Graph Queries through Datalog Optimizations *

K. Tuncay Tekle Michael Gorbovitski Yanhong A. Liu
Department of Computer Science, State University of New York at Stony Brook
tuncay,mickg,liu@cs.sunysb.edu

Abstract

This paper describes the use of a powerful graph query language for querying programs, and a novel combination of transformations for generating efficient implementations of the queries. The language supports graph path expressions that allow convenient use of both vertices and edges of arbitrary kinds as well as additional global and local parameters in graph paths. Our implementation method combines transformation to Datalog, recursion conversion, demand transformation, and specialization, and finally generates efficient analysis programs with precise complexity guarantees. This combination improves an $O(VE)$ time complexity factor using previous methods to $O(E)$, where V and E are the numbers of graph vertices and edges, respectively. We also describe implementations and experiments that confirm the analyzed complexities.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Constraint and logic languages, Very high level languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns; D.3.4 [*Programming Languages*]: Processors—Optimization; F.2.2 [*Analysis of Algorithms and Problems Complexity*]: Nonnumerical Algorithms and Problems—Computations on discrete structures; F.3.2 [*Logics and Meanings of Programs*]: Semantics of programming languages—Program analysis; H.2.3 [*Information Systems*]: Database Management—Query languages; H.2.4 [*Information Systems*]: Systems—Query processing, Rule-based databases

General Terms Languages, Performance

Keywords Complexity analysis, Datalog, demand-driven evaluation, graph query languages, program analysis, program transformation, optimization

1. Introduction

Graph queries can be used to express many problems from different areas, including program analysis in particular. Such queries can help find bugs [12], detect malicious virus patterns [8], report security violations [25], check temporal safety properties [3], etc. Efficient hand-written implementations for program analyses are difficult to develop, verify, and maintain, and query languages for specifying such analysis problems are desirable for ensuring the correctness of analyses while reducing the effort of implementations. However, higher-level query languages often lack efficient implementations or complexity guarantees. An automated approach to generating efficient implementations with complexity guarantees is needed for practical uses of such languages.

This paper describes the use of a powerful graph query language for querying programs, and a novel combination of transformations for automatically generating efficient implementations of the queries. We show that a wide range of program analysis problems can be expressed using queries in the language. The language supports graph path expressions that allow convenient use of both vertices and edges of arbitrary kinds as well as additional global and local parameters in graph paths. Our implementation method for the language combines transformation to Datalog, recursion conversion, hypothesis permutation, demand transformation, and specialization, and finally generates efficient analysis programs with precise complexity guarantees.

The first step of our method transforms a graph query into a set of rules and a query in Datalog. Datalog is an important rule-based language for inference using facts in databases. Much research has been done on implementation of Datalog [1]. Top-down computation starts from the query, generates subqueries using the hypotheses of the rules whose conclusions match the query, and does so repeatedly until the subqueries match the given facts. Bottom-up computation matches existing facts with hypotheses of rules, generates new facts from conclusions of rules whose hypotheses match existing facts, and does so repeatedly until desired facts or all facts are inferred. In particular, there is a method for generating optimal algorithm and data structures specialized for the rules for bottom-up computation, and calculating complexities for the generated programs [20]. However, this method may infer facts that are not relevant to answering a given query, and hence may be unnecessarily expensive.

To solve the problem of inferring facts not relevant to answering a given query, we perform a transformation similar to the well-known magic-set transformation [4], which transforms a given set of rules into a new set of rules that infer only facts relevant to the query. We call it *demand transformation*, because the transformation makes the computation driven by demand. Demand transformation may reduce the number of inferred facts, and therefore the complexity, asymptotically. However, its effectiveness is affected drastically by the form of recursion in recursive rules, and by the order of hypotheses. To solve this problem, before demand transformation, we first perform recursion conversion, which generates different forms of recursive rules. We then generate different permutations of hypothesis orders of the rules.

Recursion conversion, hypothesis permutation, and demand transformation yield different versions of a set of rules. The complexity of each version is calculated automatically using the method in [20]. The version that has the best complexity is then chosen. Choosing the best complexity is impossible in general, so our

* This work was supported in part by NSF under grants CCF-0964196 and CCF-061391 and by ONR under grant N000140910651.

PPDP'10, July 26–28, 2010, Hagenberg, Austria.

method chooses the versions that have provably better complexity than the rest. Using heuristics, the method succeeds in choosing a single set of rules for all the graph query examples we have encountered.

The above transformations may result in a set of rules that contain unnecessary predicates, rules, and constant arguments. Specialization, also known as partial evaluation [16], is used to remove these unnecessary parts. It does not reduce asymptotic time complexity, but may reduce both the memory usage and running time by a constant factor, and moreover, may produce rules that are much smaller and simpler. Finally, we generate an efficient program from the final set of rules, together with the calculated complexity, using the method in [20].

This combination of transformations improves an $O(VE)$ time complexity factor using previous methods to $O(E)$, where V and E are the numbers of graph vertices and edges, respectively. We show precise time complexity analysis for an example program analysis problem using this method. We also describe implementations of the example analysis problem and experiments for analyzing a set of programs of varying sizes, and confirm the calculated complexities. Additionally, we compare our results against XSB [24], a state-of-the-art top-down evaluation engine that employs tabling strategies, and bddbddb [15], a bottom-up Datalog evaluation engine that employs binary decision diagrams for storing and manipulating relations. For both systems, tight complexity guarantees are not available, and we demonstrate that they show different behavior with respect to different forms of rules without a method of predetermining the best form.

Even though the graph query language and most of the transformations to Datalog were proposed before [19], and similar languages have been used for program transformations [27], this paper is the first substantial use of the language to give precise complexity analyses for program analyses that are more complex than analyses possible without the power of the language [18]. We also show that our novel combination of transformations is necessary for generating efficient implementations that are both asymptotically better, and with better constants than possible before [19, 26]. Finally, we conduct the first substantial experimental evaluation of different combinations of transformations that our method uses, and compare them with implementations of rules in XSB and bddbddb.

The rest of the paper is organized as follows. Section 2 describes the graph query language, the Datalog language, and the cost model. Section 3 shows that a variety of program analyses can be expressed in the language. Section 4 describes the transformations and their combination. Section 5 discusses implementation and experiments. Section 6 compares with related work and concludes.

2. Language and cost model

Graph query language. We use a graph query language that can specify the existence of paths with various properties proposed in [19]. For our examples, we use the language to query the control-flow graphs of programs. We consider control-flow graphs whose vertices correspond to program points, and labeled edges correspond to operations. We use edge labels that reflect only information relevant to the analysis of interest. Consider an assignment statement $a = 5$ in a program. If we are interested in analyzing reaching definitions, then this statement may be represented by the label `def(a)`, indicating a definition of (i.e., assignment to) a. One may use several abstractions to represent one statement, therefore multiple edges between two vertices are possible. For the statement $x = y + 1$, edges can correspond to the definition of x, and the usage of y. We denote the entry point of a program as `start`.

Many analyses can be performed on a control-flow graph. For example, the use of an uninitialized variable in a program can be determined by finding a path starting from the entry point in the program such that a variable is never initialized but eventually used on the path.

