Interactive, Scalable, Declarative Program Analysis: From Prototype to Implementation

William C. Benton
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
willb@cs.wisc.edu

Charles N. Fischer
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
fischer@cs.wisc.edu

Abstract

Static analyses provide the semantic foundation for tools ranging from optimizing compilers to refactoring browsers and advanced debuggers. Unfortunately, developing new analysis specifications and implementations is often difficult and error-prone. Since analysis specifications are generally written in a declarative style, logic programming presents an attractive model for producing executable specifications of analyses. However, prior work on using logic programming for program analysis has focused exclusively on solving constraints derived from program texts by an external preprocessor. In this paper, we present DIMPLE, an analysis framework for Java bytecodes implemented in the Yap Prolog system [8]. DIMPLE provides both a representation of Java bytecodes in a database of relations and a declarative domain-specific language for specifying new analyses as queries over this database. DIMPLE thus enables researchers to use logic programming for every step of the analysis development process, from specification to prototype to implementation. We demonstrate that our approach facilitates rapid prototyping of new program analyses and produces executable analysis implementations that are speed-competitive with specialized analysis toolkits.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis; D.1.6 [*Programming Techniques*]: Logic Programming; D.2.m [*Software Engineering*]: Miscellaneous—Rapid prototyping

General Terms Performance, Design, Languages

Keywords Program analysis, Java, Bytecodes, Prototyping, Logic programming, Tabled Prolog

1. Introduction

Program analyses provide static answers to questions about the dynamic behavior of a program. A researcher who wishes to develop a new program analysis must engage in two separate activities: devising a specification for the analysis and engineering a suitably efficient implementation. Developing a correct specification can be difficult and error-prone, as a correct analysis specification must properly treat all of the features and subtle corner cases in a given language. However, even given a correct specification, the effort necessary to produce an efficient implementation may be substantial or prohibitive. The implementer's task is difficult due to the disconnect between a formal analysis specification and an executable implementation.

Analysis designers typically specify analyses in a declarative style: whether as a system of constraints [2, 12, 33, 21, 27], as type inference rules [28, 38, 10, 1], or as a fixed-point of a system of dataflow equations. Many analyses also admit natural specifications as reachability queries on context-free languages or as logic programs [29, 11, 36, 40, 42, 22]. General solvers for these sorts of problems have benefited from advances in logic programming language implementation; several are quite efficient. However, most existing solvers are far from ideal for rapid prototyping of new analyses.

Consider an idealized version of the process a researcher undertakes when developing a new program analysis. The designer:

1. Decides which abstract domains are interesting, and how best to model program properties as mathematical structures;
2. decides how individual kinds of program statements contribute to the analysis results;
3. writes a preprocessor to extract analysis relations from program source code; and finally
4. develops some sort of solver (or uses a preexisting specialized solver) for queries on the generated relations.

Since the analysis itself is likely to be specified declaratively, it would be ideal to use an analysis framework that could execute such a specification with little or no modification. Unfortunately, almost all existing analysis frameworks that support declarative specifications only aid users with the final step of this process: namely, providing a solver for analysis rules in a particular formalism. Many require extensive preprocessing or time-consuming automated optimizations before running an analysis, thus discouraging an interactive development style, casual experimentation, and rapid prototyping of new analyses.

We have developed DIMPLE, a fully-featured declarative analysis framework for Java. It is "fully-featured" in that it provides functionality for every step of the analysis design and development process. It is "declarative" in two ways: first, user-specified rules are written in a declarative language; also notably, DIMPLE itself is primarily implemented in the Yap Prolog system [8]. DIMPLE consists of a typed intermediate representation of Java bytecodes and a domain-specific language that allows users to specify program analysis implementations in an essentially declarative fashion, much as they might write an analysis specification. Unlike most

PPDP'07 July 14–16, 2007, Wrocław, Poland.
Copyright © 2007 ACM 978-1-59593-769-8/07/0007. . . $5.00

program analysis frameworks, DIMPLE represents program texts, derived relations, analysis rules, and analysis results uniformly – all as relations in a Prolog database. DIMPLE thus enables analysis designers to develop declarative specifications for every stage of analysis development and evaluation: deciding which domains are under consideration and which program statements are relevant, extracting relations from relevant program statements, and answering queries based on a system of relations and analysis rules. Experimental results confirm that, for a large class of analysis problems, DIMPLE is speed-competitive with best-of-breed program analysis frameworks and constraint solvers.

DIMPLE is essentially different from other analysis frameworks in that it offers two important capabilities:

1. DIMPLE provides a total, round-trip solution to analysis design and evaluation. That is, an analysis designer may use DIMPLE to produce a declarative specification for every component of an analysis. DIMPLE allows the user to define procedures for any necessary preprocessing of the program text in Prolog. Given declarative, user-supplied specifications, DIMPLE then automatically generates a statement processing routine that generates relations from the program text and the rules that govern analysis results. The underlying tabled Prolog system then acts as a solver for the analysis rules and relations.
2. DIMPLE provides a relational, declarative model for specifying program analyses. An analysis designer may thus execute an analysis specification that is very similar to the sort of specification that might appear in a technical paper.

There exist several excellent solvers that have been used to great effect for analysis problems. However, these tools require that the analysis designer use some external tool to develop a preprocessor that translates from program text to relations. Some analysis frameworks, such as those available within research compilers, enable users to develop preprocessors as well as solvers within the same tool, but DIMPLE is the first such framework to enable round-trip analysis development in a declarative style.

In the DIMPLE framework, program texts (in an intermediate representation of bytecode instructions), analysis rules, and analysis results are stored in a database of Prolog relations. Since analysis results are in the same format as the program to be analyzed, it is possible for analysis designers to reuse the results of costly analyses as program annotations – and, in fact, to store these annotations directly in the same database as the program text instead of recalculating the analysis results later.

1.1 Overview of contributions

This paper presents the DIMPLE program analysis framework, which features:

1. A system for encoding an intermediate representation of Java bytecodes as a database of facts.
2. A declarative framework for program analysis, which enables analysis users to prototype, develop, and debug new program analysis specifications by generating analysis implementations directly from specifications.
3. A mechanism for debugging program analysis specifications that combines the efficient execution of tabled evaluation with the tracing capacity of conventional Prolog evaluation.

We also present an evaluation of the suitability of the DIMPLE framework for developing program analyses. We discuss DIMPLE implementations of two analyses; we demonstrate both the conciseness of the specifications and the high performance of DIMPLE-specified analyses on realistic, substantial Java programs.

2. Overview of DIMPLE

DIMPLE comprises two parts. The front-end translates from Java bytecodes to the DIMPLE IR: a set of Prolog relations that fully describe the input application and library classes. The back-end provides a framework of rules and a domain-specific analysis language to implement program analyses as declarative queries on a database of relations.

We have implemented the DIMPLE front-end as a whole-program transformation that extends the Soot compiler framework. Soot [41] converts from stack-based bytecode to a typed three-address representation called Jimple and generates a conservative method call graph. Our transformation then translates from Jimple's abstract syntax to the concrete syntax of a database of DIMPLE IR relations. As we have stated, methods are modeled as sets of statements and all intraprocedural control flow is modeled by an explicit control-flow graph. It is therefore possible to use DIMPLE to implement either flow-sensitive or flow-insensitive analyses; the analysis developer need merely decide whether or not to ignore control flow when declaring analysis rules.

We have implemented the DIMPLE back-end in the Yap Prolog system [8]. The back-end consists of a domain-specific language for specifying analyses, a code generator that translates from the domain-specific language to instrumented Prolog rules, and relations and rules that describe various aspects of Java's semantics and type system. The DIMPLE back-end makes use of the Yap system's support for *tabled execution*. Tabled Prolog [7, 32] supports a *table* annotation on certain predicates. The results of a tabled predicate are memoized; thus, each is evaluated at most once for each tuple of arguments. Furthermore, a tabled Prolog system can automatically find a finite fixed point of recursive and mutually recursive tabled predicates. This property enables analysis designers to specify many relations that naturally admit a left-recursive definition in a straightforward and direct way, without considering nonterminating evaluation.

