Deductive Formulation and Implementation of a Close-to-Source Points-To Analysis

David Johnston
Iowa State University
Ames, Iowa
dwtj@iastate.edu

ABSTRACT

Inspired by previous work [7][5], we designed and implemented a program analysis using a purely-deductive style. Our field-sensitive, Andersen's style, points-to analysis for Java is written using a variant of Datalog. This analysis is encoded as relatively concise set of definitions and inference rules. These definitions and rules directly and declaratively model the analysis, and the result is essentially identical to an inference-rule-based mathematical formalization of the analysis. The analysis is built on top of a general-purpose, highly-declarative, close-to-source representation of Java code; a Java compiler plugin was written to automatically extract this representation from user code.

We have found this close-to-source and purely-deductive style is a natural fit for prototyping analyses, especially because such implementations can so closely correspond to their mathematical formulations. Our main hope is that our compiler plugin and the extended analysis example in this paper is useful to other analysis writers who may be interested in prototyping their own analyses using this style.

1. INTRODUCTION

1.1 Why Purely-Deductive?

This project grew out of an interest in trying to find a methodology to directly and declaratively prototype program analyses. Unlike much prior work, we are less concerned with our implementations' potential performance and scalability and more concerned with their expressiveness, soundness, development/maintenance costs, and implementation verifiability.

Prior work on purely-deductive program analyses [7][5] led us to believe that the highly-declarative nature of the purely-deductive style might be a good fit for prototyping analyses. For us, the main attraction of the purely-deductive approach was the closeness of an analysis's formal mathematical specification and its implementation: the purely-deductive style potentially makes the implementation directly correspond to its formalism.

To begin validating this idea, we went through the process of implementing a points-to analysis over the Java language. In this paper we discuss our resulting implementation's design in some detail. This paper might thus be considered a preliminary case study of taking an existing analysis and formulating it in the purely-deductive style. Our presentation expects some familiarity with Java and with the basic ideas behind points-to analyses, but we expect from the reader no experience with logic programming.

Like prior work [5], we chose to implement our analysis using LogicBlox, a Datalog-based deductive database. *Datalog* is a subset of the Prolog logic programming language, and it has been used as the query language for a number of deductive databases [11].

A *deductive database* is a database where logical inference rules are encoded in the database. These rules are automatically used to deduce new facts from old facts. Facts may be either values in the database or relations between these values. The basic idea behind this programming model is that as new facts are added to the database, existing inference rules automatically "fire" for any set of database values satisfying the precondition; when a rule fires, its consequents—whose variables are bound to the values satisfying the preconditions—are added to the database. In this way, the database itself automatically computes the least solution of a set of facts and rules.

The semantics of this programming model are informally explained as part of our extended example. The power of this approach is that rules can have complex dependencies (including mutually inductive definitions). The result is a highly expressive and declarative programming model which should be familiar to those with any background in deductive logic.

1.2 Why Close-to-Source?

Prior work has shown that the purely-deductive style turns out to be very appropriate for implementing various points-to analyses [14][7][13][12][6]. One similarity among all of these works is that, like many other program analyses in the Java ecosystem, the client's code comes from bytecode. Our approach is different, because it is built on top of a representation which comes directly from the client's source code. We believe (but do not validate the idea) that this point of difference may have important ramifications on some analyses.

When a program analysis is formally designed, discussed, reasoned about, or implemented, there is a particular formal language serving as the target of the analysis. This is

ACM ISBN N/A.
DOI: N/A

the particular language over which the analysis will operate and of which the analysis "directly speaks". This formal language serves as the foundations on which any analysis is built. We might in general call this language the canonical representation, the language being analyzed, or just *the analyzed language.*[1]

Like other design decisions, the choice of an analyzed language for a particular application always comes with tradeoffs along various dimensions: soundness, precision, simplicity, performance, scalability, development/maintenance costs, implementation verifiability, interoperability with other tools, client usability, etc. The evaluation of these tradeoffs usually requires that the designer has a certain analysis or a certain range of analyses in mind.

It seems natural to loosely characterize a potential analyzed language based on its *closeness-to-source*. Consider the various analyzed languages for a Java program when it is compiled, transformed by the Soot framework, and executed. We might say that

- Java compiler AST is very close-to-source
- JVM bytecode is relatively far-from-source
- Jimple lies somewhere in-between.

The closeness of an analyzed language to its source language is thus a spectrum. Each language has its uses, and each comes with tradeoffs. When an analysis (or class of analyses) is built on top of one analyzed language or another, the implementation may end up being more or less clear, rigorous, performant, or otherwise appropriate.

Our work thus diverges from previous work because we use a close-to-source analyzed language rather than a close-to-bytecode approach. We made this choice out of our desire for precision, declarativeness, and generality in analysis prototyping. We describe our analyzed language in § 2.1.

Our close-to-source approach stands in contrast with much prior work on Java program analyses. Most are formulated over a further-from-source language like JVM bytecode or Soot's Jimple. This compilation (or series of compilations) may lose some information which may be relevant to some analyses. Our representation still retains certain information related to Java's static semantics which would otherwise be lost in compilation.

1.3 Deductive Close-to-Source Points-To

In this work, we have chosen to implement a points-to analysis over Java. For our purposes, we think that points-to is a good choice for us to investigate because it has been so well studied and discussed:

- The rich literature on points-to includes many different approaches to this classic problem to draw from. This gives us a range of designs which we can attempt to prototype and compare.
- Points-to analysis has been a foundational program analysis to many other client analyses, and its usefulness has thus been well established.
- A good deal of inquiry has been done on points-to as it specifically relates to Java and the JVM (e.g. [10]).

[1] If we take the use of the term "analysis" very broadly, what we here call the analyzed language might go by other names in other communities and contexts. For example, a PL researcher might call their analyzed language a core calculus, and a compiler writer might call theirs an intermediate representation.

	Allocation	Assignment	Field store	Field load
Instruction	$a : dst := new\ C$	$dst := src$	$dst.f := src$	$dst := src.f$
Edge	$a \rightarrow dst$	$src \rightarrow dst$	$src \rightarrow dst.f$	$src.f \rightarrow dst$
Rules	$\dfrac{a \rightarrow dst}{a \in pt(dst)}$	$\dfrac{src \rightarrow dst \quad a \in pt(src)}{a \in pt(dst)}$	$\dfrac{src \rightarrow dst.f \quad a \in pt(src) \quad b \in pt(dst)}{a \in pt(b.f)}$	$\dfrac{src.f \rightarrow dst \quad a \in pt(src) \quad b \in pt(a.f)}{b \in pt(dst)}$

Figure 1: *Taken from [9].* Core inference rules from an instruction-oriented points-to analysis. Our points-to analysis uses the inference rule style seen here, but our set of rules handles a large variety of Java expressions.