The graph query language supports graph path expressions that allow convenient use of both vertices and edges of arbitrary kind as well as additional global and local parameters in graph paths. Figure 1 gives a grammar of the language.

query	→	var,...,var: pexp
pexp	→	path
	|	pexp ∧ pexp | pexp ∨ pexp | ¬pexp
path	→	ov epath ov ... epath ov
ov	→	[v] | ϵ
epath	→	label
	|	epath ∧ epath | epath ∨ epath | ¬epath
	|	epath* | epath epath | epath ∧ constraint
	|	`local` var,...,var: path
label	→	$p(a_1,...,a_k)$ | _
v, p, a	→	const | var | _
constraint	:	boolean expression
var	:	identifiers (denoted by letters `t` to `z`)
const	:	literals

Figure 1. Grammar for the graph query language.

A graph query consists of a list of variables to be returned and a path expression. A path expression is a path or a conjunction, disjunction, or negation of path expressions. A path is a sequence of edge paths separated by optional vertices. An edge path is an edge label; a conjunction, disjunction, or negation of edge paths; a repetition (denoted *) of an edge path, a concatenation of edge paths, an edge path with a constraint, or a path with local variables. Repetition of an edge path means a concatenation of any number of the edge paths. Negation of an edge path means the nonexistence of the edge path. A path with local variables means that local variables in the path may take different values each time the path is repeated. An edge label is a predicate with arguments or a wildcard label (denoted _). A wildcard label holds for any edge. A vertex, predicate, or argument is a variable, constant, or a wildcard variable. A wildcard variable (denoted _) is treated like a local variable.

The meaning of a graph query is all the values of return variables such that the path expression holds.

Datalog rules and queries. *Datalog* is a declarative language for defining facts and rules that are used to infer new facts from given facts [1]. A Datalog rule is of the form:

$$p(x_1, ..., x_k) :- p_1(x_{11}, ..., x_{1k_1}), ..., \ p_h(x_{h1}, ..., x_{hk_h}).$$

where h is a natural number, each p_i (respectively p) is a predicate of k_i (respectively k) arguments, each x_{ij} and x_i is either a constant or a variable, and variables in x_i's must be a subset of the variables in x_{ij}'s. A predicate with arguments is called an *atom*. If $h = 0$, then there are no p_i's or x_{ij}'s, and x_i's must be constants, in which case $p(x_1, ..., x_k)$ is called a *fact*. An atom on the right side of a rule is called a *hypothesis*, and the atom on the left side is called the *conclusion*.

Datalog can be extended with negation and constraints. We use Datalog with *stratified* negation, i.e., the conclusion of any rule and a negated hypothesis of any rule are not mutually recursive. The set of rules generated from any query in the graph query language we use is guaranteed to be stratified. A hypothesis that is not negated is called a positive hypothesis. We use constraints where each variable occurring in a constraint is bound by a positive hypothesis.

A Datalog rule is *unsafe* if there is a variable argument of the conclusion that does not appear as an argument of a positive hy-

pothesis. Unsafe Datalog rules are expensive for bottom-up computation, since values of variables that appear in the conclusion and not in positive hypotheses cannot be determined by matching existing facts with hypotheses, and each such variable may take all constants in the facts.

The meaning of a rule is that if there is a substitution of variables in the rule with constants such that all positive hypotheses instantiated using the substitution are facts, all negative hypotheses instantiated using the substitution are not facts, and the constraints hold under the substitution, then the instantiated conclusion is a fact.

A query is of the form $q(y_1, .., y_k)?$, where q is a predicate of k arguments. The meaning of a query given a set of Datalog rules and facts is the set of all facts of q that are given or can be inferred using the rules, restricted by the constants in the query, if any.

For ease of presentation, in our examples, letters `t` to `z` denote variables, and all other symbols are constants.

Graph queries and Datalog facts. Since we transform our queries to Datalog, we need to represent the graph data as Datalog facts. For each edge label $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ between vertices x and y, the corresponding Datalog fact is $\texttt{p(x,y,a}_1\texttt{,...,a}_k\texttt{)}$.

Cost model. Optimal bottom-up computation of the meaning of a set of Datalog rules [20] takes time proportional to the number of combinations of facts that make all positive hypotheses true for each rule, plus the number of given facts since given facts must be read in. If there are negated hypotheses in a rule, negation is performed as a lookup, and constraints are checked after the substitution. We split each rule into rules of at most two positive hypotheses, since this yields better time complexity.

When expressing time complexity, we use V and E to refer to the number of vertices and edges, respectively, in the control-flow graph. For an edge label `l`, we use `#l` to refer to the number of edges in the graph labeled with `l`. Similarly, we use `#p` to denote the number of facts for a Datalog predicate `p`.We use $\texttt{\#p.i}_1\texttt{,..,i}_n\texttt{/j}_1\texttt{,...,j}_m$ to denote the maximum number of combinations of different values of the $\texttt{i}_1$,...,$\texttt{i}_n$th arguments of the facts of predicate `p` (given or inferred), given any fixed value for the $\texttt{j}_1$,...,$\texttt{j}_m$th arguments.

3. Applications of graph queries

We show a variety of program analysis problems specified as graph queries, and illustrate the power of the language with queries that use different language features. The queries are shown in Figure 2 and explained below.

(i) Uninitialized variables. We use this example as our running example. An edge corresponding to the definition of a variable x is labeled `def(`x`)`, and an edge corresponding to the use of a variable x is labeled `use(`x`)`. The query shown in Figure 2 returns the set of pairs of program point w and variable x such that x is not defined or used before w, and used for the first time at w.

(ii) Hash values in a map. In Java, it is illegal to change the hash value of an object while it is in a HashMap [6]. We use `add/rem_map(`x`,`y`)` to denote adding/removing y to/from map x, and `change_hash(`x`)` to denote changing the hash value of x. The query shown in Figure 2 returns the set of program points w at which an object's hash is changed after it has been added to a HashMap and not removed subsequently.

(iii) Expensive loops. Concatenation to a string is an expensive computation if done repeatedly. We use `concat(`x`,`y`)` to represent the operation that concatenates y to x. The query shown in Figure 2 returns the set of pairs of program point w and variable x, such that a string is concatenated to x after program point w, and there is a loop containing the program point w.

Examples (i) to (iii) show that variables on vertices make the analyses powerful by adding both the flexibility of returning arbitrary information from the graph, and relating vertices in the query.

(iv) Live branches. The semantics of MATLAB implies that an if-branch with a set s as the condition is taken if s is nonempty and all elements of s are positive numbers. Dead-code elimination for if-branches is possible if the branch can never be taken. The query shown in Figure 2 returns the set of program points w such that the if-branch at w is not removable by dead-code elimination. This is done by finding a path in the program such that all elements added to a set x are guaranteed to be positive. We use `add(`x`,`y`)` to denote the addition of y to set x, and `if(`x`)` to denote an if-branch with condition x.

This example shows that the use of local variables in queries helps imposing properties on each edge in a path while ensuring global properties at the same time.

Many more examples can be shown, but we do not show them here since they are not conceptually different. Such examples include the specification of malicious virus patterns [8], security violations in programs and operating systems [2, 7, 25], and temporal safety properties [3].