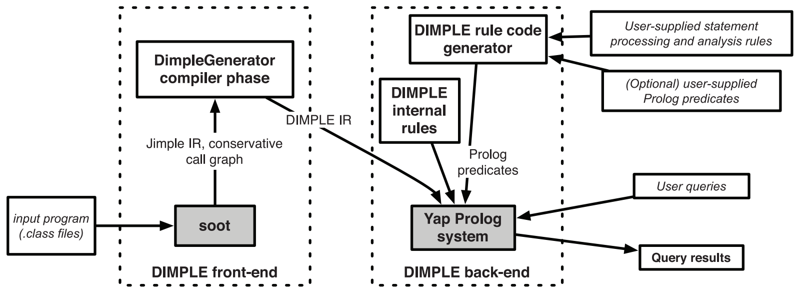
3. Using DIMPLE

We shall now present a macro-level view of DIMPLE and discuss a typical user's interactions with the DIMPLE system while developing a new analysis. Figure 1 provides an overview of DIMPLE's architecture; we shall refer to each component in the following overview.

A DIMPLE user is likely to begin by creating databases of DIMPLE IR relations for several interesting programs, in order to evaluate a new analysis on representative inputs. This task requires the DimpleGenerator application, which processes Java class files. DimpleGenerator extends the Soot compiler framework and produces a DIMPLE IR database with relations corresponding to every statement in a given application and in each of its libraries. The DIMPLE IR also includes relations describing the application and library class hierarchies and relations describing a conservative approximation of the method call graph. (We discuss the DIMPLE IR further and provide some example relations in Section 4.)

The user may then load the IR database into the Prolog system and begin asking queries about the program. This sort of interactive experimentation can often be a productive prelude to developing the actual statement processing and analysis rules. By exploring the sorts of statements and relations that may be relevant to a particular analysis in advance of specifying the analysis, the user may be able to note subtle details that might have otherwise been overlooked. After some experimentation, the user will begin to use the DIMPLE analysis language to specify a statement processing routine and the analysis. (We discuss the DIMPLE analysis language in Section 5.)

Most analyses will only need a subset of the relations in the DIMPLE IR database. All analyses will benefit from simplifying



DIMPLE front-end
DimpleGenerator compiler phase
Jimple IR, conservative call graph
soot
input program (.class files)
DIMPLE IR
DIMPLE back-end
DIMPLE rule code generator
User-supplied statement processing and analysis rules
(Optional) user-supplied Prolog predicates
DIMPLE internal rules
Prolog predicates
Yap Prolog system
User queries
Query results

Figure 1. High-level overview of DIMPLE's architecture. Shaded items indicate infrastructure developed by other groups; italics indicate user-supplied inputs

the program relations by preprocessing the input program and deriving analysis-specific relations from the DIMPLE IR. (At the very least, doing so will result in a clearer presentation.) Consider the following DIMPLE IR relation, which corresponds to the Java statement `r12 = r14`, where `r12` is a `long` local in method `foo` and `r14` is an `int` local in method `foo`:

```
stmt(pc1483, assignStmt(local(r12, method('foo'), primtype(long)),
                        local(r14, method('foo'), primtype(int )))).
```

It may be more convenient for the analysis designer to derive a relation that omits some details that are not interesting for a particular analysis. Perhaps the analysis designer is interested in knowing that an assignment occurred, but some details, like the program counter, method name, and variable types, are not relevant for a particular analysis problem. In that case, the designer could preprocess the above statement to a much simpler relation, like this one:

```
assign(r12, r14).
```

Of course, the derived relations can be more involved than simple projections of structures into simpler structures. The *statement processing routine* decides which statements are interesting for a particular analysis, and it may use arbitrarily complex criteria to do so. To give two realistic examples, particular derived relations may select only those statements that involve assignments to locals of reference types, or only those statements that might throw unchecked exceptions. A DIMPLE user will specify a statement processing routine in terms of declarative rules in the DIMPLE analysis language; these rules may use user-defined predicates on DIMPLE IR statements, DIMPLE's internal rules (which include facts and relations dealing with Java's type system and semantics), or any Prolog code. By combining arbitrarily complex selection and projection rules, analysis users can develop extremely powerful preprocessors in a declarative style. (We discuss the part of the DIMPLE analysis language that defines statement processing routines in Section 5.1).

Once the analysis designer has declared statement processing rules, DIMPLE can preprocess the input program. DIMPLE affords several options here: for prototyping, when a user may be interested in having the entire program available, the statement processing routine will simply assert derived relations into memory. If the user is interested in developing a production version of a completed analysis (or simply in using DIMPLE as an advanced preprocessor for another solver), it is possible to have the statement processing routine write derived relations to a file, discarding the IR database so that an analysis will operate strictly on a database of derived relations.

Keeping the IR database for entire program in memory incurs a cost in heap usage, but has two key benefits: if one is prototyping an analysis, one may decide that new or different derived relations might be useful. Since the whole program is available, one may add these effortlessly. In addition, having access to the whole program offers an advantage when debugging analyses: when the whole program is available, DIMPLE allows users to see precisely which IR statements contributed to each derived relation.

The analysis designer may then prototype analysis rules interactively, by issuing queries to the underlying tabled Prolog system. The user may then commence developing analysis rules by writing them in DIMPLE's rule definition language. (See Section 5.2 for more information on the rule definition language.) DIMPLE's code generator takes statements in the rule definition language and converts them to tabled Prolog rules. The code generator also generates support code to aid debugging analysis rules. The analysis designer will not invoke this support code directly; rather, DIMPLE uses it to present derivations of why particular analysis results hold. (See Section 5.3 for more information on this capability.)

By default, generated analysis rules are asserted into the Prolog database. The code generator can also write the generated rules and compiler directives to a file, which can be compiled and loaded. Each of these is appropriate for different situations, and DIMPLE gives users the opportunity to decide which to use at a given stage in the analysis development process. The former leads to slightly slower execution times but affords greater flexibility for prototyping, since asserted rules may be retracted or abolished. The latter option is most appropriate for production analyses, since it offers the fastest possible execution, but does not offer the opportunity to interactively change the analysis.

We shall now discuss the individual DIMPLE components in greater detail.

4. The DIMPLE IR

We have developed a typed intermediate representation (IR) for Java bytecodes that has an essentially declarative flavor: programs are represented as sets of relations, analyses are specified in a domain-specific language that adds rules and relations to the database, and analyses are executed by satisfying queries on generated rules and relations. Java classes are represented in terms of subtyping relationships, sets of method declarations and sets of field declarations. Methods are represented as sets of statements. All intraprocedural control flow is explicit and is modeled by a control-flow graph for each method. The abstract syntax of our representation is loosely based on the Jimple IR for Java bytecodes, which is a component of the Soot compiler framework. (We call our representation DIMPLE to reflect its *declarative* character and as an homage to prior intermediate representations like Simple [17] and Jimple.)

DIMPLE, like Jimple, is based on three-address code or register quadruples. Each statement consists of variables corresponding to operands, an operation, and a variable in which to place the result. As a consequence, each statement has a simple, "flat" form: all operands are either local variables, immediate values, or addresses, not complex expressions. Object fields and array elements are treated in a "load/store" fashion: they are loaded from memory, operated on as locals, and stored back to memory. By contrast, the Java virtual machine executes bytecode instructions that operate on a stack. Since the stack is untyped, Java provides separate bytecode instructions for each potential operand type. [24] (As an example, there are eight different bytecode instructions to load values from an array, depending on the element type: one for object references and one for each of Java's primitive types.) There are also many "special case" bytecode instructions that incorporate immediate values in their opcodes. Of course, the wide variety of bytecode instructions – and the very nature of stack-based evaluation – greatly complicates analysis of unmodified bytecode. By basing DIMPLE on an existing intermediate representation of bytecode that features fewer kinds of operations than does bytecode, it is possible to express analyses in terms of a simpler set of rules than it would if DIMPLE were to operate on bytecode directly.