- Much high-quality implementation work has gone into crafting flexible and powerful points-to analyses for Java [8][3][5]. This opens the possibility of developing meaningful comparisons of performance, scalability, soundness, precision, etc.

The use of inference rules has long been a part of formalizing and specifying points-to analyses. Andersen's thesis [4] formally presents a variety of rules for generating abstractions and constraints to support C's various language features. Lhoták and Hendren [9], when presenting their Soot-based framework for implementing Java points-to analyses, describe the core of their analyses as a set of four inference rules (reproduced here in Figure 1). In these cases, inference rules are natural way to formally specify (parts of) their imperative implementation. More recent work (in particular work from Lam [7][14] and Smaragdakis [5][13][12][6]) has taken the use of inference rules in points-to analyses a step further by writing their analysis implementations as a set of inference rules.

Java points-to—even purely-deductive points-to—has thus been well-studied. But our formulation is fundamentally different from these works: ours is close-to-source and expression-oriented, not close-to-bytecode and instruction-oriented. The rich literature gives us the opportunity and challenge to take existing points-to analysis strategies and to make them work over Java itself, not over JVM bytecode, and from there to make meaningful comparison to prior work.

For us, using a close-to-source analyzed language takes the purely-deductive analysis approach to its next logical step: expressive and sound deductive analyses are benefitted by expressive and sound foundations. By taking the close-to-source approach, we can use the power and flexibility of the purely-deductive style to handle any meaningful static feature of the Java language, even those which might be lost in the translation to JVM bytecode (e.g. generics).

Our analysis follows a classical design, though it will seem naive by comparison to today's state of the art. Our analysis is

- inclusion-based (i.e. Andersens's style [4]),
- field-sensitive,
- flow-insensitive, and
- context-insensitive.

The traditional formulation of such an analysis is with respect to a set of *pointer abstractions* (a.k.a. variables), some set of *object abstractions* (a.k.a. locations), and some set of *inclusion constraints* (a.k.a. subset constraints). Given

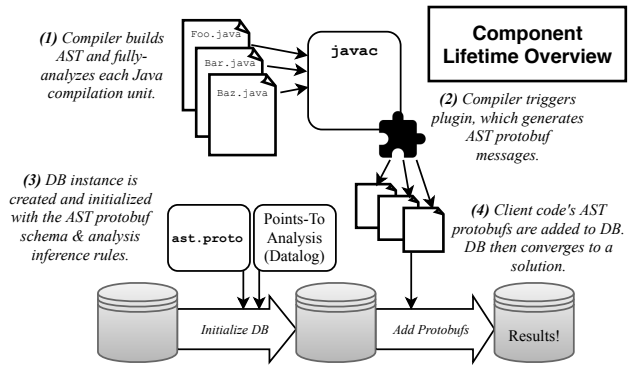


Figure 2: An overview of the interactions between our system's components, described in four steps. We implemented `ast.proto`, the compiler plugin, and the points-to analysis. Before the database begins computing the results, three sources of information are added to a deductive database instance: the specification of our close-to-source representation of Java (written as a Protobuf specification file), the points-to analysis rules (written in Datalog), and the client's code (encoded as protobufs by our compiler plugin). Once these inputs have been added to a database instance, it will automatically compute a solution. A client can then query this database to obtain the results.

some client code encoded in the analyzed language, a points-to analysis is designed to construct these abstractions, to deduce the constraints between them, and then to ultimately produce its points-to solutions.

1.4 Contributions

Our formulation of points-to differs from past work because, previously, Java analyses have been formulated over some lower-level *instruction*-based analyzed language, like JVM bytecode or Jimple. Our formulation is novel because it is formulated over Java *expressions*. The basic ideas of the analysis are all standard, but the details of the expression-oriented analysis required a certain amount of adaptation. Our resulting design is explained in § 2.2 and § 2.3.

Our implementation work was focused on three components:

1. `ast.proto`, which simply specifies our close-to-source analyzed language,
2. a `javac` compiler plugin, which constructs the analyzed language from client code, and
3. the points-to analysis itself, written in Datalog.

An overview of how these components interact to compute the analysis's solution is given in Figure 2.

The first two components are involved in taking the client's source code and constructing a representation of it in our close-to-source analyzed language. The analyzed language itself was specified by writing Google Protocol Buffers. An Oracle `javac` plugin was used to inspect fully-analyzed AST of client Java source code and to fill these protobufs accordingly. As we will explain, these protobufs are used to send facts about the user's code to our deductive database. We will discuss this process in greater detail in § 2.1.

The program analysis itself is written as a set of Datalog rules and constraints. The details of this component's design and implementation take up the bulk of this paper, § 2.2 and § 2.3.

Our concrete contributions are:

- The preliminary development of an easy-to-integrate Java compiler plugin to extract a close-to-java-source model from a compiler invocation.
- The preliminary formulation of a traditional points-to analysis whose analyzed language is close-to-java-source.
- The preliminary implementation of this analysis using a purely-deductive style in a Datalog-based deductive database.
- A detailed explanation of this analysis's design and implementation. We hope that this might serve as an extended example for anyone who might be interested to learn how to implement their own analyses in the purely-deductive style.

Despite the relative detail that we go into, this work is still quite incomplete. The implementation described here cannot yet be said to cover even Java's core language features. (We go into detail about some of the analysis's limitations in § 3.) Additionally, this work offers no firm evidence about the potential utility or generality of our close-to-source purely-deductive analysis methodology. We are instead presenting our design and implementation of components which we believe will be useful first steps towards such a work. We can, however, say that our preliminary experience programming in the purely-deductive style has so far left us optimistic that this may be a good strategy for prototyping program analyses.[2]

2. METHODOLOGY

2.1 The Analyzed Language

We have spoken of using analyzed languages which are *closer* to source because one can certainly go too far: it would of course be pathological for almost any Java program analysis to be designed and implemented over, say, unparsed source string manipulation.