4. Generating efficient implementations

To generate an efficient implementation for a graph query, our method does (i) transformation to Datalog, (ii) recursion conversion and hypotheses permutation, (iii) demand transformation, (iv) specialization, and (v) program generation. The generation takes as input the graph query and the graph data, and produces efficient implementations and corresponding complexity guarantees. We demonstrate the steps on our running example, the uninitialized variables query.

Step 1: Transformation to Datalog. This step transforms a graph query into a set of rules and a query in Datalog extended with negation and constraints. The resulting rules naturally capture the query structure, and are subsequently drastically optimized and efficiently implemented.

(1) Preprocessing. The query is preprocessed as follows. (i) If there is any edge label, whose predicate is `_` and which has no arguments, then that label is replaced by the label `edge()`. The facts are updated as follows: for each pair of vertices x and y such that there is a fact $\texttt{p(x,y,a}_1\texttt{,...,a}_k\texttt{)}$, a fact `edge(x,y)` is introduced. (ii) For all remaining occurrences of `_`, each occurrence is replaced with a new local variable, distinct for each occurrence. (iii) After these are applied, if there is any edge label $\texttt{v(a}_1\texttt{,...,a}_k\texttt{)}$, where `v` is a variable, then this label is replaced with $\texttt{label(v,a}_1\texttt{,...,a}_k\texttt{)}$. The facts are updated as follows: for each fact $\texttt{p(x,y,a}_1\texttt{,...,a}_k\texttt{)}$. given, a new fact $\texttt{label(p,x,y,a}_1\texttt{,...,a}_k\texttt{)}$ is added.

(2) Construction of rules. The query is recursively processed to obtain Datalog rules and a query. For this task, a function f is defined that maps the query to a Datalog query, and that maps subexpressions of the query to atoms. Also, rules are added to an initially empty set R during the application of f. Given a query q preprocessed as above, $f(q)$ returns a Datalog query, and upon return, R contains the set of rules. We use two new variables $\texttt{v}_s$ and $\texttt{v}_t$ that do not appear in the query for insertion as source and target vertices, respectively.

- For an edge label e of form $p(a_1, ..., a_k)$, $f(e) =$ $\texttt{p(v}_s\texttt{,v}_t\texttt{,a}_1\texttt{,...,a}_k\texttt{)}$. For example, assuming we use `y` and `z` for $\texttt{v}_s$ and $\texttt{v}_t$, $f(\texttt{def(x)}) = \texttt{def(y,z,x)}$.

(i) Uninitialized vars	(ii) Hash values in a map
$w, x:$ [start] $(\neg(\texttt{def}(x) \vee \texttt{use}(x)))^*$ $[w]$ $\texttt{use}(x)$	$w:$ [start] $\_^*$ $\texttt{add\_map}(x, y)$ $(\neg\texttt{rem\_map}(x, y))^*$ $[w]$ $\texttt{change\_hash}(y)$
(iii) Expensive loops	**(iv) Live branches**
$w, x:$ [start] $\_^*$ $[w]$ $\texttt{concat}(x, \_)$ $\_^*$ $[w]$	$w:$ [start] $(\neg\texttt{add}(x, \_))^*$ $(\texttt{add}(x, y) \wedge y > 0)$ $(\texttt{local } z: ((\texttt{add}(x, z) \wedge z > 0) \vee \neg\texttt{add}(x, \_)))^*$ $[w]$ $\texttt{if}(x)$

Figure 2. Example queries for program analysis.

- For a constraint c, a fresh predicate name $\texttt{p}_c$ is used. $f(c) = \texttt{p}_c\texttt{(v}_1\texttt{,...,v}_n\texttt{)}$, where $\texttt{v}_1, ..., \texttt{v}_n$ are the variables in c.
- For an edge path e,
  - if e is of form $e_1 \wedge ... \wedge e_n$, and each $f(e_i) = \texttt{p}_i\texttt{(v}_{i1}\texttt{,...,v}_{ik_i}\texttt{)}$, then $f(e) = \texttt{p(v}_1\texttt{,...,v}_k\texttt{)}$, where p is a fresh predicate name, and $\texttt{v}_1$,...,$\texttt{v}_k$ is the subsequence of $\texttt{v}_{11}$,...,$\texttt{v}_{nk_n}$ that are variables and appear anywhere else in the query except e. In this case, the following rule is added to R:

    ```
    p(v1,...,vk) :- p1(v11,...,v1k1), ...,
                    pn(vn1,...,vnkn).
    ```

  - if e is of form $e_1 \vee ... \vee e_n$, then $f(e)$ is exactly as for the conjunction case above. However, in this case, n rules of the following form are added to R:

    ```
    p(v1,...,vk) :- pi(vi1,...,vik_i).
    ```

    For example, $f(\texttt{def}(x) \vee \texttt{use}(x)) = \texttt{defuse(y,z,x)}$, and the following rules are added to R:

    ```
    defuse(y,z,x) :- def(y,z,x).  (R1)
    defuse(y,z,x) :- use(y,z,x).  (R2)
    ```

  - if e is of form $\neg e_1$, and $f(e_1) = \texttt{p}_1\texttt{(v}_{11}\texttt{,...,v}_{1k_1}\texttt{)}$, then $f(e) = \texttt{p(v}_1\texttt{,...,v}_k\texttt{)}$, where p is a fresh predicate name, and $\texttt{v}_1$,...,$\texttt{v}_k$ is the subsequence of $\texttt{v}_{11}$,...,$\texttt{v}_{1k_1}$ that are variables and appear anywhere else in the query except e. In this case, the following rule is added to R:

    ```
    p(v1,...,vk) :- not p1(v11,...,v1k1).
    ```

    For example, $f(\neg(\texttt{def}(x) \vee \texttt{use}(x))) = \texttt{ndu(y,z,x)}$, and the following rule is added to R:

    ```
    ndu(y,z,x) :- not defuse(y,z,x). (R3)
    ```

  - if e is of form e_1*, and $f(e_1) = \texttt{p}_1\texttt{(v}_{11}\texttt{,...,v}_{1k_1}\texttt{)}$, then $f(e) = \texttt{p(v}_s\texttt{,v}_t\texttt{,v}_1\texttt{,...,v}_k\texttt{)}$, where p is a fresh predicate name, and $\texttt{v}_1$,...,$\texttt{v}_k$ is the subsequence of $\texttt{v}_{13}$,...,$\texttt{v}_{1k_1}$ that are variables and appear anywhere else in the query except e. In this case, the following two rules are added to R, where $\texttt{v}_f$ is a fresh variable:

    ```
    p(vs,vs,v1,...,vk).
    p(vs,vt,v1,...,vk) :- p(vs,vf,v1,...,vk),
                          p1(vf,vt,v13,...,vnk1).
    ```

    For example, $f((\neg(\texttt{def}(x) \vee \texttt{use}(x)))*) = \texttt{ndus(y,z,x)}$, and the following fact and rule are added to R:

    ```
    ndus(y,y,x).                                  (R4)
    ndus(y,z,x) :- ndus(y,t,x), ndu(t,z,x).       (R5)
    ```

  - if e is of form $e_1 e_2 ... e_n$, and $f(e_i) = \texttt{p}_i\texttt{(v}_{i1}\texttt{,...,v}_{ik_i}\texttt{)}$, then $f(e) = \texttt{p(v}_s\texttt{,v}_t\texttt{,v}_1\texttt{,...,v}_k\texttt{)}$, where p is a fresh predicate name, and $\texttt{v}_1$,...,$\texttt{v}_k$ is the subsequence of $\texttt{v}_{11}$,...,$\texttt{v}_{nk_n}$ that are variables and appear anywhere else in the query except e. The following rule is added to R, where each $\texttt{v}_{fi}$ is a fresh variable.