We shall now examine some representative DIMPLE structures in order to convey the declarative flavor of DIMPLE:

local(*L*, *M*, *T*). Indicates that a local variable with name *L* and of type *T* exists in method *M*. The name, *L*, is an atom and is unique throughout the entire program and its library code. The method, *M*, is represented as a structure with one constituent: an atom corresponding to the method's Java bytecode signature.[1] There is an example local/3 relation in Figure 2.

stmt(*PC*, *Stmt*). Indicates a statement of kind *Stmt* at globally-unique program counter *PC*. *Stmt* is a structure corresponding to one of 14 different IR statement types. The set of IR statements include assignments, heap loads and stores, object monitor entrances and exits, jumps, method invocations, returns, and exception generation.

assignStmt(*LHS*, *RHS*) This sort of structure must only appear as the *Stmt* constituent of a stmt/2 structure. It indicates that the value described by *RHS* is to be stored in the location described by *LHS*. Either, but not both, of *LHS* and *RHS* *may* be a structure describing a reference to a field of a heap object in a valid DIMPLE program; at least one must be a local/3 structure. The examples in Figure 2 show statements copying a value from local *Lb* to local *La*, loading a value from the *F* field of the object referred to by *Lb*, and writing a value to the *F* field of the object referred to by *La*.

ifStmt(*La*, *PC*) As with assignStmt/2, an ifStmt/2 must only appear as the *Stmt* constituent of a stmt/2 structure. ifStmt/2 describes a conditional branch: if the boolean value stored in *La* is true, control will transfer to statement at the program counter specified by *PC*.

5. The DIMPLE Analysis Language

The DIMPLE analysis language consists of two parts: a *statement processing language* and a *rule definition language*. The statement processing language describes statement processing routines, or subprograms that decide what derived relations should hold given the presence of certain program statements. The rule definition language facilitates descriptions of analysis rules in terms of these derived relations. A DIMPLE user, then, will go through several steps when developing a new analysis:

[1] A bytecode signature is a "mangled" name that uniquely identifies a method. It consists of the name of the declaring class, the method name, and encodings of the parameter and return types. Please see the Java Virtual Machine Specification [24] for more information.

1. Developing Prolog procedures to be used before the statement processing routine (as an additional preprocessor on the program text), or during the statement processing routine.
2. Specifying statement processing rules, indicating that certain derived relations should be added to the Prolog database if their free variables can be instantiated in some condition. DIMPLE will then automatically generate a statement processing routine based on these rules.
3. Specifying analysis rules. These rules define analysis result relations as functions of derived relations. DIMPLE automatically translates these into tabled Prolog rules; it also generates special "tracing versions" of predicates to assist in debugging.

DIMPLE exploits the tabled execution model in order to allow users to develop analyses in an essentially declarative style. This essentially declarative style is very similar to the formal definitions of analysis algorithms used in the literature. In this way, DIMPLE provides expressive power similar to other declarative database approaches for program analysis, such as the Datalog-based `bddbddb` system [42]. In addition to the capabilities afforded by tabled or Datalog-style execution, DIMPLE exposes additional functionality. Because DIMPLE is implemented in tabled Prolog, the efficiency and clarity of tabled execution and the full power of Prolog are both available to users. Prolog's extralogical features, such as the cut, assignment, and assertion and retraction of relations and rules, are available to statement processing routines and custom preprocessing code.

5.1 The statement processing language

Once an analysis designer has decided how to map concrete program statements to elements of abstract domains, it is necessary to develop a routine to extract program statements of interest. DIMPLE's *statement processing language* automatically generates such a statement processing routine given a set of rules describing when particular statements are interesting. These rules may simply refer to DIMPLE IR statements: for example, asserting that a relation holds between two local variables when an assignment between them occurs in the program text. DIMPLE statement processing rules may also refer to types and methods: for example, we may only be interested in variables of certain types, or statements that execute in particular methods. In fact, DIMPLE affords analysis designers the opportunity to develop arbitrarily complex statement processing rules, since the statement processing routine may use unrestricted Prolog code and has access to a full representation of the program under analysis, including a class hierarchy and conservative call graph approximation.

Statement processing rules take the form Head <−− Body. *Head* must be a functor with at least one variable constituent; this variable must appear in *Body* as well. *Body* may contain relations about statements (e.g., stmt/2) as well as relations about the types of expressions within statements, relations about the class hierarchy, relations about a conservative method call graph, and user-defined relations. The default DIMPLE statement processing routine will exhaustively identify where every *Body* relation holds, adding each unique corresponding *Head* to the database exactly once. The statement processing rules are executed in order, so relations added to the database by the head of some rule *r* are available to the body of every statement processing rule executed after *r*. The statement processing routine also keeps track of why each derived relation holds. As a result, an analysis designer can use a simple query to determine which source program statements and conditions led to a particular relation.

The case studies in sections 7 and 8 include representative examples of the statement processing language.

Example relation	Description
local (r12, method('<java.lang.Object:␣<init>()V>'), reftype('java.lang.Object')).	Declares a local `r12` of type `Object` in the constructor for `Object`.
stmt(pc4201, assignStmt(La, Lb)).	Represents an assignment between two locals; i.e. `La = Lb;`
stmt(pc4202, assignStmt(La, instanceFieldRef(Lb, F))).	Represents a load from an instance field; i.e. `La = Lb.F;`
stmt(pc4203, assignStmt(instanceFieldRef(La, F), Lb)).	Represents a store to an instance field; i.e. `La.F = Lb;`

Figure 2. Example DIMPLE IR relations

5.2 The rule definition language

As we have seen, DIMPLE's statement processing routine interprets the user's specifications. As it runs, the statement processing routine populates the database with relations describing the state of the program as is relevant to the current analysis. Statement processing rules act like filters: the statement processing routine finds all solutions to each, in order, but each rule is only solved for once.

DIMPLE represents analysis rules in a similar way to preprocessor rules (analysis rules use the <== operator instead of <−−), but the meaning of analysis rules is very different. The DIMPLE code generator produces tabled Prolog rules from analysis rules. Therefore, analysis rules are like procedures that may call each other. (Statement processing rules, on the other hand, may call other Prolog procedures, DIMPLE internal rules, or relations generated by statement processing rules that have already executed.)

Many specialized analyses and transformations depend on more general analyses. For example, analyses to determine whether or not an object is stack-allocable typically either incorporate or rely upon a points-to analysis. Other examples include constant propagation, partial evaluation, and some schemes for type inference – all of which depend on a reaching definitions analysis. Because DIMPLE analyses and programs are stored in the same format, it is extremely straightforward to annotate programs with analysis results and serialize analysis results for use in later sessions. A user might develop an analysis, query for an exhaustive solution to the analysis rules, and save the relations derived from the analysis results to the database for use by the statement processing rules of more specialized analyses.

The case studies in sections 7 and 8 include representative examples of the rule definition language.

5.3 Code generation and tracing

Tabled evaluation provides several benefits. Many sorts of procedures enjoy improved execution efficiency when evaluated with tabling. Furthermore, tabled evaluation admits natural, declarative definitions of left-recursive procedures. However, these benefits come at two costs: a memory cost and a development cost.

The memory cost is rather straightforward to treat. Tables for procedures may take up a great deal of memory, and the balance between space and execution time may not justify tabling certain predicates. As a result, DIMPLE allows users to designate certain analysis rules as untabled; the code generator will simply generate standard Prolog rules for these.

The development cost presents a rather trickier problem. Typically, it is not possible to trace a tabled procedure, since it will be evaluated at most once for a given tuple of arguments. For many applications, the tradeoff between ease of debugging and execution time would be acceptable. However, we are interested in enabling researchers to prototype new analyses rapidly. As part of this process, a researcher may be interested in determining precisely *why* a particular spurious analysis result holds.

The DIMPLE code generator solves this problem by generating two versions of rules: *standard versions*, which are tabled and execute normally, and *tracing versions*, which are not tabled and can execute in a metainterpreter that produces a rule trace. Given a standard rule R, we generate the tracing version R' as follows:

1. Give R' a new name that does not belong to any extant procedures; create a relation in the database indicating that the name for R' is the tracing version of the rule R.
2. Make a copy of R's body; this will become the body for R'. Map every call to a non-recursive procedure in this new body with a call to the tracing version of that procedure. Note that we do not generate tracing versions of relations generated by the statement processing routine. The code generator only replaces calls to non-recursive procedures in order to ensure that the metainterpreter will terminate when asked to explain any terminating relation.

