Efficient and Effective Handling of Exceptions in Java Points-To Analysis

George Kastrinis and Yannis Smaragdakis

Dept. of Informatics, University of Athens, Greece
{gkastrinis,smaragd}@di.uoa.gr

Abstract. A joint points-to and exception analysis has been shown to yield benefits in both precision and performance. Treating exceptions as regular objects, however, incurs significant and rather unexpected overhead. We show that in a typical joint analysis most of the objects computed to flow in and out of a method are due to exceptional control-flow and not normal call-return control-flow. For instance, a context-insensitive analysis of the Antlr benchmark from the DaCapo suite computes 4-5 times more objects going in or out of a method due to exceptional control-flow than due to normal control-flow. As a consequence, the analysis spends a large amount of its time considering exceptions.
We show that the problem can be addressed both effectively and elegantly by coarsening the representation of exception objects. An interesting find is that, instead of recording each distinct exception object, we can collapse all exceptions of the same type, and use one representative object per type, to yield nearly identical precision (loss of less than 0.1%) but with a boost in performance of at least 50% for most analyses and benchmarks and large space savings (usually 40% or more).

1 Introduction

Points-to analysis is a fundamental static program analysis. It consists of computing a static abstraction of all the data that a pointer variable (and, by extension, any pointer expression) can point to during program execution. Points-to analysis is often the basis of most other higher-level client analyses (e.g., may-happen-in-parallel, static cast elimination, escape analysis, and more). It is also inter-related with call-graph construction, since the values of a pointer determine the target of dynamically resolved calls, such as object-oriented dynamically dispatched method calls or functional lambda applications.

An important question regarding points-to analysis (as well as client analyses based on it) concerns the handling of exceptions, in languages that support exception-based control flow. The emphasis of our work is on Java—a common target of points-to analysis work—but similar ideas are likely to apply to other languages, such as C# and C++. This is an important topic because exceptional control flow cannot be ignored for several client analysis (e.g., information leak or other security analyses) and if handled crudely it can destroy the precision of the base points-to analysis.

In the past, most practical points-to analysis algorithms have relied on conservative approximations of exception handling [19, 20]. The well-known points-to analysis libraries Spark [19] and Paddle [18] 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 sound but highly imprecise because it ignores the information about what exceptions can propagate to a catch site. For clients that care about exception objects specifically (e.g., computing which `throw` statement can reach which `catch` clause), precise exception handling has been added on top of a base points-to analysis [7–9]. Fu and Ryder's "exception-chain analysis" [8] is representative. It works on top of SPARK, with its conservative modeling of exceptions, but then performs a very precise analysis of the flow of exception objects. However, this approach has a high computational overhead. Furthermore, the approach does not recover the precision lost for the base points-to results for objects that do not represent exceptions.

Based on the above, the DOOP framework [2] (which is also the context of our work) has introduced a joint points-to and exception analysis [1]. DOOP expresses exception-analysis logic in modular rules, mutually recursive with the points-to analysis logic: Exception handling can cause variables to point to objects (of an exception type), can make code reachable, etc. Points-to results are, in turn, used to compute what objects are thrown at a `throw` statement. The exception analysis logic on its own is "as precise as can be" as it fully models the Java semantics for exceptions. Approximation is only introduced due to the static abstractions used for contexts and objects in the points-to analysis. Thus, 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. The result is an analysis that achieves very high precision and performance for points-to results, while also matching the precision of techniques specifically for exception-related queries, as in the Fu and Ryder exception-chain analysis.

The motivation for our work is that, despite the benefits of the DOOP approach, there is significant room for improvement. The joint points-to and exception analysis performs heavy work for exception objects alone. An indicative metric is the following: Consider the number of objects pointed to by method parameters or its return value vs. the objects thrown and not caught by the current method or by methods called by it. The former number represents the objects that flow into or out of each method due to normal control-flow, while the latter shows the objects that flow out of the method due to exceptions. Our experiments show that the latter number is often several times larger than the former. (We present full results later.) This is counterintuitive and suggests that the analysis performs unexpectedly much work on exceptions alone.

To address this issue we observe that most client analyses do not care about exception objects specifically. They do, however, care about the impact of exceptions to the rest of the points-to and call-graph facts. For instance, the effectiveness of a client analysis such as static cast elimination is not impacted in practice by the few optimization opportunities that lie inside exception handlers or that involve objects of an exception type. But the analysis *is* impacted by code possibly executed only because of exception handling, or variables that point to extra objects as a result of an exception handler's execution. In other words, we would like *precise handling of exceptions only to the extent that they impact the precision of the base points-to analysis, even if the information over exception objects themselves is less precise*. (Note that this is very different from the SPARK or PADDLE handling of all exceptions through a single global variable: That ap-

proach does adversely impact the precision and performance of the base analysis—e.g., it more than doubles the number of edges of a context-sensitive call-graph [1, Fig.12].)

Therefore, our approach consists of coarsening the representation of exception objects in two ways. First, we treat exception objects context-insensitively, even for an otherwise context-sensitive analysis.[1] Second, we merge exception objects and represent them as a single object per-dynamic-type. The per-type treatment is important for maintaining precision, since the main purpose of an exception object is to trigger appropriate exception handling code (i.e., a `catch` clause keyed on the type of the object caught).

We find that this approach is both easy to specify and implement, as well as highly effective. For instance, for a 1-object-sensitive analysis we obtain a 60% average speedup for the "antlr" benchmark and a 225% average speedup for the "eclipse" benchmark of the DaCapo suite (with similar speedups for other benchmarks) just by employing the "merge exception objects per-type" idea. This speedup is accompanied by significant space savings in the analysis. Crucially, the performance increase does not entail any loss of precision for results unrelated to exception objects. All precision metrics of the analysis remain virtually identical. Namely, the numbers of call-graph nodes and edges, methods that can be successfully devirtualized, and casts that can be statically eliminated remain the same up to at least three significant digits.

In summary, the contributions of our work are as follows:

- We give a concise and general model of flow-insensitive, context- and field-sensitive points-to analyses and call-graph construction for a language with exceptions. Although a joint exception and points-to analysis has been formulated before [1], it was expressed by-example. In contrast, we give a small, closed set of rules and definitions of input domains. That is, we present all the relevant detail of the analysis in a closed form, assuming a simplified intermediate language as input.
- We present measurements demonstrating that the impact of exceptions on points-to analysis performance metrics is significant. A points-to analysis that tries to model exceptions precisely ends up spending much of its time and space computing results for exception-based control-flow.
- We define on top of our model two simple ways to coarsen the representation of exception objects without affecting any other aspect of the points-to or exception logic.
- We show that our approach is very effective in practice, yielding both significant speedup and space savings. Our technique is the default in the upcoming version of the Doop framework as it gains performance without adversely impacting precision.

