From Datalog Rules to Efficient Programs with Time and Space Guarantees

Yanhong A. Liu and Scott D. Stoller
State University of New York at Stony Brook

This paper describes a method for transforming any given set of Datalog rules into an efficient specialized implementation with guaranteed worst-case time and space complexities, and for computing the complexities from the rules. The running time is optimal in the sense that only useful combinations of facts that lead to all hypotheses of a rule being simultaneously true are considered, and each such combination is considered exactly once in constant time. The associated space usage may sometimes be reduced using scheduling optimizations to eliminate some summands in the space usage formula. The transformation is based on a general method for algorithm design that exploits fixed-point computation, incremental maintenance of invariants, and combinations of indexed and linked data structures. We apply the method to a number of analysis problems, some with improved algorithm complexities and all with greatly improved algorithm understanding and greatly simplified complexity analysis.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*constraint and logic languages*; D.3.4 [**Programming Languages**]: Processors—*code generation,optimization*; E.1 [**Data**]: Data Structures—*arrays, lists, queues, records*; E.2 [**Data**]: Data Storage Representations—*linked representations*; F.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; H.2.3 [**Information Systems**]: Database Management—*query languages* ; H.2.4 [**Information Systems**]: Systems—*query processing, rule-based databases*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Complexity analysis, Datalog, data structure design, incremental computation, indexing, indexed representations, linked representations, program transformation, optimization, recursion, tabling

1. INTRODUCTION

Many computational problems are most clearly and easily specified using relational rules. Examples include database queries and problems in program analysis, model

This work was supported in part by NSF under grants CCR-0204280, CCR-0306399, CCF-0613913, and CCR-9876058 and ONR under grants N00014-01-1-0109, N00014-02-1-0363, N00014-04-1-0722, and N00014-09-1-0651. This article is a revised and extended version of a paper that appeared in *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 172–183, Uppsala, Sweden, August 2003. Authors' address: Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400. Tel: (631)632-{8463,1627}, Email: {liu,stoller}@cs.sunysb.edu. Contact author: Y.A. Liu.

checking, and security analysis. Datalog [Ceri et al. 1990; Abiteboul et al. 1995] is an important rule-based language for specifying how new facts can be inferred from existing facts. It is sufficiently powerful for expressing many practical analysis problems, such as pointer analysis for programs.

While a Datalog program can be easily implemented in, say, a Prolog system, evaluated using various existing methods, or rewritten using methods such as magic sets to allow more efficient evaluation, such an implementation is typically for fast prototyping. This is partly due to relatively weak integration with the rest of the applications. Moreover, the running times of Datalog programs implemented using these methods can vary dramatically depending on the order of rules and the order of hypotheses in a rule, and even less is known about the space usage. Developing and implementing efficient algorithms specialized for any given set of rules and with time and space guarantees, and thus also easily pluggable into applications, is a nontrivial, recurring task.

This paper describes a powerful, fully automatable method for generating efficient specialized algorithms and implementations from Datalog rules. The heart of this paper consists of two main results:

—The first is a method that, given any set of Datalog rules, transforms them into an efficient specialized implementation that, given any set of facts, computes exactly the set of facts that can be inferred.

—The second is a method that computes the guaranteed worst-case time and space complexities of the implementation from the set of rules and allows easy simplification of the complexity formulas based on characterizations of the set of facts.

The running time is optimal in the sense that only useful combinations of facts that lead to all hypotheses of a rule being simultaneously true are considered, and each such combination is considered exactly once in constant time. For space complexity, our method separately analyzes the output space and the auxiliary space. The auxiliary space may sometimes be reduced using scheduling optimizations to eliminate some summands in the space complexity formula.

These two results are formally derived together using a systematic algorithm development method [Paige and Koenig 1982; Cai and Paige 1988; Paige 1989], generalized in this paper to support the design of necessary and more sophisticated data structures. The formal derivation starts with a fixed-point specification and has three steps: (i) transform the fixed-point expression into a loop that handles a single new fact in each iteration, (ii) replace expensive computations in the loop with efficient incremental operations, and (iii) design data structures, built from records, arrays, and linked lists, that efficiently support every incremental operation. The analysis of the complexities is based on a thorough understanding of the transformation process, reflecting the complexities of the implementation back to the rules.

We apply these results to a number of nontrivial analysis problems. We obtain improved algorithm complexities for some problems and gain a deeper understanding of all the algorithms. The complexity analysis based on the rules is quite easy, significantly easier than the ad hoc analysis done previously for individual problems.

The rest of the paper is organized as follow. Sections 2 and 3 introduce Datalog rules and our derivation approach, respectively. Sections 4 and 5 describe our formal derivation that involves (i) incremental computation of expensive set expressions and (ii) design of a combination of indexed and linked data structures, respectively. Section 6 presents complexity analysis and optimality of the derived complete algorithms. Section 7 describes extensions. Section 8 presents example applications. Section 9 discusses related work and concludes.

2. PROBLEM

We consider finite sets of relational rules of the form

$$P_1(X_{11}, ..., X_{1a_1}) \wedge ... \wedge P_h(X_{h1}, ..., X_{ha_h}) \rightarrow Q(X_1, ..., X_a) \tag{1}$$

where h is a finite natural number, each P_i (respectively Q) is a relation of finite number a_i (respectively a) of arguments, each X_{ij} and X_k is either a constant or a variable, and each variable in the arguments of Q must also be in the arguments of some P_i. If $h = 0$, then there is no P_i or X_{ij}, and each X_k must be a constant, in which case $Q(X_1, ..., X_a)$ is called a *fact*. For the rest of the paper, "rule" refers only to the case where $h \geq 1$, in which case each $P_i(X_{i1}, ..., X_{ia_i})$ is called a *hypothesis* of the rule, and $Q(X_1, ..., X_a)$ is called the *conclusion* of the rule.

Such rules and facts are captured exactly by Datalog [Ceri et al. 1990; Abiteboul et al. 1995], a database query language based on the logic programming paradigm. Recursion in Datalog allows queries not expressible in relational algebra or relational calculus.

Example. We use transitive closure of edges in a graph as a running example. An edge from a vertex `u` to a vertex `v` is represented by a fact `edge(u,v)`. The following two rules capture transitive closure, i.e., all pairs of vertices `u` and `v` such that there is a path from `u` to `v`.

```
edge(u,v) → path(u,v)
edge(u,w) ∧ path(w,v) → path(u,v)
```

The meaning of a set of rules and a set of facts is the least set of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the rules. The problem considered in this paper is to efficiently compute this set.

A variable that occurs multiple times in a hypothesis is called an *equal card*; it forces a fact that matches the hypothesis to have the same value in those argument positions. A variable that occurs only once in only one hypothesis and not in the conclusion of a rule is called a *wild card*; its name does not affect the meaning of the rule.

For ease of exposition, we give a formal derivation of the algorithms and complexities for rules with at most two hypotheses, where equal cards and wild cards may occur only in rules with one hypothesis, and where arguments of relations appear to be grouped and possibly reordered. We will see in Section 7 that rules with more hypotheses or with equal cards and wild cards in rules with multiple hypotheses can be easily reduced to rules with at most two hypotheses and where equal cards and wild cards occur only in rules with one hypothesis, and that this does not affect the complexity guarantees. We will see through the derivation that grouping and reordering of arguments do not affect the results either. Precisely, our formal derivation considers rules and facts of the following forms:

$$\begin{array}{ll} \text{form 2:} & \texttt{P1(X1s,Ys,C1s)} \wedge \texttt{P2(X2s,Ys,C2s)} \\ & \rightarrow \texttt{Q2(X1s,X2s,Y's,C3s)} \\ \text{form 1:} & \texttt{P(Zs,As)} \rightarrow \texttt{Q1(Z's,Bs)} \\ \text{form 0:} & \texttt{Q(Cs)} \end{array} \tag{2}$$

Each of `X1s`, `X2s`, `Ys`, `Y's`, `Zs`, and `Z's` abbreviates a group of variables. Each of `C1s`, `C2s`, `C3s`, `As`, `Bs`, and `Cs` abbreviates a group of constants. Variables in `Y's` and `Z's` are subsets of the variables in `Ys` and `Zs`, respectively. In form 2, variables in `Ys` are exactly those shared between the two hypotheses. Each variable or constant in a group may occur multiple times in the group, except for `X1s`, `X2s`, and `Ys` in the hypotheses in form 2. This exception ensures that there is no equal card in rules with two hypotheses. Also, there is no wild card in rules with two hypotheses, since each variable occurs in both hypotheses or in a hypothesis and the conclusion.