Given these tracing versions, a user can drill-down to fully explain any individual analysis result. As an example, consider a user who wishes to explain why the relation v_pt holds with arguments `r12` and `pc45`. When the user asks DIMPLE to explain why v_pt(r12,pc45) holds, the metainterpreter will consult the tracing versions of the various clauses for v_pt. Assume that, for this example, the relevant clause of v_pt is recursive and depends on an assign/2 relation generated by the statement processing routine: namely,

```
v_pt(Ref,Obj) <== assign(Ref,Int), v_pt( Int ,Obj).
```

The generated tracing version of this clause would be very similar, except that it would not be declared as a tabled procedure. If there were calls to non-recursive analysis rules inside the body of this clause, then they would be replaced with calls to tracing versions. However, the call to assign/2 would not change, since assign/2 is a fact generated by the statement processing routine; the recursive call to v_pt/2 would not change, since the code generator does not replace calls to recursive rules. Therefore, the tracing version of this clause, in standard Prolog syntax, would consist of:

```
dimpleTRACE_v_pt(Ref,Obj) :− assign(Ref,Int), v_pt(Int,Obj).
```

In this example, the metainterpreter will see that assign(Ref,Int) holds with Ref = `r12` and Int = `r14`; it will then consult the table for v_pt/2 and see that v_pt(Int ,Obj) holds with Int = `r14` and Obj = `pc45`. Therefore, the user is able to see which rule led to the (perhaps spurious) result; if one wishes to trace further, e.g., to determine why v_pt(r14,pc45) holds, one may iteratively query the metainterpreter for more details.

6. Experimental evaluation

We used DIMPLE to implement two analyses: a subset-based points-to analysis (Section 7) and an effects inference analysis (Section 8). We evaluated the performance of these implementations on thirteen benchmarks from the SPECjvm98[2] [37] and DaCapo [4] suites. (We used version 2006-10-MR2 of the DaCapo suite. Note also that we

[2] SPEC and SPECjvm are registered trademarks of the Standard Performance Evaluation Corporation.

are using the compiled versions of these applications as inputs to our analysis and are *not* generating competitive benchmark results by actually running the applications.) These are realistic and substantial Java applications. The names and characteristics of the benchmarks we used are detailed in Figure 3; we show the number of classes in each application, the number of classes transitively reachable from its `main` method (that is, including library classes), and the total IR statements for the class and all of its dependencies in the Java standard library.

We present performance results for "production" versions of analyses: that is, we have used the statement processing routine to preprocess the program database for each benchmark to a new file so that only derived relations of interest are in memory. (If a DIMPLE user were prototyping an analysis, it might be desirable to leave the entire statement database intact and accessible at the expense of using additional memory; this capability is unnecessary for production analyses.) We have also instructed the code generator to write new rules to a file (which is then compiled statically), rather than assert these rules dynamically; this results in faster executions but means that users cannot reliably retract and alter rules.

All performance numbers are from DIMPLE running under Yap 5.1.1 on a Pentium 4-based Linux machine running at 3 ghz with 2 gb of RAM. While it is possible to configure Yap to evaluate tabled queries with multiple threads, we did not do so.

benchmark	app classes	total classes	total IR statements	suite
`antlr`	228	1594	416174	dacapo
`bloat`	360	1757	466886	dacapo
`chart`	1246	4237	1189539	dacapo
`db`	14	1448	361797	specjvm
`eclipse`	413	1819	455273	dacapo
`hsqldb`	577	1483	370384	dacapo
`jack`	67	1501	379645	specjvm
`javac`	182	1616	404305	specjvm
`jess`	159	3844	948596	specjvm
`luindex`	349	1618	396145	dacapo
`pmd`	2779	2249	555476	dacapo
`raytrace`	35	1469	366874	specjvm
`xalan`	1830	1487	367150	dacapo

Figure 3. Benchmark workloads under which we evaluated DIMPLE implementations of analyses

```
/* newly−allocated objects */
pt(La,Id) <−−
    stmt(Id, assignStmt(local(La,Ma,Ta)), newExpr(Loc,Type)).

/* assignments between locals */
assign(La,Lb) <−−
        stmt(_, assignStmt(local(La,Ma,Ta), local(Lb,Mb,Tb))),
    reference_type(Ta).

/* static fields (i.e., globals) */
s_load(La,Field) <−−
    stmt(Id, assignStmt(local(La,Ma,Ta), staticFieldRef(Field))),
    reference_type(Ta).
s_store(Field, La) <−−
    stmt(Id, assignStmt(staticFieldRef(Field), local(La,Ma,Ta))),
    reference_type(Ta).

/* formal and actual parameters */
formal(La, Index, Method) <−−
    stmt(Id, identityStmt(local(La,Ma,Ta), parameterRef(Method, Index))),
    reference_type(Ta).
formal(La, this, Method) <−−
    stmt(Id, identityStmt(local(La,Ma,Ta), thisRef(Method, Type))),
    reference_type(Ta).
actual(La, Index, Method) <−−
    unitActual(Callsite, Index, local(La, Ma, Ta)),
    unitMaySelect(Callsite, Method), reference_type(Ta).

/* return values */
ret_caller(La, Method) <−−
    stmt(Id, assignStmt(local(La,Ma,Ta), X)), invocation(X,_),
    unitMaySelect(Id, Method), reference_type(Ta).
ret_callee(La, Method) <−− stmt(Id, returnStmt(local(La,Ma,Ta))),
    reference_type(Ta), containsStmt(Method, Id).

/* instance fields */

% La = Lb.F
load(La,F,Lb) <−−
    stmt(Id, assignStmt(local(La,Ma,Ta),
                        instanceFieldRef(local(Lb,Mb,Tb), F))),
    reference_type(Ta).

% La.F = Lb
store(La,F,Lb) <−−
    stmt(Id, assignStmt(instanceFieldRef(local(La,Ma,Ta), F),
                        local(Lb,Mb,Tb))),
    reference_type(Tb).
```

Figure 4. Andersen's analysis: select statement processing rules (rules treating arrays and exceptions are omitted)

7. Case study: Andersen's analysis

Consider a family of fundamental program analyses: *points-to* analyses, which provide answers to the question: "Which (abstract) memory locations *might* this reference-valued variable refer to at runtime?" Andersen's analysis [2] is a points-to analysis; it provides a reasonable tradeoff between precision and worst-case execution time for many applications. Andersen's analysis also enjoys an intuitive, succinct specification in terms of constraints on a points-to graph: one could easily describe it to an implementer by writing the most important analysis rules on a cocktail napkin. However, producing a good imperative program that implements Andersen's analysis is a rather difficult task – early implementations were quite slow and did not scale. The first truly scalable implementation of an Andersen-style analysis, due to Heintze and Tardieu [16], was reported *seven years* after Andersen's dissertation was published.

In this section, we present a DIMPLE implementation of Andersen's points-to analysis for Java. (Andersen's analysis was initially designed for C; several groups [3, 36, 42, 33] have refined it to take advantage of Java's type system and lack of unrestricted pointers; these various extensions are fundamentally similar.) We have adapted the analysis rules from the `bddbddb` implementation of a context-insensitive subset-based points-to analysis, due to Whaley and Lam [42]. We developed the statement processing rules in order to translate from DIMPLE IR statements to the relations used by Whaley and Lam's specification.

Andersen's analysis is a *flow-* and *context-insensitive*, *inclusion-based* points-to analysis. This means that subprograms are treated as sets of statements (rather than as graphs of statements); that analysis results for a particular method are merged among all call sites of that method; and that a variable may refer to a subset of the locations that another variable refers to. (Readers who are interested in more information on categories of points-to analyses should refer to Hind [19, 18] for a thorough overview of the field.)