A more realistic candidate for general-purpose close-to-source program analyses would be existing Java compiler ASTs, but to facilitate correct and maintainable analyses, we are looking for something a bit further from the source than this: something more rigorously structured, like a core calculus, but still retaining all of Java's static semantics.[3]

[2] For example, while implementing our system, most of the developmental pain points came from writing the Java compiler plugin, which just needs to directly adapt Java AST to a set of protobuf messages. Developing our analysis prototype was relatively painless by comparison.

[3] In our opinion, existing Java compiler ASTs are not ideal for general purpose close-to-source program analyses because they contain certain unnecessary ambiguities in their representation of Java source. These arise from combination of performance constraints and complexities of the Java language, particularly ambiguities of Java's grammar which can only be resolved after the compiler's semantic analyses. This is not a criticism of the design of these systems,

What we are looking for is a somewhat "cleaned-up" representation of Java AST, a representation which can be analyzed "offline", that is, by tools running outside of the compiler process itself. So our approach was this: we wrote a compiler plugin for Oracle's `javac` to analyze the compiler's data structures, and to extract such a representation for each compilation unit.[4]

2.1.1 Java as Protobuf Messages

We chose to encode our analyzed language using Google's Protocol Buffers. This choice was made for a few reasons:

1. A `.proto` file can specify mutually recursive objects and hierarchical relations between them.
2. A `.proto` file is very declarative and easy to understand. These files have a flavor somewhat similar to a formal grammar, except that terms have named and typed fields.
3. Efficient code for reading and writing compactly-encoded protobufs can be compiled from a `.proto` specification for a variety of languages. Thus, Java source code represented in a collection of protobufs should be usable by a wide variety of tools.

We wrote a protobuf specification file, `ast.proto`, to serve as the canonical design of our close-to-source analyzed language. The protobuf messages are used to encode a wide variety of facts about the client's source code. Different protobuf message types are used to encode different language features (e.g. class/method/variable declarations, unary/binary/assignment expressions, etc.).

Because these protobufs form the foundations of our methodology, we will briefly describe two examples. Listing 1 shows two message types appearing in the specification file to encode two different kinds of Java expressions.

Listing 1: Two Java expression protobuf messages.

```
message ConditionalExpr {
  required SomeExpr condition = 1;
  required SomeExpr true_expr = 2;
  required SomeExpr false_expr = 3;
}

message AssignExpr {
  oneof to {
    LocalVariableAccessExpr  to_local_variable =  1;
    FieldAccessExpr          to_field =           2;
    ArrayComponentAccessExpr to_array_component = 3;
  }
  required SomeExpr from = 4;
}
```

The first, `ConditionalExpr`, is the message type used to encode conditional expressions (a.k.a. ternary expressions) when they appear in source. Each of its three fields is marked `required`, meaning that every conditional expression must have the three indicated sub-expressions. These sub-expressions each have type `SomeExpr`, meaning that the field could be any kind of expression.

The second message type, `AssignExpr`, is used to encode assignment expressions. The `oneof` group named `to` has three fields; each field is used to distinguish between the three mutually exclusive possibilities:[5] the left hand side is either a local variable access, a field variable access, or an array variable access. The last field indicates that every assign expression needs to a right hand side, and any kind of expression may appear here.

2.1.2 Java as Datalog Facts

Our points-to analysis is implemented using LogicBlox, a Datalog-based deductive database system. This system has native support for importing and exporting data contained in protobufs. Similar to how a `.proto` specification file can be compiled to Java or C++ APIs, a `.proto` file can be compiled to a set of LogicBlox database entity types and relations between. Once these definitions have been installed, it is thus simple and efficient to load our information about Java source into LogicBlox, and to have the database automatically start performing our analysis.

So, as described in the last section, our compiler plugin generates protobuf messages from the client's code. When such messages are loaded into the database, they are transformed into a set of *facts*, that is, entities and relations between them.

When a protobuf message representing some term is loaded into the database, a uniquely identifiable entity is automatically added to the database to represent that particular term. This is done for each of its sub-terms, each of their sub-terms, and so on, recursively. Additionally, for every message with a field directly holding a sub-message, a fact is added to the database, relating the message to its sub-message.

Here is an example. Suppose that a Java program contains some conditional expression e of the form e_{cond} `?` e_{true} `:` e_{false}, where each $e_{\_}$ is a metavariable denoting some sub-expression. Adding this expression's protobuf message to the database will add all of the facts in Listing 2:

Listing 2: Datalog facts of a conditional expression.

```
1  ConditionalExpr(e).
2  SomeExpr(e_cond).
3  SomeExpr(e_true).
4  SomeExpr(e_false).
5  ConditionalExpr_condition[e] = e_cond.
6  ConditionalExpr_true_expr[e] = e_true.
7  ConditionalExpr_false_expr[e] = e_false.
8  ...
```

Such facts are auto-generated from a protobuf message when it is added to the database. These are the seven facts added to the database because of some message which encodes expression e.[6] The first four facts are all entity declarations. The last three facts are relations between them.

For example, `ConditionalExpr_condition[_] = _` is the predicate relating `ConditionalExpr` entities to their (unique)

only an acknowledgement that they have been designed for a very precise set of program analyses, and that we want our analyses to build on top of this fundamental set of analyses.

[4] This compiler plugin was implemented via a combination of the Java Annotation Processing API [1] (which is implemented by all standards-compliant Java compilers) and the Compiler Trees API [2] (which is a well-supported public API of `javac`, but not supported by other Java compilers such as Eclipse). Our plugin's design is this: as the compiler finishes analyzing a compilation unit, our plugin receives a callback; during this callback, we inspect the compiler's data structures and write the protobufs to disk.

[5] A technical note: the `oneof` keyword means that *at most* one of the options is set, not *exactly* one.

[6] The ellipsis at the end is meant to suggest that more facts may be added to the database because of its sub-expressions.

associated `condition` expressions. The first, second, and fifth facts together might thus be read, "e is a particular conditional expression and e_{cond} is a particular expression of some kind; e's `condition` is e_{cond}."

2.2 Specifying the Analysis: Object and Pointer Abstractions

So far in our pipeline, we have used protobufs to extract descriptions of Java source code from a Java compiler and then loaded these protobufs into our deductive logic database, thus populating the database with a set of source-level entities and source-level relations between these entities.