Note that different relation names in these forms may refer to the same relation. We use different names for different occurrences of relations so that in the description that follows, we can tell which one is from where. For similar reasons, we use different names for different groups of constants and variables.

3. APPROACH

We use a set-based language for the formal derivation and analysis. The language is based on SETL [Schwartz et al. 1986; Snyder 1990] extended with a fixed-point operation [Cai and Paige 1988]; we allow sets of heterogeneous elements and extend the language with pattern matching.

Primitive data types include sets, tuples, and maps, i.e., binary relations represented as sets of 2-tuples. Their syntax and operations on them are summarized in Figure 1.

$\{X_1 \text{ ... } X_n\}$	a set with elements X_1,...,X_n
$[X_1 \text{ ... } X_n]$	a tuple with elements X_1,...,X_n in order
$\{[X_1\ Y_1] \text{ ... } [X_n\ Y_n]\}$	a map that maps X_1 to Y_1, ..., X_n to Y_n
$\{\}$	empty set
exists X **in** S	whether set S is not empty and, if not, binding variable X to any element of S
$S+T,\ S-T$	union and difference, respectively, of sets S and T
S **with** X, S **less** X	$S+\{X\}$ and $S-\{X\}$, respectively
$S \subseteq T$	whether set S is a subset of set T
X **in** S, X **notin** S	whether or not, respectively, X is an element of set S
$\#S$	number of elements of set S
dom(M)	domain set[1] of map M, i.e., $\{X : [X\ Y]$ **in** $M\}$
$M\{X\}$	image set of X under map M, i.e., $\{Y : [X\ Y]$ **in** $M\}$

Fig. 1. Sets, tuples, maps, and operations on them.

We use the notation below for pattern matching a value against a *tuple pattern*, i.e., a tuple whose components may be constants or variables. It returns false if

X is not a tuple of length n, if any Y_i is a constant but the ith component of X is not the same constant, or if any Y_i and Y_j are a same variable but the ith and jth components of X are not a same constant; otherwise, it binds each Y_i that is a variable to the ith component of X and returns true.

$$X \textbf{ of } [Y_1 \ ... \ Y_n] \qquad \text{matching value } X \text{ against tuple pattern } [Y_1 \ ... \ Y_n] \qquad (3)$$

We use the notation below for set comprehension. Each variable Y_i enumerates elements of set S_i; for each combination of values of $Y_1, ..., Y_n$, if the value of Boolean expression Z is true, then the value of expression X forms an element of the resulting set.

$$\{X : Y_1 \textbf{ in } S_1, ..., Y_n \textbf{ in } S_n \mid Z\} \qquad \text{set comprehension} \qquad (4)$$

Each Y_i can also be a tuple pattern, in which case each enumerated element of set S_i is first matched against the pattern before expressions Z and X are evaluated. If $\mid Z$ is omitted, Z is implicitly the constant *true*.

LFP(S_0, F) denotes the least fixed point of F that includes set S_0. In other words, it is the smallest set S that satisfies the condition $S_0 \subseteq S$ and $F(S) = S$.

We use standard control constructs **while**, **for**, and **if**, and we use indentation to indicate scope. We abbreviate assignments of the form $X := X$ **op** Y as X **op** $:= Y$.

The fixed-point specification

Using the above language, we represent a fact of the form `Q(Cs)` using `[Q Cs]`, which abbreviates a tuple whose first component is a constant `Q` and whose other components are the constants in `Cs`; and we represent a hypothesis of the form `P(Xs,Cs)` using `[P Xs Cs]`, where `P` and the components in `Cs` are constants and those in `Xs` are variables.

Let `e0` be the set of all given facts. Since all rules of the same form are processed in the same way, we will describe the compilation method for only one rule of form 1 and one rule of form 2. Given any set of facts `R`, for the rule of form 1, let `e1(R)` be the set of facts `Q1(Z's,Bs)` such that `P(Zs,As)` is in `R`, i.e., `e1(R)` is the set of facts that can be inferred from `R` using the rule of form 1 once. For the rule of form 2, let `e2(R)` be the set of facts `Q2(X1s,X2s,Y's,C3s)` such that `P1(X1s,Ys,C1s)` and `P2(X2s,Ys,C2s)` are in `R`, i.e., `e2(R)` is the set of facts that can be inferred from `R` using the rule of form 2 once. That is,

```
e0    = {[Q Cs] : Q(Cs) in givenFacts}
e1(R) = {[Q1 Z's Bs] : [P Zs As] in R}
e2(R) = {[Q2 X1s X2s Y's C3s] :                    (5)
              [P1 X1s Ys C1s] in R and
              [P2 X2s Ys C2s] in R}
```

The meaning of the given set of rules and facts is

`LFP(e0,F)`, where `F(R) = R+e1(R)+e2(R)`. (6)

[1]Note the difference between the *domain* of an *argument of a relation* and the *domain set* of a *map*. The former is the set of possible values for the argument of the relation; the latter is the domain of the first component of the map.

Note that we allow sets to contain elements of different types, i.e., facts of different relations. Such union types allow simpler and clearer algorithm derivation at a high level before data structure design.

Compilation and analysis

Transforming a set of rules into an efficient implementation has three steps. Step 1 transforms the fixed-point specification into a **while**-loop. The idea is to perform a small update operation in each iteration. The fixed-point expression in (6) is equivalent to

`LFP({},F)`, where `F(R) = R+e0+e1(R)+e2(R)` (7)

and is transformed into the following loop. When it terminates, `R` is the result.

```
R := {}
while exists x in e0+e1(R)+e2(R)-R:                (8)
  R with:= x
```

Step 2 transforms expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression *exp* in the loop with a fresh variable, say E, and maintain the invariant $E = exp$ by inserting appropriate initializations and updates to E where variables in *exp* are initialized and updated, respectively. Step 3 designs appropriate data structures for representing each set so that operations on it can be implemented efficiently. The idea is to design sophisticated linked structures, whenever possible, based on how sets and set elements are accessed, so that each operation can be performed in worst-case constant time and with at most a constant (a small fraction) factor of overall space overhead. Note, however, to compile Datalog rules, indexed structures (arrays) must also be exploited extensively in order to achieve the best running time.

These three steps are called dominated convergence [Cai and Paige 1988], finite differencing [Paige and Koenig 1982; Paige 1986], and real-time simulation [Paige 1989; Cai et al. 1991], respectively, by Paige et al. Details of Steps 2 and 3 for computing (8) are described in the two subsequent sections. Step 2 is the driving force; Liu [Liu 2000] gives references to much work that exploits similar ideas, but the key idea here is to maintain appropriate auxiliary maps for efficient incremental computation. Step 3 is the enabling mechanism; the main difficulty here is that linked structures using based representations [Paige 1989; Cai et al. 1991] do not suffice, and sophisticated indexed structures must also be used extensively and set up carefully.

The complexity results are obtained by carefully bounding the numbers of facts actually used and produced by the rules rather than approximating them crudely using sizes of separate domains of arguments.

Correctness

Correctness of the transformations follows from the correctness of Paige et al.'s three steps, as proven in [Cai and Paige 1988; Paige and Koenig 1982; Paige 1986; 1989; Cai et al. 1991], the correctness of our use of auxiliary maps in Step 2, and the correctness of our extensions to Step 3. For Step 2, determining auxiliary maps is not trivial, but correctness of incremental computation using them follows from simple properties of maps. For Step 3, designing appropriate data structures is difficult, but correctness of the basic operations supported by the resulting data structures follows from simple properties of records, arrays, and linked lists. Correctness of the complexity results follows from the careful analysis in Section 6 of the sizes of the sets and maps used and the number of combinations of facts considered by the algorithm.

4. INCREMENTAL COMPUTATION

We transform (8) to compute expensive set expressions in the loop incrementally in each iteration. That is, we hold the values of expensive expressions in variables, initialize the values of these variables for the initial value of `R` before entering the loop, use the values of these variables where the values of the corresponding expressions are needed, and update the values of these variables incrementally as the value of `R` is updated. This eliminates repeated recomputations of the expensive expressions in the loop.

Identifying expensive subexpressions and auxiliary maps

A set expression is expensive if it is a set former (4) or involves high-level set operations such as union and difference. Therefore, the expensive expressions in (8) are `e0`, `e1(R)`, `e2(R)`, and `e0+e1(R)+e2(R)-R`. We use fresh variables `E0`, `E1`, `E2`, and `W` to hold their respective values and thus have the following invariants:

```
E0 = e0 = as in (5)
E1 = e1(R) = as in (5)
E2 = e2(R) = as in (5)                          (9)
W = e0+e1(R)+e2(R)-R = E0+E1+E2-R
```

As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant for `E1`. Clearly, `E1` can be initialized to `{}` at the initialization `R:={}`. `E1` can also be updated easily at the update `R with:=x` as follows: if `x` is of the form `[P Zs As]`, then `E1` is updated by adding the corresponding `[Q1 Z's Bs]` if it is not already in `E1`, otherwise nothing needs to be done.