In the following sections we define an abstraction of context-sensitive points-to analysis and enhance it with exception handling logic (Section 2), present our technique in this abstract model (Section 3), detail its performance in a series of experiments (Section 4), and discuss related work in more detail (Section 5).

[1] *Context-sensitivity* is a general approach that achieves tractable and usefully high precision in points-to analyis. It consists of qualifying local program variables, and possibly (heap) object abstractions, 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 contexts.

2 Background: Model of Points-To Analysis

We next present a model of context-sensitive, flow-insensitive points-to analysis algorithms, as well as the enhancement of the model for computing exception information in mutual recursion with the analysis. Interestingly, the logical formalism that we use in our model is quite close to the actual implementation of the analysis in the Doop framework, under simplifications and omissions that we describe.

2.1 Base Points-To Analysis

We model a wide range of flow-insensitive points-to analyses together with the associated call-graph computation as a set of customizable Datalog rules, i.e., monotonic logical inferences that repeatedly apply to infer more facts until fixpoint. Our rules do not use negation in a recursive cycle, or other non-monotonic logic constructs, resulting in a declarative specification: the order of evaluation of rules or examination of clauses cannot affect the final result. The same abstract model applies to a wealth of analyses. We use it to model a context-insensitive Andersen-style analysis, as well as several context-sensitive analyses, both call-site-sensitive [25, 26] and object-sensitive [23].

The input language is a simplified intermediate language with a) a "new" instruction for allocating an object; b) a "move" instruction for copying between local variables; c) "store" and "load" instructions for writing to the heap (i.e., to object fields); d) a "virtual method call" instruction that calls the method of the appropriate signature that is defined in the dynamic class of the receiver object. This language models well the Java bytecode representation, but also other high-level intermediate languages. (It does not, however, model languages such as C or C++ that can create pointers through an address-of operator. The techniques used in that space are fairly different—e.g., [12, 13].) The specification of our points-to analysis as well as the input language are in line with those in the work of others [10, 21], although we also integrate elements such as on-the-fly call-graph construction and field-sensitivity.

Specifying the analysis logically as Datalog rules has the advantage that the specification is close to the actual implementation. Datalog has been the basis of several implementations of program analyses, both low-level [2, 17, 24, 29, 30] and high-level [5, 11]. Indeed, the analysis we show is a faithful model of the implementation in the Doop framework [2]. Our specification of the analysis (Figures 1-2) is an abstraction of the actual implementation in the following ways:

- The implementation has many more rules. It covers the full complexity of the language, including rules for handling reflection, native methods, static calls and fields, string constants, implicit initialization, threads, and a lot more. The Doop implementation currently contains over 600 rules in the common core of all analyses, as opposed to the dozen-or-so rules we examine here. (Note, however, that these dozen rules are the most crucial for points-to analysis. They also correspond fairly closely to the algorithms specified in other formalizations of points-to analyses in the literature [22, 28].)
- The implementation also reflects considerations for efficient execution. The most important is that of defining indexes for the key relations of the evaluation. Furthermore, it designates some relations as functions, defines storage models for relations

(e.g., how many bits each variable uses), designates intermediate relations as "materialized views" or not, etc.

V is a set of variables	H is a set of heap abstractions
M is a set of methods	S is a set of method signatures (including name)
F is a set of fields	I is a set of instructions (e.g., invocation sites)
T is a set of class types	$\mathbb{N}$ is the set of natural numbers
HC is a set of heap contexts	C is a set of contexts

ALLOC (*var : V, heap : H, meth : M*)	FORMALARG (*meth : M, i : $\mathbb{N}$, arg : V*)
MOVE (*to : V, from : V*)	ACTUALARG (*invo : I, i : $\mathbb{N}$, arg : V*)
LOAD (*to : V, base : V, fld : F*)	FORMALRETURN (*meth : M, ret : V*)
STORE (*base : V, fld : F, from : V*)	ACTUALRETURN (*invo : I, var : V*)
VCALL (*base : V, sig : S, invo : I*)	THISVAR (*meth : M, this : V*)
HEAPTYPE (*heap : H, type : T*)	LOOKUP (*type : T, sig : S, meth : M*)
INMETHOD (*instr : I, meth : M*)	SUBTYPE (*type : T, superT : T*)
VARPOINTSTO (*var : V, ctx : C, heap : H, hctx : HC*)	
CALLGRAPH (*invo : I, callerCtx : C, meth : M, calleeCtx : C*)	
FLDPOINTSTO (*baseH : H, baseHCtx : HC, fld : F, heap : H, hctx : HC*)	
INTERPROCASSIGN (*to : V, toCtx : C, from : V, fromCtx : C*)	
REACHABLE (*meth : M, ctx : C*)	
RECORD (*heap : H, ctx : C*) = ***newHCtx : HC***	
MERGE (*heap : H, hctx : HC, invo : I, ctx : C*) = ***newCtx : C***	

Fig. 1. Our domain, input relations, output relations, and constructors of contexts.

Figure 1 shows the domain of our analysis (i.e., the different sets that comprise the space of our computation), its input relations, the intermediate and output relations, as well as two constructor functions, responsible for producing new objects that represent contexts. We explain some of these components below:

- The input relations are standard and correspond to the intermediate language for our analysis. For instance, the ALLOC relation represents every instruction that allocates a new heap object, *heap*, and assigns it to local variable *var* inside method *meth*. (Note that every local variable is defined in a unique method, hence the *meth* argument is also implied by *var* but included for conciseness of later rules.) There are input relations for each instruction type (MOVE, LOAD, STORE and VCALL) as well as input relations encoding the type system and symbol table information. For instance, LOOKUP matches a method signature to the actual method definition inside a type.
- The main output relations of our points-to analysis and call-graph computation are VARPOINTSTO and CALLGRAPH. The VARPOINTSTO relation links a variable (*var*) to a heap object (*heap*). (A heap object is identified by its allocation site.) Both the variable and the heap object are qualified by "context" elements in our analysis: a plain context for the variable and a heap context for the heap object. Similarly, the CALLGRAPH relation qualifies both its source (an invocation site) and its target (a method) with contexts. Other intermediate relations (FLDPOINTSTO, INTERPROCASSIGN, REACHABLE) correspond to standard concepts and are introduced for conciseness and readability.
- The base rules are not concerned with what kind of context-sensitivity is used. The same rules can be used for a context-insensitive analysis (by only ever creating a single context object), for a call-site-sensitive analysis, or for an object-sensitive analysis, for any context depth. These aspects are completely hidden behind constructor