We will next define how our analysis *abstracts* various concrete terms of the analyzed language to construct the abstract values of our analysis. Like the points-to analyses from which we are drawing inspiration, we define two domains of abstract values: pointer abstractions (with database type `Ptr`) and object abstractions (with database type `Object`). These abstractions are used to (approximately) model the many possibilities which may happen at runtime. They are important to our analysis because as we will explain in § 2.3, the results of our analysis are encoded in `PointsTo`, a relation from `Ptrs` to `Objects`.

2.2.1 Object Abstractions

Our analysis constructs object abstractions to statically represent a certain set of objects which may exist at runtime. Currently, there are just two simple rules for constructing object abstractions: one for class object abstractions and another for array object abstractions. We briefly define and describe the former below; the latter is similar.

The rule is this: every new class expression has its own uniquely constructed object abstraction. A new class expression's object abstraction is meant to statically represent all of the runtime class instances which can ever be instantiated by this expression. This rule is written using traditional inference rule notation as,

ClassObject

$$\frac{\texttt{NewClassExpr(e)}}{\texttt{ClassObject(o)} \quad \texttt{ClassObjectOfExpr(e) = o}}$$

and this rule can be encoded in the deductive database with the following code:

Listing 3: Datalog for `ClassObjects`.

```
// Entity Declarations:
Object(o) -> .
ClassObject(o) -> Object(o).

// Entity's Constructor Declaration:
ClassObject_of_expr[e] = o ->
    NewClassExpr(e),
    ClassObject(o).
lang:constructor(`ClassObject_of_expr).

// Entity Construction Inference Rule:
ClassObject(o), ClassObject_of_expr[e] = o <-
    NewClassExpr(e).
```

This code has three parts:

1. **Entity Declarations:** This says that `Object` is some (top-level) type of entity which can exist in the database and that `ClassObject` is a type of database entity subtyping `Object`.
2. **Constructor Declaration:** This says that the relation `ClassObject_of_expr` can uniquely associate a new class expression with one of our object abstractions. Furthermore, this relation can be used to create unique `ClassObject` instances.
3. **Entity Construction Inference Rule:** This inference rule will drive the construction of our object abstractions. A distinct instance will be constructed for every new class expression in the database, and the two entities will be associated via the constructor relation.

The first two parts are not so important, they just show the Datalog syntax which we use to declare entities and their constructors. It is the last of these three parts, the *inference rule*, that one should note. In particular, notice that this rule's meaning is the same as the preceeding formal inference rule; they only differ in syntax.

Throughout the rest of the paper, we will not be using the Datalog syntax, opting to instead use the more traditional inference rule syntax. However, the above example illustrates how these inference rules are easily and clearly encoded in Datalog.

2.2.2 Pointer Abstractions

Possibly the most fundamental question in the design of a points-to analysis is "where should we track points-to sets?" Many formulations have been proposed in the literature, but in this work we try to follow a simple and traditional design: our analysis is field-sensitive and context-insensitive. Loosely speaking, this means that we have (up to) one points-to set for each local variable/expression, but we have (up to) one points-to set for each field of each object abstraction. To make this design concrete, we now discuss our pointer abstractions and the `PtrOf` function.

A pointer abstraction is a static abstraction designed to represent some set of pointers which may arise at runtime. Essentially, a pointer abstraction is some entity at which our analysis tracks a points-to set.

The `PtrOf` function is used to map terms to these pointer abstractions. Each of our abstract values (both object abstractions and pointer abstractions) are generally "associated" with a particular term.[7] In this section, we will describe the rules which associate various terms (or in the last case, an abstraction-term pair) with some pointer abstraction. Clearly defining this function is important, because:

- The function's co-domain is the set of entities at which points-to sets are tracked.
- The function is used prevalently in the definition of our points-to analysis to access pointer abstraction values from concrete terms. (See § 2.3.)

As we will see in the examples below, the precise meaning of `PtrOf` depends upon the kind of term. We will give a precise meaning of the function in each of our examples; other cases are similar to one of these examples.

Example 1: Static Fields. The simplest case arises from a class variable declaration (i.e., a field marked `static`).

[7] We saw this kind of "association" between concrete terms and abstract entities previously with object abstractions, where `ClassObject_of_expr` served as a one-to-one mapping between our new class expressions and our class object abstractions.

This is the simplest case to model because (ignoring complications from class loading) there will be just one runtime variable related to a class variable declaration, and (ignoring complications from reflection) all uses of a class variable are statically unambiguous.

We now define a pair of induction rules: the first defines the `PtrOf` any class field declaration, whereas the second defines the `PtrOf` any expression which *uses* a class field.

The first rule says that for every class field *declaration*, we construct a single unique pointer abstraction for that field.

PtrOf:StaticField

```
VariableDecl_has_modifier(f, STATIC)
-------------------------------------
StaticFieldPtr(p)    PtrOf(f) = p
```

This second rule says that for every *expression* which accesses a static field (i.e. reads or writes it), we associate that expression with this variable declaration's pointer abstraction.[8]

PtrOf:StaticFieldAccess

```
          FieldAccessExpr(e)
FieldAccessExpr_mode(e) = STATIC
 VariableDecl(f)    PtrOf(f) = p
-------------------------------
          PtrOf(e) = p
```

Looking at the rules' conclusions, notice that both rules define the `PtrOf` function for a certain class of terms; however only the first rule is responsible for constructing pointer abstractions.

The meaning of the `PtrOf` some variable declaration is likely clear: the `PtrOf` a variable is the pointer abstraction whose points-to set statically represents all of the objects which may be assigned to that variable at runtime. But what do we mean for an *expression* to be associated with some pointer abstraction?

We define the `PtrOf` some expression to be a pointer abstraction which is meant to statically represent the evaluation of that expression at runtime; the points-to set of this pointer abstraction should contain any object abstraction to which the expression may evaluate.

Because our analysis is context insensitive, local variables (including parameters) will all be modeled similarly to this class field variables example: a variable declaration implies the existence of a unique pointer abstraction, and the `PtrOf` all access expressions to that variable is this same pointer abstraction. However, because our analysis is field sensitive, we will see in Example 5 that things are not quite as simple for instance fields.

Example 2: Binary Minus Expressions. A binary minus expression always evaluates to primitive values, and never to reference values.[9] There is no need to associate a pointer abstraction with this kind of expression, since its points-to set would always be empty.

