Using Datalog for Fast and Easy Program Analysis

Yannis Smaragdakis[1,2] and Martin Bravenboer[3]

[1] University of Massachusetts, Amherst, MA 01003, USA,
yannis@cs.umass.edu
[2] University of Athens, Athens 15784, Greece,
smaragd@di.uoa.gr
[3] LogicBlox Inc., Two Midtown Plaza, Atlanta, GA 30309, USA,
martin.bravenboer@acm.org

Abstract. Our recent work introduced the Doop framework for points-to analysis of Java programs. Although Datalog has been used for points-to analyses before, Doop is the first implementation to express full end-to-end context-sensitive analyses in Datalog. This includes key elements such as call-graph construction as well as the logic dealing with various semantic complexities of the Java language (native methods, reflection, threading, etc.).
The findings from the Doop research effort have been surprising. We set out to create a framework that would be highly complete and elegant without sacrificing performance "too much". By the time Doop reached maturity, it was a full order-of-magnitude faster than Lhoták and Hendren's Paddle—the state-of-the-art framework for context-sensitive points-to analyses. For the exact same logical points-to definitions (and, consequently, identical precision) Doop is more than 15x faster than Paddle for a 1-call-site sensitive analysis, with lower but still substantial speedups for other important analyses. Additionally, Doop scales to very precise analyses that are impossible with prior frameworks, directly addressing open problems in past literature. Finally, our implementation is modular and can be easily configured to analyses with a wide range of characteristics, largely due to its declarativeness.
Although this performance difference is largely attributable to architectural choices (e.g., the use of an explicit representation vs. BDDs), we believe that our ability to efficiently optimize our implementation was largely due to the declarative specifications of analyses. Working at the Datalog level eliminated much of the artificial complexity of a points-to analysis implementation, allowing us to concentrate on indexing optimizations and on the algorithmic essence of each analysis.

1 Introduction

Points-to analysis is one of the most fundamental static program analyses. It consists of computing a static approximation of all the data that a pointer variable or expression can reference during program run-time. The analysis forms the basis for practically every other program analysis and is closely inter-related with mechanisms such as 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.

In recent work [1, 2], we presented Doop: a versatile points-to analysis framework for Java programs. Doop is crucially based on the use of Datalog for specifying the program analyses, and on the aggressive optimization at the Datalog level, by programmer-assisted indexing of relations so that highly recursive Datalog programs evaluate near-optimally. The optimization approach accounts for several orders of magnitude of performance improvement: unoptimized analyses typically run over 1000 times more slowly. The result is quite surprising: compared to the prior best-comparable system Doop often achieves speedups of an order-of-magnitude (10x or more) for several important analyses, while yielding identical results. This performance improvement is not caused by any major algorithmic innovation: we discuss in Section 3 how performance is largely a consequence of the optimization opportunities afforded by using a higher-level programming language (Datalog). Declarative specifications admit automatic optimizations and at the same time enable the user to identify and apply straightforward manual optimizations.

An important aspect of Doop is that it is full-featured and "all Datalog". That is, Doop is a rich framework, containing context insensitive, call-site sensitive, and object-sensitive analyses for different context depths, all specified modularly as variations on a common code base. Additionally, Doop achieves high levels of completeness, as it handles complex Java language features (e.g., native code, finalization, and privileged actions). As a result, Doop emulates and often exceeds the rich feature set of the Paddle framework [7], which is the state-of-the-art in terms of completeness for complex, context-sensitive analyses. All these features are implemented entirely in Datalog, i.e., declaratively. Past points-to analysis frameworks (including those using Datalog) typically combined imperative computation and some declarative handling of the core analysis logic. For instance, the bddbddb system [10, 11] expresses the core of a points-to analysis in Datalog, while important parts (such as normalization and call-graph computation—except for simple, context-insensitive, analyses) are done in Java code. It was a surprise to researchers even that a system of such complexity can be usefully implemented declaratively. Lhoták [6] writes: "*[E]ncoding all the details of a complicated program analysis problem (such as the interrelated analyses [on-the-fly call graph construction, handling of Java features]) purely in terms of subset constraints [i.e., Datalog] may be difficult or impossible.*"

The more technical aspects of Doop (including the analysis algorithms and features, as well as our optimization methodology) are well-documented in prior publications [1, 2, 9]. Here we only intend to give a brief introduction to the framework and to extrapolate on our lessons learned from the Doop work.