For set expressions such as `e2(R)` formed by joining elements from sets, efficient incremental computation may require maintaining auxiliary maps. To update `E2` incrementally with the update `R with:= x`, if `x` is of the form `[P1 X1s Ys C1s]`, then we consider all matching tuples `[P2 X2s Ys C2s]` in `R` and add the corresponding tuple `[Q2 X1s X2s Y's C3s]` to `E2`. To form the tuples to add, we need to efficiently find the appropriate values of `X2s`, so we maintain an auxiliary map that maps variables in `Ys` to variables in `X2s` for all `[P2 X2s Ys C2s]` in `R`. We store this map in variable `P2YsX2s`, indicating that it is built from `P2` and maps variables in `Ys` to variables in `X2s`:

```
P2YsX2s = {[Ys X2s] : [P2 X2s Ys C2s] in R}          (10)
```

Note that, if arguments of the hypothesis `P2(X2s,Ys,C2s)` start with variables in `Ys`, followed by variables in `X2s`, and possibly followed by constants, then, by definition of `P2YsX2s`, the tuples in `P2YsX2s` are the same as the initial part of the arguments of the facts of `P2` in `R`. Therefore, `P2YsX2s` is not needed and facts of `P2` that are in `R` can be used in place of `P2YsX2s`. Having no shared variables, i.e., `Ys` being empty, is a trivial case of this.

Symmetrically, if `x` is a tuple of `P2`, we need to consider each matching tuple of `P1` and add the corresponding tuple of `Q2` to `E2`. To efficiently form the tuples to add, we maintain

```
P1YsX1s = {[Ys X1s] : [P1 X1s Ys C1s] in R}          (11)
```

We call the first set of arguments in an auxiliary map the *anchor* and the second set of arguments the *non-anchor*. Being able to directly find only the matching tuples allows us to consider only combinations of facts that make both hypotheses simultaneously true and to consider each combination only once.

Example. For the transitive closure example, `E0`, `E1`, `E2`, and `W` are defined straightforwardly. We also maintain the auxiliary map `edgewu = {[w u] : [edge u w] in R}`, which is an inverse of `edge`. Auxiliary map `pathwv` is not needed, since facts of `path` that are in `R` can be used in place of `pathwv`.

Initializations and incremental updates

Variables holding the values of expensive subcomputations and auxiliary maps are initialized together with the assignment `R:={}` and updated incrementally together with the assignment `R with:= x` in each iteration.

By definitions (9), (10) and (11), when `R` is `{}`, we have:

```
E0 = {[Q Cs] : Q(Cs) in givenFacts}
E1 = {}
E2 = {}
W = E0 = {[Q Cs] : Q(Cs) in givenFacts}              (12)
P2YsX2s = {}
P1YsX1s = {}
```

and when `x` is added to `R` in the loop body, these variables can be updated as follows. Note that `P2YsX2s{Ys}` denotes the set of `X2s`'s such that `[Ys,X2s]` is in `P2YsX2s`, and `P2YsX1s{Ys}` is similar; they are examples of the image set operation defined in Figure 1.

```
if x of [P Zs As]:
  E1 with:= [Q1 Z's Bs]
  if [Q1 Z's Bs] notin R: W with:= [Q1 Z's Bs]
if x of [P1 X1s Ys C1s]:
  E2 +:= {[Q2 X1s X2s Y's C3s]: X2s in P2YsX2s{Ys}}
  W  +:= {[Q2 X1s X2s Y's C3s]: X2s in P2YsX2s{Ys}
           | [Q2 X1s X2s Y's C3s] notin R}
  P1YsX1s with:= [Ys X1s]                            (13)
if x of [P2 X2s Ys C2s]:
  E2 +:= {[Q2 X1s X2s Y's C3s]: X1s in P1YsX1s{Ys}}
  W  +:= {[Q2 X1s X2s Y's C3s]: X1s in P1YsX1s{Ys}
           | [Q2 X1s X2s Y's C3s] notin R}
  P2YsX2s with:= [Ys X2s]
W less:= x
```

Adding these initializations and updates and using `W` in place of `e0+e1(R)+e2(R)-R` in (8), we obtain the following complete code. It is easy to see that `W` serves as the workset.

```
initialize using (12)
R := {}
while exists x in W:                                  (14)
  update using (13)
  R with:= x
```

Eliminating dead code

To compute the result `R`, only `W`, `P2YsX2s`, and `P1YsX1s` are needed. So `E0`, `E1`, and `E2` are dead. Eliminating them from (14), we obtain the following algorithm:

```
W := {[Q Cs] : Q(Cs) in givenFacts}
P2YsX2s := {}
P1YsX1s := {}
R := {}
while exists x in W:
  if x of [P Zs As]:
    if [Q1 Z's Bs] notin R: W with:= [Q1 Z's Bs]
  if x of [P1 X1s Ys C1s]:
    W +:= {[Q2 X1s X2s Y's C3s]: X2s in P2YsX2s{Ys}      (15)
              | [Q2 X1s X2s Y's C3s] notin R}
    P1YsX1s with:= [Ys X1s]
  if x of [P2 X2s Ys C2s]:
    W +:= {[Q2 X1s X2s Y's C3s]: X1s in P1YsX1s{Ys}
              | [Q2 X1s X2s Y's C3s] notin R}
    P2YsX2s with:= [Ys X2s]
  W less:= x
  R with:= x
```

Cleaning up

Finally, the code is cleaned up to contain only uniform element-level operations for data structure design. First, we decompose `R` into `Ri`'s, where each `Ri` is for a single relation that occurs in the rules. Similarly, we decompose `W` into `Wi`'s. For a relation `Qi` that occurs in the conclusion of a rule, we write `RQi` and `WQi` instead of `Ri` and `Wi`. We also eliminate relation names from the first component of tuples, and transform the **while**-clause and pattern matching clauses to iterate over `Wi`'s.

Example. For the transitive closure example, after representing `R` as `Redge` and `Rpath` and representing `W` as `Wedge` and `Wpath`, we obtain the algorithm in Figure 2. Note that the first two cases in (15) are merged for this example since both rules for this example have a hypothesis that contains `edge`. Also, `Rpath`, i.e., facts of `path` that are in `R`, is used in place of an auxiliary relation `pathwv`.

Then, we do the following three sets of transformations.

(i) Transform set-level operations (unions here) into loops that use element-level operations. Specifically, replace addition of a set $\{X : Y \textbf{ in } S \,|\, Z\}$ to a set T with

```
Wedge := {[u v] : edge(u,v) in givenFacts}
Wpath := {}
edgewu := {} //inverse map for edge
Redge := {}
Rpath := {}
while Wedge != {} or Wpath != {}:
  while exists [u,w] in Wedge:
    if [u w] notin Rpath: Wpath with:= [u w]        //rule 1
    Wpath +:= {[u v]: v in Rpath{w} | [u v] notin Rpath}
                             //rule 2, and use of Rpath
    edgewu with:= [w u]      //update of inverse map
    Wedge less:= [u,w]
    Redge with:= [u,w]
  while exists [w,v] in Wpath:
    Wpath +:= {[u v]: u in edgewu{w} | [u v] notin Rpath}
                             //rule 2, and use of inverse map
    Wpath less:= [u,w]
    Rpath with:= [u,w]
```

Fig. 2. Transitive closure algorithm after decomposing relations R and W.

a **for**-loop that adds elements one at a time: **for** Y **in** S : **if** Z : T **with** := X. Additionally, replace enumeration of facts Q(Cs) from givenFacts with reading of facts Q(Cs) from input one at time, denoted **while** read Q(Cs).

(ii) Replace tuples and tuple operations with maps and map operations. Specifically, replace tuples of more than two components with tail nested tuples of two components, e.g., $[X\ Y\ Z\ V]$ becomes $[X\ [Y\ [Z\ V]]]$. Then, for each 2-tuple Z and map M, replace **while exists** Z **in** M : ... Z... with

```
while exists $X$ in dom($M$) :
  while exists $Y$ in $M\{X\}$ :
    ... $[X\ Y]$...
```

and replace **for**-loops similarly. Finally, replace $M \neq \{\}$ with **dom**$(M) \neq \{\}$; replace $[X\,Y]$ **notin** M with X **notin dom**(M) **or** Y **notin** $M\{X\}$, where **or** uses short-circuit semantics; replace M **with** := $[X\,Y]$ with **if** X **notin dom**(M) : $M\{X\} := \{\}$ followed by $M\{X\}$ **with** := Y; and replace M **less** := $[X\,Y]$ with $M\{X\}$ **less** := Y followed by **if** $M\{X\} = \{\}$: **dom**(M) **less** := X.