functions RECORD and MERGE, following the usage and naming convention of earlier work [28]. RECORD takes all available information at the allocation site of an object and combines it to produce a new heap context, while MERGE takes all available information at the call site of a method and combines it to create a new context. (Hence, the name "MERGE" refers to merging contexts and is unrelated to the idea of merging exception objects per-type, which we discuss later in this paper.) These functions are sufficient for modeling a very large variety of context-sensitive analyses.[2] Note that the use of such constructors is not part of regular Datalog and can result in infinite structures (e.g., one can express unbounded call-site sensitivity) if care is not taken.

```
InterProcAssign (to, calleeCtx, from, callerCtx) ←
    CallGraph (invo, callerCtx, meth, calleeCtx),
    FormalArg (meth, i, to), ActualArg (invo, i, from).
InterProcAssign (to, callerCtx, from, calleeCtx) ←
    CallGraph (invo, callerCtx, meth, calleeCtx),
    FormalReturn (meth, from), ActualReturn (invo, to).

RECORD (heap, ctx) = hctx,
VarPointsTo (var, ctx, heap, hctx) ←
    Reachable (meth, ctx), Alloc (var, heap, meth).
VarPointsTo (to, ctx, heap, hctx) ←
    Move (to, from), VarPointsTo (from, ctx, heap, hctx).
VarPointsTo (to, toCtx, heap, hctx) ←
    InterProcAssign (to, toCtx, from, fromCtx),
    VarPointsTo (from, fromCtx, heap, hctx).
VarPointsTo (to, ctx, heap, hctx) ←
    Load (to, base, fld), VarPointsTo (base, ctx, baseH, baseHCtx),
    FldPointsTo (baseH, baseHCtx, fld, heap, hctx).
FldPointsTo (baseH, baseHCtx, fld, heap, hctx) ←
    Store (base, fld, from), VarPointsTo (base, ctx, baseH, baseHCtx),
    VarPointsTo (from, ctx, heap, hctx).

MERGE (heap, hctx, invo, callerCtx) = calleeCtx,
Reachable (toMeth, calleeCtx),
VarPointsTo (this, calleeCtx, heap, hctx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ←
    VCall (base, sig, invo),
    VarPointsTo (base, callerCtx, heap, hctx), HeapType (heap, heapT),
    Lookup (heapT, sig, toMeth), ThisVar (toMeth, this).
```

Fig. 2. Datalog rules for the points-to analysis and call-graph construction.

[2] Explaining the different kinds of context-sensitivity produced by varying RECORD and MERGE is beyond the scope of this paper but is fully covered in past literature [28]. To give a single example, however, a 1-call-site-sensitive analysis with a context-sensitive heap has $C = HC = I$ (i.e., both the context and the heap context are a single instruction), **RECORD** (*heap, ctx*) = ***ctx*** and **MERGE** (*heap, hctx, invo, callerCtx*) = ***invo***. That is, when an object is allocated, its (heap) context is that of the allocating method, and when a method is called, its context is its call-site.

Figure 2 shows the points-to analysis and call-graph computation. The rule syntax is simple: 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, the very last rule says that if the original program has an instruction making a virtual method call over local variable *base* (this is an input fact), and the computation so far has established that *base* can point to heap object *heap*, then the called method is looked up inside the type of *heap* and several further facts are inferred: that the looked up method is reachable, that it has an edge in the call-graph from the current invocation site, and that its *this* variable can point to *heap*. Additionally, the MERGE function is used to possibly create (or look up) the right context for the current invocation.

2.2 Adding Exceptions

We can now easily add exception handling to our input language and express a precise exception analysis via rules that are mutually recursive with the base points-to analysis rules. The algorithm is essentially that of [1] but stated more concisely: we hide exception handler lookup details by assuming a more sophisticated input relation CATCH.

THROW (*instr : I, e : V*)	CATCH (*heapT : T, instr : I, arg : V*)
THROWPOINTSTO (*meth : M, ctx : C, heap : H, hctx : HC*)	

Fig. 3. Datalog input and output relations for the exception analysis

```
ThrowPointsTo (meth, ctx, heap, hctx) ←
    Throw (instr, e), VarPointsTo (e, ctx, heap, hctx),
    HeapType (heap, heapT), ¬Catch (heapT, instr, _), InMethod (instr, meth).
ThrowPointsTo (meth, callerCtx, heap, hctx) ←
    CallGraph (invo, callerCtx, toMeth, calleeCtx),
    ThrowPointsTo (toMeth, calleeCtx, heap, hctx),
    HeapType (heap, heapT), ¬Catch (heapT, invo, _), InMethod (invo, meth).
VarPointsTo (arg, ctx, heap, hctx) ←
    Throw (instr, e), VarPointsTo (e, ctx, heap, hctx),
    HeapType (heap, heapT), Catch (heapT, instr, arg).
VarPointsTo (arg, callerCtx, heap, hctx) ←
    CallGraph (invo, callerCtx, toMeth, calleeCtx),
    ThrowPointsTo (toMeth, calleeCtx, heap, hctx),
    HeapType (heap, heapT), Catch (heapT, invo, arg).
```

Fig. 4. Datalog rules for the Exception analysis

Figure 3 presents the input and output relations for our analysis. The input relations enhance the language-under-analysis with `catch` and `throw` instructions, with Java-like semantics. The THROW (*i,e*) relation captures throwing at instruction *i* an expression object that is referenced by local variable *e*. The CATCH (*t,i,a*) relation connects an instruction *i* that throws an exception of dynamic type *t* with the local variable *a* that will be assigned the exception object at the appropriate catch-site. Although CATCH does not directly map to intermediate language instructions, one can compute it easily from such

low-level input. Furthermore, hiding the definition of CATCH allows modeling of exception handlers at different degrees of precision—e.g., a definition of CATCH may or may not consider exception handlers in-order.

Figure 4 shows the exception computation, in mutual recursion with the points-to analysis. Two syntactic constructs we have not seen before are "_", meaning "any value", and "¬", signifying negation. The relation we want to compute is THROWPOINTSTO, which captures what exception objects a method may throw at its callers. As can be seen, VARPOINTSTO is used in the definition of THROWPOINTSTO and vice versa.

3 Coarsening the Representation of Exceptions

Although a precise joint points-to and exception analysis algorithm offers significant benefits [1], we next show that there is large room for improvement. The analysis ends up spending much of its time and space computing exception flow. We propose ideas for coarsening the representation of exception objects to address this issue and yield more efficient analyses, without sacrificing precision.

3.1 Motivation

Consider the size of the THROWPOINTSTO relation for an analysis. This represents the total flow of objects out of methods due to exceptions. For a context-sensitive analysis, this number is a good metric of the work performed by the analysis internally for reasoning about exceptions. It is interesting to compare this number with a similar metric over the VARPOINTSTO relation, namely the subset of VARPOINTSTO facts that concern variables that are either method arguments or return values. This represents the total flow of objects in and out of methods due to normal call and return sequences.

