Exception Analysis and Points-to Analysis: Better Together

Martin Bravenboer
Department of Computer Science
University of Massachusetts, Amherst
martin.bravenboer@acm.org

Yannis Smaragdakis
Department of Computer Science
University of Massachusetts, Amherst
yannis@cs.umass.edu

ABSTRACT

Exception analysis and points-to analysis are typically done in complete separation. Past algorithms for precise exception analysis (e.g., pairing throw clauses with catch statements) use precomputed points-to information. Past points-to analyses either unsoundly ignore exceptions, 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). We show that this separation results in significant slowdowns or vast imprecision. The two kinds of analyses are interdependent: neither can be performed accurately without the other. The interdependency leads us to propose a joint handling for performance and precision. We show that our exception analysis is expressible highly elegantly in a declarative form, and can apply to points-to analyses of varying precision. In fact, our specification of 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. Our combined approach achieves similar precision relative to exceptions (exception-catch links) as the best past precise exception analysis, with a runtime of seconds instead of tens of minutes. At the same time, our analysis achieves much higher precision of points-to information (an average of half as many values for each reachable variable for most of the DaCapo benchmarks) than points-to analyses that treat exceptions conservatively, all at a fraction of the execution time.

Categories and Subject Descriptors. F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; D.1.6 [**Programming Techniques**]: Logic Programming

General Terms. Algorithms, Languages, Performance

1. INTRODUCTION AND MOTIVATION

Static program analysis is a domain of mutual recursion. A well-known scenario is that of points-to analysis and call-graph construction. *Points-to* analysis computes what objects (represented by a static abstraction, such as their allocation site) a program variable can point to. *Call-graph construction* computes which methods can call which others. To compute accurate points-to information, we need call-graph information—for instance, an assignment "`x = y.f()`" necessitates knowing what methods can be called, to find what objects `x` can point to. Similarly, computing the call-graph requires having points-to information: knowing what objects a variable can point to is necessary for resolving virtual method calls. The benefits of defining a joint analysis that computes points-to and call-graph information gradually together (commonly called *on-the-fly call-graph construction*) have been well documented in past precision studies [19].

In this paper, we show that a different kind of mutually recursive definition has an even more significant impact on the precision and speed of the defined analyses. *Exception analysis* (i.e., a computation of what control-flow is induced by throwing exceptions) and points-to analysis are mutually recursive in object-oriented languages. The logical mutual recursion between the two analyses is not hard to see conceptually. First, exception analysis depends on points-to information mainly because call-graph information determines which exception handlers can be in the dynamic scope of a thrown exception. Other dependencies also exist—e.g., in a clause "`throw e;`" the points-to information of variable `e` determines what object can be thrown, which also determines which `catch` clauses can be executed. Second, points-to information directly depends on exception analysis. Exceptions change the control flow of a program, so they enable or disable object assignments. Furthermore, throwing an exception directly introduces an object assignment: that of the thrown object being assigned to the formal variable in the `catch` clause.

Motivation. From a practical standpoint, integrating precise handling of exceptions when performing points-to analysis is a virtual necessity. In the past, practical points-to analysis algorithms have relied on conservative approximations of exception handling [17, 19]. (We present a full treatment of related work in Section 7 but discuss the closest comparables here.) The well-known points-to analysis libraries Spark [17] and Paddle [16] both model exception throwing as an assignment to a single global variable for all exceptions thrown in a program. The variable is then read at the site of an exception catch. This approach is highly imprecise because it ignores the information about what exceptions can propagate to a catch site. Conservative exception handling is a dominant factor in precision metrics for these points-to frameworks, especially for more precise points-to analyses. For example, we measured the average number of object abstractions that a reachable variable can point to, as computed by a 1-object-sensitive analysis for the DaCapo antlr benchmark. Conservative exception handling (as in Spark or Paddle) yields 21 objects per variable, while our precise exception handling computes just 10 objects per variable. That is, imprecise exception handling alone is responsible for a 2× worse precision for the points-to analysis! (Full results for more programs and metrics are reported in Section 5.)

Spark and Paddle's purpose is to perform a points-to analysis and only incidentally deal with exceptions, as required for soundness. For instance, neither of them explicitly computes exception information (e.g., a list of exceptions that can be thrown by a

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

method). On the opposite side are precise exception handling algorithms proposed in past work [8–10]. Fu and Ryder's "exception-chain analysis" [9] is representative, and probably the most advanced. Such specialized exception analyses work as a second step, on top of a pre-computed points-to analysis and call-graph (first step). Fu and Ryder's analysis works on top of SPARK, with its conservative modeling of exceptions, as discussed. Thus, this approach models the interdependencies of the points-to and exception analyses crudely, but then performs an exception analysis that is fully precise, to the extent that the result is not affected by the noise added to the points-to analysis in the first step. Even when the precision of this exception analysis is not affected, the run-time overhead of such a separate analysis is significant: Fu et al. report exception analysis times measured in *minutes* for computing a *single* exception-catch link. Furthermore, we have the paradox that the *precise* exception analysis of the second step has to suffer the cost of first computing the imprecise call-graph (we show that this can be up to 6× larger than the precise call-graph), which is due to the *imprecise* exception analysis of the first step!

Our approach. To address these shortcomings, we propose a joint exception analysis and points-to analysis (also with on-the-fly call-graph construction) for Java. Perhaps surprisingly, not much past work has concentrated on points-to analysis in the presence of exceptions, except to theoretically characterize the worst case complexity of the fully precise versions of the problem [3]. We introduce points-to analysis in the presence of exceptions using a modular specification. Indeed, our exception analysis logic on its own is "as precise as can be" since it fully models the Java exception handling semantics. The approximation introduced is only due to the static abstractions used for contexts and objects in the points-to analysis. Thus, our exception analysis is specified in a form that applies to points-to analyses of varying precision, and the exception analysis transparently inherits the points-to analysis precision.

Our joint analysis is implemented in the DOOP pointer analysis framework for Java.[1] DOOP builds on the idea of specifying pointer analysis algorithms declaratively, using Datalog: a logic-based language for defining (recursive) relations. In other work [1], we show how DOOP carries the declarative approach further than past work by describing the full end-to-end analysis in Datalog and optimizing aggressively through exposition of the representation of relations (e.g., indexing) to the Datalog language level. As a result, the declarative pointer analysis implementations of DOOP outperform the previous state of the art in context-sensitive pointer analysis by an order of magnitude.

In summary, our paper makes the following contributions:

- We present a modular exception analysis specification, applicable to a variety of base points-to analyses and operating mutually recursively.
- We experimentally demonstrate the benefits of joint exception and points-to analysis.
  - Compared to past points-to analyses with conservative exception handling, we achieve much higher precision, especially for context-sensitive analyses. We demonstrate this precision in metrics such as the size of the points-to sets (which, for the most precise analysis, is reduced in half on most analyzed programs).
  - Compared to past precise exception analyses, we achieve comparable precision in a small fraction of the analysis time—in the order of 90sec instead of 4,000+sec.