(iii) Make all element-level updates easy by testing membership first. Specifically, replace S **with** := X with **if** X **notin** S : S **with** := X and replace S **less** := X with **if** X **in** S : S **less** := X. There are three exceptions. First, removal from a Wi does not need the additional test, since the removed element is retrieved from Wi. Second, addition to Ri does not need the additional test, since elements are moved from Wi to Ri one at a time and each element is put into Wi, and thus Ri, only once. Third, addition to PiYsXis does not need the additional test if the corresponding hypothesis Pi(Xis,Ys,Cis) has no constant arguments, since the element to add corresponds to an element in some Wj and each element is put into Wj, and thus the corresponding element put into PiYsXis, only once.

Example. For the transitive closure example, after applying the three sets of transformations, we obtain the algorithm in Figure 3. The comments show the corresponding statements in Figure 2.

```
while read edge(u,v):                         //Wedge :=...
  if u notin dom(Wedge): Wedge{u} := {}       //
  if v notin Wedge{u}: Wedge{u} with:= v      //
Wpath := {}
edgewu := {}
Redge := {}
Rpath := {}
while dom(Wedge) != {} or dom(Wpath) != {}:   //while Wedge!={} or...
  while exists u in dom(Wedge):         //while exists [u,w] in Wedge
    while exists w in Wedge{u}:         //
      if u notin dom(Rpath) or w notin Rpath{u}://if [u w] notin Rpath
        if u notin dom(Wpath): Wpath{u} := {}     //Wpath with:=[u w]
        if w notin Wpath{u}: Wpath{u} with:= w    //
      for v in Rpath{w}:                               //Wpath +:=...
        if u notin dom(Rpath) or v notin Rpath{u}:     //
          if u notin dom(Wpath): Wpath{u} := {}        //
          if v notin Wpath{u}: Wpath{u} with:= v       //
      if w notin dom(edgewu): edgewu{w} := {}      //edgewu with:=...
      edgewu{w} with:= u                           //
      Wedge{u} less:= w                      //Wedge less:= [u,w]
      if Wedge{u} = {}: dom(Wedge) less:= u  //
      if u notin dom(Redge): Redge{u} := {}      //Redge with:= [u,w]
      Redge{u} with:= w                          //
  while exists w in dom(Wpath):         //while exists [w,v] in Wpath
    while exists v in Wpath{w}:         //
      for u in edgewu{w}:                              //Wpath +:=...
        if u notin dom(Rpath) or v notin Rpath{u}:     //
          if u notin dom(Wpath): Wpath{u} := {}        //
          if v notin Wpath{u}: Wpath{u} with:= v       //
      Wpath{u} less:= w                      //Wpath less:= [u,w]
      if Wpath{u} = {}: dom(Wpath) less:= u  //
      if u notin dom(Rpath): Rpath{u} := {}      //Rpath with:= [u,w]
      Rpath{u} with:= w                          //
```

Fig. 3. Transitive closure algorithm after all clean-up transformations.

Correctness

The derivation of the algorithm in (15) from (6) explains how to choose the iteration step, what invariants and auxiliary values to maintain, and the exact transformations for iteration, initializations, incremental updates, and dead-code elimination. Each transformation is easily seen to preserve correctness, and together the derivation provides a proof for the correctness of the resulting algorithm. That is, the derived algorithm in (15) correctly computes the sets of facts specified by least fixed-point specification in (6). Additionally, each of the cleaning-up transformations is easily seen to preserve correctness. Therefore, we have the following theorem.

Theorem 1. The algorithm derived using transformations for incremental computation is correct with respect to the least fixed-point specification in (6).

5. DATA STRUCTURE DESIGN

We describe how to guarantee that each set operation in the cleaned-up version of (15) takes worst-case $O(1)$ time. All operations are among the following kinds: set initialization $S := \{\}$, computing domain set $\mathbf{dom}(M)$, computing image set $M\{X\}$, emptiness test $S = \{\}$ and $S \neq \{\}$, element retrieval in **for** X **in** S and **while exists** X **in** S, membership test X **in** S and X **notin** S, and element addition S **with** X and deletion S **less** X. We use *associative access* to refer to membership test (X **in** S and X **notin** S) and computing image set ($M\{X\}$). Such an operation requires the ability to locate an element (X) in a set (S or $\mathbf{dom}(M)$).

Based representations

Consider using a singly linked list for each set, for the domain set of each map, and for each of the image sets of each map. Let each element in a domain set linked list contain a pointer to its image set linked list. In other words, represent a set as a linked list, and represent a map as a linked list of linked lists. It is easy to see that, if associative access can be done in worst-case $O(1)$ time, so can all other primitive operations. To see this, note that computing a domain set or an image set simply returns a pointer to the set; retrieving an element from a set only needs to locate any element in the set; and adding or deleting an element from a set can be done in constant time after doing an associative access. An associative access would take linear time if a linked list is naively traversed to locate an element. A classical approach to address this problem is to use hash tables [Aho et al. 1983] instead of linked lists. However, this gives average, rather than worst-case, $O(1)$ time for each operation, and has the overhead of computing hashing-related functions for each operation.

Paige et al. [Paige 1989; Cai et al. 1991] describe a technique for designing linked structures that support associative access in worst-case $O(1)$ time with little space overhead for a general class of set-based programs. Consider a piece of code that retrieves elements from set W and tests whether such an element is in set S:

```
for $X$ in $W$, or while exists $X$ in $W$
   ...$X$ in $S$..., or ...$X$ notin $S$...,
   or ...$M\{X\}$... where the domain set of $M$ is $S$
```

We want to locate value X in set S after it has been located in set W. The idea is to use a set B, called a *base*, to store values for both W and S, such that retrieval of a value from W also locates the value in S. Base B is a set (this set is only conceptual) of records, with a K field storing the key (i.e., value).

—Set S is represented using an S field of B: records of B whose keys belong to S are connected by a linked list whose links are stored in the S field; records of B whose keys are not in S store a special value for undefinedness in the S field.

—Set W is represented as a separate linked list of pointers to records of B whose keys belong to W.

Thus, an element of S is represented as *a field in* the record, and S is said to be *strongly based* on B; an element of W is represented as *a pointer to* the record, and W is said to be *weakly based* on B. Figure 4 shows the based representation for set W containing elements x_1 and x_3 and set S containing elements x_1 and x_2. Such a representation allows an arbitrary number of weakly based sets but only a constant number of strongly based sets. Essentially, base B provides a kind of indexing to elements of S starting from elements of W.

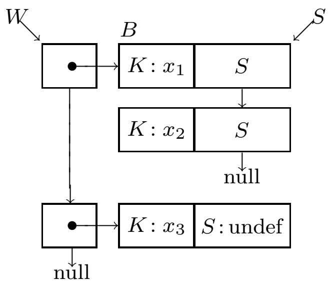

Fig. 4. Based representation for sets S and W using base B.

However, often a non-constant number of sets must be strongly based for constant-time associative access [Liu et al. 2001; Liu and Yu 2002; Liu et al. 2004], and this is particularly the case here for compiling general forms of rules. Specifically, in the cleaned-up version of (15), there is associative access in the domain of each component of the

(i) result sets `RQi`'s and worksets `WQi`'s for relations `Qi`'s that occur in the conclusions of rules, e.g., the result set `Rpath` and workset `Wpath` in Figure 3, for testing whether a fact of `Qi` to be added to `WQi` is already in `RQi` or `WQi`,

(ii) anchors of the auxiliary maps `PiYsXis`'s, i.e., the components corresponding to `Ys`'s in `PiYsXis`'s, e.g., the domain set of `edgewu` in Figure 3, by the image set operations `PiYsXis{Ys}`'s, and

(iii) auxiliary maps `PiYsXis`'s if the corresponding hypothesis `Pi(Xis,Ys,Cis)` has constant arguments, for testing whether a tuple to be added to `PiYsXis` is already in it.

Since each value accessed in the domain of a non-last component yields an image set for the domain of the next component whose values need to be accessed efficiently again, and there are a non-constant number of values in the domain of a component, these non-constant number of image sets can not be all strongly based directly on the set of possible domain values. Therefore, based representations do not apply. Nevertheless, we may extend them to use arrays for all the non-constant numbers of image sets, as described below. This guarantees worst-case constant running time for each operation.