    ```
    p(vs,vt,v1,...,vk) :- p1(vs,vf2,v13,...,v1k1),
                          p2(vf2,vf3,v23,...,v2k2),
                          ...,
                          pn(vfn,vt,vn3,...,vnkn).
    ```

  - if e is of form $\texttt{local } var_1, ..., var_n : e_1$, and $f(e_1) = \texttt{p}_1\texttt{(v}_{11}\texttt{,...,v}_{1k_1}\texttt{)}$, then $f(e) = \texttt{p(v}_1\texttt{,...,v}_k\texttt{)}$, where p is a fresh predicate name, and $\texttt{v}_1$,...,$\texttt{v}_k$ is the subsequence of $\texttt{v}_{11}$,...,$\texttt{v}_{1k_1}$ that are variables and not in $var_1, ..., var_n$. The following rule is added to R:

    ```
    p(v1,...,vk) :- p1(v11,...,v1k1).
    ```

- For a path e of form $ov_1\ e_1\ ...\ ov_n\ e_n\ ov_{n+1}$, a placeholder vertex with a fresh variable name is inserted for each optional vertex ov_i that is not specified. For example, in our running example, a placeholder vertex [u] is inserted at the end of the query. After this, if each $f(e_i) = \texttt{p}_i\texttt{(v}_{i1}\texttt{,...,v}_{ik_i}\texttt{)}$, then $f(e) = \texttt{p(v}_1\texttt{,...,v}_k\texttt{)}$, where p is a fresh predicate name, and $\texttt{v}_1$,...,$\texttt{v}_k$ is the subsequence of $\texttt{v}_{11}$,...,$\texttt{v}_{nk_n}$ that are variables and appear anywhere else in the query except e. The following rule is added to R:

    ```
    p(v1,...,vk) :- p1(ov1,ov2,v13,...,v1k1), ...,
                    pn(ovn,ovn+1,vn3,...,vnkn).
    ```

For example, if we denote the path expression in the running query pe, then $f(pe) =$ `result(w,x)`, and the following rule is added to R:

```
result(w,x) :- ndus(start,w,x), use(w,u,x). (R6)
```

- For a path expression e, if e is a negation, conjunction or disjunction of path expressions, then we proceed precisely as we did for the negation, conjunction and disjunction cases for edge paths.
- For a query q in the form $var_1, ..., var_n : p$, we define $f(q) = f(p)$`?`. For the running example, if we denote the query q, then $f(q) =$ `result(w,x)?`.

(3) Postprocessing. Postprocessing removes unsafe rules. First, for each atom generated for constraints, we replace the atom with the constraint it was generated for. Then, if any rule in the result is unsafe, we perform the following: (i) For each pair of vertices x and y such that there is a fact `p(x,y,`a_1`,...,`a_n`)`, we introduce a fact `edge(x,y)`. Also, for each constant c that appears in the facts as arguments, we introduce a fact `any(c)`. (ii) For each rule whose conclusion has arguments that are not bound by positive hypotheses, for each unbound argument a, if a is among the first two arguments of the conclusion (say a_1 and a_2), we add a hypothesis `edge(`a_1`,`a_2`)`, otherwise we add a hypothesis `any(a)`. Finally, we remove any duplicate hypotheses added. For the running example, rules `(R3)` and `(R4)` are modified to obtain the final set of rules below.

```
defuse(y,z,x) :- def(y,z,x).                    (R1)
defuse(y,z,x) :- use(y,z,x).                    (R2)
   ndu(y,z,x) :- edge(y,z), any(x),             (R3)
                 not defuse(y,z,x).
  ndus(y,y,x) :- edge(y,z), any(x).             (R4)
  ndus(y,z,x) :- ndus(y,t,x), ndu(t,z,x).       (R5)
  result(w,x) :- ndus(start,w,x), use(w,u,x). (R6)
```

The time complexity of computation using each rule is given in the left column of Figure 3. The bottleneck is the complexity for `(R5)`, $O(V \times$`#ndu`$)$; since `#ndu` is bounded by $O(E \times$`#any`$)$ based on `(R3)`, this complexity is $O(V \times E \times$`#any`$)$.

Step 2: Recursion conversion and hypothesis permutation. This step generates different forms of the rules from Step 1 with the same semantics. It is essential because different forms of rules may have drastically different running time and space usage after demand transformation and specialization in the subsequent steps.

This step first performs recursion conversion to obtain both left- and right-recursive forms of recursive rules. This uses the transformations described previously [26]. For example, for `(R5)`, an alternative rule with the same semantics is:

```
  ndus(y,z,x) :- ndu(y,t,x), ndus(t,z,x).     (R5')
```

This step then permutes hypotheses that are not constraints or negations in each rule; constraints and negations are placed immediately after all of their arguments are bound. For example, for `(R6)`, an alternative rule with a different order of hypotheses is:

```
  result(w,x) :- use(w,u,x), ndus(start,w,x). (R6')
```

Finally, a new set of rules is generated for each combination of different recursive forms of rules and different permutation of hypotheses in rules. We avoid unnecessary combinations using three heuristics described below.

1. For a recursively defined predicate p, if there is a hypothesis whose predicate is p and its first argument is a constant, then we only generate the left-recursive form for the recursive rule that defines p, and respectively if the second argument is a constant, then we only generate the right-recursive form. These forms are asymptotically better to use, since after demand transformation, the chosen recursive form will be asymptotically faster than the alternative form.
2. Among two permutations in each rule, if the predicate of one of the hypotheses h_1 is a predicate for which facts are given, and the predicate of the other hypothesis h_2 is a predicate defined by rules, then we always order the hypotheses so that h_1 is first and h_2 is second. This reduces the time complexity, since after demand transformation, the demand for h_2 will be stricter.
3. If the positive hypotheses of the original rule do not share any variables, then we use the given order. This is due to the fact that the join of these hypotheses costs the same in either direction when no variables are shared, so we can ignore the alternative order.

Original rules		After demand transformation	
`(R1)`	$O($`#def`$)$	`(R1)`	$O($`#def`$)$
`(R2)`	$O($`#use`$)$	`(R2)`	$O($`#use`$)$
`(R3)`	$O(E \times$`#any`$)$	`(R3d)`	$O(E \times$`#dem2.2/1`$)$
`(R4)`	$O(E \times$`#any`$)$	`(R4d)`	$O($`#dem`$)$
`(R5)`	$O(V \times$`#ndu`$)$	`(R5d1)`	$O($`#ndu`$)$
		`(R5d2)`	$O($`#ndu`$)$
`(R6)`	$O($`#use`$)$	`(R6d)`	$O($`#use`$)$

Figure 3. Time complexities for the original rules and the rules after demand transformation.