Example 3: Conditional Expressions. A conditional expression may evaluate to values from either of its subexpressions. We want the conditional expression to have a points-to set which is distinct from either of these sub-expressions, one which includes all of the object abstractions from both of the sub-expressions.[10][11] The inference rule to construct this pointer abstraction is:

PtrOf:ConditionalExpr

```
          ConditionalExpr(e)
-------------------------------------
ConditionalExprPtr(p)    PtrOf(e) = p
```

Example 4: Type Casts. There is little reason to construct a distinct points-to set for a type cast expression: it will always be equivalent to the points-to set of its subexpression. Therefore, we do not construct a pointer abstraction for this kind of expression. Instead, we have the following rule,

PtrOf:TypeCast

```
          TypeCastExpr(e)
TypeCastExpr_casted_expr(e) = e'
--------------------------------
        PtrOf(e) = PtrOf(e')
```

which just says that the pointer abstraction associated with a type cast is the same as the pointer abstraction associated with its sub-expression.

Example 5: Instance Fields. This example demonstrates how, in order to make our analysis field-sensitive, we "parameterize" instance fields with object abstractions. How such parameterization is handled is what makes the analysis not quite completely naive.

An instance field access expression needs its own pointer abstraction on which we can "merge" together multiple points to sets. This need to merge is just like the conditional expression which we saw Example 3, In that previous case, the merge came from two subexpressions, but for a class object field expression, the merge may come from many points-to sets: it depends on the points-to set of the base expression being dereferenced.

We will thoroughly explain merging and dereferencing in § 2.3.2, but the point make here is that we need to create a points-to set for the field access expression into which we can merge various points-to sets:

PtrOf:InstanceFieldAccessExpr

```
          FieldAccessExpr(e)
FieldAccessExprMode(e) = INSTANCE
---------------------------------
InstanceFieldAccessExprPtr(p)    PtrOf(e) = p
```

Instance field access expressions are like other expressions which we have seen, where we construct just one pointer abstraction per expression. But pointer abstractions for the fields themselves are not quite so simple.

Because our analysis is field-sensitive, we don't just have a single pointer abstraction for each instance field declaration (like we saw for static fields). Rather, our instance fields are parameterized on object abstractions; this means that for every object abstraction `o` of some class type `T` and for every instance field on that type, we construct a distinct pointer abstraction (and thus a distinct points-to set). Here

[8] Because our analysis is context insensitive, every expression will be associated with (at most) one pointer abstraction instance: the pointer abstractions of expressions are not parameterized on some calling context, object context, or any other kind of context.

[9] This is not quite true in Java because of auto-boxing, but that language detail is not yet handled by our analysis. This point is discussed in § 3.

[10] We create a distinct points-to set for a conditional expression for the usual reason: to increase the precision of our analysis. Otherwise, the conditional expression and its sub-expressions would all have to share a single points-to set.

[11] We will see in § 2.3 how exactly these inclusion constraints work, but for now, we know that the conditional expression needs its own pointer abstraction.

is the rule for this[12][13]:

PtrOf:ClassObjectField

$$\frac{\texttt{ClassObject(o)} \quad \texttt{ClassObjectInstanceField(o, f)} \quad \texttt{VariableDecl(f)}}{\texttt{ClassObjectPtr(p)} \quad \texttt{PtrOf}(\langle \texttt{o}, \texttt{f} \rangle) = \texttt{p}}$$

Thus, in relation to a single instance field declaration, many pointer abstractions (and thus distinct points-to sets) may need to be constructed in relation to it:

- PtrOf:InstanceFieldAccessExpr implies a pointer abstraction for a each instance field access expression;
- PtrOf:ClassObjectField implies a pointer abstraction for certain pairs of object abstractions and fields.

Examples Summary. The five preceeding examples demonstrated the key ways in which terms of the analyzed language are associated with pointer abstractions via the `PtrOf` function. Every term of the analyzed language has `PtrOf` rules similar to one of these five:

1. *Static Fields (Make One; Use It Across Accesses)* The `PtrOf` a static field declaration is one which is instantiated just for that declaration. The `PtrOf` each static field access expression will be the same as the `PtrOf` the static field declaration which it accesses.
2. *Binary Minus (Don't Have One):* There is no `PtrOf` a binary minus expression (i.e. there are no rules in this case).
3. *Conditional Expressions (Make One):* The `PtrOf` a conditional expression is instantiated just for that expression.
4. *Type Cast Expressions (Use Nested One):* The `PtrOf` a type cast expression is the `PtrOf` its subexpression.
5. *Instance Fields (Make Many):* The `PtrOf` an instance field access expression is instantiated just for that expression. For some instance field declaration, a distinct pointer abstractions is instantiated for each class object abstraction with that field.

2.3 Specifying the Analysis: Points-To Inference Rules

In the last section, we thoroughly described the meanings of the abstract values used by our analysis and how a set of inference rules are used to construct them from a program encoded in our analyzed language. In this section, we will use these abstract values along with the terms of the analyzed language to construct the `PointsTo` relation—the result of our analysis. In particular, we will write inference rules whose preconditions concern abstract values and terms of the program and whose conclusions are assertions about one of two interconnected relations: `PointsTo` and `Includes`.[14]

[12] Here, `ClassObjectInstanceField` is just a helper relation saying that the class type allocated in association with object abstraction `o` includes the instance field declaration `f`. This relation is inferred from the new class expression associated with the class object abstraction and the fields declared on this class.

[13] The construction of multiple pointers for each instance field declaration may be easier to understand when visualized. See Listing 6 and Figure 4 for an example.

[14] We say that these two relations are "interconnected" because, as we will see, their induction rules are defined in terms of one another, that is, they are mutually inductively defined.

Our Datalog declarations for these relations are as follows:

Listing 4: Declarations for `PointsTo` and `PointsTo`

```
PointsTo(p, o) -> Ptr(p), Object(o).
Includes(sup, sub) -> Ptr(sup), Ptr(sub).
```

The first declaration simply reads "`PointsTo` is a relation from pointer abstractions to object abstractions"; we take a fact of the form `PointsTo(p, o)` to mean that some pointer abstraction `p` may point to some object abstraction `o`.[15]

The second declaration reads, "`Includes` is a relation between pointer abstractions." We take a fact of the form `Includes(supPtr, subPtr)` to mean that all objects in the points-to set of `subPtr` are included in the points-to set of `supPtr`.[16] Our inference rule to ensure this is

IncludesPropagatesPointsTo

$$\frac{\texttt{Includes(supPtr, subPtr)} \quad \texttt{PointsTo(subPtr, o)}}{\texttt{PointsTo(supPtr, o)}}$$

The inference rules which construct `PointsTo` and `Includes` facts are the essence of our analysis.[17] Different rules are used for different kinds of terms. In the rest of this section, we present a sampling of such rules which should be illustrative, since other rules are similar to one of these.