Data structures

The data structures need to support the three kinds of associative access (i) to (iii) described above and the following two kinds of element retrieval. Note that an associative access of kind (iii) is not needed if the relation `Pi` from which `PiYsXis` is built does not appear in any conclusion, because in that case, every tuple `[Ys Xis]` added to `PiYsXis` is built from a unique given fact `Pi(Xis,Ys,Cis)` and thus must be new. Element retrieval is by traversals in the domain of each component of the

(i) worksets `Wi`'s, by the nested **while**-loops transformed from the single **while**-loop in (15), and

(ii) non-anchors of the auxiliary maps `PiYsXis`'s, by the nested **for**-loops in the cleaned-up version of (15) that add elements to the worksets `Wi`'s.

We describe a uniform method for representing all these sets and maps, using an array for each non-constant number of sets that have associative access, a linked list for each set that is traversed by loops, and both an array and a linked list when both kinds of operations are needed.

Consider all domains from which arguments of relations take values. For each domain `D`, we map the values in `D` one-to-one to the integers from 1 to `#D`, and use these integers to refer to the values in `D`. Recall that `Qi`'s denote relations that occur in the conclusions of rules. We represent `RQi`'s, `WQi`'s and other `Wi`'s, and `PiYsXis`'s, respectively, as follows.

—Each `RQi` of, say, `a` components, is represented using an `a`-level nested array structure. The first level is an array indexed by values in the domain of the first component of `RQi`; the `k`-th element of the array is `null` if there is no tuple of `RQi` whose first component has value `k`, and otherwise is `true` if `a=1`, and otherwise is recursively an (`a-1`)-level nested array structure for the remaining components of tuples of `RQi` whose first component has value `k`.

—Each `WQi` is represented the same as `RQi` with two additions. First, for each array, we add a linked list linking indices of non-null elements of the array. Second, to each linked list, we add a tail pointer, i.e., a pointer to the last element, to form a queue. We combine the array, the head of the linked list, and the tail pointer in a record. Each other `Wi` is represented simply as a nested queue structure (without the underlying arrays), one level for each component of `Wi`, linking the elements (which correspond to indices of the arrays) directly.

—Each `PiYsXis` for which associative access of kind (iii) is needed uses a nested array structure as `RQi` and `WQi` do and additionally linked lists (without the tail pointers) for each component of the non-anchor as `WQi` does. Each other `PiYsXis` uses a nested array structure only for the anchor, where elements of arrays for the last component of the anchor are each a nested linked-list structure (without the tail pointers or the underlying arrays) for the non-anchor. Finally, if an `Ri` is used in place of an `PiYsXis`, the corresponding data structure must be imposed on `Ri`.

Note that we did not discuss representations for relations `Ri`'s that do not occur in the conclusion of any rule and are not used in place of any auxiliary map. These sets contain only given facts, not newly inferred facts. They are not used in any way by our derived algorithms, except that their elements are simply taken from the given facts via the `Wi`'s. Elements of `RQi`'s and other `Ri`'s could be linked together as we do for `WQi`'s and other `Wi`'s if these result sets need to be traversed in subsequent computations.

A small natural improvement is to avoid using completely separate data structures for the different kinds of tuples in `Ri`'s, `Wi`'s, and `PiYsXis`'s. For all kinds of tuples whose first components are from the same domain, we use a single 1st-level array of records, as a base, for the domain, and use a field for each kind of tuples that shares the 1st-level array. This does not change the asymptotic complexities but allows the use of a single indexing operator to locate the first component of multiple tuples that are always accessed next to each other, e.g., `Ri` and `Wi` in each of the three cases of (15), and `P2YsX2s` and `P1YsX1s` in each of the last two cases of (15). This also allows all the data structures to fall back to completely based representations when there is no associative access into a non-constant number of sets.

Example. For the transitive closure example, we use numbers from 1 to `#vertex` to refer to the vertices. A base for the domain of all vertices is used, since both arguments of both `edge` and `path` are from this domain. The resulting data structure is explained below and depicted for a small graph in Figure 5, where the edges drawn with solid lines are already processed and the edges drawn with dashed lines are not yet processed.

Elements of the base are stored in an array `vertex` indexed by the vertices, for efficient access of the first component of `Rpath`, `Wpath`, and `edgewu`. Each element `u` of the base array is a record of six fields.

An `RpathArray` field of `u` is for `Rpath`; it is null if no element of `Rpath` starts with `u` and otherwise is an array for the second component of `Rpath`, indexed by the vertices and whose element at `v` is true if `[u,v]` is an element of `Rpath` and null otherwise. An `RpathList` field of `u` is a linked list of indices of vertices `v` for all tuples `[u,v]` in `Rpath`.

A similar `WpathArray` field of `u` is used for `Wpath`. A linked list `Wpath` with tail pointer is used to link indices of the base array elements whose `WpathArray` field is not null. A `WpathQueue` field of `u` is a linked list with tail pointer linking indices of non-null elements of the array in `WpathArray`.

A linked list `Wedge` with tail pointer is used to link indices of vertices in the first component of `Wedge`. A `WedgeQueue` field of `u` is a linked list of indices of successor vertices of `u`.

An `edgewuList` field of `u` is used for the inverse map `edgewu`; it is a linked list of indices of predecessor vertices of `u`.

Correctness

In the resulting data structures, we can easily observe the following. First, the nested array structure for each `RQi` allows each component in a result tuple to be accessed in worst-case constant time. Second, the nested queue structure for each `Wi` plus nested array structure for each `WQi` allow each element retrieval in a traversal by a while-loop and each element access, addition, and deletion to be

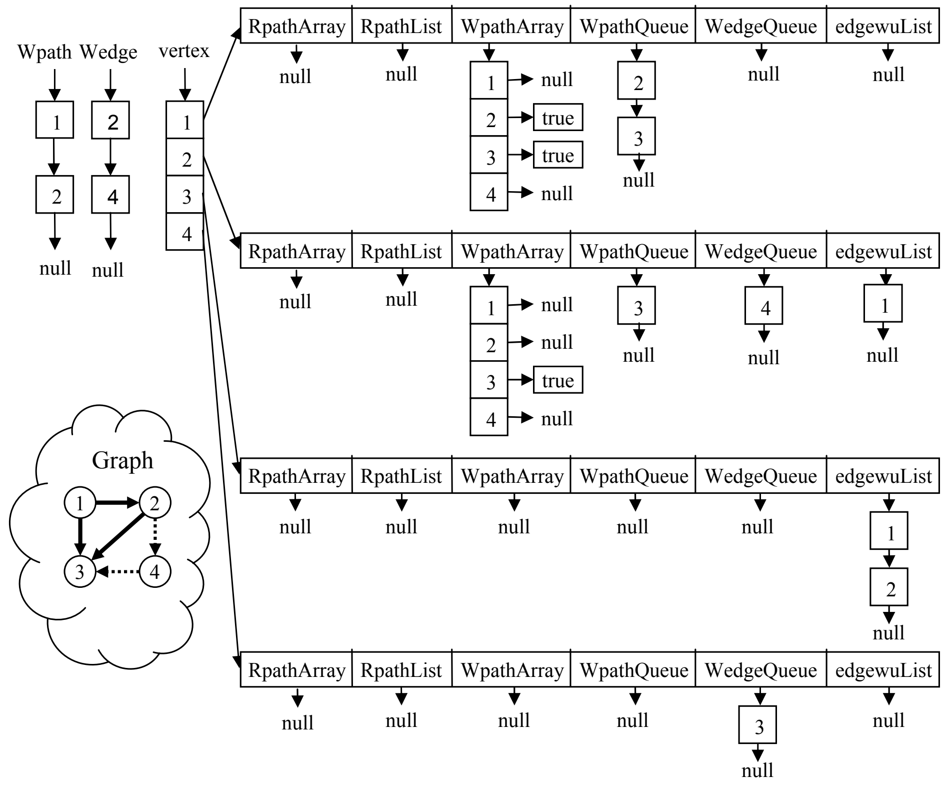

Fig. 5. Data structure for the transitive closure algorithm. To reduce clutter, tail pointers for queues in the `WpathQueue` and `WedgeQueue` fields are not shown.

done in worst-case constant time. Third, the nested array structure for `PiYsXis` plus nested linked-list structure for the non-anchor of `PiYsXis` allows each needed associative access, element retrieval using a for-loop, and element addition to be done in worst-case constant time. Therefore, we have the following theorem.

Theorem 2. The data structures designed for all `Ri`'s, `Wi`'s, and `PiYsXis`'s allow each operation in the algorithm derived using transformations for incremental computation to be done in worst-case constant time.