[1] Available at `http://doop.program-analysis.org`

2. BACKGROUND: POINTS-TO ANALYSIS IN DATALOG

The DOOP framework expresses points-to analysis algorithms in Datalog, a declarative language. The use of Datalog enables elegant and modular analysis specifications, but also easy illustration of insights: the declarative nature of our algorithm is a large reason why we are able to illustrate it so succinctly in the next section. As we show in other work [1], the use of Datalog with appropriate optimization not only does not slow down the analysis, but makes DOOP much faster than competing pointer analysis frameworks.

The use of deductive databases and logic programming languages for program analysis has a long history (e.g., [5, 21]) and has raised excitement again recently [6, 11, 25, 26]. Like our work, much of the past emphasis has been on using the Datalog language. Datalog is a logic programming language originally introduced in the database domain. At a first approximation, one can view Datalog as either "SQL with full recursion" or "Prolog without constructors/functions". The essence of the language is its ability to define recursive relations. Relations (or equivalently *predicates*) are the main Datalog data type. Computation consists of inferring the contents of all relations from a set of input relations. For instance, in our pointer analysis domain, it is easy to represent the relevant actions of a Java program as relations, typically stored as database tables. Consider two such relations, `AssignHeapAllocation(?heap, ?var)` and `Assign(?to, ?from)`. (We follow the convention of capitalizing the first letter of relation names, while writing variable names in lower case and prefixing them with a question-mark.) The former relation represents all occurrences in the program of an instruction "`a = new A();`" where a heap object is allocated and assigned to a variable. That is, a pre-processing step takes a Java program (in our implementation this is in intermediate, bytecode, form) as input and produces the relation contents. A static abstraction of the heap object is captured in variable `?heap`—it can be concretely represented as, e.g., 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) <-
2    AssignHeapAllocation(?heap, ?var).
3 VarPointsTo(?heap, ?to) <-
4    Assign(?to, ?from), VarPointsTo(?heap, ?from).
```

The Datalog program consists of a series of *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. We use the left arrow symbol (`<-`) to separate the inferred fact (the *head*) from the previously established facts (the *body*). For instance, lines 3-4 above say 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 (lines 1-2), as well as the recursion in the definition of `VarPointsTo` (lines 3-4).

The declarativeness of Datalog makes it attractive for specifying complex program analysis algorithms. Particularly important is the ability to specify recursive definitions—as we discussed in the Introduction, 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

elegance of the declarative approach is evident when contrasted with common implementations of points-to analyses.

Datalog evaluation is guaranteed to be *bottom-up*, meaning that known facts are propagated using the rules until a maximal set of derived facts is reached. This is also the link to the data processing intended domain of Datalog: evaluation of a rule can be thought of as a sequence of relational algebra joins and projections. For instance, the evaluation of lines 3-4 in our above example can be thought of as: Take the join of relation `Assign` with relation `VarPointsTo` over the first column of both (because of common field `?from`) and project the join result on fields `?to` and `?heap`. The result of the projection is added to relation `VarPointsTo` (skipping duplicates) and forms the value of `VarPointsTo` for the next iteration step. Application of all rules iterates to fixpoint. Note that this means that the evaluation of a Datalog program comprises two distinct kinds of looping/iteration activities: the relational algebra joins and projections, and the explicit recursion of the program. The former kind of looping is highly efficient through traditional database optimizations (e.g., for join order, group-fetching of data from disk, locality of reference, etc.).

We use a commercial Datalog engine, marketed by LogicBlox Inc. This version of Datalog allows "stratified negation", i.e., negated clauses, as long as the negation is not part of a recursive cycle. It also allows specifying that some relations are functions, i.e., 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. We will see these features in action in our algorithm specification, next.

3. JOINT POINTS-TO AND EXCEPTION ANALYSIS

We next describe our joint points-to and exception analysis in Doop. To a large extent, the reason the algorithm can be specified concisely and modularly is that it is in declarative form.

The relevant parts of exception handling in Java consist of declaring and throwing exceptions (which are regular Java objects), as well as catching them. An extra element, the Java `finally` clause, often causes headaches for static analysis purposes [3] but is a non-issue in our context. We perform all analysis on bytecode. At that level, all uses of `finally` have already been translated away by the Java compiler: the finally block is copied appropriately (in JDK 1.4 and later) and executed on any possible exit point, normal or exceptional, of the `try` block or any local `catch` block.

The logic of our analysis models the Java exception handling semantics. Namely, from the perspective of the Doop points-to analysis, exceptions introduce an interprocedural layer of assignments over the normal source code. At `throw` statements, normal objects flow into 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.

The exception handling logic is shown in Figures 1-6, with the meaning of input predicates described in Figure 7 and the meaning of computed predicates described in Figure 8. Recall that the input predicates are straightforward relational representations of the bytecode instructions of a program and are produced by a preprocessing step. We next describe the important elements of the rules.

- We use some extensions and notational conventions in the code. First, some of our relations are functions, and the functional notation "`Relation[?domainvar] = ?val`" is used instead of the relational notation, "`Relation(?domainvar, ?val)`". Semantically the two are equivalent, only 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. Second, the colon (`:`) in relation names is just a regular character with no semantic significance—we use common prefixes ending with a colon as a lexical convention for grouping related predicates, such as `ExceptionHandler:Before` and `ExceptionHandler:InRange`. Third, numeric inequality on values is supported with standard operators (<, <=), as in Figure 1. (Adding an input-domain-ordering relation to Datalog is a very common extension that makes the language equivalent to the PTIME complexity class, i.e., every polynomial-time problem is expressible in Datalog and every Datalog computation is guaranteed polynomial-time [12, p.225].)

- The analysis has two quite separate parts: Figures 1-4 and Figures 5-6. The first reason these parts are distinct is that the former

```
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.
```

Figure 1: Instructions in the range of an exception handler. A handler is in range if it is defined in the same method and the instruction is between the exception handler's begin index and end index.

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

PossibleExceptionHandler(?handler, ?subtype, ?instr) <-
  PossibleExceptionHandler(?handler, ?type, ?instr),
  Superclass(?subtype, ?type).
```

Figure 2: Possible exception handlers. An exception handler is possible if it is in range and it handles the exception type or any of its supertypes.

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

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

InfeasibleExceptionHandler(?handler, ?type, ?instr) <-
  PossibleExceptionHandler(?handler, ?type, ?instr),
  ExceptionHandler:Before(?previous, ?handler),
  PossibleExceptionHandler(?previous, ?type, ?instr).
```

Figure 3: Filter for possible exception handlers. Exception handlers are ordered in the input program. An exception handler is infeasible if another possible exception handler is before it. Defining "infeasible" only for handlers that would be otherwise "possible" is useful for performance reasons, although not conceptually necessary.

