHandler(?handler, ?type, ?instruction),
  ExceptionHandler:Before(?previous, ?handler),
  PossibleExceptionHandler(?previous, ?type, ?instruction).

/**
 * All possible handlers of an exception type for an instruction.
 */
```

```
PossibleExceptionHandler(?handler, ?type, ?instruction) ->
  ExceptionHandlerRef(?handler),
  Type(?type),
  InstructionRef(?instruction).

PossibleExceptionHandler(?handler, ?type, ?instruction) <-
  ExceptionHandler:InRange(?handler, ?instruction),
  ExceptionHandler:Type[?handler] = ?type.

PossibleExceptionHandler(?handler, ?subtype, ?instruction) <-
  ExceptionHandler:InRange(?handler, ?instruction),
  ExceptionHandler:Type[?handler] = ?type,
  Superclass(?subtype, ?type).

/**
 * Instructions that are in the range of an exception handler.
 */
ExceptionHandler:InRange(?handler, ?instruction) ->
  ExceptionHandlerRef(?handler),
  InstructionRef(?instruction).

ExceptionHandler:InRange(?handler, ?instruction) <-
  Instruction:Method[?instruction] = ?method,
  ExceptionHandler:Method(?handler, ?method),
  Instruction:Index[?instruction] = ?index,
  ExceptionHandler:Begin[?handler] = ?begin,
  ?begin <= ?index,
  ExceptionHandler:End[?handler] = ?end,
  ?index < ?end.

/**
 * Transitive closure of ExceptionHandler:Previous.
 */
ExceptionHandler:Before(?before, ?handler) ->
  ExceptionHandlerRef(?before),
  ExceptionHandlerRef(?handler).

ExceptionHandler:Before(?previous, ?handler) <-
  ExceptionHandler:Previous[?handler] = ?previous.

ExceptionHandler:Before(?before, ?handler) <-
```

```
  ExceptionHandler:Before(?middle, ?handler),
  ExceptionHandler:Previous[?middle] = ?before.

InRangeOfExceptionHandler(?instruction) -> InstructionRef(?instruction).
InRangeOfExceptionHandler(?instruction) <-
  ExceptionHandler:InRange(_, ?instruction).
```

```
/***********************************************************
 * Special objects
 *
 * Some objects are so common that they heavily impact performance if
 * every allocation is distinguished or a context-sensitive heap
 * abstraction is used. In many cases, this precision is not actually
 * useful for a points-to analysis, so handling them in a less precise
 * way is beneficial.
 ***********************************************************/

/**
 * Objects that should not be allocated as normal.
 */
HeapAllocation:Special(?heap) -> HeapAllocationRef(?heap).

/**
 * Objects that should use a context-insensitive heap abstraction.
 */
HeapAllocation:ContextInsensitive(?heap) ->
  HeapAllocationRef(?heap).

HeapAllocation:Special(?heap) <-
  HeapAllocation:ContextInsensitive(?heap).

/**
 * Objects that should be merged to some heap abstraction (implies context-
    insensitive)
 */
HeapAllocation:Merge[?heap] = ?mergeHeap ->
  HeapAllocationRef(?heap),
  HeapAllocationRef(?mergeHeap).

```

```
/**
 * Join with AssignHeapAllocation for performance.
 */
AssignNormalHeapAllocation(?heap, ?var, ?inmethod) <-
  AssignHeapAllocation(?heap, ?var, ?inmethod),
  ! HeapAllocation:Special(?heap).

HeapAllocation:Special(?heap) <-
  HeapAllocation:Merge[?heap] = _.

AssignContextInsensitiveHeapAllocation(?mergeHeap, ?var, ?inmethod) <-
  AssignHeapAllocation(?heap, ?var, ?inmethod),
  HeapAllocation:Merge[?heap] = ?mergeHeap.

AssignContextInsensitiveHeapAllocation(?heap, ?var, ?inmethod) <-
  AssignHeapAllocation(?heap, ?var, ?inmethod),
  HeapAllocation:ContextInsensitive(?heap).


/*************************************************************
 * Exceptions
 *************************************************************/

/*
 // Context-Insensitive Treatment:
 // This is the original, precise and straightforward treatment
 // of throwables. They were allocated context insensitively. This still
 // produced huge ThrowPointsTo sets and caused slowdowns.
 */

HeapAllocation:ContextInsensitive(?heap) <-
  HeapAllocation:Type[?heap] = ?heaptype,
  Type:Value(?throw:"java.lang.Throwable"),
  AssignCompatible(?throw, ?heaptype).


// Type Representatives Treatment:
// The optimized treatment represents every exception (i.e., throwable)
// object by a unique representative of the same type. All exception
// objects of the same type are therefore merged. This means that points-to
// results for throwables are not accurate! Only the type will be right.
```

```

TypeToHeap(?heap, ?heaptype) <-
  HeapAllocation:Type[?heap] = ?heaptype,
  Type:Value(?throw:"java.lang.Throwable"),
  AssignCompatible(?throw, ?heaptype).

// Quadratic but so local that it shouldn't matter, ever.
HeapRepresentative[?heap] = ?representativeHeap <-
  agg<<?representativeHeap = min(?otherHeap)>>(TypeToHeap(?otherHeap,
     HeapAllocation:Type[?heap])).

HeapAllocation:Merge[?heap] = ?mergeHeap <-
  HeapRepresentative[?heap] = ?mergeHeap.
```

References

[1] Ole Agesen. The Cartesian product algorithm. In *ECOOP '95, Object-Oriented Programming: 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 2–51, 1995.

[2] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In Laura Dillon, editor, *ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis*, New York, NY, USA, July 2009.

[3] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2009. ACM.

[4] Martin Bravenboer and Yannis Smaragdakis. Pick your contexts well: Understanding object-sensitivity. In *POPL '11: 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Austin, Texas, USA, 2011. ACM.

[5] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of points-to analysis of Java in the presence of exceptions. *IEEE Trans. Softw. Eng.*, 27(6):481–512, 2001.

[6] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08: Proc. of the 30th int. conf. on Software engineering*, pages 391–400, New York, NY, USA, 2008. ACM.

[7] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness testing of Java server applications. *IEEE Trans. Softw. Eng.*, 31(4):292–311, 2005.

[8] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of java web services for robustness. In *ISSTA '04: Proceedings of the 2004 International Symposium on Software Testing and Analysis*, 2004.

[9] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with Datalog. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pages 2–27. Spinger, 2006.

[10] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM.

[11] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[12] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53, 2008.

[13] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In Michael D. Ernst and Thomas P. Jensen, editors, *PASTE*, pages 6–12. ACM, 2005.

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

[15] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 308–319, 2006.

[16] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Precise concrete type inference for object-oriented languages*, pages 324–340, New York, NY, USA, 1994. ACM.

[17] Thomas Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

[18] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, pages 189–233, Englewood Cliffs, NJ, 1981. Prentice-Hall, Inc.

[19] Olin Shivers. *Control-Flow Analysis on Higher-Order Languages*. PhD thesis, Carnegie Melon University, May 1991.

[20] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280. ACM Press, 2000.

[21] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[22] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.