Time and space trade-offs

When elements in a set are sparse over a domain, array representations may result in non-optimal use of space. Note that initialization of the arrays does not affect the time complexity, as per the note in [Aho et al. 1983, Exercise 2.12]. When a set over a domain is sparse, we could use linked lists instead of arrays for accessing the set elements. This makes the space usage for this domain optimal but incurs an extra factor of the length of the lists for the time complexity. When worst-case time is not a concern, one could also use hash tables in place of arrays or linked lists, yielding another set of trade-offs involving also the overheads of hashing.

6. DETERMINING COMPLEXITY

We describe how to compute time and space complexities precisely from the rules, and express the complexities in terms of characterizations of the facts. The size of each rule is considered a constant. The idea is to analyze precisely the number of facts actually processed, avoiding approximations that use only the sizes of individual argument domains.

Size parameters and basic constraints

We use `P.i` to denote the projection of `P` on its `i`-th argument. We use `P.I`, where `I` = `{i1,i2,...,ik}`, to denote the projection of P on its `i1`-th, `i2`-th, ..., and `ik`-th arguments.

The analysis uses the following sizes to characterize the set of given facts, called *relation size*, *domain size*, *argument size*, and *relative argument size*, respectively:

—`#P`: the number of facts that actually hold for relation `P`.

—`#D(P.i)`: the size of the domain from which `P.i` takes its value.

—`#P.i`: the number of different values that `P.i` can actually take.
`#P.I`: the number of different combinations of values that elements of `P.I` together can actually take. For `I` $= \emptyset$, we take `#P.I` $= 1$.

—`#P.i/j`: the maximum number of different values that `P.i` can actually take for each possible value of `P.j`, where `i` $\neq$ `j`.
`#P.I/J`: the maximum number of different combinations of values that elements of `P.I` together can actually take for each possible combination of values of elements of `P.J`, where `I` $\cap$ `J` $= \emptyset$. For `I` $= \emptyset$, we take `#P.I/J` $= 1$. For `J` $= \emptyset$, we take `#P.I/J` = `#P.I`.
If `j` or an element of `J` can take on only a particular constant value, say `c`, we specify that by following `j` or the element of `J`, respectively, with `=c`.

Example. For the transitive closure example, `#edge` is the number of pairs in relation `edge`, i.e., the number of edges in the graph; `#D(edge.1)` is the number of vertices; `#edge.1` is the number of vertices that are sources of edges; `#edge.1/2` is the maximum number of predecessors of a vertex, i.e., the maximum in-degree of vertices; and `#edge.1/2=c` is the number of predecessors of vertex `c`.

It is easy to see that the following basic constraints hold:

`#P` = `#P.{1,...,a}` for relation `P` of `a` arguments
`#P.i` $\leq$ `#D(P.i)`
`#P.I` $\leq$ `#P.J` for `I` $\subseteq$ `J`
`#P.(I` $\cup$ `J)` $\leq$ `#P.I` $\times$ `#P.J/I` and `#P.J/I` $\leq$ `#P.J` for `I` $\cap$ `J` $= \emptyset$

These imply commonly used constraints, including in particular

`#P` $\leq$ `#D(P.1)` $\times \ldots \times$ `#D(P.a)`

for relation `P` of `a` arguments, which is especially useful when `#P` is not an input parameter, i.e., when `P` occurs in the conclusion of a rule.

Example. For the transitive closure example, let `vertex` be the domain of the arguments of `edge`, and thus also the domain of the arguments of `path`. We have

$$\begin{aligned}
&\texttt{\#path.2/1} \le \texttt{\#path.2} \le \texttt{\#D(path.2)} = \texttt{\#vertex}\\
&\texttt{\#path} \le \texttt{\#D(path.1)} \times \texttt{\#D(path.2)} = \texttt{\#vertex}^2\\
&\texttt{\#edge.1/2} \le \texttt{\#edge.1} \le \texttt{\#D(edge.1)} = \texttt{\#vertex}\\
&\texttt{\#edge} \le \texttt{\#D(edge.1)} \times \texttt{\#D(edge.2)} = \texttt{\#vertex}^2
\end{aligned}$$

Time complexity and optimality

In our derived algorithms, each fact is added to `W` once and then moved from `W` to `R` once. Each fact that makes the hypothesis of a rule of form 1 true and each combination of facts that makes both hypotheses of a rule of form 2 simultaneously true is considered exactly once, called a *firing* of the corresponding rule. To see that each combination of facts that makes both hypotheses `P1` and `P2` of a rule simultaneously true is considered only once, note that the auxiliary map entry for a fact `f` of `P1` or `P2` is built after retrieving `f` from a workset and used afterwards. So, a fact `f1` of `P1` combines once with each fact of `P2` retrieved before `f1` is retrieved, and each fact of `P2` retrieved after `f1` is retrieved combines once with `f1`.

It is therefore easy to see that the time complexity is the total number of firings of all rules, analyzed below. Since each firing as defined above may imply a new fact as an instance of the conclusion, it must, in general, be considered at least once. In this sense the running times of our derived algorithms are optimal.

For each rule `r`, let `r.#firedTimes` denote the total number of times `r` is fired. Use `IXs` to denote the set of indices of arguments `Xs` in a hypothesis. For a rule of form 1 in (2), we have

$$\texttt{r.\#firedTimes} = \texttt{\#P}. \tag{16}$$

For a rule of form 2 in (2), we have

$$\begin{aligned}\texttt{r.\#firedTimes} \le \texttt{min(\#P1} \times \texttt{\#P2.IX2s/IYsC2s},\\ \texttt{\#P2} \times \texttt{\#P1.IX1s/IYsC1s).}\end{aligned} \tag{17}$$

where `IYsCis` includes the constant values for the `Cis` components.

Consider any given set of rules. Let characteristics of facts be given in terms of the four kinds of sizes defined above, and consider the constraints on these sizes described above. The total time complexity is the sum of `#firedTimes` over all rules, minimized symbolically with respect to the given sizes and the constraints. In particular, if a relative argument size is needed but not given, we use the corresponding non-relative argument size; if an argument size `#P.I` is used but not given, we use the minimum of (i) the product of domain sizes for arguments of `P` that are in `I` and (ii) the argument size of `P` for arguments that are a superset of `I`, if given.

Example. For the transitive closure example, the time complexity is the sum of `#edge` for the first rule and $\texttt{min(\#edge}\times\texttt{\#path.2/1,\#path}\times\texttt{\#edge.1/2)}$ for the second rule. When only parameters `#edge` and `#vertex` are given, this sum is bounded by $\texttt{min(\#edge}\times\texttt{\#vertex,\#vertex}^3\texttt{)}$ based on the constraints above. Simplifying it based on $\texttt{\#edge} \le \texttt{\#vertex}^2$, we obtain the worst-case time complexity $O(\texttt{\#edge}\times\texttt{\#vertex})$.

Note that in (17), both ways of bounding the number of firings are excellent for presenting and understanding the time complexity, because they are precisely defined and they capture the worst-case time more tightly than in conventional complexity analysis. The symbolic minimization is just for expressing the complexity using conventional input sizes, as used typically in conventional complexity analysis, although it generally gives looser bounds. Also, such symbolic minimization is difficult to automate since the formulas are not linear.

Example. Consider the second rule in the transitive closure example. The first bound `#edge`×`#path.2/1` says that the complexity is linear in the number of edges and in the maximum number of vertices reachable from a single vertex, and the latter is `#vertex` in the worst case. The second bound, `#path`×`#edge.1/2` says that the complexity is linear in the number of resulting `path` pairs and in the maximum in-degree of vertices, and the former is `#vertex`^{2} in the worst case, and the latter is `#vertex` in the worst case. The worst-case complexity expressed using only `#edge` and `#vertex` is looser.

Note also that our time complexity analysis takes into account the number of rules, because the complexity is a sum of firings over all rules. The number of hypotheses of each rule is considered a constant since it is one or two; we will see that rules with more than two hypotheses are analyzed as more rules with one or two hypotheses. The analysis could easily be refined to take into account the number of arguments of the relations since we know exactly how many arguments are used for lookups, membership tests, etc. for each rule, but such quantities are usually considered constant factors.