```
ExceptionHandler[?type, ?instr] = ?handler <-
  PossibleExceptionHandler(?handler, ?type, ?instr),
  not InfeasibleExceptionHandler(?handler,?type,?instr).
```

Figure 4: Precise exception handler for type and instruction. The appropriate exception handler for a type and instruction is the single possible exception handler that is not disqualified by the "infeasible" handlers filter.

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

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

Figure 5: 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 `?callerMethod`. `VarPointsTo`: If there is such an exception handler, then the exception `?heap` is assigned to the formal parameter `?param` of the exception handler.

```
ThrowPointsTo(?heap, ?method) <-
  Throw(?instr, ?var),
  VarPointsTo(?heap, ?var),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?instr],
  Instruction:Method[?instr] = ?method.

VarPointsTo(?heap, ?param) <-
  Throw(?instr, ?var),
  VarPointsTo(?heap, ?var),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?instr] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

Figure 6: Propagation of exceptions for throw instructions. `ThrowPointsTo`: A method throws an exception if there is a throw statement in the method, and the thrown variable `?var` points to an object `?heap` that is not immediately caught by an exception handler. `VarPointsTo`: If the object is caught, then it is assigned to the formal parameter of the exception handler.

consists of fully general rules that apply to *any* Doop points-to analysis. The latter part is the code linking the exception analysis to the main relations of the host points-to analysis. The rules shown in Figures 5-6 are for a context-insensitive points-to analysis. As we will see in the next section, for context-sensitive analyses (e.g., 1-call-site-sensitive, 1-object-sensitive, 1-call-site-sensitive with heap cloning) these rules have more complex counterparts but with essentially the same logic flow. The extra complexity is due to the representation of contexts.

- Another way to view the two distinct parts of the exception analysis logic is by noting that Figures 1-4 represent a completely *intraprocedural* computation. Their purpose is to finally compute relation `ExceptionHandler` of Figure 4. Note that this relation is a function and computes, for every instruction and exception type, a single (intraprocedural) exception handler that gets called for the corresponding exception. In contrast, Figures 5-6 add the *interprocedural* logic: they define a relation `ThrowsPointsTo` which encodes which exception *objects* (not types) a certain method can throw. This is a crucial element of the joint handling of exception analysis and points-to analysis. Since the purpose of points-to analysis is to compute which objects a program name can refer to, it is natural to do the same for exception objects and this leads to straightforward interfacing with the rest of the analysis. The `ThrowPointsTo` relation is defined based on the

`Instruction:Method[?instruction] = ?method`
Method that an instruction is in.

`Instruction:Index[?instruction] = ?index`
Index (program counter) of an instruction in a method body.

`ExceptionHandler:Method(?handler, ?method)`
Method with which an exception handler is associated.

`ExceptionHandler:Previous[?handler] = ?previous`
Exception handler `?previous` is the exception handler immediately before exception handler `?handler`. Both exception handlers are associated with the same method.

`ExceptionHandler:Begin[?handler] = ?begin`
Exception handler `?handler` is active starting at the index (program counter) `?begin` in the array of bytecode instructions of the method body. The `?begin` index is inclusive.

`ExceptionHandler:End[?handler] = ?end`
Exception handler `?handler` stops being active at the index `?end` in the array of bytecode instructions of the method body. The `?begin` index is exclusive.

`ExceptionHandler:Type[?handler] = ?type`
Exception handler `?handler` handles exceptions of `?type`. The exception handler is called if the thrown exception is an instance of `?type` or a subclass of `?type`. In Java, this corresponds to the type T in `catch(T exc){...}`.

`ExceptionHandler:FormalParam[?handler] = ?param`
If exception handler `?handler` is invoked, then the thrown exception is assigned to parameter `?param` of the exception handler. In Java, this corresponds to the variable `e` in `catch(T e){...}`.

`HeapAllocation:Type[?heap] = ?heaptype`
The type of an object allocated at site `?heap`.

`Throw(?instr, ?var)`
Throw instruction with key `?instr` throws the value of variable `?var`.

`DirectSuperclass[?subclass] = ?superclass`
Class `?subclass` extends class `?superclass`.

Figure 7: Input Predicates

`ExceptionHandler:Before(?before, ?handler)`
Transitive closure of `ExceptionHandler:Previous`.

`ExceptionHandler:InRange(?handler, ?instruction)`
Exception handlers whose range includes an instruction `?instruction`. In Java bytecode, exception handlers have a begin and end index of the instructions they handle exceptions for.

`PossibleExceptionHandler(?handler, ?type, ?instruction)`
If an exception of type `?type` is thrown directly or indirectly at `?instruction`, then it might be handled by exception handler `?handler`. This predicate ignores the order of exception handlers, so there might be more than one exception handler for a combination of a type and an instruction.

`InfeasibleExceptionHandler(?handler,?type,?instruction)`
For `?instruction`, an exception of type `?type` is never handled by `?handler` because there is an exception handler before `?handler` that handles this type of exception.

`ExceptionHandler[?type, ?instruction] = ?handler`
An exception of a specific `?type`, thrown at an `?instruction`, is handled by an exception handler `?handler`.

`ThrowPointsTo(?heap, ?method)`
Method declaration `?method` may throw heap abstraction `?heap`.

`VarPointsTo(?heap, ?var)`
Variable `?var` may have heap abstraction `?heap` as value.

`CallGraphEdge(?invocation, ?tomethod)`
Method invocation `?invocation` may invoke method `?tomethod`.

`Superclass(?subclass, ?class)`
Transitive closure of `DirectSuperclass` (irreflexive).

Figure 8: Computed Predicates

main relations of the points-to analysis, the VarPointsTo relation and the CallGraphEdge relation. These, in turn, depend on ThrowPointsTo, since the exception analysis adds rules that produce more VarPointsTo facts. In essence, all interprocedurality is handled through the mutually recursive computations of the call-graph, the points-to relation, and the exception throwing relations.

We can observe that the declarative definition of the analysis is the main reason for its power with such conciseness. Defining the complex mutual interdependencies manually would have been very hard. Consider that every Doop points-to analysis already has multiple rules (not shown) that cause the computation of VarPointsTo to depend on call-graph information (e.g., because method reachability is checked, because the result of a method may be assigned to a variable, or because a variable may be a method formal argument and we need to know which method is a possible target at a call site). Similarly, there are many rules that cause the call-graph computation to depend on VarPointsTo (with the most notable case being that of a variable used as a method receiver, when needing to resolve a virtual method call). To these rules, we are adding more complex mutual recursion, 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. To summarize, our analysis can be expressed so concisely in Datalog because of three factors: the high-level operations on relations, the automatic incrementalization, and the mutually recursive definitions. It is interesting to consider how our analysis would be expressed in a different language that offers high-level operations on relations, such as Jedd [18], the substrate of the Paddle program analysis framework. Indeed, the analysis of Figures 1-6 (just like the rest of the Doop logic) has a direct mapping to Jedd. The biggest difference is that recursion needs to be replaced by explicit iteration, where the programmer needs to specify how each relation is used to iteratively compute others, manually emulating the implicit Datalog fixpoint computation.