2.3.1 The `PointsTo` and `Includes` Relations

We explain the connection between these two relations by working through a simple example, `CondExpr.java`, whose code is given in Listing 5 and whose points-to graph is given in Figure 3.

[15] In this paper, we have at various points informally referred to "the points-to set of some pointer abstraction p". We can define this notion formally to be the function from pointer abstractions to sets of object abstractions such that

$$PointsToSet(p) = \{o : PointsTo(p, o)\}$$

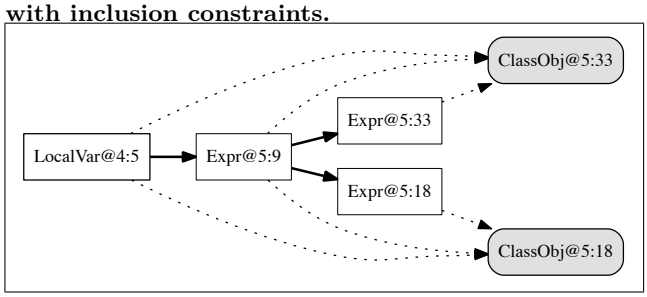
[16] Asserting `Includes(supPtr, subPtr).` is equivalent to saying $PointsToSet(subPtr) \subseteq PointsToSet(supPtr)$. This is why inclusions are often called subset constraints.

[17] A key aspect of our Datalog-based analysis implementation is that the deductive database is responsible for incrementally adding `PointsTo` and `Includes` facts until the least solution is reached. We, the analysis designers, need only specify the analysis's inference rules. This has the benefit of simplifying the design and implementation of the analysis, but it also means that we cannot use any of the clever algorithms out there for efficiently computing the dynamic transitive closure of a points-to graph. We must instead rely on the deductive database's built-in optimizations, which we suspect will not scale as well as these customized methods.

Listing 5: `CondExpr.java`

```
1  package ptr;
2  class CondExpr {
3    static void main() {
4      Object o;
5      o = (true) ? new Object() : new Object();
6    }
7  }
```

Figure 3: Solved points-to graph of `CondExpr.java` with inclusion constraints.



In our figures of points-to graphs such as this one, dotted edges will be used to denote `PointsTo` facts, and solid edges denote `Includes` facts; grey nodes denote object abstractions, and white nodes denote pointer abstractions.

Every new class expression is associated with its own class object abstraction, as discussed in § 2.2.1. Additionally, the `PtrOf` each new class expression is its own pointer abstraction, and this pointer abstraction will always point to just one object abstraction.[18]

Looking at Listing 5, we can see two such expression on line 5; these correspond to nodes "Expr@5:18" and "Expr@5:33" of our points-to graph. In the points-to graph, the *dotted* edges from these expressions to "ClassObj@5:33" and "ClassObj@5:18", respectively, denote the `PointsTo` facts between these pointer abstractions and object abstractions. The following inference rule is responsible for adding such `PointsTo` facts to the database[19]:

PropagateUpNewClass

$$\frac{\texttt{NewClassExpr(e)} \quad \texttt{PtrOf(e)=p} \quad \texttt{ClassObject\_of\_expr(e)=o}}{\texttt{PointsTo(p, o)}}$$

As we described in the preceeding section, we create a pointer abstraction for each conditional expression. In this example, we have such an expression, labeled "Expr@5:9" in our points to graph. Previously, we were motivated to introduce a pointer abstraction for this kind of expression to create a points-to set at which we could merge together the points-to sets of two subexpressions. We can now precisely describe the logic used to do such a "merge" using the following inference rule:

MergeUpConditionalExpr

$$\frac{\begin{array}{c}\texttt{ConditionalExpr(e)} \quad \texttt{PtrOf(e) = p} \\ \texttt{(ConditionalExpr\_true\_expr(e) = e'} \vee \\ \texttt{ConditionalExpr\_false\_expr(e) = e')} \\ \texttt{PtrOf(e') = p'}\end{array}}{\texttt{Includes(p, p')}}$$

Unlike the preceding inference rule, which added individual `PointsTo` facts to the database, this rule directly[20] adds `Includes` facts to the database.

We can see represented in our points-to graph two `Includes` facts which were added by this rule: they are the bold edges going rightward from "Expr@5:9" (the conditional expression's pointer abstraction) into "Expr@5:18" and "Expr@5:33" (the subexpressions' pointer abstractions). Because of these inclusion facts and IncludesPropagatesPointsTo, *the conditional expression points to each of our object abstractions*, which we can see as dotted edges.

Lastly for this example, consider the local variable assignment on line 5. Intuitively, we want a rule which says that the pointer abstraction of an assigned-to local variable includes everything from the right-hand side's pointer abstraction. We give an `Includes` rule which achieves this:

PropagateAcrossLocalAssignment

$$\frac{\begin{array}{c}\texttt{AssignExpr(e)} \\ \texttt{PtrOf(AssignExpr\_to\_local\_variable(e)) = lhsPtr} \\ \texttt{PtrOf(AssignExpr\_from(e)) = rhsPtr}\end{array}}{\texttt{Includes(lhsPtr, rhsPtr)}}$$

The bold edge from the local variable's pointer abstraction ("LocalVar@4:5") to the conditional expression's pointer abstraction ("Expr@5:9") is a consequence of this rule. Thus, by IncludesPropagatesPointsTo, the local variable declaration points to everything which the conditional expression points to, that is, both object abstractions. The points-to graph includes the two dotted edges which denote these two `PointsTo` facts: "LocalVar@4:5" to "ClassObj@5:18" and "LocalVar@4:5" to "ClassObj@5:33".

2.3.2 *Object Dereferencing*

This section uses a pair of examples to illustrate how our analysis handles variable dereferencing. We only directly discuss dereferencing class objects (i.e. instance field loads & stores), but array dereferencing is similar.