Additional constraints that capture dependencies among relations and relation arguments can be constructed from the rules to further bound the sizes for symbolic minimization. They can provide more precise results of symbolic minimization for rules that have longer chains of non-circular dependencies among relations and relation arguments. They can also help understand the complexity in terms of output size, rather than input size alone. Specifically, we can bound, for each rule `r`, the number of instances of the conclusion, denoted `r.#addedFacts`, based on the number of instances of the hypotheses combined, and we can bound the number of instances of a hypothesis, denoted `#P(Xs)`, by summing all facts that match the hypothesis from the rules that can conclude these facts and from the given facts. For the latter, we can approximate by considering all facts of relation `P`. These yield the following constraints, where `P.conclRules` denotes the set of rules where `P` occurs in the conclusion.

$$\texttt{r.\#addedFacts} \leq \texttt{r.\#firedTimes}, \quad \text{where } \texttt{r.\#firedTimes} \text{ is defined above}$$

$$\texttt{\#P(Xs)} \leq \texttt{\#}\{\texttt{P(Xs)}:\texttt{P(Xs) in givenFacts}\} + \sum_{\texttt{r in P.conclRules}} \texttt{r.\#addedFacts}$$

While it is possible to consider facts that match also the arguments, not just the relation name `P`, we have not found the need of it in applications.

Space complexity

We consider the space needed besides the space taken by the input. The total such space is the sum of the space needed for each of the result sets `RQi`'s and other `Ri`'s, worksets `WQi`'s and other `Wi`'s, and auxiliary maps `PiYsXis`'s, described separately as follows:

—`RQi`'s are only for relations that occur in the conclusions of the rules, i.e., relations for which new facts may be inferred. For each such relation `Qi` of, say, `a` arguments, the space of RQi is for the a-level nested array structures that RQi uses. These arrays are indexed by the values in the domains and thus take

$$\texttt{\#D(Qi.1)} \times \ldots \times \texttt{\#D(Qi.a)}$$

space. Other Ri's take the same amount of space as the given facts for the corresponding relations take.

—WQi's use the same amount of space for their nested-array structures as RQi's use. The queues for WQi's take no more space than the arrays. The queues for other Wi's take the same amount of space as the given facts for the corresponding relations take.

—PiYsXis's are only for relations that occur in the hypotheses of rules of form 2. If associative access of kind (iii) is needed, then the space of PiYsXis is for the arrays used for accessing all the components; linked lists for the components in the non-anchor take no more space. Otherwise, the space is taken by the arrays for the components in the anchor plus linked lists for the non-anchor.

Let the domains of Ys be DY1 to DYj and of Xis be DXi1 to DXik. If associative access of kind (iii) is needed, the total space for PiYsXis is

$$\texttt{\#DY1} \times \ldots \times \texttt{\#DYj} \times \texttt{\#DXi1} \times \ldots \times \texttt{\#DXik}.$$

Otherwise, the product after the anchor is replaced with the amount of space taken by the nested linked-list structures for the non-anchor, one such structure for each element of the arrays for the last component of the anchor; it is hard to sum the space used by these structures directly, but it is easy to express it as the difference between the space for a nested linked-list structure for all components and the space for a nested linked-list structure for the anchor. So the total space for PiYsXis is

$$\texttt{\#DY1} \times \ldots \times \texttt{\#DYj} + \texttt{\#Pi.(IYs}\cup\texttt{IXis)} - \texttt{\#Pi.IYs}$$

We call the space taken by result sets RQi's *output space*, and the space taken by auxiliary maps PiYsXis's *auxiliary space*. Worksets WQi's take the same space asymptotically as RQi's, and other Wi's and Ri's take no more space asymptotically than the given facts take.

In our data structures for the auxiliary map for each individual relation, and for the result set for each individual rule, arrays are used only where needed—each array supports constant-time associative access by index for a non-constant number of sets that have associative access—and linked structures with minimum space overhead are used for the rest. However, optimizations that schedule the order in which elements in the worksets are considered may allow reuse of space by considering relations and rules in a certain order. Therefore the total space used may sometimes be reduced using scheduling optimizations to eliminate some summands in the space complexity formula.

Example. For the transitive closure problem, the output path takes space $\texttt{\#D(path.1)}\times\texttt{\#D(path.2)}$, which is $O(\texttt{\#vertex}^2)$. The auxiliary space usage is $\texttt{\#D(edge.2)}+\texttt{\#edge.\{2,1\}}-\texttt{\#edge.2}$, which is $O(\texttt{\#edge})$ for vertex being the domain of the arguments of edge, i.e., $\texttt{vertex} = \texttt{edge.1}\cup\texttt{edge.2}$.

Note that our space complexity analysis takes into account the number of rules, the number of hypotheses in rules, and the number of arguments of relations, because the complexity is a sum of space taken by all relations used in all rules.

7. EXTENSIONS

The time complexity of the compilation process described above is linear in the number of rules. It can handle additions of new rules incrementally—one just needs to add pattern matching clauses and data structures that correspond to the new rules. The generated algorithms can handle additions of facts incrementally—one just needs to start with the previously computed result `R` and add new facts to the workset `W`.

Datalog rules with more than two hypotheses

For rules with more than two hypotheses, we can both transform them into rules with two hypotheses and generalize our derivation process to handle such rules directly.

The transformations simply introduce auxiliary relations with necessary arguments to hold the results of combining two hypotheses at a time. Precisely, we repeatedly apply the following transformations to each rule with more than two hypotheses until only rules with at most two hypotheses are left: (i) replace any two hypotheses, say $P_i(X_{i1}, ..., X_{ia_i})$ and $P_j(X_{j1}, ..., X_{ja_j})$, of the rule with a new hypothesis, $Q(X_1, ..., X_a)$, where Q is a fresh relation, and X_k's are variables in the arguments of P_i or P_j that occur also in the arguments of other hypotheses or the conclusion of this rule, and (ii) add a new rule $P_i(X_{i1}, ..., X_{ia_i}) \wedge P_j(X_{j1}, ..., X_{ja_j}) \rightarrow Q(X_1, ..., X_a)$. For a rule with h hypotheses, there are $(2h-3)!!$ (i.e., $1 \times 3 \times \cdots \times (2h-3)$) ways of decomposing it into rules with two hypotheses, but h is typically a very small constant, most often no more than two. This observation is supported by our example applications, as discussed in Section 8.

Each decomposition leads to certain time and space complexities, calculated easily using our method; the only modification is that the space taken by the introduced auxiliary relations should be counted as auxiliary space not output space. The complexities resulting from different decompositions can be compared to determine which one is best in terms of time, space, or possibly both. Note that there may be a trade-off between time and space complexities, if no decomposition leads to the smallest complexities for both time and space.

For example, the following rules define `path2`, which is the set of all pairs of vertices `u` and `v` such that there is a path of even length from `u` to `v`. The second rule has three hypotheses and can be decomposed in three ways: combining the first two hypotheses first, the first and third hypotheses first, and the last two hypotheses first.

```
edge(u,w) ∧ edge(w,v) → path2(u,v)
edge(u,w) ∧ edge(w,x) ∧ path2(x,v) → path2(u,v)
```

The third way yields the following rules and the time complexity $O(\texttt{\#edge}\times\texttt{\#vertex})$, because both rules have this complexity, as can be analyzed as for the transitive closure example. The output space is $O(\texttt{\#vertex}^2)$ for `path2`, again as for the transitive closure example. The auxiliary space is $O(\texttt{\#vertex}^2)$, worse than $O(\texttt{\#edge})$

for the transitive closure example, because of the space taken by the auxiliary relation `path1`.

```
edge(w,x) ∧ path2(x,v) → path1(w,v)
edge(u,w) ∧ path1(w,v) → path2(u,v)
```

The first way yields the following, with `min(#edge`$\times$`#edge.2/1,#edge`$\times$`#edge.1/2)` and `min(#edge2`$\times$`#path2.2/1,#path2`$\times$`#edge2.1/2)` as bounds for their respective numbers of firings. So the time complexity is $O(\texttt{\#vertex}^3)$, worse than the third way. The output space and auxiliary space are both $O(\texttt{\#vertex}^2)$, for similar reasons as the third way.

```
edge(u,w) ∧ edge(w,x) → edge2(u,x)
edge2(u,x) ∧ path2(x,v) → path2(u,v)
```

The second way can be analyzed equally easily. It leads to the same output space and auxiliary space as above, but an even worse time complexity $O(\texttt{\#vertex}^4)$, because it may consider many more combinations of facts for the first and third hypotheses that are not connected by an edge in between for the second hypothesis.

Note that our method generates correct algorithms and data structures, together with time and space complexity formulas, regardless of which decomposition is used. Choosing a decomposition is part of finding an optimal algorithm at a high level. When a user does not care about such optimizations, any decomposition can be used.