Another element to note is that the exception handling logic is, in a sense, fully precise. This is not a formal statement and we do not offer a proof for it, 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. Specifically, the intraprocedural rules (Figures 1-4) just express the local Java exception handling semantics precisely. Of course, there is an approximation introduced in our exception analysis, but this comes directly from the abstraction of the host points-to analysis. That is, the logic of Figures 5-6 depends on the static abstraction of heap objects and method/variable contexts chosen. If one were to imagine an "ideal" host points-to analysis, the logic of Figures 5-6 (with a small adaptation to express the "perfect" context) would also produce a fully precise exception analysis. This feature offers a convenient precision knob and is not shared by any other exception analysis in the literature.

4. CONTEXT-SENSITIVITY

Doop supports a range of pointer analyses, all expressed as variations over a common code trunk. The framework currently supports a context-insensitive analysis, a 1-call-site-sensitive analysis, a 1-object-sensitive analysis, as well as versions of the latter two with a context-sensitive heap. Context-sensitivity is important for exception analysis: the results of past precise exception analyses [8–10] can be emulated by our joint exception and points-to analysis only with context-sensitivity.

4.1 Doop Context-Sensitive Pointer Analysis Background

Context-sensitive points-to analysis intends to add precision by having the static abstraction for a program variable be not just the variable's declaration (e.g., the method, location, and name of the variable) but also some context information. The context is typically a sequence of the N top call-sites in the calling stack or of the static abstractions of the receiver objects for the N top calling stack methods. In the former case the analysis is called an *N-call-site-sensitive* analysis and in the latter an *N-object-sensitive* analysis, for various values of N. For instance, a 1-call-site-sensitive analysis can distinguish between objects that flow to variable v in method m when m is called by instruction i of method foo and objects that flow to v in m when m has been called by any different program instruction.

Even more precise analyses add a *context-sensitive heap* (a technique also known as *heap cloning*): the abstraction for objects is enhanced with a context, similar to the context of variables.

Context-sensitive points-to analysis in Doop is, to a large extent, similar to the previously discussed 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.

```
1 Assign(?calleeCtx, ?formal, ?callerCtx, ?actual) <-
2   CallGraphEdge(?callerCtx, ?invocation,
3                 ?calleeCtx, ?method),
4   FormalParam[?index, ?method] = ?formal,
5   ActualParam[?index, ?invocation] = ?actual.
6
7 VarPointsTo(?heap, ?toCtx, ?to) <-
8   Assign(?toCtx, ?to, ?fromCtx, ?from),
9   VarPointsTo(?heap, ?fromCtx, ?from).
```

The example shows how a derived Assign relation (unlike the input relation Assign in the basic example of Section 2) is computed, based on the call-graph information, and then used in deriving a context-sensitive VarPointsTo relation. In Doop these rules can be used for a 1-call-site-sensitive or a 1-object-sensitive analysis. (For more complex contexts, we need to add extra variables, since Datalog does not allow constructors and therefore cannot support value combination. However, we use a simple macro system to abstract over the number of context variables, so the rules are generated from a common code pattern, for any number of contexts.)

The example also shows an element significant for our later experimental discussion: the context-sensitive call-graph. In order to perform a context-sensitive analysis, the context information needs to be propagated in the CallGraphEdge relation. This context-sensitive call-graph is not a user-visible structure but we have found it to be a valuable metric for understanding the precision and cost of our exception analysis.

4.2 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 call-graph. Furthermore, the ThrowPointsTo relation needs to become context-sensitive, in much the same way as VarPointsTo. Figure 9 straightforwardly adapts Figure 5 to the context-sensitive setting.

Why is context-sensitivity important, however? The goal of our joint analysis is employ the well-understood precision mechanisms of a standard points-to analysis in order to match the precision of

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

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

Figure 9: Context-Sensitive Exception Analysis

(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. [8, Figures 6,7] which are used to motivate improvements over the original [10] DataReach algorithm.

A simpler but illustrative example is shown in Figure 10 [8, Figure 4]. If different calls to `BufferedInputStream.read` are not distinguished, 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` (Figure 11). 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. 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.

5. EXPERIMENTS

We evaluate the benefits of our joint points-to and precise exception analysis in three ways. First, we evaluate the precision and performance compared to a points-to analysis with imprecise handling of exceptions. Second, we explore approximations to determine the features contributing to the added precision. Third, we compare the precision of our exception analysis to related work on exception-flow analysis [10].

5.1 Precision

We compare the precision of our joint exception and points-to analysis to a pointer analysis that uses an imprecise exception analysis, accurately reflecting the handling of exceptions in SPARK and PADDLE. (Note that the DOOP points-to analyses are logically equivalent to the PADDLE analyses and produce identical results [1].) 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.

We evaluate four analyses: context-insensitive (insens), 1-call-site-sensitive (1 call), 1-call-site-sensitive with a context-sensitive heap abstraction (1H call), and 1-object-sensitive (1 obj). We analyze the DaCapo benchmark programs, v.2006-10-MR2, with JRE

```
  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 10: 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)
```

Figure 11: Exception-catch links for Fig. 10 using context-insensitive and 1-object-sensitive analyses, focusing on native methods. The striked out results are eliminated by the 1-object-sensitive analysis.

1.4 (j2re1.4.2_18).[2] We use a 64-bit machine with two quad-core Intel Xeon E5345 2.33GHz CPUs. However, the LogicBlox Datalog engine uses only one core. The machine has 16GB of RAM.

Figure 12 presents the results of this experiment. The table shows statistics on the main products of a pointer analysis: the context-(in)sensitive call-graph and context-(in)sensitive points-to information. The statistics of the imprecise exception analyses are relative to the corresponding statistics of the precise exception analyses (i.e. four rows up). The imprecise exception analysis does not compute the exceptions potentially thrown by each method, therefore the corresponding cells of Figure 12 are empty (-).

For context-sensitive pointer analysis, we present two sets of statistics: one corresponds to end-user visible results and the other to the primary internal complexity metrics. Namely, the first group (*'after dropping contexts'*) of *context-insensitive* statistics drops all contexts after a context-sensitive analysis. For example, for the 1 obj analysis, the relation `VarPointsTo(h,ctx,v)` is projected to `VarPointsTo(h,v)` showing which objects a program variable can point to. The second group of results (*'before dropping contexts'*) is *context-sensitive*, although it does drop the context of the heap abstraction for the 1H call analysis, so that it is comparable with the rest. For example, for 1H call the relation `VarPointsTo(hCtx,h,vCtx,v)` is projected to `VarPointsTo(h,vCtx,v)`. These context-sensitive results are relevant for client analyses that consider contexts. Also, the context-sensitive statistics are an indication of the size of the relations that the analysis operates on. For the context-sensitive results and a context-insensitive analysis, we repeat the context-insensitive results to ease comparison.