2 Background: Points-To Analysis in Datalog

Doop's primary defining feature is the use of Datalog for its analyses. Architecturally, however, an important factor in Doop's performance discussion is that it employs an *explicit* representation of relations (i.e., all tuples of a relation are represented as an explicit table, as in a database), instead of using Binary Decision Diagrams (BDDs), which have often been considered necessary for scalable points-to analysis [6, 7, 10, 11].

We use a commercial Datalog engine, developed 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.

Datalog is a great fit for the domain of program analysis and, as a consequence, has been extensively used both for low-level [5, 8, 11] and for high-level [3, 4] analyses. The essence of Datalog is its ability to define recursive relations. Mutual recursion is the source of all complexity in program analysis. For a standard example, the logic for computing a callgraph depends on having points-to information for pointer expressions, which, in turn, requires a callgraph. We can easily see such recursive definitions in points-to analysis alone. Consider, for instance, two relations, `AssignHeapAllocation(?heap, ?var)` and `Assign(?to, ?from)`. (We follow the Doop convention of capitalizing the first letter of relation names, while writing variable names in lower case and prefixing them with a question-mark.) The former relation represents all occurrences in the Java 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 Doop 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) <- AssignHeapAllocation(?heap, ?var).
2 VarPointsTo(?heap, ?to) <- 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. In our syntax, the left arrow symbol (`<-`) separates the inferred fact (the *head*) from the previously established facts (the *body*).

The key for a precise points-to analysis is context-sensitivity, which consists of qualifying program variables (and possibly object abstractions—in which case the context-sensitive analysis is said to also have a *context-sensitive heap*), with context information: the analysis collapses information (e.g., "what objects this method argument can point to") over all possible executions that result in the same context, while separating all information for different contexts. Object-sensitivity and call-site-sensitivity are the main flavors of context sensitivity in modern points-to analyses. They differ in the contexts of a context, as well as in when contexts are created and updated. Here we will not concern ourselves with such differences—it suffices to know that a context-sensitive analysis qualifies its computed facts with extra information.

Context-sensitive analysis in Doop is, to a large extent, similar to the above context-insensitive logic. The main changes are due to the introduction of Datalog variables

representing contexts for variables (and, in the case of a context-sensitive heap, also objects) in the analyzed program. For an illustrative example, the following two rules handle method calls as implicit assignments from the actual parameters of a method to the formal parameters, in a 1-context-sensitive analysis with a context-*insensitive* heap. (This code is the same for both object-sensitivity and call-site-sensitivity.)

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

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

(Note that some of the above relations are functions, 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.)

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

For deeper contexts, one needs to add extra variables, since pure Datalog does not allow constructors and therefore cannot support value combination. We have introduced in DOOP a macro system to hide the number of context elements so that such variations do not pollute the analysis logic.

Generally, the declarative nature of DOOP often allows for very concise specifications of analyses. We show in an earlier publication [2] the striking example of the logic for the Java cast checking—i.e., the answer to the question "can type A be cast to type B?" The Datalog rules are almost an exact transcription of the Java Language Specification. A small excerpt, with the Java Language Specification text included in comments, can be seen in Figure 1.

3 Discussion: DOOP and Large-Scale Development in Datalog

Perhaps the main lesson learned from our experience with DOOP and its definition in Datalog is quite simple: *Datalog is not an abstract logic and does not magically yield automatic programming capabilities, but it is still much higher-level than current mainstream programming languages.*

Recent Datalog research has often concentrated on generalizing the language (to full first-order logic and higher-order logics), and on applying automated reasoning techniques. Although this is certainly a valuable direction, we believe that one should not lose sight of the fact that Datalog is already a very high-level language when compared to mainstream general purpose languages, such as Java, C++, or C#. It is, therefore, perhaps more interesting to examine Datalog not as a proxy for a logic but as an application programming language. Many of the benefits that we obtained with DOOP are

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

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

Fig. 1. Excerpt of Datalog code for Java cast checking, together with Java Language Specification text in comments. The rules are quite faithful to the specification.

directly due to such an approach. Of course, this raises the question of whether plain Datalog is expressive enough for general application programming. As we saw, even for the domain of points-to analysis, researchers were highly skeptical of the feasibility of expressing all elements (including those consisting mostly of tedious engineering) of a complex analysis in Datalog. We believe that this is precisely what is missing at this point in the evolution of Datalog. The language needs to be developed as a real programming language, with appropriate library support (for, e.g., graphics, communication, etc., APIs), tool support, a mature engine (for advanced automatic optimization of rule evaluation and efficient representation of relations), and possibly expressiveness enhancements (e.g., macros, exponential-search, or other high-order capabilities). A final element, which we are still debating whether it is essential or an intermediate state, is the ability to manually optimize a Datalog program, by exposition of an easy-to-understand cost model and appropriate interfacing with the engine.