We defined statement processing rules to extract relevant relations from the IR database. (Only a subset of all the DIMPLE IR relations affect points-to information.) Figure 4 shows how we encoded these in preprocessing the program text:

1. Object allocation statements create a new abstract object, which we name by the program counter of the allocation site. Object allocation statements imply an *immediate points-to* (pt/2) relation between the local variable receiving the object reference and the newly-allocated object.
2. Assignment statements (including heap and array loads and stores) indicate that the variable or location on the left-hand side

```
% case 1:
v_pt(Ref,Id) <== pt(Ref,Id).

% case 2:
v_pt(Ref,Id) <== assign(Ref, RefI), v_pt(RefI, Id).

% case 3:
v_pt(Ref, Id) <== s_load(Ref, F), s_store(F, RefI), v_pt(RefI, Id).

% case 4:
v_pt(Ref, Id) <== formal(Ref, I, M), actual(RefI, I, M), v_pt(RefI, Id).

% case 5:
h_pt(Obj1, F, Obj2) <==
    store(Ref1, F, Ref2), v_pt(Ref1, Obj1), v_pt(Ref2, Obj2).

% case 6:
v_pt(Ref, Id) <==
    load(Ref1, F, Ref), v_pt(Ref1, Id1), h_pt(Id1, F, Id).

% case 7:
v_pt(Ref, Id) <==
    ret_caller (Ref, Method), ret_callee(RefI, Method), v_pt(RefI, Id).
```

Figure 5. Andersen's analysis: complete analysis rules

of the assignment may refer to a superset of the locations that the right-hand side of the assignment may refer to. Standard assignments between locals imply an assign/2 relation; loads and stores of instance fields imply load/3 and store/3 relations. (We treat array accesses – not shown in the figure – as field accesses to a distinguished field name and static fields as global variables.)

3. Method invocations indicate that the local variables corresponding to formal parameters (indicated by formal/3 relations) must refer to a superset of all abstract objects referred to by variables passed as actual parameters (indicated by actual/3 relations). (Recall that we use a precomputed conservative approximation of the dynamic call graph; the unitMaySelect/2 relation indicates that a particular program counter may invoke a particular method.)

Let us now consider the actual analysis rules. Andersen's analysis is perhaps simpler to understand if we consider it as a graph problem. An exhaustive solution to the points-to question, as given by the set of all tuples under the v_pt/2 relation, is simply the transitive closure of the assignment relation from abstract heap objects *Id* to reference variables *Ref*. With this in mind, we can consider the rules, as shown in Figure 5:

1. v_pt(Ref,Id) holds when *Ref* immediately points to *Id*.
2. v_pt(Ref,Id) holds when *Ref* has received a reference from some other variable *RefI*, and v_pt(RefI,Id) holds.
3. v_pt(Ref,Id) holds when *Ref* may have been loaded from a static field *F*, *F* may have had some reference *RefI* stored to it, and v_pt(RefI,Id) holds.
4. v_pt(Ref,Id) holds when *Ref* is formal parameter *I* of some method *M*, *RefI* is actual parameter *I* of a call site that may invoke *M*, and v_pt(RefI,Id) holds.
5. h_pt(Obj1,F,Obj2) holds when some field *F* of an abstract heap object *Obj1* may refer to the abstract heap object *Obj2*.
6. v_pt(Ref,Id) holds when *Ref* may have been loaded from a field that has had *Id* stored to it.
7. v_pt(Ref,Id) holds when *Ref* may receive its value from a method that returns some *RefI* such that v_pt(RefI,Id) holds.

		unsound exceptions		*sound exceptions*	
benchmark	input relations	points-to pairs ($\times 10^6$)	CPU time (s.)	points-to pairs ($\times 10^6$)	CPU time (s.)
antlr	347290	12.29	8.15	12.31	8.13
bloat	476635	19.39	13.66	19.41	13.54
chart	586337	4.72	3.92	4.74	3.88
db	267418	5.39	3.25	5.41	3.38
eclipse	401474	12.04	12.08	12.06	11.91
hsqldb	304451	6.62	4.81	6.64	4.56
jack	278813	5.93	3.94	5.95	3.81
javac	333841	6.77	4.65	6.79	4.29
jess	793032	32.91	27.23	32.94	25.34
luindex	322634	7.30	7.26	7.32	7.42
pmd	427279	10.65	7.00	10.67	7.01
raytrace	270804	5.38	3.52	5.39	3.50
xalan	302150	6.17	4.13	6.18	4.13

Figure 6. DIMPLE performance for points-to analysis

Figure 6 shows the performance for this analysis on each of our benchmarks: the number of relations returned by the statement processing routine, the number of pairs *Ref* × *Obj* for which v_pt(Ref,Obj) holds, and the CPU time to calculate exhaustive points-to results. We implemented two versions of the analysis: one treats exception objects in an unsound manner; the other features a sound (but imprecise) conservative treatment of exceptions. (The sound analysis conflates all exception objects into a single `bottom` object. It is actually faster than the unsound analysis even though it finds more aliasing. Since there are fewer abstract objects, some calculations on the points-to graph are simpler.) Note that DIMPLE calculated analysis results for each application in under thirty seconds; most results were available in under ten seconds.

8. Case study: effects inference

In this section, we present an *effects inference* analysis to aid program understanding. In so doing, we show an example of using DIMPLE to extend an existing analysis. We also demonstrate how the interactive nature of DIMPLE makes it easy to validate intuitions and apply these to improving an analysis. Before we introduce our particular analysis, we shall discuss the problem in more detail.

Standard type systems characterize the ranges of values that expressions may produce and that variables may assume. Effect systems [25] extend type systems to also characterize the computational effects (e.g., reads or writes to shared state) of expressions, statements, and methods. Thus, a type system might tell us that method `foo` takes an `int` parameter and returns a reference to `Object`, but an effect system would also tell us that method `foo` may write values in regions *X* and *Y* and read values from regions *X* and *Z*. (A region is simply a subset of the heap.) Just as a type system may use explicit type annotations (as in Java or C) or infer types for variables and expressions (as in ML or Haskell), effect systems may either require user annotations of effecting behavior or infer this information.

There are many applications of effects inference analysis. Several notable examples include finding expression scheduling constraints (as in Lucassen and Gifford's original work [25]); automatically providing annotations for a model checker or specification language [35]; and, most commonly, improving *region-based memory management* [39], in which object lifetimes are inferred at compile-time to enable a stack discipline for dynamic allocations, so that an entire region of objects may be deallocated at once. The analysis we present in this section is designed to aid program understanding and debugging, so it does not calculate region lifetimes; rather, it answers the question: *given some method X, which abstract locations may X read or write?* We shall use the `Point` class declared in Figure 7 as an example as we introduce our analysis. Before we do so, we shall review some relevant Java features.

```
public class Point {
  private float x, y;

  public Point(float x, float y) {
    this.x = x; this.y = y;
  }

  public void setX(float x) { this.x = x; }
  public void setY(float y) { this.y = y; }

  public float getX() { return this.x; }
  public float getY() { return this.y; }

  public void translate(float dx, float dy) {
    this.x = this.x + dx;
    this.y = this.y + dy;
  }
}
```

Figure 7. A simple `Point` class

8.1 Side effects in Java

Java enforces a strict divide between heap data and stack data. In particular, Java supports references instead of unrestricted pointers. References may only refer to heap locations (objects or arrays); it is not possible to create a reference to a stack value.

Parameters are passed by value in Java. Therefore, the only data that can be shared between methods – the sort of data that we are interested in inferring effects on – are heap objects. The only way to access or modify heap data (thus, shared state) is via an array element reference or an object field reference.

Because Java is a typed language, it is not possible to refer to an object of type *C* via a reference of type *D*, where *D* is not a supertype of *C*. Doing so will result in either a compile-time or run-time error. At the IR (or bytecode) level, all field accesses are to fully-qualified names, including a field name and the name of its declaring class. Therefore, we can examine the IR for the `Point.setX` method and determine that it may only modify one abstract location: the `Point.x` field of some object that is an instance of `Point` or some subclass thereof. In the language of effect systems, we could say that `Point.setX` writes into the region `Point.x`.

We could devise a very basic effects inference for Java by preprocessing a program database to reject everything except heap reads and writes, method invocations, and the conservative call graph. We would write our analysis to indicate that a heap read had a READ effect on the field it accessed, that a heap write had a WRITE effect on the field that it accessed, and that a method invocation statement had the union of all effecting statements from every method that might be dispatched from that call site. Such a basic analysis would use Java's type system to provide extremely coarse memory disambiguation. Its results would be sound but perhaps not very useful – there would be no way to distinguish between writes to some field *C*.*F* through references *X* and *Y*, even if it were possible to statically guarantee that *X* and *Y* did not refer to the same object.