[2]We do not benchmark fop, because for whole program static analysis the benchmark has missing dependencies.

legend

precise	=	pointer analysis with precise exception analysis	var points-to = *total* and *mean (per variable)* entries in var points-to relation
imprecise	=	pointer analysis with imprecise exception analysis	throw points-to = *total* and *mean (per method)* entries in throw points-to relation
nodes, edges	=	call-graph nodes, call-graph edges	

prog	analysis		after dropping contexts (context-insensitive)						before dropping contexts (context-sensitive)						time (sec)
			nodes	edges	var points-to		throw points-to		nodes	edges	var points-to		throw points-to		
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
		1H call	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
		1H call	×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
bloat	precise	insens	6K	47K	8.4M	137	1.9M	302	6K	47K	8.4M	137	1.9M	302	74
		1 call	6K	47K	3.7M	60	1.2M	189	47K	274K	28M	86	8.7M	186	272
		1H call	6K	46K	3.7M	60	1.2M	189	46K	259K	27M	84	8.6M	185	4909
		1 obj	6K	46K	2.7M	45	1.0M	161	81K	2.9M	16M	32	7.3M	91	868
	imprecise	insens	×1.0	×1.0	×1.0	+5	-	-	×1.0	×1.0	×1.0	+5	-	-	48
		1 call	×1.0	×1.0	×1.1	+6	-	-	×1.0	×1.0	×1.1	+4	-	-	181
		1H call	×1.0	×1.0	×1.1	+6	-	-	×1.0	×1.0	×1.1	+4	-	-	3529
		1 obj	×1.0	×1.0	×1.2	+7	-	-	×1.0	×1.9	×1.4	+12	-	-	1563
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
		1H call	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
		1H call	×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
		1H call	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
		1H call	×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
hsqldb	precise	insens	4K	18K	1.9M	55	855K	229	4K	18K	1.9M	55	855K	229	39
		1 call	4K	18K	483K	14	441K	118	18K	71K	2.5M	21	1.6M	90	56
		1H call	4K	18K	477K	14	440K	118	18K	66K	2.4M	20	1.6M	89	159
		1 obj	4K	18K	401K	12	356K	96	38K	891K	2.6M	10	1.5M	39	170
	imprecise	insens	×1.0	×1.0	×1.2	+8	-	-	×1.0	×1.0	×1.2	+8	-	-	19
		1 call	×1.0	×1.0	×1.7	+10	-	-	×1.0	×1.0	×1.4	+8	-	-	36
		1H call	×1.0	×1.0	×1.7	+10	-	-	×1.0	×1.0	×1.4	+8	-	-	144
		1 obj	×1.0	×1.0	×2.0	+11	-	-	×1.1	×3.2	×3.0	+18	-	-	828
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
		1H call	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
		1H call	×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
luindex	precise	insens	4K	19K	1.9M	54	939K	232	4K	19K	1.9M	54	939K	232	31
		1 call	4K	19K	512K	14	494K	122	19K	74K	2.7M	21	1.7M	90	48
		1H call	4K	19K	506K	14	493K	122	19K	69K	2.5M	20	1.7M	89	159
		1 obj	4K	19K	434K	12	382K	95	39K	905K	2.8M	10	1.6M	41	149
	imprecise	insens	×1.0	×1.0	×1.2	+9	-	-	×1.0	×1.0	×1.2	+9	-	-	20
		1 call	×1.0	×1.0	×1.8	+12	-	-	×1.0	×1.0	×1.4	+8	-	-	36
		1H call	×1.0	×1.0	×1.8	+12	-	-	×1.0	×1.0	×1.4	+8	-	-	146
		1 obj	×1.0	×1.0	×2.0	+13	-	-	×1.1	×3.6	×3.2	+19	-	-	987
lusearch	precise	insens	5K	22K	2.2M	54	1.1M	231	5K	22K	2.2M	54	1.1M	231	34
		1 call	5K	22K	609K	15	547K	118	22K	85K	3.2M	22	2.0M	88	53
		1H call	5K	22K	603K	15	545K	118	22K	80K	3.0M	21	1.9M	87	186
		1 obj	5K	22K	500K	13	397K	86	43K	927K	3.0M	11	1.6M	37	153
	imprecise	insens	×1.0	×1.0	×1.2	+9	-	-	×1.0	×1.0	×1.2	+9	-	-	22
		1 call	×1.0	×1.0	×1.7	+11	-	-	×1.0	×1.0	×1.3	+7	-	-	39
		1H call	×1.0	×1.0	×1.7	+11	-	-	×1.0	×1.0	×1.3	+7	-	-	171
		1 obj	×1.0	×1.0	×1.9	+12	-	-	×1.1	×3.8	×3.2	+21	-	-	1091
pmd	precise	insens	5K	26K	3.3M	72	1.5M	271	5K	26K	3.3M	72	1.5M	271	52
		1 call	5K	26K	1.0M	22	692K	127	26K	106K	5.5M	30	2.4M	91	79
		1H call	5K	26K	1.0M	21	691K	127	26K	100K	5.2M	29	2.3M	90	263
		1 obj	5K	26K	926K	20	581K	107	51K	1.1M	5.0M	15	2.2M	43	231
	imprecise	insens	×1.0	×1.0	×1.2	+14	-	-	×1.0	×1.0	×1.2	+14	-	-	29
		1 call	×1.0	×1.0	×1.8	+18	-	-	×1.0	×1.0	×1.4	+10	-	-	59
		1H call	×1.0	×1.0	×1.8	+18	-	-	×1.0	×1.0	×1.4	+11	-	-	251
		1 obj	×1.0	×1.0	×1.9	+18	-	-	×1.1	×3.8	×2.7	+23	-	-	1390
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
		1H call	4K	17K	459K	14	458K	123	18K	65K	2.3M	20	1.6M	92	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
		1H call	×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

Figure 12: Precise versus imprecise exception analysis. Results for imprecise analysis are relative to corresponding precise analysis.

Var Points-To Precision. The primary benefit of our analysis is evident in the precision of the points-to results. This effect is more pronounced for more precise (context-sensitive) analyses. For these analyses, 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. In particular, for a 1 obj analysis (which is the most precise of the analyses we tried in terms of points-to sets sizes) the increase in precision due to improved exception handling is 1.9× or better (i.e., a program variable is on average predicted to point to roughly half as many objects) for 7 of our 10 benchmarks!

The *context-sensitive* var points-to relation is comparatively even smaller when using our precise exception analysis. For the 1 obj analysis, 6 out of our 10 benchmarks exhibit 3× or better increase in precision. That is, more than two-out-of-three inferred facts relating a variable, under a context, to a heap object were due entirely to the imprecision of exception handling! This provides evidence that the context abstraction is used much more effectively to increase precision under our joint analysis.

Call-Graph Precision.[3] 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 [19].

The *context-sensitive* call-graph is also not significantly affected, except in the case of the 1 obj analysis. This is quite expected, since this analysis uses objects as contexts. Therefore, the size of the context-sensitive call-graph is highly related to the size of points-to sets: 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. The effect of imprecise exception handling is minor in terms of nodes (about a 10% increase) but major in terms of edges (a 1.9× to 6.1× increase).