Table 1 shows the results of comparing these two measures for several different analyses: insensitive, call-site sensitive, object-sensitive, and type-sensitive [28], with a context-sensitive heap. The results are over five of the benchmarks in the DaCapo benchmark suite, analyzed with Oracle JDK 1.6. (A full description of our experimental setting can be found in the next section.) Entries with a dash instead of a number did not terminate within the time allotted (90mins).

For the context-insensitive analysis (first results column), the ratio can be understood in intuitive terms: the antlr ratio of 0.22, for instance, means that, on average, 4.5 times more objects are possibly thrown out of a method than passed into it or returned through regular call and return sequences. This is a counterintuitive result. Human reasoning about how a method interacts with its callers is certainly not dominated by exception objects. Therefore, we see that the joint points-to and exception analysis perhaps pays a disproportionate amount of attention (and expends much effort) on exceptions.

3.2 Coarse Exceptions

To reduce the cost of reasoning about exception objects, we propose two simple approaches for coarsening the representation of exception objects. The first is to represent

		insens	1obj+H	2obj+H	1type+H	1call+H
antlr	objs passed	697	-	10,440	3,955	17,486
	objs thrown	3,123	-	164,392	20,783	44,118
	ratio	**.22**	-	**.06**	**.19**	**.40**
bloat	objs passed	829	-	-	5,681	46,952
	objs thrown	4,112	-	-	32,905	78,593
	ratio	**.20**	-	-	**.17**	**.60**
eclipse	objs passed	637	15,750	-	6,570	18,690
	objs thrown	4,064	138,361	-	37,634	42,140
	ratio	**.16**	**.11**	-	**.17**	**.44**
luindex	objs passed	383	8,473	-	3,328	8,413
	objs thrown	2,544	60,897	-	18,928	25,297
	ratio	**.15**	**.14**	-	**.17**	**.33**
xalan	objs passed	668	-	-	7,480	18,895
	objs thrown	3,876	-	-	41,351	43,376
	ratio	**.17**	-	-	**.18**	**.44**

Table 1. Objects on method boundaries compared to exception objects thrown by a method (measured in thousands)

exception objects context-insensitively. This is a rather straightforward idea—even in context-sensitive analyses, several different kinds of objects (e.g., string constants) are more profitably represented context-insensitively. Even before our current work, the Doop framework had the ability to represent exceptions context-insensitively with the right choice of flags. The second approach consists of not just omitting context for exception objects, but also merging the objects themselves, remembering only a single representative per (dynamic) type. That is, all points-to information concerning exception objects is merged "at the source"—all objects of the same type become one.

This is a fitting approach for exception objects because it relies upon intuition on how exception objects are used in practice. Specifically, the intuition is that exception objects have mostly control-flow significance (i.e., they are used as type labels determining what exception handler is to be executed) and little data-flow impact (i.e., the data stored in exception objects' fields do not affect the precision of an overall points-to analysis). In other words, an exception object's dynamic type alone is an excellent approximation of the object itself. Our measurements of the next section show that this is the case.

Figure 5 shows the changes to earlier rules required to implement the two approaches. The original logic of allocating an object is removed and replaced with two cases: if the allocated object is not an instance of an exception type, then the original allocation logic applies. If it is, then the object is allocated context-insensitively (by using a constant context instead of calling the Record function to create a new one). Furthermore, in the case of merging exception objects, the object itself is replaced by a representative object of its type (arbitrarily chosen to be the object of the same type with the minimum internal identifier). Note that the definition of an exception type consists of merely looking up all types used in `catch` clauses—the definition could also be replaced by the weaker condition of whether a type is a subtype of `Throwable`.

Commmon core of coarsening logic: object allocation rule is replaced by refined version

CHContext (*"ConstantHeapCtx"*) ← True.
ExceptionType (*t*) ← Catch (*superT, _, _*), Subtype (*t, superT*).

~~Record (*heap, ctx*) = ***hctx***~~,
~~VarPointsTo (*var, ctx, heap, hctx*) ←~~
~~Reachable (*meth, ctx*), Alloc (*var, heap, meth*).~~

Record (*heap, ctx*) = ***hctx***,
VarPointsTo (*var, ctx, heap, hctx*) ←
Reachable (*meth, ctx*), Alloc (*var, heap, meth*),
HeapType (*heap, heapT*), ¬ExceptionType (*heapT*).

Additional Rule (over common core) for Context-insensitive treatment

VarPointsTo (*var, ctx, heap, hctx*) ←
Reachable (*meth, ctx*), Alloc (*var, heap, meth*), HeapType (*heap, heapT*),
ExceptionType (*heapT*), CHContext (*hctx*).

Additional Rules (over common core) for merging exceptions by use of representative objects

Representative (*heap, reprH*) ←
HeapType (*heap, heapT*), *reprHeap* = **min**<HeapType (***?***, *heapT*)>.

VarPointsTo (*var, ctx, reprH, hctx*) ←
Reachable (*meth, ctx*), Alloc (*var, heap, meth*), HeapType (*heap, heapT*),
ExceptionType (*heapT*), CHContext (*hctx*), Representative (*heap, reprH*).

Fig. 5. Changes over the rules of Figures 2 and 4 for the two treatments that coarsen the representation of exception objects. The object allocation rule is shown striken out to indicate that it is replaced by a new, conditional version, immediately below. The rules introduce a constant heap context, CHContext, as well as auxiliary relations ExceptionType (*t : T*) and Representative (*heap : H, reprH : H*).

There are some desirable properties of replacing objects with per-type representatives at their creation site. Most importantly, this approach leaves the rest of the analysis unchanged and can maintain all its precision features. Compared to past approaches to merging exceptions (e.g., the single-global-variable assignment of SPARK or PADDLE) we can maintain all the precision resulting from considering exception handlers in order, filtering caught exceptions, and taking into account the specific instructions under the scope of an exception handler. These have been shown to be important features for the precision and performance of the underlying points-to analysis. Ignoring the order of exception handlers, for instance, results in a much less precise context-sensitive call-graph, with 50% more edges [1, Fig.13].

4 Experiments

We next present the results of our experiments with the two ideas for coarsening the representation of exception objects. As we will see, our approach yields substantial performance improvements without sacrificing virtually *any* precision. This is a rather surprising result. Given how crucial the handling of exceptions has been for the precision of the joint points-to and exception analysis, one would expect that representing exception objects crudely (by merging them per-type) would have serious precision implications. For comparison, Bravenboer and Smaragdakis attempted a different approximation: they represented the THROWPOINTSTO relation context-insensitively (i.e., by dropping the *ctx* argument) and found this to significantly hurt the precision of the points-to analysis [1, Sec.5.2], e.g., increasing points-to sets by 10%.[3]

Our implementation is in the DOOP framework and was run on the LogicBlox Datalog engine, v.3.9.0. We use a 64-bit machine with a quad-core Xeon E5530 2.4GHz CPU (only one thread was active at a time) and 24GB of RAM. We analyzed the DaCapo benchmark programs, v.2006-10-MR2 with JDK 1.6.0_30. The choice of JDK is highly significant for Java static analysis. Earlier published DOOP results [1, 2, 28] were for JDK 1.4. We chose to present JDK 1.6 results since it is recent, much larger, and more representative of actual use. However, results for JDK 1.4 can also be found in the first author's M.Sc. thesis, available at `http://cgi.di.uoa.gr/~gkast/MSc_Thesis.pdf`.