8.2 A simple effects inference analysis

We can develop a more precise analysis by building on the points-to analysis from Section 7. Java's type system provides an extremely coarse form of memory disambiguation; we can use a points-to analysis to discriminate among locations with finer granularity. We shall consider abstract objects to be regions. Therefore, an inferred effect will consist of READ or WRITE, an abstract object (or the distinguished location `global`), and a field name. (We discuss but do not show the DIMPLE rules for the analysis we present in this section; many are substantially similar to the rules for the improved analysis that we present in Section 8.3.)

```
public void foo() {
      Point pt = new Point(0,0);
      pt.translate(1,1);
}
```

Figure 8. `foo` exposes a shortcoming of the simple analysis

This simple effects inference analysis proceeds as follows: First, it preprocesses the input database, extracting statements of interest. For the purposes of this analysis, we are exclusively interested in method invocations and reads and writes to heap locations. The analysis rules define the reads/3 and writes/3 relations to describe side effects, as follows:

reads(PC, Loc, F) holds when:

1. the statement at program counter *PC* reads from the instance field *F* from a reference *R* and v_pt(R,Loc) holds;[3] or
2. *Loc* is the atom `global`, and the statement at program counter *PC* reads the static field *F*; or
3. the statement at program counter *PC* may invoke a method that contains some statement *PC*′ for which reads(PC',␣Loc,␣F) holds.

writes(PC, Loc, F) holds in analogous situations as does reads/3, except that writes/3 holds when fields are written to instead of read from.

Andersen's analysis is a better discriminator between memory locations than is Java's type system, but it is limited by its insensitivity to contexts. Recall that Andersen's analysis merges analysis results for every context in which a method is invoked. (Here we construe "context" broadly to include call stack strings and receiver objects.) Specifically, consider the `foo` method from Figure 8.

A novice Java programmer would correctly note that the scope of `foo`'s side effects are confined to the object constructed in its first line. However, because the simple effects analysis is using Andersen's analysis to disambiguate between memory locations, it will not fare as well. Since `foo` only consists of two method invocations (a constructor and `translate`), its READ and WRITE sets are the unions of those sets for all of the methods it invokes. Because of context-insensitivity, these sets may be very large. The constructor has WRITE effects for the `x` and `y` fields of every object that may be referred to by `this` – that is, every `Point` object that has been created with that constructor! The `translate` method likewise has READ and WRITE effects for the `x` and `y` fields of every object that is pointed to by a reference that has had the `translate` method invoked on it.

8.3 Parameterizing on receiver objects

It seems plausible that the READ or WRITE sets of most instance methods will contain primarily fields of the receiver object (i.e., `this`). Were this the case, we could parameterize our analysis so that when we encountered an effect involving `this`, instead of keeping track of all possible objects that might be referenced by `this`, the READ and WRITE sets would merely record an effect to `this`. Given this sort of parameterized analysis, for example, the WRITE sets for `Point.setX` would include only the `x` field of `this`. When our analysis encountered an invocation like `pt.setX(y)`, it could then calculate which objects `pt` might refer to, and generate WRITE sets for the method invocation by instantiating `this` in the WRITE set for `setX` with each such object.

[3] The rules treating array references are similar to those treating instance fields; thus, we have omitted them here as we did in Section 7.

	simple analysis				*parameterized analysis*			
benchmark	input relations	reads/ call	writes/ call	CPU time (s.)	input relations	reads/ call	writes/ call	CPU time (s.)
`antlr`	596121	775.65	775.69	6.74	701468	312.49	123.27	4.65
`bloat`	818542	1362.86	1362.82	12.42	937325	338.01	101.65	7.41
`chart`	1103095	315.13	315.18	3.24	1381994	104.98	47.55	2.66
`db`	461756	571.73	571.78	2.90	548614	174.26	50.17	1.79
`eclipse`	681224	816.21	816.13	10.37	790904	266.79	87.47	8.00
`hsqldb`	524218	616.24	616.30	3.96	611857	205.88	68.16	2.62
`jack`	485683	566.38	566.43	3.21	576476	177.80	54.45	2.07
`javac`	557415	626.42	626.36	3.75	656425	178.55	48.70	2.23
`jess`	1363131	2415.52	2415.51	26.02	1592712	518.13	91.65	11.10
`luindex`	559118	644.11	644.18	7.00	653728	211.68	69.72	5.48
`pmd`	754279	730.86	730.90	6.08	889959	239.13	87.00	4.03
`raytrace`	467481	543.48	543.53	2.94	556009	167.16	47.78	1.84
`xalan`	519984	594.69	594.75	3.73	606898	196.82	63.71	2.46

Figure 9. Performance results for effect inference analyses

```
 isthis (L) <−− formal(L,this,M).
 isthis (L2) <−−
     formal(L,this ,M),
     assign(L2, L),
      all (Source,assign(L2, Source),[L]).

receiver(Id, L) <−−
     unitActual(Id, this, local(L, _, _)).

/∗ globals ∗/
read_global(Id, F) <−−
     stmt(Id, assignStmt(local(_,_,_), staticFieldRef(F))).

write_global(Id, F) <−−
     stmt(Id, assignStmt(staticFieldRef(F), local(_,_,_))).

/∗ instance fields ∗/
read_effect(Id, this, F) <−−
     stmt(Id, assignStmt(local(_,_,_), instanceFieldRef(local(L,_,_), F))),
      isthis (L).

 write_effect(Id, this, F) <−−
     stmt(Id, assignStmt(instanceFieldRef(local(L,_,_), F), local(_,_,_))),
      isthis (L).

read_effect(Id, L, F) <−−
     stmt(Id, assignStmt(local(_,_,_), instanceFieldRef(local(L,_,_), F))),
     \+ isthis (L).

 write_effect(Id, L, F) <−−
     stmt(Id, assignStmt(instanceFieldRef(local(L,_,_), F), local(_,_,_))),
     \+ isthis (L).

/∗ invocations ∗/
callgraph_edge(Id, Callee) <−− unitMaySelect(Id, method(Callee)).

/∗ methods containing effecting statements ∗/
in_method(Id, Method) <−− containsStmt(Method, Id),
     (read_effect(Id, Lr, Fr) ; write_effect(Id, Lw, Fw)).
in_method(Id, Method) <−− containsStmt(Method, Id),
     (read_global(Id, Fr) ; write_global(Id, Fw)).

method_contains(Method, Id) <−−
          in_method(Id, Method).
```

Figure 10. Select statement processing rules for parameterized effects inference

Of course, a plausible intuition doesn't provide sufficient justification for building and validating a new analysis, no matter how straightforward it is to do so in a given framework. Fortunately, DIMPLE makes it easy to collect empirical evidence for our intuitions: we may simply query the program database to see if certain conditions obtain.

In order to do so, we define an isthis /1 predicate that holds when a variable is a `this` reference – that is, when it refers exclusively to the receiver object of the current method. (The DIMPLE IR generator attempts to minimize the lifetimes of locals. Therefore, each method has one local variable that corresponds to `this` and possibly many variables that alias that local.) Then, we can use DIMPLE to query the program text and determine whether our intuition is correct; viz., most instance field accesses are to `this` instead of to some other object.

```
reads(Id, global, F) <==
    read_global(Id, F).

reads(Id, O, F) <==
    read_effect(Id, L, F), v_pt(L, O).

reads(Id, this, F) <==
    read_effect(Id, this, F).

method_reads(Method, Loc, F) <==
    method_contains(Method, Id), reads(Id, Loc, F).

reads(Id, Loc, F) <==
     callgraph_edge(Id, Callee),
     method_reads(Callee, this, F),
          receiver(Id, V),
          v_pt(V, Loc).

reads(Id, Loc, F) <==
     callgraph_edge(Id, Callee),
     method_reads(Callee, Loc, F),
          Loc \= this.
```

Figure 11. Select analysis rules for parameterized effects inference (rules treating WRITE effects are omitted)

For the benchmark programs we examined, this particular intuition happens to be correct. As a result, we know that it is probably worthwhile to develop a new analysis that generates parameterized summaries of effects information for methods; our analysis can then instantiate these summaries at call sites to indicate that effects to `this` may only impact objects referenced by the receiver at the call site. Figure 10 contains the statement processing rules for this improved analysis, and Figure 11 contains the analysis rules. (We omit rules related to WRITE effects in the analysis rules presented here, as they are very similar to the rules for READ effects.)