For the running example, there are two choices of recursion forms, and 16 hypothesis orders for each form. Thus, 32 different versions exist. Using heuristic 1 above, we obtain only the left-recursive form `(R5)`, not `(R5')` since the predicate of the first hypothesis of `(R6)` is `ndus` and its first argument is a constant (`start`). Using heuristic 2, we obtain only the reversed hypothesis order for `(R6)`, i.e., `(R6')`, since `use` is a predicate for which facts are given, and `ndus` is a predicate defined by rules. Using heuristic 3, we use the given orders for `(R3)` and `(R4)` since their positive hypotheses do not share any variables. Therefore, we are left with only one ordering for each rule.

Step 3: Demand transformation. This step performs, for each rule set obtained from Step 2, demand transformation. It adds new rules and new hypotheses to original rules so that only facts relevant to answering the query are inferred. This uses a novel transformation that achieves the same effect as magic set transformation [4], but is much simpler since the rules generated from graph queries are of a particular form. It is the center piece of our method and is described in detail in the next section.

After demand transformation, we calculate the complexity of each transformed rule set, and choose the one with the best complexity via comparison of the obtained formulas. Comparing the time complexity of two sets of rules is not possible in general, but for all the graph query examples we have encountered, it is possible to choose one set of rules with the best complexity. In case multiple rule sets have non-comparable complexities, the method proceeds on all rule sets, and the output contains multiple programs with different complexities.

For the running example, the resulting set of rules with the best complexity is for the original set of rules but with `(R6)` replaced by `(R6')`. It contains `(R1)`, `(R2)`, and the following rules; recall that rules are split into rules with at most two positive hypotheses each:

```
    ndu(y,z,x) :- dem2(z,x), edge(y,z),   (R3d)
                  not defuse(y,z,x).
   ndus(y,y,x) :- dem(y,y,x).             (R4d)
split(y,z,t,x) :- dem(y,z,x), ndu(t,z,x). (R5d1)
   ndus(y,z,x) :- split(y,z,t,x),         (R5d2)
                  ndus(y,t,x).
   result(w,x) :- use(w,u,x),             (R6d)
                  ndus(start,w,x).
dem(start,w,x) :- use(w,u,x).             (D1)
    dem(y,t,x) :- split(y,z,t,x).         (D2)
     dem2(z,x) :- dem(y,z,x).             (D3)
```

The time complexity of the resulting rules is given in the right column of Figure 3. It is reduced asymptotically, including dropping an $O(V)$ factor from `(R5)`, and the reduction of all $O($`#any`$)$ factors to tighter factors. The bottleneck complexity is reduced to $O(E\times$`#dem2.2/1`$)$ from $O(V \times E\times$`#any`$)$.

Step 4: Specialization. This step applies specialization and deterministic unfolding to the result from Step 3, to remove unnecessary predicates, arguments, and rules. Specialization uses a simplified version of partial evaluation, as described previously [26].

For specialization, we define a function f that takes an atom $\mathtt{p(a}_1\mathtt{,...,a}_k\mathtt{)}$ as an argument, and returns $\mathtt{p}_f\mathtt{(v}_1\mathtt{,...,v}_l\mathtt{)}$, where $\mathtt{p}_f$ is a fresh name, and $\mathtt{v}_1$,...,$\mathtt{v}_l$ is the subsequence of $\mathtt{a}_1$,...,$\mathtt{a}_k$ that are variables. For a set of rules R, and a query `q?`, we add `q` to a queue Q. For each atom a in Q, for each rule in R of the form $\mathtt{c}$ `:-` $\mathtt{h}_1\mathtt{,...,}$ $\mathtt{h}_n$ such that there exist two substitutions θ and θ' such that $\theta(\mathtt{c}) = \theta'(a)$, we perform two steps. First, for each $\mathtt{h}_i$, we add $\theta(\mathtt{h}_i)$ to Q. Second, we add the following rule to the output:

$\theta(\mathtt{c})$ `:-` $f(\theta(\mathtt{h}_1))$, ..., $f(\theta(\mathtt{h}_n))$. (Rs)

We also unfold hypotheses. For each rule r of the form $\mathtt{c}$ `:-` $\mathtt{h}_1$, ..., $\mathtt{h}_n$, for each hypothesis $\mathtt{h}_i$, if there is only one rule of the form `c' :- h'` for which there is a substitution θ such that $\theta(\mathtt{h}_i) = \mathtt{c'}$, we replace $\mathtt{h}_i$ in r with $\theta($`h'`$)$. Unfolding a hypothesis whose predicate is defined by more than one rule may decrease space, but increase time by a constant factor since the size of the rules become larger. We compare the performance of two unfolding strategies in the experiments section. A decision needs to be made for when to stop unfolding. We choose to stop unfolding at each recursive predicate, and we only unfold hypotheses that are defined by one rule, because it guarantees improvements in both time and space. This unfolding scheme is called deterministic unfolding [16].

This step does not reduce the asymptotic complexity, but reduces both running time and space by constant factors. For the running example, the resulting rules are `(R1)`, `(R2)`, and the following:

```
     ndu(y,z,x) :- dem_s(z,x), edge(y,z),  (R3ds)
                   not defuse(y,z,x).
ndus_s(start,x) :- dem_s(start,x).         (R4ds)
 split_s(z,t,x) :- dem_s(z,x), ndu(t,z,x). (R5d1s)
    ndus_s(z,x) :- split_s(z,t,x),         (R5d2s)
                   ndus_s(t,x).
    result(w,x) :- use(w,u,x), ndus_s(w,x).(R6ds)
     dem_s(w,x) :- use(w,u,x).             (D1s)
     dem_s(t,x) :- split_s(z,t,x).         (D2s)
```

This step removes the predicate `dem2` and rule `(D3)` that defines it; the first argument of predicate `dem` in rules `(R4d)`, `(R5d1)`, `(D1)`, and `(D2)`; the first argument of predicate `split` in rules `(R5d2)` and `(D2)`; and the first argument of predicate `ndus` in rules `(R4d)`, `(R5d2)`, and `(R6d)`.

Specialization applied after demand transformation does not change the asymptotic time complexity. However, when it is effective, it (i) reduces the space used by the computation by removing arguments of predicates that are guaranteed to be constants, (ii) reduces the time by a constant factor, and (iii) makes the resulting set of rules smaller and simpler.

Step 5: Program generation. This step generates efficient implementations with specialized data structures for the set of rules from Step 4. This uses the method by Liu and Stoller [20]. It guarantees that the generated implementation has the analyzed complexity. For the running example, the generated program in Python is 171 lines, and the generated program in C++ is 3534 lines.

We have shown that the application of the above five steps, and the order in which they are applied are crucial in obtaining efficient implementations for graph queries. After obtaining a set of Datalog rules, and a query whose set of answers are equivalent to the graph query, we use Datalog optimizations to asymptotically reduce the complexity of the resulting rules, and finally generate an efficient implementation of the optimized rules.

5. Demand transformation

Not all facts are needed for answering a query. Top-down evaluation does a query-driven, left-to-right evaluation of the rules, avoiding inferring facts not needed in this process for answering the query. Demand transformation transforms a set of rules and a query into a new set of rules, such that all the facts that can be inferred from the new set of rules contain only facts that would be inferred in a top-down evaluation of the original rules. It adds new rules that define needed facts for each hypothesis in each rule, and adds hypotheses to the original rules to restrict computation to infer only needed facts.