Throw Points-To Precision. Recall that the throw points-to relation expresses what exception objects can be thrown by each method (under a context, in the context-sensitive throw points-to case). Therefore it is a major metric of the precision of our *exception* analysis, and the question is how it is affected when varying the precision of the *points-to* analysis.

When moving from an insensitive analysis to a context-sensitive analysis, 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. However, the increased precision for throw points-to is less pronounced than the increased precision for the *context-insensitive* var points-to relation. This is easy to understand: for a context-insensitive pointer analysis all invocations of a method return the same points-to set, i.e., parameters of an invocation "leak" to other invocations. Thrown exceptions are less directly dependent on parameters.

Similar to var points-to, the 1 obj analysis is the most precise analysis for throw points-to. For this analysis, the mean number of exceptions thrown per *context-sensitive* method is always significantly lower than the mean per *context-insensitive* method. Also, the mean of the 1 obj analysis is always substantially lower than for the 1 call analyses. This illustrates that object-sensitivity is a useful context abstraction for calculating throw points-to.

The 1H call analysis with precise exception analysis minimally improves the precision of throw points-to compared to 1 call. This corresponds to conclusions in related work that the 1H call analysis is not effective for object-oriented programs.

[3]Except for the antlr benchmark, we have not yet configured dynamically loaded classes. This means that some parts of the benchmarks are not reachable.

				var points-to		throw points-to	
cs	o	f	e	sens	insens	sens	insens
×	×	×	1.0M	3.4M 9	598K 10	1.9M 31	579K 115
×		×	×1.5	×1.2 +2	×1.0 +0	×1.1 +3	×1.1 +11
×	×		×2.6	×1.8 +6	×1.2 +2	×1.9 +23	×1.9 +103
×			×2.6	×1.9 +7	×1.3 +3	×1.9 +23	×1.9 +103
	×	×	×1.1	×1.2 +2	×1.1 +1	- -	×1.9 +108
		×	×1.6	×1.5 +4	×1.2 +2	- -	×2.1 +126
	×		×2.7	×2.5 +11	×1.4 +4	- -	×3.4 +276
			×2.7	×2.5 +12	×1.5 +5	- -	×3.4 +276
imprecise			×6.1	×4.6 +28	×2.0 +11	- -	- -

cs = context-sensitive throw points-to
o = order of exception handlers is considered
f = caught exceptions are filtered (not thrown)
e = context-sensitive call-graph edges
sens = context-sensitive, total and mean
insens = context-insensitive, total and mean

Figure 13: Precision of approximations for exception analysis with 1-object-sensitive pointer analysis and DaCapo's antlr.

Analysis Time. Besides the precision improvements, the most striking result is the performance improvement of using precise exception analysis with the 1 obj pointer analysis: 14× for antlr, 12× for chart, between 5× and 10× for most of the benchmark programs, and, as a minimum, 1.8× for bloat.

The analysis time comparison is a trade-off in most cases. Compared to an imprecise exception analysis, a precise exception analysis computes more information: the throw points-to relation. This relation is big, usually not much smaller than 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. For 1 obj the benefit of precise information about thrown exceptions clearly compensates for the cost of computing it. The overhead of an imprecise exception analysis is basically unacceptable in this case.

For the insens, 1 call, and 1H call analyses, precise exception analysis is usually a bit slower than imprecise exception handling. Therefore, the "only" benefit for these analyses is increased precision. However, in recent years several researchers have pointed out that object sensitivity is the most useful context abstraction for object-oriented programs, which makes our precision and performance improvements for the 1 obj analysis highly relevant.

5.2 Approximations for Exception Analysis

Having shown the major precision improvement of fully precise analysis of exceptions, the question is what this improvement should be attributed to. To explore this, we focus on the 1-object-sensitive analysis and antlr. We evaluate the precision and performance by gradually removing features from our "fully precise" analysis, introducing the following approximations: *(1)* context-insensitive method throws abstraction: we consider a `ThrowPointsTo` relation with no context; *(2)* unordered handlers: we ignore the order of exception handlers; *(3)* no filtering of caught exceptions: a method throws all exceptions thrown in its dynamic scope, even if an exception is caught. This implies that the index of an instruction does not need to be compared to the range of an exception handler.

Figure 13 presents the results. The configuration of features (indicated by ×) of the first row is equal to our precise exception analysis. The statistics of all other configurations are relative to the results for this analysis. As can be seen, every approximation introduces a major increase of call graph edges, var points-to or throw points-to.

The most interesting approximations are: ignoring the order of exception handlers (second row) and making throw points-to context-insensitive (first row in second part). The latter is tempting for reducing memory consumption, however the sum of context-sensitive throw points-to and var points-to is not lower for this approximation.

5.3 Exception-Flow Analysis

While the precise exception analysis has a major impact on the size of the call-graph, var points-to and throw points-to, the impact of the increased precision is especially clear with a client analysis that focuses on exceptions. We next evaluate such a client analysis, namely calculating exception-catch links (*e-c links*).

An e-c link is a tuple of an instruction that may throw an exception and a handler that might handle this exception. Fu et al. use e-c links to determine the test suite coverage of exception handling code [8–10]. Their method first uses a sophisticated custom static analysis to determine a set of possible e-c links. Next, it determines the number of covered e-c links by injecting faults at throw-sites and monitoring which e-c links are covered by a test suite. The percentage of possible e-c links that have been covered is then used as a coverage metric. The precision of the possible e-c link analysis is crucial for this coverage metric to be meaningful and practical. The Fu et al. exception analysis works as a filter on e-c links produced by a context-insensitive pointer analysis that uses the imprecise exception handling we evaluated in Section 5.1. We compare this approach to our joint points-to and exception analysis.

Setup. We contacted Chen Fu to learn more about the exact setup of the experiment reported in past literature [10]. This proved to be difficult to reproduce exactly for several reasons: *(1)* the analysis implementation has evolved to become a specialized analysis [9] and cannot be used in its current form to perform the original, general exception-flow analysis; *(2)* the number of I/O related e-c links highly depends on the specification of dynamically loaded classes, but the list used by Fu et al. was not available; *(3)* the resulting e-c links could not be compared because the results of the original experiments (beyond numerical aggregates) are not available. As a result, our e-c links are not necessarily a sub- or superset of the e-c links reported by Fu et al.—we have spent considerable effort to do a faithful comparison, but emphasize that these issues need to be considered when comparing the results. In conceptual terms (e.g., based on published examples of cases handled) we see no reason why the Fu and Ryder analysis would be more precise than ours, based on the algorithm description.

For this experiment we use JRE 1.3 (jre1.3.1_20) to make a comparison possible. We selected the smallest and the largest benchmark used by Fu et al.: ftpd-0.6 and muffin-0.9.3a. Our e-c link client analysis is a query on the results of the joint points-to and exception analysis. If a native method throws an exception e and the parameter of an exception handler points to e, then we conclude that this is an e-c link. This is slightly more general than the analysis of Fu et al. [8, 10], since our approach also handles rethrown exceptions. This can result in more e-c links than discovered by Fu et al. Note that the test coverage application of e-c links does not require soundness (i.e., missing e-c links is acceptable). Our results are sound, under the assumption that we model native code correctly and dynamic class loading is configured correctly.