There are three primary questions we would like to answer with our experiments:

1. Can we reduce the cost of points-to and exception analysis by coarsening the representation of exception objects, without sacrificing precision?
2. Is the "simple" coarsening approach of treating exceptions context-insensitively sufficient or do we get significant extra benefit from merging exception objects per-type?
3. Do our techniques address the motivation of Table 1, i.e., produce results that roughly match human expectations when reasoning about objects that flow in and out of methods due to exceptions vs. normal call-returns?

[3] We repeated several experiments from [1] in our setting for validation but do not report them here since the results are effectively the same as in that publication.

Tables 2 and 3 show the time and space savings of the coarsening techniques over a large set of analyses, ranging from context-insensitive to a highly-precise 2-object-sensitive with a 2-context-sensitive heap (2obj+2H). The analysis variety includes a mix of call-site-, type-, and object-sensitive analyses. Entries with a dash instead of a number are due to analyses that did not terminate within 90 mins. Entries with neither a number nor a dash mean that we did not run the corresponding experiment. (This only happened for experiments on the full-sensitive treatment of exceptions, which we omitted because the main trends were already clear from a smaller subset of our measurements—those for benchmarks and analyses also shown earlier in Table 1.)

As can be seen, the results demonstrate a substantial benefit in the "bottom-line" performance of the joint analysis from representing exception objects coarsely. Furthermore, the simple approach of dealing with exceptions context-insensitively is clearly insufficient. The advantage of merging objects over merely eliding context can be as high as a 3.4x boost in performance, and rarely falls below a 50% speedup. Space savings tell a similar story, to a lesser but still large extent.

The major question we are addressing next is whether these significant performance improvements entail sacrifices in precision. This requires us to first state the question appropriately. Since all exception objects of the same type are merged, it makes little sense to query points-to information for variables holding such objects (e.g., local variables inside exception handlers). Every such variable will appear to spuriously point to any object of the same dynamic type. Instead, what we want to do is examine the impact of merging exception objects on *the rest* of the points-to analysis. That is, a client analysis that cares about exception objects themselves should not employ our techniques. The question, however, is whether an analysis that applies over the whole program will be affected by the coarse exception representations.

Tables 4 and 5 show precision metrics for our benchmarks and a subset (for space reasons) of our analyses. We show the size of the computed call-graph, in terms of both nodes (i.e., methods) and edges, as well as the number of virtual calls that cannot be statically resolved and casts that cannot be statically be proven safe. (The total number of reachable virtual calls and casts are given for reference.) From our past experience, the call-graph metrics are generally excellent proxies for the overall precision of an analysis, and even tiny changes are reflected on them.

As can be seen, the precision of the program analysis remains virtually unaffected by the coarse representations of exceptions. This confirms that our merged exception objects still carry the essence of the information that the rest of the program needs from them.

The final question from our experiments is whether these two techniques address the motivating measurements of Section 3.1. Table 6 shows the same metrics of (context-sensitive) objects passed to/from methods vs. thrown for the analysis using our coarse representations of exceptions. As can be seen, the handling of exceptions context-insensitively does not suffice to bring the relative ratio of the metrics close to expected values, but merging exception objects per-type does. Specifically, for all values of the "merge" column, the total number of objects passed via calls and returns is several times higher than the number of objects potentially thrown. Thus, the analysis is allocating

		insens	1obj	1obj+H	2obj+H	2obj+2H	1type+H	2type+H	1call	1call+H
antlr	sens	120	338	-	3207	-	543	400	245	975
	insens	111	319	1089	574	2785	334	209	238	543
	merge	75	199	899	249	2313	217	121	128	420
	sen/ins	**1.08**	**1.06**	-	**5.58**	-	**1.62**	**1.91**	**1.02**	**1.79**
	ins/mer	**1.48**	**1.60**	**1.21**	**2.30**	**1.20**	**1.53**	**1.72**	**1.85**	**1.29**
bloat	sens	120	1065	-	-	-	826	1921	426	3403
	insens	120	1057	2337	-	-	483	553	429	1795
	merge	68	432	1727	-	-	292	162	208	1496
	sen/ins	**1.00**	**1.00**	-	-	-	**1.71**	**3.47**	**1.00**	**1.79**
	ins/mer	**1.76**	**2.44**	**1.35**	-	-	**1.65**	**3.41**	**2.06**	**1.19**
chart	sens									
	insens	240	2932	-	-	-	1597	699	591	1334
	merge	138	1434	-	999	-	1253	256	319	1115
	sen/ins									
	ins/mer	**1.73**	**2.04**	-	-	-	**1.27**	**2.73**	**1.85**	**1.19**
eclipse	sens	91	314	2243	-	-	892	800	230	1099
	insens	90	315	800	1059	-	508	348	231	642
	merge	52	140	634	623	-	269	187	90	500
	sen/ins	**1.00**	**1.00**	**2.80**	-	-	**1.75**	**2.30**	**1.00**	**1.71**
	ins/mer	**1.73**	**2.25**	**1.26**	**1.69**	-	**1.88**	**1.86**	**2.56**	**1.28**
jython	sens									
	insens	96	636	2023	-	-	-	-	237	613
	merge	60	129	944	-	-	258	768	98	429
	sen/ins									
	ins/mer	**1.60**	**4.93**	**2.14**	-	-	-	-	**2.42**	**1.43**
luindex	sens	71		838			411			524
	insens	67	168	395	391	2247	239	168	133	288
	merge	45	86	281	153	1982	142	86	67	195
	sen/ins	**1.06**		**2.12**						**1.82**
	ins/mer	**1.48**	**1.95**	**1.40**	**2.55**	**1.13**	**1.68**	**1.95**	**1.98**	**1.47**
lusearch	sens									
	insens	69	185	428	459	3024	262	169	145	302
	merge	45	97	299	214	2723	158	87	71	210
	sen/ins									
	ins/mer	**1.53**	**1.90**	**1.43**	**2.14**	**1.11**	**1.66**	**1.94**	**2.04**	**1.43**
pmd	sens									
	insens	105	274	567	427	2330	327	213	181	392
	merge	66	153	379	188	2076	207	129	100	280
	sen/ins									
	ins/mer	**1.59**	**1.79**	**1.49**	**2.27**	**1.12**	**1.57**	**1.65**	**1.81**	**1.40**
xalan	sens	116		-	-	-	1091			1214
	insens	118	463	1139	-	-	579	349	249	672
	merge	70	219	836	-	-	470	200	126	521
	sen/ins	**1.00**		-			**1.88**	-	-	**1.80**
	ins/mer	**1.68**	**2.11**	**1.36**	-	-	**1.23**	**1.74**	**1.97**	**1.29**