Given a set R of rules and a query `q?`, we first create a dummy rule `x :- q.` with a fresh predicate `x`, and add this rule to a workset W of needed rules. We take rules out of W, one at a time, to process until it is empty. For rule r, from W, of the form $\mathtt{c}$ `:-` $\mathtt{h}_1$, $\mathtt{h}_2$, ..., $\mathtt{h}_n$, we do the two steps below for each hypothesis $\mathtt{h}_i$.

Step 1 determines needed facts for $\mathtt{h}_i$. If some arguments of $\mathtt{h}_i$ are constants or are variables that occur also to the left of $\mathtt{h}_i$ in r, then we denote them a_1,...,a_k, and add the following rule to the set of resulting rules:

$\mathtt{demand}_{h_i}\mathtt{(a}_1\mathtt{,...,a}_k\mathtt{)}$ `:-` $\mathtt{h}_1$, $\mathtt{h}_2$, ... , $\mathtt{h}_{i-1}$. (D*)

where $\mathtt{demand}_{h_i}$ is a fresh predicate.

Step 2 restricts computation using rules that define the predicate of $\mathtt{h}_i$. For each rule r' in R of the form `c' :-` $\mathtt{h'}_1$, ..., $\mathtt{h'}_n$, if there is a substitution θ such that $\theta(\mathtt{h}_i) = \mathtt{c'}$, we create a new rule `(Rd)` as follows:

`c' :-` $\theta(\mathtt{demand}_{h_i}\mathtt{(v}_1\mathtt{,...,v}_k\mathtt{))}$, $\mathtt{h'}_1$, ... , $\mathtt{h'}_n$. (Rd)

Then, we remove, from `(Rd)`, each $\mathtt{h'}_i$ such that its predicate is `edge` or `any`, and all of its variable arguments appear in $\theta(\mathtt{demand}_{h_i}\mathtt{(v}_1\mathtt{,...,v}_k\mathtt{))}$. These hypotheses can be removed since they are now bound by the new hypothesis, and safety is preserved by that hypothesis. We add `(Rd)` to the output and W.

For the running example, the query is `result(w,x)?`. For rule `(R6')`, step 1 adds the rule `(D1)`, shown below, and step 2 inserts a first hypothesis in `(R5)` to obtain `(R5d)` below, which is then split into rules `(R5d1)` and `(R5d2)` as shown before:

```
dem(start,w,x) :- use(w,u,x).               (D1)
   ndus(y,z,x) :- dem(y,z,x), ndus(y,t,x),  (R5d)
                  ndu(t,z,x).
```

Demand transformation is akin to several transformations, the most well-known being magic set transformation (MST) [4]. Demand transformation differs from MST by the absence of annotations for given predicates. Demand transformation does not always

preserve stratification of negation, and methods are proposed [21] to always yield stratified rules. In the following theorem, we show that for rules generated for graph queries, the rules obtained after demand transformation contain only stratified negation.

Theorem 1. *Let R and q be the set of rules and query obtained from a graph query as described, and let R' be the set of rules obtained after demand transformation of R with respect to q. Both R and R' contain only stratified negation.*

Proof. For a rule generated from an expression e in a graph query, if the rule has a negated hypothesis, the negated predicate refers to a predicate for a subexpression of e, therefore the negation is stratified for all rules in R. In R', we add positive demand hypotheses to rules, and rules that define the predicates of demand hypotheses. The added hypotheses cannot violate stratification since they are positive. The rules that define the demand predicates only contain positive hypotheses, since the last hypothesis of a rule cannot appear as the hypothesis of those rules, and a negated hypothesis is always the last hypothesis of a rule if it exists. Therefore, the added rules in R' cannot violate stratification. □

There are two reasons for demand transformation's success in reducing asymptotic complexity for graph queries. Focusing on our examples for program analysis, first, most queries for program analyses start from the entry point of the program, which is a constant. Other constants occasionally occur in edge labels in queries. Having constant vertices significantly reduces the complexity of transitive closure after demand transformation is applied. Secondly, `edge` and `any` hypotheses are usually removed after demand transformation, since demand hypotheses usually bind the arguments of those hypotheses. The effect depends heavily on the form of recursion and order of hypotheses.

Note that specialization may be applied without applying demand transformation first. There are cases when demand transformation without specialization obtains better asymptotic complexity than specialization without demand transformation. However, if specialization alone provides the same complexity as demand transformation, then it is more preferable to obtain the set of rules from specialization since the rules become simpler and smaller. In our running example, applying specialization directly to `(R5)` would yield `(R5d1s)` and `(R5d2s)`, but demand transformation and specialization applied in order also yields the same rules.

Theorem 2 shows that when specialization yields rules with the same complexity as demand transformation, demand transformation and specialization applied in sequence yields the same set of rules as only applying specialization.

We say that a set of rules R_S obtained by specializing R is *simpler* than R if there is a predicate p such that for every rule r that defines p in R, its counterparts in R_S have fewer variables in arguments than r. We say that a set of rules R is in *no-copy normal form*, if there is no rule in R with only one hypothesis such that the argument list of the conclusion is a permutation of the argument list of the hypothesis.

Theorem 2. *For any specialization method S, if S obtains a set of rules R_S that is simpler than the original set of rules R, and is in no-copy normal form, then there is a form of recursion and an ordering of hypotheses of R, say R', such that demand transformation, S, and unfolding applied in sequence to R' produces R_S.*

Proof. If R_S is simpler than R, then there is a constant c in R that is propagated to a rule r by S to specialize r by removing an argument v. This means that v always takes the value c in r due to the hypotheses that refer to it. Therefore, there is a form of recursion and an ordering of hypothesis of R, say R', such that demand transformation will add a demand hypothesis that binds v in r, due to its assignment to c at the hypotheses referring to it, and the rule that defines that demand predicate will reflect that v takes the value c. When S is applied to the rules obtained after demand transformation to R', and unfolding is performed, the rule that defines the demand predicate is unfolded, and then the obtained constant from unfolding is propagated, and unnecessary constants and rules are eliminated. As a result, demand transformation, S, and unfolding applied in sequence to R' produces R_S. □

6. Experiments

We have implemented the method described in Python. As the final output of our method, the implementation emits both Python code, and Patton [23] code that is transformed automatically to C++ code by Patton, which is finally compiled by GCC.

We show the results of experiments using the running example on the control-flow graphs of six benchmark programs of varying size written in Python. The programs *chunk*, *bdb*, *tarfile*, and *pickle* are from the Python library; *Fortran* is a Fortran2003 implementation; *RBAC* is an implementation of an RBAC (Role Based Access Control) standard. In some figures, we omit one of the programs to avoid label overlapping. The experiments were conducted on a 3.0 GHz Intel Q9650 with 4 GB of memory, running SuSE Linux, and using Python 2.6.1 and GCC 4.3.3.

Running time and memory usage of the generated implementations. We have shown via automatic complexity analysis that the rules obtained after Step 1 in Section 4 are asymptotically worse than the final set of rules. The implementation of those rules only completes the smallest benchmark in 9.2 seconds, and cannot complete the rest of the benchmarks in less than 10 minutes.