In the last example, we saw an inference rule which adds an individual object abstraction to a points-to set, and we saw a pair of inference rules which adds inclusion constraints between pointer abstractions. These are two patterns which we see among the rules defining `PointsTo` and `Includes`.

In this section, we will see a third pattern, where the firing of an inference rule adds a variable number of inclusion constraints; exactly where these inclusion constraints are added to the graph depends upon the contents of a base expression's points-to set.[21] As we will see, these inclusion constraints arise from expressions which "use" pointer abstractions parameterized on object abstractions. (We saw

[18] In the example points-to graphs in this section, for every grey object abstraction node, there is a pointer abstraction with the same source position nearby whose only outgoing edge goes to the object abstraction. Looking for these pairs of nodes can help make reading the graphs easier.

[19] This is an example of a syntax-directed inference rule which directly adds individual `PointsTo` facts to the database. There are only a few such rules in our system. We have many more inference rules which add `Includes` facts to the database. The next rule discussed is of the latter kind.

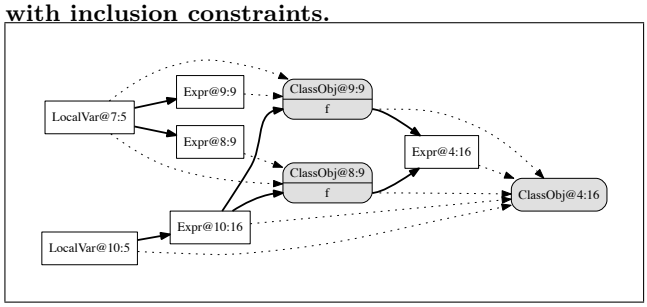
[20] Recall that by the IncludesPropagatesPointsTo rule, a single `Includes` fact may *indirectly* imply many `PointsTo` facts.

[21] In some other Andersen's style points-to analyses, constraints from this last pattern have been called "complex" constraints.

Listing 6: `FieldLoad.java`

```
1  package ptr;
2  class FieldLoad {
3    static class OneField {
4      Object f = new Object();
5    }
6    static void main() {
7      OneField a;
8      a = new OneField();
9      a = new OneField();
10     Object o = a.f;
11   }
12 }
```

Figure 4: Solved points-to graph of `FieldLoad.java` with inclusion constraints.



such pointer abstractions in § 2.2.2, Example 5.)

Loading From an Instance Field. First, consider Listing 6. Here we can see two instances of `OneField` are instantiated and assigned to a local variable, `a`; field `f` of both instances is initialized on line 4. Figure 4 visualizes this code's solution. The notation of this points-to graph is the same as in the preceding graph except for the representation of instance fields' pointer abstractions.

Unlike the preceding example, this code instantiates classes with instance fields. We began discussing how our analysis handles instance fields in § 2.2.2. There we said that because we are designing our analysis to be field-sensitive, every instance field of every `OneField` instance needs to be modeled with its own distinct points-to set, that is, with its own pointer abstraction. There we formally denoted these pointer abstractions as constructions from a pair of the form `(o, f)`. Here we visualize these pointer abstractions as "contained within" their corresponding object abstraction: one pointer abstraction is contained within "ClassObj@7:18" for that instance's field `f`; another pointer abstraction is contained within "ClassObj@8:9" for this other instance's field `f`.

Notice in our points-to graph, that both of our field `f` pointer abstractions point to "ClassObj@4:16". (This is as expected because of the field's initializer on line 4.) Given that this object abstraction is in these points-to sets, how exactly did it end up in the points-to set of `Object o` via the field load on line 9? The key to the answer is the following inference rule:

MergeInstanceFieldAccess

```
        PtrOf(e) = exprPtr
  InstanceFieldAccessExpr_accesses(e) = f
InstanceFieldAccessExpr_base(e) = baseExpr
    PointsTo(PtrOf(baseExpr), baseObj)
      ClassObject_field(baseObj(f))
-------------------------------------------
      Includes(exprPtr, fieldPtr)
```

This rule constructs various `Includes` facts from the access expression to the fields which may be accessed. *Which fields may be accessed is determined by examining the contents of the points-to set of the base expression.* The name of this rule comes from the idea that the points-to set of an instance field access expression is the union of the points-to sets of these various fields: the contents of the fields have been "merged" to make the expression's points-to set.

In our example, this rule will fire twice to construct the `Includes` facts from "Expr@9:16" to both "(ClassObj@7:18, `f`)" and also "(ClassObj@8:9, `f`)". (These facts can be seen in our graph as two dark edges leaving "Expr@9:16"). Why does this rule make edges to the `f` field of these object abstractions? Because the points-to set of the base expression, the local variable `a`, has been found to include both of these object abstractions. (These facts can be seen in our graph as dotted edges going out from "LocalVar@7:5").

This rule is quite different from what we saw in § 2.3.1: the preconditions are not only based on the client code and our abstract values; the preconditions also depend upon the results of our analysis. A rule like this makes `PointsTo` and `Includes` mutually inductive definitions. In Datalog, we can simply state such dependencies, and the deductive database will automatically converge to the least solution without writing a custom solving algorithm. This declarativeness illustrates why Datalog-based analyses may be an attractive option in contexts such as analysis prototyping.

Storing to an Instance Field. We will explain one more of our analysis's inference rules using one more example. Like in the preceding example, we will see a rule related to dereferencing where `PointsTo` is used as a precondition for inferring `Includes` facts. In particular, this example includes an assignment expression which stores values into an instance field of a dereferenced object; the rule will be responsible for propagating object abstractions from the right-hand side's points-to set to certain instance field points-to sets indicated by the left-hand side. Which points-to sets are indicated by the left hand side will depend upon our `PointsTo` relation.

Consider Listing 7 and its solved points-to graph, Figure 5. As with the last example, two different `OneField` instances are assigned to local variable `a`, and `a` is dereferenced.

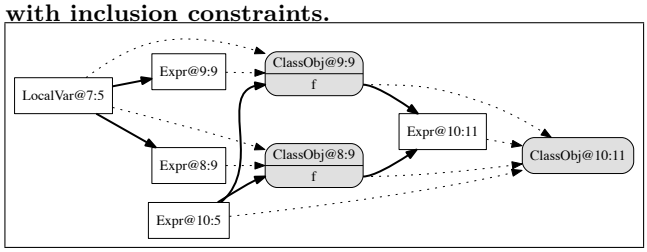
Notice that this points-to graph's structure is almost identical to the last graph; they only differ in that there is no second local variable node and there are some different node labels. Though they are similar, the reasoning that led to this solution is different from the last.