Figure 9 shows performance numbers for the simple and parameterized analyses on each benchmark. For each benchmark, we show the number of effects inferred (to abstract locations) for each method call site. Note that the parameterized analysis, while not as precise as a context-sensitive analysis, dramatically improves the precision of our analysis. (Note also that the effects analyses are more time-efficient than the points-to analyses – this is the case because they do not need to calculate *exhaustive* points-to information and may simply query on demand.)

9. Related work

Work related to the research we have reported here falls into two categories: declarative representations of programs and program analysis specifications, and techniques for efficiently solving declarative analysis queries. We discuss notable results in each of these areas that are most relevant to our work. We conclude by placing DIMPLE in context in the field and recapitulating its contributions.

9.1 Declarative frameworks for analysis

Dawson, Ramakrishnan, and Warren [9] argued that a general-purpose logic programming system (the XSB Prolog system) could be used to evaluate declarative formulations of program analysis problems. Their work demonstrated that the evaluation model of tabled Prolog is suitable for answering program analysis queries efficiently and completely. However, their evaluation was restricted to analyses of functional and logic programs consisting of tens to hundreds of lines of code and therefore did not evaluate the scalability of their techniques.

Heintze [14] demonstrated that many program properties could be faithfully approximated by sets – and, thus, that many program analyses could be formulated as a system of set constraints. Heintze and Jaffar [15] present an overview of work in this area; in Section 9.2, we describe some notable tools based on this abstraction.

Reps [29] demonstrated that many interprocedural dataflow analyses could be formulated as reachability problems on context-free languages. Since there is a well-understood correspondence between context-free languages and declarative programs that recognize them, this approach implies declarative formulations of a large class of analysis problems. Reps, Schwoon, Jha, and Melski [30] later generalized this approach to include analysis problems that could be specified as reachability problems on weighted pushdown systems.

Several researchers have investigated the class of problems that are expressible as CFL-reachability problems. Notably, Melski and Reps [26] gave a general algorithm for translating from any CFL-reachability problem to a set-constraint satisfaction problem. Later, Kodumal and Aiken [20] provided an algorithm for converting from *Dyck CFLs*, a subset of all context-free languages, to systems of set constraints; their algorithm is less general than the Melski-Reps reduction, but produces more efficient implementations in the special case of Dyck CFLs. Many static analysis problems can be expressed as Dyck CFLs; a representative example is given by Sridharan et al. [36], who formulated demand-driven points-to analysis for Java with a Dyck CFL.

Liu and Moore [13] encoded the semantics for the Java Virtual Machine in the ACL2 theorem prover by developing an interpreter for the JVM in pure Lisp. As a consequence of their work, it is possible to use ACL2 to reason about Java programs. In a similar vein, Leroy [23] developed a certified compiler for a C-like language in the Coq proof assistant. There is, of course, a strong and obvious analogy between using a theorem prover or proof assistant to reason about programs and using a logic programming system to reason about programs.

9.2 Solving declarative analysis queries

Prior to the work described in Section 9.1, Reps [31] showed how to automatically derive demand-driven versions of dataflow analysis algorithms – that is, analysis procedures that calculate exact results for a particular subset of the program or for a particular program point. His technique relied on the magic-sets transformation: applying this transformation to a logic program implementing an exhaustive analysis algorithm results in a demand-driven version of the algorithm.

Saha and Ramakrishnan [34] adapted techniques for incremental and goal-driven evaluation of tabled Prolog in order to formulate incremental and demand-driven versions of program analyses. They evaluated their work on a version of Andersen's analysis for C, treating programs consisting of tens of thousands of preprocessed statements. As with the subset-based points-to analysis we present in Section 7, their analysis is flow-, field- and context-insensitive. (It is difficult to derive a sound field-sensitive analysis for C, since C admits many unsafe features. In contrast, the points-to analysis we presented is flow- and context-insensitive, but field-*sensitive*.) Their work focuses on applying techniques for improving the performance of logic programs to improving the utility of logic programming as a tool for program analysis.

Several special-purpose systems have been developed to allow declarative specifications of program analyses. The BANE toolkit [12] and its successor BANSHEE [21] generate specialized solvers for program analyses specified as constraint-satisfaction problems. An analysis designer would use these tools by developing a preprocessor to extract relevant constraints from the program text and then declare the analysis itself in terms of constraints on terms or sets. BANSHEE enjoys a rich type system and static checking of type-safety for analyses. BANSHEE also achieves high performance – a BANSHEE implementation of Andersen's analysis for C analyzed hundreds of thousands of lines of preprocessed C code in seconds and millions of lines of preprocessed C code in under a minute. Like the work of Saha and Ramakrishnan, BANSHEE also provides support for incremental evaluation of analysis queries.

The `bddbddb` system of Whaley et al. [42, 22] is a specialized implementation of Datalog based on *binary decision diagrams* [6], or BDDs. The BDD representation represents boolean functions as directed acyclic graphs, and is able to exploit redundancy in relations in order to solve queries on large relations efficiently. Indeed, the BDD-based implementation of Datalog in `bddbddb` can even enable users to solve queries on relations whose explicit representations would not fit in memory on any extant computer. Whaley and Lam demonstrate the performance of two `bddbddb` implementations of Andersen's analysis [42]: the context-insensitive implementation is roughly comparable to the performance we have achieved in a similar analysis with DIMPLE. Where the BDD-based representation excels, however, is in handling context-sensitive analyses – many of which would produce results too large to represent explicitly in a Prolog database. However, the performance (and tractability) of BDD-based analysis approaches depends on the size of the BDD graph, which in turn depends on the *variable ordering* that the BDD user has chosen. Bollig and Wegener showed that algorithmically choosing a "good or optimal" variable ordering for a BDD is NP-complete [5]. `bddbddb` requires the user to specify a variable ordering, but it features a machine learning-based process for automatically identifying a good ordering candidate from among several heuristically determined possibilities.

9.3 DIMPLE in context

The work we discussed in Section 9.1 primarily treats declarative formalisms for programs and program analysis problems. Obviously, the choice of a formalism for expressing analyses is somewhat subjective. However, in our opinion, the relational model of logic programming approaches (including DIMPLE) is rather easier to use than constraint- or set-based models.

In contrast, the work mentioned in Section 9.2 treats implementation techniques for efficient tools to handle evaluating analysis problems expressed in terms of various formalisms. DIMPLE already benefits from advances in logic programming system implementation; the efficient execution of a tabled Prolog system makes whole-program analyses feasible. However, users who wish to exploit capabilities not directly available in the underlying Prolog system (such as bddbddb's support for the very large relations required for context-sensitive analyses) could easily use DIMPLE as a front-end for a specialized solver. Such an approach would use the DIMPLE IR, statement processing language, and analysis language, but would override the DIMPLE code generator by writing a Prolog procedure that translates from user analysis rules to the Datalog format expected by bddbddb, to set constraints, or to any other format or formalism expected by an external tool.

Perhaps the best characterization of our work is that DIMPLE provides an interface, framework, and language to facilitate using a logic programming system for program analysis. The related work under discussion either presents formalisms for program analysis or advances the state of the art of logic programming, perhaps with immediate application for program analysis problems. Since we have developed an application that exploits many of the features of an advanced logic programming system, contributions that improve logic programming systems are complementary to ours.

The DIMPLE system enables analysis designers to rapidly prototype new program analyses in an interactive fashion. Unlike every other system under discussion, DIMPLE enables analysis designers to use logic programming for every phase of the analysis development process. DIMPLE is also unique in that the entire program text is available to the analysis designer for prototyping. However, DIMPLE enables users to discard irrelevant relations for efficient execution of production analyses.