Table 2. Execution time (seconds) for a variety of analyses on various benchmarks

		insens	1obj	1obj+H	2obj+H	2obj+2H	1type+H	2type+H	1call	1call+H
antlr	sens	649	996	-	4608	-	1433	1228	945	3174
	insens	649	996	2560	1536	3686	963	735	945	1740
	merge	544	683	2048	978	3072	675	518	661	1433
	sen/ins	**1.00**	**1.00**	-	**3.00**	-	**1.48**	**1.67**	**1.00**	**1.82**
	ins/mer	**1.19**	**1.45**	**1.25**	**1.57**	**1.19**	**1.42**	**1.41**	**1.42**	**1.21**
bloat	sens	460	1126	-	-	-	1433	2150	1228	5120
	insens	461	1126	2457	-	-	923	992	1228	3379
	merge	363	663	1843	-	-	586	505	773	2969
	sen/ins	**1.00**	**1.00**	-	-	-	**1.55**	**2.16**	**1.00**	**1.51**
	ins/mer	**1.26**	**1.69**	**1.33**	-	-	**1.57**	**1.96**	**1.58**	**1.13**
chart	sens									
	insens	968	3072	-	-	-	2764	1536	1945	3276
	merge	653	1740	-	-	-	1945	811	1331	2560
	sen/ins									
	ins/mer	**1.48**	**1.76**	-	-	-	**1.42**	**1.89**	**1.46**	**1.27**
eclipse	sens	428	748	4505	-	-	2048	1945	694	2867
	insens	429	748	2048	2252	-	1433	1012	694	1638
	merge	300	405	1433	1536	-	679	602	444	1433
	sen/ins	**1.00**	**1.00**	**2.20**	-	-	**1.43**	**1.92**	**1.00**	**1.75**
	ins/mer	**1.43**	**1.84**	**1.42**	**1.46**	-	**2.11**	**1.68**	**1.56**	**1.14**
jython	sens									
	insens	604	1228	3584	-	-	-	-	980	1740
	merge	472	609	2355	-	-	798	1740	601	1331
	sen/ins									
	ins/mer	**1.27**	**2.01**	**1.52**	-	-	-	-	**1.63**	**1.30**
luindex	sens	346		2252	-	-	929			1638
	insens	346	496	1126	1126	2764	645	549	508	874
	merge	265	317	716	565	2150	399	327	349	684
	sen/ins	**1.00**		**2.00**	-	-				**1.87**
	ins/mer	**1.30**	**1.56**	**1.57**	**1.99**	**1.28**	**1.62**	**1.67**	**1.45**	**1.27**
lusearch	sens									
	insens	367	517	1228	1228	3379	681	550	548	926
	merge	277	334	723	690	2867	424	327	370	722
	sen/ins									
	ins/mer	**1.32**	**1.54**	**1.69**	**1.77**	**1.17**	**1.60**	**1.68**	**1.48**	**1.28**
pmd	sens									
	insens	602	817	1536	1331	2969	948	785	814	1331
	merge	505	585	1017	839	2457	647	560	631	1126
	sen/ins									
	ins/mer	**1.19**	**1.39**	**1.51**	**1.58**	**1.20**	**1.46**	**1.40**	**1.29**	**1.18**
xalan	sens	614		-	-	-	2457			2969
	insens	614	1331	3174	-	-	1638	1126	940	1843
	merge	487	710	2252	-	-	823	684	659	1536
	sen/ins	**1.00**		-	-	-	**1.50**			**1.61**
	ins/mer	**1.26**	**1.87**	**1.41**	-	-	**2.00**	**1.64**	**1.42**	**1.20**

Table 3. Disk footprint (MB) for a variety of analyses on various benchmarks

			edges	meths	v-calls	poly v-calls	casts	fail casts
antlr	1obj+H	sens	-	-	-	-	-	-
		insens	59075	8886	33467	1924	1767	985
		merge	59075	8886	33467	1924	1767	985
	2obj+H	sens	55445	8714	32976	1712	1709	611
		insens	55445	8714	32976	1712	1709	611
		merge	55445	8714	32976	1712	1709	611
	1type+H	sens	59738	8916	33507	1948	1770	1070
		insens	59738	8916	33507	1948	1770	1070
		merge	59738	8916	33507	1948	1770	1070
	1call+H	sens	60797	8961	33631	1985	1778	1037
		insens	60797	8961	33631	1985	1778	1037
		merge	60797	8961	33631	1985	1778	1037
bloat	1obj+H	sens	-	-	-	-	-	-
		insens	65672	10116	31049	2067	2815	1911
		merge	65672	10116	31049	2067	2815	1911
	2obj+H	sens	-	-	-	-	-	-
		insens	-	-	-	-	-	-
		merge	-	-	-	-	-	-
	1type+H	sens	66697	10150	31089	2137	2818	2045
		insens	66697	10150	31089	2137	2818	2045
		merge	66697	10150	31089	2137	2818	2045
	1call+H	sens	70340	10200	31214	2129	2829	2007
		insens	70340	10200	31214	2129	2829	2007
		merge	70340	10200	31214	2129	2829	2007
chart	1obj+H	sens						
		insens	-	-	-	-	-	-
		merge	-	-	-	-	-	-
	2obj+H	sens						
		insens	-	-	-	-	-	-
		merge	59027	12510	31111	1610	2765	1055
	1type+H	sens						
		insens	79871	16044	39462	2725	3858	2445
		merge	79896	16044	39462	2730	3858	2450
	1call+H	sens						
		insens	81865	16134	39724	2887	3887	2480
		merge	81890	16134	39724	2892	3887	2485
eclipse	1obj+H	sens	49279	9408	23505	1386	1984	1092
		insens	49279	9408	23505	1386	1984	1092
		merge	49282	9408	23505	1386	1984	1092
	2obj+H	sens	-	-	-	-	-	-
		insens	44792	9188	22852	1168	1912	729
		merge	44795	9188	22852	1168	1912	729
	1type+H	sens	51161	9452	23634	1438	1987	1198
		insens	51161	9452	23634	1438	1987	1198
		merge	51162	9452	23634	1438	1987	1198
	1call+H	sens	52800	9511	23716	1507	2000	1154
		insens	52800	9511	23716	1507	2000	1154
		merge	52800	9511	23716	1507	2000	1154

Table 4. Metrics concerning precision for a variety of analyses. Jython omitted for space.