The key difference from the last example is that the fields are now initialized via an instance field access expression on line 10. Here, variable `a` is dereferenced to *store* values as part of this assignment expression. That both fields point to the object abstraction is indicated in the graph by the two dotted edges coming out of the fields.

Listing 7: `FieldStore.java`

```
1  package ptr;
2  class FieldStore {
3    static class OneField {
4      Object f;
5    }
6    static void main() {
7      OneField a;
8      a = new OneField();
9      a = new OneField();
10     a.f = new Object();
11   }
12 }
```

Figure 5: Solved points-to graph of `FieldStore.java` with inclusion constraints.



But why exactly did the analysis construct these two dotted edges? As with the last example, *the key is the points-to set of the dereferenced base expression.* In this case, the dereferenced expression is the variable `a`, which may point to both of the `OneField` object abstractions; thus, it will be these two object abstractions' `f` fields which are indicated by the left-hand side of the assignment; thus, both of these fields may point to any of the object abstractions that the right-hand side may point to (in this case, just "ClassObj@10:11"). This is why we add the two `Includes` facts to "Expr@10:11", which in turn implies the two `PointsTo` facts to "ClassObj@10:11", the two dotted edges in question.

We have seen and discussed all of the parts of this reasoning except for the rule which constructs the `Includes` facts across the assignment. The idea behind the rule is that given some assignment expression which stores to an instance field,

1. get the points-to set of the field access expression's base expression,

2. for each of the object abstractions in the base expression's points-to set, get the appropriate field pointer abstraction within that object abstraction, and

3. for each of these fields' pointer abstractions, include everything from the right-hand side's points-to set.

Our analysis encodes this idea formally with the following rule:

ScatterAcrossInstanceFieldAssignment

```
    AssignExpr_to_field(assignExpr) = lhsExpr
!VariableDecl_has_modifier_str(lhsExpr, "STATIC")
 PtrOf(FieldAccessExpr_base(lhsExpr)) = basePtr
           PointsTo(basePtr, baseObj)
      FieldAccessExpr_accesses(lhsExpr) = f
          PtrOf((baseObj, f)) = fieldPtr
      AssignExpr_from(assignExpr) = rhsExpr
             PtrOf(rhsExpr) = rhsPtr
-------------------------------------------
          Includes(fieldPtr, rhsPtr)
```

The contents of the RHS's points-to set are going to be copied to various other points-to sets, and the points-to set to which fields' points-to sets depends upon the base expression's points-to set. This base expression's points-to set may grow dynamically as the solution is computed. We use the term "scatter" in the rule's name to suggest this copying processes's variability and dynamism.

3. FUTURE WORK

This paper has presented the key parts of the analysis which has been implemented. The implemented analysis is still quite incomplete. Some missing features negatively impact the analysis's precision, while others negatively impact its soundness. The implementation's incompleteness sometimes comes from the compiler plugin (which generates the protobuf representation) and sometimes comes from the Datalog rules (which specify the analysis itself).

One current source of unnecessary imprecision comes from the fact that our analysis does not differentiate between reference and primitive types: the design conservatively and naively assumes that all values are reference types, and thus need to be tracked by our analysis. However, variables and expressions of primitive type need not be tracked, since they can never store or evaluate to object types.

Method calls, class subtyping, exception throwing, interfaces, and generic types are all core Java language features which are not yet fully supported by our analysis, and this lack of support for them obviously leads to unsoundness in our analysis. These are all glaring omissions from our design, but a more subtle source of unsoundness in the analysis is the missing support for Java's auto-boxing feature: object abstractions ought to be added to auto-boxing expressions which implicitly instantiate objects from primitives.

Work has started on the implementation of method call support. In the current implementation, points-to set contents are propagated from call expressions' arguments–including receiver arguments–to methods' formal parameters'. However, the analysis does not yet have a correct way of identifying all of the methods which may be called by a virtual method call. We expect that we will be able to fix this problem without too much difficulty by modifying the information encoded in the protobufs and then adding some dereferencing-based inference rules like we saw in § 2.3.2; this would make our analysis simultaneously a call-graph analysis and a points-to analysis (a design which has been used in a number of other works).

In addition to these problems with features, we are interested in improving the usability of the system. Our compiler plugin, which generates the protobufs, is trivial to integrate into any standard Java build system, but the rest of the analysis system more cumbersome to use. Starting the deductive

database and initializing it with the analysis and protobufs is currently done using a customized Gradle build script. This was convenient for development of our prototype, because it let us easily chain together a number of simple tools, but this complexity shouldn't ought to be better packaged for users. Our initial thought is that this functionality could all be included within the compiler plugin.

One rather different direction for this work would be experiments with enhanced type systems over Java using Java annotations. This could be similar in spirit to pluggable type checkers.

We believe that for this work to move forward much work needs to be done to validate its correctness and its scalability; both can be done via comparisons to previous work. Particularly interesting comparisons could be made with Doop [12], which also uses the same Datalog variant as we do to write and compute points-to analysis variants. Does our analysis produce the same results as theirs? Is one a superset of the other? Does our close-to-source implementation scale similarly to their close-to-bytecode analysis? Does our expression-oriented analysis track a similar number of points-to sets as their variable-oriented analysis? These questions could be asked quite directly by running both analyses within a single deductive database instance and then querying the database directly for the answers.

4. REFERENCES

[1] The Annotation Processing API. https://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/package-summary.html.

[2] The Compiler Tree API. https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/.

[3] T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net.

[4] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[5] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[6] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices*, 48(6):423–434, 2013.

[7] 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 *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM.

[8] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: A retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[9] O. Lhoták and L. Hendren. Scaling java points-to analysis using spark. In *International Conference on Compiler Construction*, pages 153–169. Springer, 2003.

[10] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for java. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 73–79. ACM, 2001.

[11] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *The journal of logic programming*, 23(2):125–149, 1995.

[12] Y. Smaragdakis and M. Bravenboer. *Using Datalog for Fast and Easy Program Analysis*, pages 245–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[13] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.

[14] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.