Such arguments are easy to see in the context of Doop. The use of Datalog in Doop is certainly not as a logic. Doop is not written as an abstract specification that a clever runtime system automatically optimizes and executes efficiently. We needed to develop an optimization methodology for highly recursive programs and to introduce indexes manually, in order to attain optimal performance. The difference in performance between optimized and unoptimized Doop rules is enormous. At the same time, Doop is expressed at a much higher level than a similar implementation of a points-to analysis in Java or C++. The declarativeness of Datalog and the suitability of the LogicBlox Datalog platform for application development were crucial for Doop in more than one way:

- We relied on query optimization (i.e., intra-rule, as opposed to inter-rule, optimization) being performed automatically. This was crucial for performance and, although a straightforward optimization in the context of database relations, results in far more automation than programming in a mainstream high-level language.
- The declarativeness and modularity of Doop specifications contributed directly to performance. The surprisingly high performance of Doop compared to past frameworks is due to combining two factors: simple algorithmic enhancements, and an explicit representation instead of BDDs. Eliminating either of these factors results in complete lack of scalability in Doop. For instance, an explicit representation alone makes many standard analyses infeasible in Doop: even a 1H-object-sensitive analysis (i.e., 1-object-sensitive with a context-sensitive heap) would be completely infeasible for realistic programs. Nevertheless, we observed that this lack of scalability was due to very high redundancy (i.e., large sizes of some relations without an increase in analysis precision) in the data that the analysis was computing. The redundancy was easy to eliminate with two simple algorithmic enhancements: *1)* we perform exception analysis on-the-fly [1], computing contexts that are reachable because of exceptional control flow while performing the points-to analysis itself. The on-the-fly exception analysis significantly improves both precision and performance; *2)* we treat static class initializers context-insensitively (since points-to results are equivalent for all contexts of static class initializers), thus improving performance while keeping identical precision. These enhancements (especially the former, which results in highly recursive definitions of core relations) would be quite hard to consider in a non-declarative context. In Doop, such enhancements could be added with minor changes to the rules or with just the addition of extra rules. Once redundancy is eliminated via our algorithmic enhancements, an explicit representation (with the help of our index optimizations) becomes much faster than using BDDs.

Based on our experience, we believe that Datalog can have a bright future for application development. In a programming setting that has a dire need for higher-level programming abstractions, Datalog holds a great promise. The elements missing in order to fulfill this promise are not in the direction of greater declarativeness and automated reasoning abilities. Pursuing more complete-logic-like variants of Datalog may turn out to be an unreachable goal and is certainly not what is missing in practice: Datalog is already much more declarative than the mainstream languages currently used for application programming. Instead, it is practical elements that are missing and that can propel actual Datalog implementations to the mainstream. An interesting question is whether it is necessary for a programmer to treat a Datalog program as a program and not as a specification, i.e., whether the programmer should have the ability to understand and manually influence the program's execution cost.

In summary, the Doop framework has raised the bar in the domain of points-to analysis by introducing fast, modular, and scalable implementations of precise points-to analysis algorithms, while yielding important lessons about the architecture of such implementations. At the same time, however, we hope that Doop will be representative of future successes for Datalog application development as a whole.

Acknowledgments This work was funded by the NSF (CCF-0917774, CCF-0934631) and by LogicBlox Inc.

References

1. M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In L. Dillon, editor, *ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis*, New York, NY, USA, July 2009.
2. M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2009. ACM.
3. M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08: Proc. of the 30th int. conf. on Software engineering*, pages 391–400, New York, NY, USA, 2008. ACM.
4. E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with Datalog. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pages 2–27. Spinger, 2006.
5. M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM.
6. O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, Jan. 2006.
7. 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.
8. T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.
9. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity (the making of a precise and scalable pointer analysis). In *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2011. ACM.
10. J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.
11. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.