			edges	meths	v-calls	poly v-calls	casts	fail casts
luindex	1obj+H	sens	40004	7876	18263	1110	1521	796
		insens	40004	7876	18263	1110	1521	796
		merge	40004	7876	18263	1110	1521	796
	2obj+H	sens	-	-	-	-	-	-
		insens	36477	7702	17748	899	1463	496
		merge	36477	7702	17748	899	1463	496
	1type+H	sens	40646	7906	18303	1138	1524	889
		insens	40646	7906	18303	1138	1524	889
		merge	40646	7906	18303	1138	1524	896
	1call+H	sens	41790	7953	18492	1171	1532	837
		insens	41790	7953	18492	1171	1532	837
		merge	41790	7953	18492	1171	1532	837
lusearch	1obj+H	sens						
		insens	42977	8526	19556	1289	1622	812
		merge	42977	8526	19556	1289	1622	812
	2obj+H	sens						
		insens	39352	8344	19048	1071	1564	508
		merge	39352	8344	19048	1071	1564	508
	1type+H	sens						
		insens	43676	8558	19620	1319	1625	927
		merge	43676	8558	19620	1319	1625	934
	1call+H	sens						
		insens	45071	8626	19857	1352	1643	938
		merge	45071	8626	19857	1352	1643	938
pmd	1obj+H	sens						
		insens	46826	9277	21591	1168	1990	1210
		merge	46826	9277	21591	1168	1990	1210
	2obj+H	sens						
		insens	42988	9090	21004	942	1,931	846
		merge	42988	9090	21004	942	1,931	846
	1type+H	sens						
		insens	47539	9311	21632	1192	1993	1311
		merge	47540	9311	21632	1192	1993	1317
	1call+H	sens						
		insens	48895	9371	21843	1240	2003	1273
		merge	48895	9371	21843	1240	2003	1273
xalan	1obj+H	sens	-	-	-	-	-	-
		insens	54033	10511	25683	1857	2042	1055
		merge	54038	10511	25683	1858	2042	1055
	2obj+H	sens	-	-	-	-	-	-
		insens	-	-	-	-	-	-
		merge	-	-	-	-	-	-
	1type+H	sens	54792	10561	25760	1887	2049	1235
		insens	54792	10561	25760	1887	2049	1235
		merge	54796	10561	25760	1888	2049	1235
	1call+H	sens	56658	10613	25891	1966	2059	1203
		insens	56658	10613	25891	1966	2059	1203
		merge	56666	10613	25891	1967	2059	1203

Table 5. Metrics concerning precision for a variety of analyses (cont'd from Table 4)

		insens		1obj+H		2obj+H		1type+H		1call+H	
		insens	merge	insens	merge	insens	merge	insens	merge	insens	merge
antlr	objs passed	697	651	26,867	25,271	6,983	6,079	4,702	4,528	17,236	17,155
	objs thrown	3,123	204	14,197	965	23,856	1,730	7,084	428	12,461	822
	ratio	**.22**	**3.18**	**1.89**	**26.16**	**.29**	**3.51**	**.66**	**10.56**	**1.38**	**20.86**
bloat	objs passed	829	781	22,633	22,325	-	-	5,338	5,167	46,650	46,563
	objs thrown	4,112	257	18,697	1,189	-	-	10,140	589	20,048	1,251
	ratio	**.20**	**3.02**	**1.21**	**18.76**	-	-	**.52**	**8.76**	**2.32**	**37.22**
chart	objs passed	2,315	1,723	-	-	-	16,739	-	18,721	46,158	41,909
	objs thrown	8,331	414	-	-	-	7,231	-	1,357	19,747	1,371
	ratio	**.27**	**4.15**	**-**	**-**	**-**	**2.31**	**-**	**13.79**	**2.33**	**30.56**
eclipse	objs passed	637	539	14,045	16,875	15,234	14,357	6,025	6,471	18,311	18,186
	objs thrown	4,064	248	15,983	1,047	36,699	2,612	10,516	593	11,829	777
	ratio	**.15**	**2.17**	**.87**	**16.10**	**.41**	**5.49**	**.57**	**10.90**	**1.54**	**23.40**
jython	objs passed	801	479	43,068	35,106	-	-	-	-	19,307	-
	objs thrown	3,452	215	20,449	1,025	-	-	-	-	11,288	-
	ratio	**.23**	**2.23**	**2.10**	**34.21**	**-**	**-**	**-**	**-**	**1.71**	**-**
luindex	objs passed	383	339	7,664	7,414	3,525	3,037	3,031	2,881	8,186	8,109
	objs thrown	2,544	166	8,953	606	21,401	1,521	6,055	363	7,218	486
	ratio	**.15**	**2.03**	**.85**	**12.21**	**.16**	**1.99**	**.50**	**7.92**	**1.13**	**16.67**
lusearch	objs passed	429	385	8,085	7,822	4,519	4,010	3,319	3,165	9,005	8,928
	objs thrown	2,764	180	9,222	626	19,432	1,435	6,341	379	7,880	528
	ratio	**.15**	**2.13**	**.87**	**12.49**	**.23**	**2.79**	**.52**	**8.33**	**1.14**	**16.90**
pmd	objs passed	461	414	9,273	8,949	4,388	3,864	3,500	3,337	10,948	10,867
	objs thrown	3,008	198	10,409	695	19,961	1,465	7,100	431	8,529	577
	ratio	**.15**	**2.09**	**.89**	**12.87**	**.21**	**2.63**	**.49**	**7.73**	**1.28**	**18.81**
xalan	objs passed	668	589	24,618	24,155	-	-	6,350	5,821	18,448	18,301
	objs thrown	3,876	251	20,765	1,372	-	-	11,278	641	12,198	810
	ratio	**.17**	**2.34**	**1.18**	**17.60**	-	-	**.56**	**9.08**	**1.51**	**22.59**

Table 6. Objects on method boundaries compared to exception objects thrown by a method (measured in thousands)

its reasoning effort in a way that closely matches what one would expect intuitively, possibly indicating that large further improvements are unlikely.

5 Related Work

We next discuss in more detail some of the past work in points-to analysis combined with exceptions.

As mentioned earlier, points-to analysis frameworks SPARK [19] and PADDLE [18, 20] both use imprecise exception analysis via assignment of thrown exceptions to a single global variable. Even if this were to change to distinct per-type variables, it would still have significant precision shortcomings compared to our approach since the order and scope of exception handlers would be ignored. The Soot framework also has a separate exception analysis [16] that is not based on a pointer analysis.

The IBM Research WALA [6] static analysis library supports several different pointer analysis configurations. The points-to analyses of WALA support computing which ex-

ceptions a method can throw (analogously to our ThrowPointsTo relation), but no results of Wala's accuracy or speed have been reported in the literature. Wala allows an exception object to be represented by-type (analogously to our coarsening) but it is unclear how the underlying analysis compares to our joint exception and points-to analysis. It will be interesting to compare our analyses to Wala in future work.