The complexity of the set of rules obtained after the transformations is $O(E \times$`#dem2.2/1`$)$. A systematic manual analysis of the rules reveals that `#dem2.2/1` is bounded by the number of variables in scope at a program point, since the first argument of `#dem2` is a program point, and the second argument only takes the variables that can reach that program point via edges. We computed the average number of variables in scope (s) at each point using static analysis for each program, and the line with plus markers in Figure 4 shows the running times of set of rules with respect to $E \times s$. The resulting plot is almost linear as expected; we think that the deviations from linearity are due to the fact that the benchmarks do not exhibit worst-case time complexity.

Specialization after demand transformation reduces running time and memory usage by a constant factor, and the decision for when to stop unfolding affects the running time and memory usage. In Figures 4 and 5, *unfolding 1* denotes only unfolding predicates defined by one rule, and *unfolding 2* denotes unfolding predicates defined by possibly multiple rules with only one hypothesis.

Figure 4 shows the running time of the set of rules at different implementation stages. Unfolding 1 has the best running time, since it avoids duplicate inference for predicates defined by only one rule. On average, compared to the rules after demand transformation, specialization with unfolding 1 reduces running time by 17%.

Figure 5 shows the memory usage of the implementations at different implementation stages. We obtain the memory usage of generated Python implementations, since memory profiling for Python is very precise using Heapy[1]. As expected, all steps show a constant decrease in memory usage, and unfolding 2 uses the least memory, since it removes the most rules. On average, compared to the rules after demand transformation, specialization with unfolding 2 reduces memory usage by 26%.

Comparison with state-of-the-art top-down and bottom-up systems. We have shown that recursion conversion and hypothesis

[1] Available at http://guppy-pe.sourceforge.net

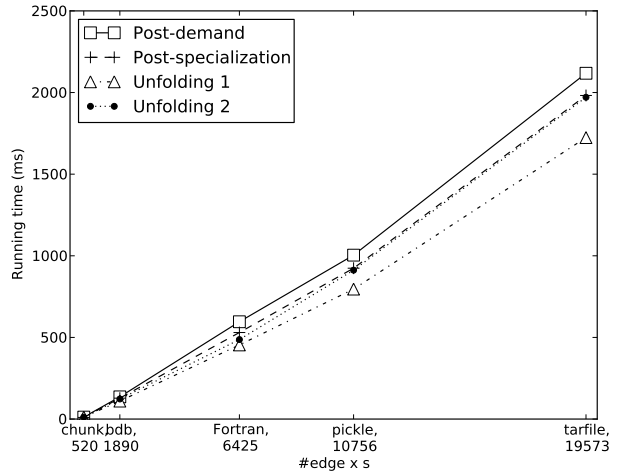

Figure 4. Running time of the implementation of rules in C++ at different implementation stages.

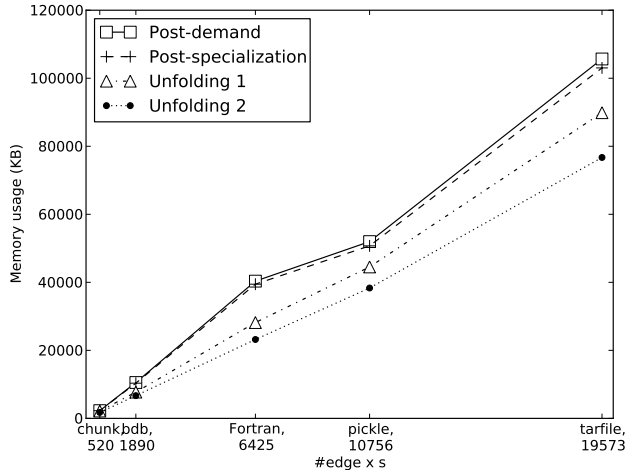

Figure 5. Memory usage of the implementation of rules in Python at different implementation stages.

permutation are important steps before demand transformation and specialization are applied for bottom-up computation. This is also true for top-down systems. A prominent top-down evaluation engine with tabling is XSB [24]. There is no known systematic analysis to find the best combination of form of recursion and hypothesis order for top-down evaluation in the literature, and we confirmed this by consulting the main developer of XSB [30].

We generated all recursion forms and hypothesis permutations for the running example, and manually ran and timed all benchmarks for all combinations in XSB. In general, the number of such combinations is exponential in program size. Among 32 possible combinations for our running example, 16 versions do not complete the benchmarks in less than 10 minutes, and all versions are slower than our generated code in C++. Note that our code generators are implemented for proof-of-concept, and they are not optimized for constants in contrast to the effort put into the development of a mature system like XSB. Figure 6 shows the running time of our generated code, the running time of the rules and query generated in Step 1 of Section 4 in XSB, and the running time of the manually found best version for XSB.

A bottom-up evaluation engine that has been used to solve large problems is bddbddb [15]. bddbddb does not employ any transformations for efficiency, but employs binary decision diagrams (BDDs) to store and manipulate the relations. It does not provide any complexity guarantees for a set of Datalog rules. We conducted experiments on bddbddb using both the original set of rules generated, and the rules yielded by our combination of transformations.

As expected, bddbddb shows asymptotically worse behavior on the original set of rules. On the final set of rules, we observed that the performance of bddbddb is highly dependent on the several options provided, especially the ordering of variables in the BDDs. We used the provided option of using machine learning to automatically find the best variable order, and also manually tried all 13 variable orderings possible. There is a large discrepancy between the results, using the worst variable ordering is up to 6 times slower, and using the automated variable ordering is up to 1.8 times slower than the manually obtained best result. The running times are shown in Figure 6.

Our experiments show that despite the large amount of effort spent to find the version of rules for minimum running time in XSB and bddbddb, the code automatically generated by our combination of transformations outperforms these systems. For all examples we have encountered, our method succeeds in finding the version of rules with the best complexity among all versions in less than a second. The complexity analysis provided by our method is confirmed by the actual running times, whereas such analysis is not available in the state-of-the-art systems compared.

7. Related work and conclusion

The design and implementation of graph query languages for program analysis has been studied extensively. These include languages for both static analysis and runtime monitoring. The study of programs as relational data has been first proposed by Linton [17] with a language called QUEL. However, early query languages such as QUEL did not allow recursion and showed poor performance due to lack of optimization. Several other languages such as JQuery [29] and ASTLOG [9] have demonstrated better performance, but they lack support for specifying path properties in forms of regular expressions with parameters. We compare our work with several languages and implementations below.

Manual implementations. One of the most popular program analysis tools is FindBugs [12], which is used to find bugs in Java programs. FindBugs only supports the specification of bug patterns via manual implementations. However, the 355 different types of bugs that can be found by the tool are well-documented, and out of the 17 common bugs described, we can express 16 of them in the language we use. The only one that cannot be expressed is for a bug that involves counting the number of occurrences of an edge; such aggregation operators are currently missing in the query language we use. Integrating the language we use and our method into a tool such as FindBugs would make it easier to add new analyses to the tool. Such analyses could then be clearly specified, and the efficient implementation can be automatically provided by our method.