Benchmarks. *FTPD* handles FTP requests using reflection based on strings in the input. Though Doop performs reflection analysis, no static analysis can handle this without configuration input. We modified the code to call methods directly, and confirmed that Fu et al. did this as well. For FTPD, our analyses report 101 reachable methods. This number differs from the 128 reported methods in [10]. We have verified by hand that the extra 27 methods are not reachable. *Muffin* involves substantial amounts of native code because its user-interface is based on AWT. We have configured the analysis to make all AWT event handling methods reachable. Also, there are a few places where reflection is used. In most cases, classes are loaded based on a simple name and string concatenation (e.g., `"CookieMonster"` loads the class `org.doit.muffin.filter.CookieMonsterFilter`).

			all	I/O	I/O sel.	time (sec)
ftpd	prec	insens	243	109	47	15
		1 call	176	68	17	18
		1 obj	164	66	▸15	15
	imprec	insens	459	325	104	12
		1 call	446	312	91	18
		1 obj	446	312	91	23
muffin	prec	insens	1109	466	237	31
		1 call	743	294	134	44
		1 obj	611	134	▸49	94
	imprec	insens	1811	1555	490	22
		1 call	1811	1555	490	40
		1 obj	1741	1485	420	86

Figure 14: Number of exception-catch links between native methods throwing exceptions and handlers in application code (all: all native methods; I/O: all native methods throwing IOException or subclasses; I/O sel.: the selection of the native I/O methods used by Fu et al.)

Results. Figure 14 presents the results of our experiment. To give more insight into the number of e-c links, we do not only report on the 32 native methods selected by Fu et al., but also include statistics for all native methods and all I/O related native methods.

Clearly, an imprecise exception analysis is largely useless for reducing the number of e-c links, regardless of the pointer analysis used. The precise exception analysis reduces the number of e-c links by a factor of 6 to 10 when using a context-sensitive analysis.

The most precise analysis of Fu et al. reports 13 possible e-c links for FTPD, and 42 for Muffin. Our 1 obj analysis reports a few more e-c links (15 and 49), but, as explained earlier, it is unclear if this is actually due to imprecision. Our analysis could actually be more precise, but trace more exception-flow. A strong indication that this is the case is provided by the reported number of e-c links for our insens analysis. Fu et al. report on the number of e-c links found using the Spark context-insensitive pointer analysis without a further exception analysis: 16 for FTPD and 112 for Muffin. Our context-insensitive analysis is comparable in precision to Spark, yet discovers 47 and 237 links. Therefore we suspect that our sound analysis explores a much larger space of exception flow and still reduces the e-c links to numbers comparable to those of Fu et al.

The analysis of Fu et al. is reported [8] to take 298 seconds for FTPD and 4,043 seconds for Muffin on a 2.8GHz P4 machine. The vast majority of the time is spent in the custom exception analysis, and not in the Spark points-to analysis. Our analysis readily outperforms the reported Fu et al. numbers, and the speedup is much higher than even a generous allowance for the difference in machine speeds.[4] Our 1 obj analysis is 20× faster for FTPD and 43× faster for Muffin.

We conclude that our *general* joint points-to and exception analysis achieves precision comparable to the *custom* exception-flow

[4]Standard benchmarks (e.g., `http://www.cpubenchmark.net`) put the Fu et al. CPU at 15% of the speed of our 2.33GHz quad-core Xeon for peak performance with all four cores of the Xeon utilized. In single-threaded performance and realistic applications, the two CPUs are probably much closer.

analysis. Moreover, the dramatic performance improvement leaves a lot of room to experiment with improving the precision by introducing more context.

6. DISCUSSION

We next discuss various factors regarding the precision or applicability of our results.

Analysis Architecture. In Doop, imprecision directly affects the speed of the analysis. Thus, a large points-to relation proportionally slows down our analysis. This is not the case in points-to analyses that employ BDDs, such as Paddle [16]. The number of distinct points-to sets may be a better indicator of the analysis cost for such analyses. Thus, the conservative exception handling of Paddle affects the precision of the analysis results, but not the running time—Paddle with conservative but imprecise exception analysis is barely slower than its mode that unsoundly ignores exceptions. (Nevertheless, in our other study [1] we show Doop to outperform Paddle by 4× to 12× on the analyses of Section 5.1, so Paddle's imprecise analysis certainly does not offer a performance sweet spot compared to Doop.)

Implicitly Thrown Exceptions. Our numbers do not include a handling of implicitly thrown exceptions. These include violations of language preconditions (e.g., `NullPointerException`) but also JVM errors, which can be thrown anywhere and are rarely caught. Whether implicit exceptions are relevant depends on the client analysis. If a sound pointer analysis is crucial, for example for program optimization, then implicitly thrown exceptions should be reported. It is likely, however, that such exceptions will require special treatment. This is just a matter of engineering and not a conceptual analysis problem because the precision of the analysis of implicitly thrown exceptions is less important: it is hardly useful to have a heap abstraction that identifies the allocation-site, or even the context of allocation for `OutOfMemoryErrors`, `NullPointerExceptions`, etc. Therefore, the impact on the amount of points-to data will be limited. For instance, the type of the implicit exception can be used as the heap abstraction. This limits the number of possible objects drastically, but ensures that the points-to information is complete.

Thread.stop. The deprecated `stop` method of a Java thread stops a thread by throwing an object in the thread that is stopped. By default, this is a `ThreadDeath` error, but any `Throwable` object can be passed to the `stop` method.

There are multiple problems with the `Thread.stop` method (hence the deprecation). Via `Thread.stop`, any instruction in the program can throw any `Throwable` object. The specified, checked exceptions of method declarations are not even relevant. (These cannot be trusted in bytecode anyway, as the bytecode may have been produced by an untrusted compiler, and the JVM verifier does not confirm the correctness of exception declarations in a method's signature.) Possible solutions range from the conservative ("detect use of `Thread.stop` and refuse to analyze the program") to the sophisticated ("perform a thread-sensitive analysis [20] of which methods can execute in which thread and what objects can be passed to the `stop` method"). Neither Doop nor other points-to frameworks (e.g., Spark and Paddle) handle `Thread.stop` but such support can be added.

7. RELATED WORK

Precise exception analysis has not been integrated in a pointer analysis before to the best of our knowledge. This is surprising, since there is a clear mutual dependency and improving the precision of one analysis directly affects the other. We claim that exception analysis should not be considered a client of pointer analysis.