Type-based approaches to dealing with exception objects have also been explored before [14, 15], in the context of a separate exception analysis (i.e., not jointly with a precise points-to analysis and not in comparison to an object-based exception representation).

bddbddb is a Datalog and BDD-based database that has been employed for points-to analysis [29, 30]. 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, bddbddb ignores the order of exception handlers and also disables filtering of caught exceptions. Both of these features are crucial for precision.

Chatterjee et al. analyze the worst-case complexity of fully-precise pointer analysis with exceptions [3]. This is a theoretical analysis with no current application to practical points-to algorithms.

Sinha et al. discuss how to represent exception flow in the control-flow graph [27]. One of the topics is handling `finally` clauses. We analyze Java bytecode, hence the complex control-flow of `finally` clauses is already handled by the Java compiler.

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 that work.

6 Conclusions

When analyzing an object-oriented program, exceptions pose an interesting challenge. If completely ignored, valuable properties of the program are lost and large amounts of code appear unexercised. If handled in isolation (either before or after points-to analysis) the result is imprecise and the analysis suffers from inefficiency. A joint points-to and exception analysis offers the answer but has significant time and space cost due to the precise representation of exception objects. We showed that we can profitably coarsen the representation of exception objects in such a joint analysis. Precision remains unaffected, for the parts of the analysis not directly pertaining to exception objects, i.e., for most common analysis clients, such as cast elimination, devirtualization, and call-graph construction. At the same time, performance is significantly enhanced. Thus the approach is a clear win and is now the default policy for exception handling in the Doop framework.

There are interesting avenues for further work along the directions of the paper. Our approach is based on standard patterns of use of exception objects. These patterns

can perhaps be generalized to other kinds of objects used as "message carriers". Furthermore, the question arises of how such patterns translate across languages. Is there an analogous concept in functional languages that can be exploited to gain scalability? Also, it remains to be seen whether similar approaches can apply to alias analysis in C++ in the presence of exceptions.

Acknowledgments. We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant and a European Research Council Starting/Consolidator grant. We also gratefully acknowledge the support of the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award. The work in this paper was significantly aided by pre-existing mechanisms that Martin Bravenboer has implemented in the Doop framework.

References

1. Bravenboer, M., Smaragdakis, Y.: Exception analysis and points-to analysis: Better together. In: Dillon, L. (ed.) ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis. New York, NY, USA (Jul 2009)
2. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications. ACM, New York, NY, USA (2009)
3. Chatterjee, R., Ryder, B.G., Landi, W.A.: Complexity of points-to analysis of Java in the presence of exceptions. IEEE Trans. Softw. Eng. 27(6), 481–512 (2001)
4. Choi, J.D., Grove, D., Hind, M., Sarkar, V.: Efficient and precise modeling of exceptions for the analysis of Java programs. SIGSOFT Softw. Eng. Notes 24(5), 21–31 (1999)
5. Eichberg, M., Kloppenburg, S., Klose, K., Mezini, M.: Defining and continuous checking of structural program dependencies. In: ICSE '08: Proc. of the 30th int. conf. on Software engineering. pp. 391–400. ACM, New York, NY, USA (2008)
6. Fink, S.J.: T.J. Watson libraries for analysis (WALA). `http://wala.sourceforge.net`
7. Fu, C., Milanova, A., Ryder, B.G., Wonnacott, D.G.: Robustness testing of Java server applications. IEEE Trans. Softw. Eng. 31(4), 292–311 (2005)
8. Fu, C., Ryder, B.G.: Exception-chain analysis: Revealing exception handling architecture in Java server applications. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering. pp. 230–239. IEEE Computer Society, Washington, DC, USA (2007)
9. Fu, C., Ryder, B.G., Milanova, A., Wonnacott, D.: Testing of java web services for robustness. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 23–34. ACM, New York, NY, USA (2004)
10. Guarnieri, S., Livshits, B.: GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In: Proceedings of the 18th USENIX Security Symposium. pp. 151–168. SSYM'09, USENIX Association, Berkeley, CA, USA (2009), `http://dl.acm.org/citation.cfm?id=1855768.1855778`
11. Hajiyev, E., Verbaere, M., de Moor, O.: Codequest: Scalable source code queries with Datalog. In: Proc. European Conf. on Object-Oriented Programming (ECOOP). pp. 2–27. Spinger (2006)
12. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: PLDI'07: Proc. ACM SIGPLAN conf. on Programming Language Design and Implementation. pp. 290–299. ACM, New York, NY, USA (2007)

13. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 226–238. ACM, New York, NY, USA (2009)
14. Jo, J.W., Chang, B.M.: Constructing control flow graph for Java by decoupling exception flow from normal flow. In: Laganà, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA (1). Lecture Notes in Computer Science, vol. 3043, pp. 106–113. Springer (2004)
15. Jo, J.W., Chang, B.M., Yi, K., Choe, K.M.: An uncaught exception analysis for Java. Journal of Systems and Software 72(1), 59–69 (2004)
16. Jorgensen, J.: Improving the precision and correctness of exception analysis in Soot. Tech. Rep. 2003-3, McGill University (Sep 2004)
17. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: 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. pp. 1–12. ACM, New York, NY, USA (2005)
18. Lhoták, O.: Program Analysis using Binary Decision Diagrams. Ph.D. thesis, McGill University (Jan 2006)
19. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using Spark. In: Hedin, G. (ed.) Compiler Construction, 12th Int. Conf. LNCS, vol. 2622, pp. 153–169. Springer, Warsaw, Poland (April 2003)
20. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. 18(1), 1–53 (2008)
21. Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of javascript applications in the presence of frameworks and libraries. Tech. Rep. MSR-TR-2012-66, Microsoft Research (Jul 2012)
22. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In: Conf. on Programming Language Design and Implementation (PLDI). pp. 305–315. ACM (Jun 2010)
23. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. 14(1), 1–41 (2005)
24. Reps, T.: Demand interprocedural program analysis using logic databases. In: Ramakrishnan, R. (ed.) Applications of Logic Databases. pp. 163–196. Kluwer Academic Publishers (1994)
25. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis. pp. 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ (1981)
26. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University (May 1991)
27. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Trans. Softw. Eng. 26(9), 849–871 (2000)
28. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity (the making of a precise and scalable pointer analysis). In: ACM Symposium on Principles of Programming Languages (POPL). pp. 17–30. ACM Press (Jan 2011)
29. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with binary decision diagrams for program analysis. In: Yi, K. (ed.) APLAS. Lecture Notes in Computer Science, vol. 3780, pp. 97–118. Springer (2005)
30. Whaley, J., Lam, M.S.: 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. pp. 131–144. ACM, New York, NY, USA (2004)