While the other analysis frameworks we have mentioned represent excellent research contributions, no other system under discussion is as suitable for prototyping and interactive, exploratory development as DIMPLE. The DIMPLE analysis developer need never leave the declarative world of Prolog and the DIMPLE IR, whether preprocessing input programs, defining statement processing routines, or declaring and evaluating analysis rules. In contrast, other systems like BANSHEE and bddbddb require the user to develop a specialized preprocessor for source text that extracts relations of interest, to declare a set of rules or constraints, and then to feed preprocessed program text and the user-declared rules into a specialized solver. If there is an error in the preprocessor or rules, the user must start over; in DIMPLE, one may simply assert or retract additional rules or relations as necessary. Other factors that inhibit casual experimentation and rapid, interactive prototyping come from implementation details: BANSHEE analyses are C programs that link with a specialized solver library. bddbddb specifications require a good BDD variable ordering. Deciding on a good ordering is nontrivial and requires either a user with a profound understanding of the BDD abstraction and the problem domain or a user who is willing to wait for the bddbddb tool to automatically apply time-consuming heuristics and learning techniques to find a good ordering.

We plan to improve DIMPLE in the future by incorporating some of the best ideas from these other systems. In particular, we intend to investigate ways to support large relations with significant redundancies, as does bddbddb. We are also interested in using DIMPLE to develop tools based on abstract interpretation or operational semantics, like the aforementioned work involving ACL2 and Coq.

10. Conclusion

This paper has presented DIMPLE, a framework that facilitates rapid prototyping, development, and implementation of program preprocessors and static analyses. Because the analysis designer can defer decisions about which program statements are relevant – or even which analysis rules are necessary – until the analysis is actually producing results, DIMPLE encourages experimentation and interactive development and provides for a spectrum of executable analyses from flexible prototypes to efficient production implementations. More generally, we have confirmed prior work that has asserted the suitability of general-purpose logic programming systems for program analysis tasks; we have also reinforced these assertions by demonstrating scalability with substantially larger input programs than were treated in prior evaluations.

DIMPLE is unique in that it is designed to encourage interactive experimentation with new analysis ideas; it also enables analysis designers to use declarative specifications for every step of the analysis design and implementation process. We have demonstrated the usability of DIMPLE by showing the actual executable specifications for two realistic, fundamental program analyses and the preprocessing phases necessary for each. Finally, we have demonstrated that DIMPLE implementations of two analyses are efficient enough for production use; our DIMPLE implementation of Andersen's analysis is in fact speed-competitive with a BDD-based specialized solver.

Acknowledgments

We wish to thank Vítor Santos Costa, Nicholas Kidd, Anne Mulhern, and the anonymous referees for comments on prior drafts of this paper. Any errors or opinions in this paper are our own.

References

[1] ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (2002), ACM Press, pp. 311–330.

[2] ANDERSEN, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] BERNDL, M., LHOTÁK, O., QIAN, F., HENDREN, L., AND UMANEE, N. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (2003), ACM Press, pp. 103–114.

[4] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (New York, NY, USA, Oct. 2006), ACM Press.

[5] BOLLIG, B., AND WEGENER, I. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers 45*, 9 (1996), 993–1002.

[6] BRYANT, R. E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys 24*, 3 (1992), 293–318.

[7] CHEN, W., AND WARREN, D. S. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43*, 1 (1996), 20–74.

[8] COSTA, V. S., DAMAS, L., REIS, R., AND AZEVEDO, R. *YAP User's Manual*. Universidade do Porto, 2000.

[9] DAWSON, S., RAMAKRISHNAN, C. R., AND WARREN, D. S. Practical program analysis using general purpose logic programming systems–a case study. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 1996), vol. 31, ACM Press, pp. 117–126.

[10] DIWAN, A., MCKINLEY, K. S., AND MOSS, J. E. B. Using types to analyze and optimize object-oriented programs. *Programming Languages and Systems 23*, 1 (2001), 30–72.

[11] ESPARZA, J., AND PODELSKI, A. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2000), ACM Press, pp. 1–11.

[12] Fähndrich, M., and Aiken, A. Program analysis using mixed term and set constraints. In *Static Analysis Symposium* (1997), pp. 114–126.

[13] Hanbing Liu, and J Strother Moore. Java Program Verification via a JVM Deep Embedding in ACL2. In *Proceedings of 17th International TPHOLs 2004* (2004), Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, Eds., vol. 3223 of *LNCS*, pp. 184–200.

[14] Heintze, N. *Set-Based Program Analysis*. PhD thesis, Pittsburgh, PA, USA, October 1992.

[15] Heintze, N., and Jaffar, J. Set constraints and set-based analysis. In *Principles and Practice of Constraint Programming* (1994), pp. 281–298.

[16] Heintze, N., and Tardieu, O. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), ACM Press, pp. 254–263.

[17] Hendren, L. J., Donawa, C., Emami, M., Gao, G. R., Justiani, and Sridharan, B. Designing the McCat compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing* (London, 1993), Springer-Verlag, pp. 406–420.

[18] Hind, M. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)* (Snowbird, UT, 2001).

[19] Hind, M., and Pioli, A. Which pointer analysis should i use? In *International Symposium on Software Testing and Analysis* (2000), pp. 113–123.

[20] Kodumal, J., and Aiken, A. The set constraint/cfl reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation* (New York, NY, USA, 2004), ACM Press, pp. 207–218.

[21] Kodumal, J., and Aiken, A. Banshee: A scalable constraint-based analysis toolkit. In *Proceedings of 12th International Static Analysis Symposium* (2005), pp. 218–234.

[22] Lam, M. S., Whaley, J., Livshits, V. B., Martin, M. C., Avots, D., Carbin, M., and Unkel, C. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (2005), ACM.

[23] Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM Press, pp. 42–54.

[24] Lindholm, T., and Yellin, F. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[25] Lucassen, J. M., and Gifford, D. K. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1988), ACM Press, pp. 47–57.

[26] Melski, D., and Reps, T. Interconvertbility of set constraints and context-free language reachability. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (New York, NY, USA, 1997), ACM Press, pp. 74–89.

[27] Milanova, A., Rountev, A., and Ryder, B. G. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering Methodology 14*, 1 (2005), 1–41.

[28] Palsberg, J. Type-based analysis and applications. In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)* (Snowbird, UT, 2001), pp. 20–27.

[29] Reps, T. Program analysis via graph reachability. *Information and Software Technology 40*, 11–12 (November/December 1998), 701–726.

[30] Reps, T., Schwoon, S., Jha, S., and Melski, D. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming 58*, 1-2 (2005), 206–263.

[31] Reps, T. W. Solving demand versions of interprocedural analysis problems. In *Proceedings of the 5th International Conference on Compiler Construction* (1994), Springer-Verlag, pp. 389–403.

[32] Rocha, R., Silva, F., and Costa, V. S. On Applying Or-Parallelism and Tabling to Logic Programs. *Theory and Practice of Logic Programming Systems 5*, 1-2 (2005), 161–205.

[33] Rountev, A., Milanova, A., and Ryder, B. G. Points-to analysis for java using annotated constraints. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2001), ACM Press, pp. 43–55.

[34] Saha, D., and Ramakrishnan, C. R. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming* (2005), ACM Press.

[35] Salcianu, A., and Rinard, M. A combined pointer and purity analysis for Java programs. TR MIT-CSAILTR-949, MIT, May 2004.

[36] Sridharan, M., Gopan, D., Shan, L., and Bodík, R. Demand-driven points-to analysis for java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2005), ACM Press, pp. 59–76.

[37] Standard Performance Evaluation Corporation. Specjvm98 benchmark suite (http://www.spec.org), 1998.

[38] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1996), ACM Press, pp. 32–41.

[39] Tofte, M., and Talpin, J.-P. Region-based memory management. *Information and Computation* (1997).

[40] Tomb, A., and Flanagan, C. Automatic type inference via partial evaluation. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2005), ACM Press, pp. 106–116.

[41] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), IBM Press, p. 13.

[42] Whaley, J., and Lam, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 2004), ACM Press.