Path queries. Regular path queries have been used in program analysis, e.g. in [10]. Parametric regular path queries [18] are regular-expression-like queries that allow the use of parameters, but do not support vertices and local parameters. Therefore, the language of parametric regular path queries is a strict subset of the query language proposed in [19] and used in this paper. The language we use also strictly contains Condate [28]. The query language of Blast [5] is also a path query language for software verification, however it operates only on a particular kind of graph

Programs	# of facts	Our method		XSB		bddbddb	
		Python	C++	Generated	Manual	Generated	Manual
chunk	367	57	12	36098	47	18354	454
bdb	926	664	110	-	215	145240	1027
RBAC	4701	2289	384	-	702	-	2296
Fortran	2890	2795	454	-	765	-	630
pickle	3201	4673	784	-	968	-	2477
tarfile	4300	10136	1724	-	3151	-	4416

Figure 6. Running time in milliseconds of implementations generated by our method, of the generated rules in XSB, and of the manually found best version of these rules in XSB, and similarly for bddbddb. - denotes incompletion in 10 minutes.

generated from the program, whereas the language we use can work with different graphs generated from the same program.

More powerful languages. PQL [14] is a more powerful program query language and is also transformed into Datalog rule. However, its implementation does not perform rule transformations as we do in this paper, or provide complexity guarantees. The resulting Datalog rules from a PQL query are evaluated using bddbddb, a BDD-based implementation of Datalog. However, as shown in Section 6, transformations affect the running time of the resulting rules significantly, and the BDD-based implementation of Datalog does not provide any complexity bounds and shows irregular behavior.

PQL is more powerful in the sense that it allows arbitrary query declarations which are Datalog-like, rather than only graph expressions. It is less expressive in the sense that it does not allow arbitrary variables on vertices for return or reuse. Since Datalog rules are generated from PQL, our combination of transformations for Datalog can be used in conjunction to provide better complexity with precise complexity guarantees. This also applies to systems that use Datalog directly to query source code such as CodeQuest [11]. Additionally, since our implementation first transforms graph queries into Datalog, we can easily add support for Datalog in the graph query language too.

Transformations for Datalog rules. Many transformations for Datalog rules have been studied for efficient implementation. We employ a bottom-up implementation strategy for obtaining time complexity guarantees as described in [20]. In contrast to bottom-up, top-down strategies have the advantage of being driven by the demand in the query, but do not provide complexity guarantees. For bottom-up implementations, demand driven computation can be mimicked by the magic-set transformation [4]. Our demand transformation has the same effect.

Specialization of rules [16] may also help reduce time complexity of rules. However, the effectiveness of demand transformation and specialization highly depends on the type of recursion in the rules. The effectiveness of demand transformation also depends on the ordering of hypotheses. Hristova [13] gives a detailed method for obtaining hypotheses orders that yield efficient implementations, and provides several applications of the method. However, it does not perform recursion conversion or specialization as studied in [22, 26]. Tekle et al. [26] combines recursion conversion and specialization for optimization, but does not employ demand transformation, and therefore can be ineffective when specialization alone is not sufficient. We use recursion conversion, hypothesis permutation, demand transformation, specialization, and program generation in order, to obtain both asymptotic speedups and reduce memory usage and running times by constant factors.

Conclusion and future work. The graph query language demonstrated with applications in this paper allows many queries to be written much more easily and clearly than in Datalog. This is shown in the running example, where a simple query corresponds to several Datalog rules. Our work is novel in (1) the use of a graph query language to effectively describe different program analyses, (2) the combination of transformations that optimizes the running time of the query, and automatically generates implementations with time complexity guarantees.

We have experimentally shown that the code generated by our method adheres to the calculated complexities, and outperforms state-of-the-art Datalog evaluation engines — XSB [24], a top-down evaluation engine, and bddbddb [15], a bottom-up evaluation engine that employs BDDs — even after significant manual effort to find the rules with the minimum running time for these engines. Therefore, using the graph query language in conjunction with our combination of transformations is a powerful tool for clearly specifying program analysis problems, and obtaining efficient implementations with precise complexity guarantees.

Future work includes further use of the graph query language and the implementation method for security and provenance applications, and possible extensions of the language.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] K. Ashcraft and D. R. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. of the 2002 IEEE Symp. on Security and Privacy*, pages 143–159, 2002.

[3] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proc. of the 30th annual ACM SIGPLAN - SIGACT Symp. on Principles of Programming Languages*, pages 97–105, 2003.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(1/2/3&4):255–299, 1991.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In *Proc. of the 11th Intl. Static Analysis Symp.*, pages 2–18, 2004.

[6] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proc. of the 21st European Conf. on Object-Oriented Programming*, pages 525–549, 2007.

[7] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proc. of the 11th Annual Network and Distributed System Security Symp.*, pages 171–185, 2004.

[8] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proc. of 12th USENIX Security Symp.*, pages 12–12, 2003.

[9] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proc. of the Conf. on Domain-Specific Languages*, page 18, 1997.

[10] O. de Moor, D. Lacey, and E. V. Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1-2):15–35, 2003.

[11] E. Hajiyev, M. Verbaere, and O. de Moor. *CodeQuest:* scalable source code queries with Datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, pages 2–27, 2006.

[12] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[13] K. Hristova. *From Rules to Efficient Algorithms for Cyber Trust Applications*. PhD thesis, Computer Science Department, SUNY Stony Brook, Dec 2007.

[14] M. S. Lam, M. Martin, V. B. Livshits, and J. Whaley. Securing web applications with static and dynamic information flow tracking. In *Proc. of the 2008 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation*, pages 3–12, 2008.

[15] 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 *Proc. of the 24th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–12, 2005.

[16] M. Leuschel. Logic program specialisation. In *Partial Evaluation*, pages 155–188, 1998.

[17] M. A. Linton. *Queries and Views of Programs Using a Relational Database System*. PhD thesis, EECS Department, University of California, Berkeley, Dec 1983.

[18] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation*, pages 219–230, 2004.

[19] Y. A. Liu and S. D. Stoller. Querying complex graphs. In *Proc. of the 7th Intl. Symp. on Practical Aspects of Declarative Languages*, pages 199–214, 2006.

[20] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 29(1), 2009.

[21] M. Meskes and J. Noack. The generalized supplementary magic-sets transformation for stratified Datalog. *Information Processing Letters*, 47(1):31–41, 1993.

[22] A. Pettorossi and M. Proietti. Program specialization via algorithmic unfold/fold transformations. *ACM Computing Surveys*, 30(3es):6, 1998.

[23] T. Rothamel and Y. A. Liu. Efficient implementation of tuple pattern based retrieval. In *Proc. of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 81–90, 2007.

[24] K. F. Sagonas, T. Swift, and D. S. Warren. XSB as a deductive database. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, page 512, 1994.

[25] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire Linux distribution for security violations. In *Proc. of the 21st Annual Computer Security Applications Conf.*, pages 13–22, 2005.

[26] K. T. Tekle, K. Hristova, and Y. A. Liu. Generating specialized rules and programs for demand-driven analysis. In *Proc. of the 12th Intl. Conf. on Algebraic Methodology and Software Technology*, pages 346–361, 2008.

[27] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *Proc. of the 28th Intl. Conf. on Software Engineering*, pages 172–181, 2006.

[28] E.-N. Volanschi. A portable compiler-integrated approach to permanent checking. *Automated Software Engineering*, 15(1):3–33, 2008.

[29] K. D. Volder. JQuery: A generic code browser with a declarative configuration language. In *Proc. of the 8th Intl. Symp. on Practical Aspects of Declarative Languages*, pages 88–102, 2006.

[30] D. S. Warren. Personal communication, 2009.