Exception Analysis. The main differences with previous exception analysis work are the increased precision and generality of our joint analysis. All of the earlier work on exception analysis propagates exceptions over a *context-insensitive* call graph. Often, this call graph is also constructed using an imprecise method, such as class hierarchy analysis. Propagating exceptions over a context-insensitive call graph does not, for example, distinguish the different contexts of file I/O and socket I/O in Figure 10. We have shown that propagating exceptions over a context-sensitive call graph substantially increases the precision of an exception analysis.

Most earlier work on exception analysis represents a potentially thrown exception by its type and does not consider points-to information. In this way, it is possible to see what exceptions a method throws, but not where they come from. Type-based exception analyses also have to be very conservative when thrown exceptions are not directly allocated by a throw statement, but, for example, are obtained via invoking a method, or retrieved from the `cause` field (a member of every `Throwable`) of a wrapper exception. Thanks to our joint analysis, we support arbitrary code leading to thrown exceptions.

Robillard et al. [22] present the tool Jex for interprocedural exception-flow analysis. Jex propagates exceptions, represented by types, over a call graph constructed using class hierarchy analysis. The results are imprecise and incomplete because *(1)* Jex does not handle `throw` statements that do not directly allocate an exception and *(2)* class hierarchy analysis produces an imprecise call graph.

Buse et al. [2] present a sophisticated tool for documenting under which conditions an exception can be thrown. This tool is an extension of Jex. It also uses a more precise call graph, computed by Spark. After computing the exceptions that can be thrown by a method, the program paths leading to throw-sites are symbolically executed to construct a predicate that describes the condition that leads to an exception. This aspect of the tool is orthogonal to our pointer analysis. The effectiveness of the tool could be improved substantially by using results of our exception analysis as input.

Jo et al. [13] present an interprocedural exception analysis that is similar to Jex, reporting the types of potentially thrown exceptions. The context-insensitive analysis depends on a class analysis, which gives the possible types of an expression. While the precision of the tool depends on the precision of the class analysis, the analysis is overall imprecise for reasons similar to Jex. *Ryu et al.* [23] extend this analysis with a concurrency analysis to support throwing exceptions between threads, specifically addressing the problem of `Thread.stop`. A points-to based solution would again result in more precise information, also for exceptions flowing to `Thread.stop`.

Leroy et al. [15] present an analysis of uncaught exceptions formalized as a type system for exceptions, accompanied by a type inference algorithm. Their system supports a limited form of context-sensitivity to obtain more precise results for polymorphic functions. Leroy et al. report that they abandoned a more precise control-flow analysis approach out of performance and scalability concerns. For integration in a compiler, this is probably still a valid concern.

Yi et al. [27] present a static analysis for finding uncaught exceptions in SML. Similarly to the work of Leroy et al., this analysis was intended for inclusion in a compiler and, therefore, sacrifices precision for performance. The mutual dependency between control-flow and exception-flow is addressed by approximation.

We explicitly compared to the exception-flow work of *Fu et al.* [8, 10] in Section 5.3 and used their e-c link metric. In later work, Fu et al. studied the problem of discovering chains of thrown exceptions [9]. Our analysis automatically supports exceptions that

get rethrown as-is. The points-to analysis also takes care of exceptions that are wrapped in a different exception and then rethrown. The analysis for other flow of data from a caught exception to a new thrown exception is orthogonal to exception-flow and pointer analysis. It is a real client analysis, without mutually recursive relationships to any points-to data.

Pointer Analysis. Exceptions have not received much attention in points-to analysis research. We show in this paper that imprecise exception analysis has a major impact, not only on performance, but also on statistics that are reported to indicate the precision benefits of context abstractions. For example, from our evaluation it is clear that imprecise exception analysis hurts an object-sensitive analysis more than it hurts a call-site-sensitive analysis.

SPARK [17] and PADDLE [16, 19] both use the imprecise exception analysis we discussed in Section 5.1. Soot also has a separate exception analysis [14] that is not based on a pointer analysis.

IBM Research's WALA [7] static analysis library is designed to support different pointer analysis configurations. The points-to analyses of WALA support computing which exceptions a method can throw, but no results of WALA's accuracy or speed have been reported in the literature. It will be interesting to compare our analyses to WALA in future work.

bddbddb is a Datalog and BDD-based database that has been employed for points-to analysis [25, 26]. The publications do not discuss exception analysis, yet the bddbddb distribution examples do propagate exceptions over the control-flow graph. One of the differences between DOOP and bddbddb is that DOOP expresses the entire analysis in Datalog and only relies on basic input facts. In contrast, the points-to analyses of bddbddb largely rely on pre-computed input facts, such as a call graph, reducing the Datalog analysis to just a few lines of code for propagating points-to data. For exception analysis, the analyses of bddbddb rely on input facts that define for every method invocation to which variable the thrown exceptions should be assigned. This ignores the order of exception handlers and also disables filtering of caught exceptions.

Sinha et al. discuss how to represent exception flow in the control-flow graph [24]. One of the topics is handling `finally` clauses, but we analyze Java bytecode and the complex control-flow of `finally` clauses is already handled by the Java compiler. Also, our analysis is flow-insensitive.

Choi et al. suggested a compact intraprocedural control-flow representation that collapses the large number of edges to exceptions handlers [4]. Our analyses are interprocedural and flow-insensitive, so not directly comparable to this work. However, we could perhaps use ideas from this representation to reduce the size of the predicate to look up an exception handler given an instruction and an exception type.

Acknowledgments. This work was funded by the NSF (CCR-0735267, CCF-0934631) and LogicBlox Inc. We thank Chen Fu and Ondřej Lhoták for their help with our experiments.

8. REFERENCES

[1] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *submission to OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, 2009.

[2] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008.

[3] 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.

[4] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. *SIGSOFT Softw. Eng. Notes*, 24(5):21–31, 1999.

[5] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems - a case study. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, 1996.

[6] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, 2008.

[7] S. J. Fink. T.J. Watson libraries for analysis (WALA). `http://wala.sourceforge.net`.

[8] 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.

[9] C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007.

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

[11] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with Datalog. In *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, 2006.

[12] N. Immerman. *Descriptive Complexity*. Springer, 1998.

[13] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe. An uncaught exception analysis for Java. *J. Syst. Softw.*, 72(1):59–69, 2004.

[14] J. Jorgensen. Improving the precision and correctness of exception analysis in Soot. Technical Report 2003-3, McGill University, 2004.

[15] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000.

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

[17] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Compiler Construction, 12th International Conference*, 2003.

[18] O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, 2004.

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

[20] J. Qian and B. Xu. Thread-sensitive pointer analysis for inter-thread dataflow detection. In *FTDCS '07: Proc. of the 11th IEEE Int. Workshop on Future Trends of Distributed Computing Systems*, 2007.

[21] T. Reps. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, 1994.

[22] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.

[23] S. Ryu and K. Yi. Exception analysis for multithreaded Java programs. In *APAQS '01: Proceedings of the Second Asia-Pacific Conference on Quality Software*, 2001.

[24] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.*, 26(9):849–871, 2000.

[25] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Asian Symposium on Programming Languages and Systems*, 2005.

[26] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, 2004.

[27] K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, 1997.