Although transforming rules is higher-level, more declarative, simpler, and clearer than treatment of more hypotheses in the derivation process, the space taken by the auxiliary relations may be unnecessary. Treatment of more hypotheses directly in the derivation process allows us to try all possible decompositions into groups of two or more hypotheses. For each group, we consider adding one fact of one hypothesis at a time and test all combinations of facts that make the other hypotheses in the group true. Note that considering a sequence of three or more hypotheses requires considering two hypotheses, or one hypothesis and one temporary intermediate relation, at a time, repeatedly, so the best running time this approach can achieve is no better than the transformational method. However, this method can avoid storing intermediate relations in considering a sequence of more than two hypotheses and thus lead to minimum possible auxiliary space. The regular path query examples in Section 8 illustrate this tradeoff.

Note that not storing auxiliary relations for intermediate results of combinations is clearly more space efficient, at no loss of time efficiency, when each intermediate result is used only once. For example, for a rule that contains multiple consecutive graph edges as hypotheses, when graph vertices are connected by single paths, each intermediate fact is used only once, so storing auxiliary relations for the intermediate facts is not useful. Instead of storing an intermediate fact, the generated algorithm immediately executes code in the branches that match this fact. In this case, the running time should also be improved, by a constant factor, because of elimination of the operations on auxiliary relations. However, when this is not the case, not storing auxiliary relation can be much less efficient in running time, even asymptotically less efficient, as shown in the experiments in Section 8.

Datalog rules with equal or wild cards and with multiple hypotheses

We can either eliminate equal cards and wild cards in rules with multiple hypotheses by simple transformations or handle them in the derivation process. Both approaches result in essentially the same algorithms and complexities. We present the transformational approach since it is higher-level, more declarative, simpler, and clearer.

If a hypothesis of a rule with multiple hypotheses contains equal cards, we introduce an auxiliary relation to hold only those facts that have the same value in the argument positions of each equal card. Precisely, for every hypothesis of a rule that contains equal cards, we (i) replace the hypothesis, say $P(X_1, ..., X_a)$, with a new hypothesis, $Q(X'_1, ..., X'_{a'})$, where Q is a fresh relation, and X'_k's are the arguments of P but with the second and later occurrences of each variable eliminated, and (ii) add a new rule $P(X_1, ..., X_a) \rightarrow Q(X'_1, ..., X'_{a'})$. Similarly, if a hypothesis of a rule with multiple hypotheses contains a wild card, we introduce an auxiliary relation that has only the arguments of the hypothesis that are not wild cards. Precisely, we do the same two things as for equal cards, except that X'_k's are the arguments of P that are not wild cards. Note that we must first remove equal cards and then remove wild cards, because removing equal cards may introduce wild cards.

For example, consider rules 1 to 3 below, where `edge(u,v,c)` denotes an edge from `u` to `v` with color `c`, and `green` and `red` are two constants denoting colors. Then, `path(u,v,c)` means that there is a path from `u` to `v` with color `c` on all edges in the path, and `greenReachRedCycle(v)` means that there is a vertex `v` reachable from a green path and being in a red cycle.

1. `edge(u,v,c)` $\rightarrow$ `path(u,v,c)`
2. `edge(u,w,c)` $\wedge$ `path(w,v,c)` $\rightarrow$ `path(u,v,c)`
3. `path(u,v,green)` $\wedge$ `path(v,v,red)` $\rightarrow$ `greenReachRedCycle(v)`

First, to remove equal cards from rule 3 which has two hypotheses, we replace rule 3 with rule $3'$ and add rule 4. Then, to remove wild cards from rule $3'$ which has two hypotheses, we replace rule $3'$ with rule $3''$ and add rule 5.

$3'$. `path(u,v,green)` $\wedge$ `redCycle(v,red)` $\rightarrow$ `greenReachRedCycle(v)`
4. `path(v,v,red)` $\rightarrow$ `redCycle(v,red)`
$3''$. `greenReach(v,green)` $\wedge$ `redCycle(v,red)` $\rightarrow$ `greenReachRedCycle(v)`
5. `path(u,v,green)` $\rightarrow$ `greenReach(v,green)`

To see that equal cards must be removed before wild cards, consider changing `path(u,v,green)` in rule 3 to `path(u,u,green)`. Then `u` is a equal card, but after the equal card is removed, `u` becomes a wild card, which is then removed.

The only effect of these two transformations on our complexity analysis is that the space taken by introduced auxiliary relations should be counted as auxiliary space not output space. This space is asymptotically no more than the space for the original given problem. Note that in the derived algorithms for the transformed rules, when rules with two hypotheses are considered, all instances of a hypothesis that differ only in the wild card components are considered only once together for different values of the wild card components, and only instances whose equal cards components are equal are considered. The domain of an equal card component is the intersection of the domains of all the components of the equal card.

Extension of Datalog rules with constraints and external functions

We can extend Datalog rules to allow other constraints as hypotheses and external functions in conclusions. The constraints are Boolean-valued functions on various domains, such as greater-than on integers or prefix-of on sequences. The external functions can be arbitrary functions on these domains. In general, constraints may use variables not appearing in the non-constraint hypotheses and may be difficult to handle [Jaffar and Maher 1994]. In all the application problems we have encountered, many of which are discussed in Section 8, all variables in the constraints also appear in the non-constraint hypotheses. This makes it much easier to handle the constraints efficiently.

For example, the following rule is used in computing the name reduction closure in SPKI trust management framework [Ellison et al. 1999; Clarke et al. 2001; Hristova et al. 2007], where `cert(k1,n,k2,ns)` denotes the name certificate stating that principal `k1`'s name `n` is principal `k2`'s name sequence `ns`, `[]` denotes the empty sequence, constraint `isHead(h,s)` denotes that `h` is the head of sequence `s`, and external function `tail(s)` denotes the tail of sequence `s`.

```
cert(k1,n1,k2,ns2) ∧ cert(k2,n2,k3,[]) ∧ isHead(n2,ns2)
→ cert(k1,n1,k3,tail(ns2))                                    (18)
```

The constraints can be evaluated straightforwardly after all variables in them are bound. Precisely, after decomposing the rules into rules with at most two hypotheses, including constraint hypotheses, we discard decompositions that contain constraints involving unbound variables. Among the remaining decompositions, those that evaluate constraints earlier have equal or better running time, because the constraints may eliminate some combinations of values. The complexity calculation considers the non-constraint hypotheses as before, but with argument domain size possibly reduced based on the number of values that can satisfy the constraint, and with the cost of constraint evaluation added.

The external functions can be evaluated any time after all variables they use are bound. The complexity calculation is as before except with the cost of evaluating the functions added.

For some kinds of constraints, it is possible and sometimes beneficial to evaluate the constraint earlier, before all the variables in it are bound, because the bound variables of the constraint can be used to restrict the unbound variables. For example, for a constraint on sequences that tests whether an element x is the head of a sequence s, we may evaluate the constraint when s is bound and x is not, because this will restrict x to be the unique head of s. This can be more efficient than first evaluating additional hypotheses to bind x and then evaluating the constraint. In the name reduction closure example above, any certificate matching the first hypothesis binds `k2` and `ns2`, and `ns2` then binds `n2` using constraint `isHead(n2,ns2)`. Using both `k2` and `n2`, rather than `k2` alone, helps reduce the number of certificates considered that match the second hypothesis.

Extension of Datalog rules with negation

Datalog rules do not contain negated hypotheses, but negation is useful for expressing some analysis problems. The most well-known semantics for Datalog with negation are stratified semantics [Abiteboul et al. 1995], well-founded semantics [Abiteboul et al. 1995], and stable model semantics [Gelfond and Lifschitz 1988].

Our derivation and complexity analysis work naturally for rules with stratified semantics. The first step of our derivation is modified to do a fixed-point computation following the order of stratification. The rest of the derivation needs no change, since our data structures support associative access for testing of both positive and negative hypotheses. The complexity analysis remains the same also, simply ignoring negations in the negative hypotheses, since they are processed using the same time and space as the corresponding positive hypotheses. We think that our derivation method should work for computing least fixed-points in other semantics, such as well-founded semantics and stable model semantics, as well. The precise derivation is a subject for further study.

Extension of Datalog rules with data constructors

Datalog rules do not contain data constructors, i.e., functors in logic programs. Recursion with data constructors yields recursively structured data and is necessary for many analysis problems, including many combinatorial optimization problems.

We have developed a general and systematic method, called incrementalization [Liu et al. 1998; Liu 2000; Liu et al. 2001], for incremental computation of recursive functions that use data constructors. The method is able to derive dynamic programming algorithms for these problems when they are specified using recursive functions [Liu and Stoller 2003; 2002]. We believe that the same ideas can be applied to specifications of these problems using Datalog rules extended with data constructors as well as with arithmetic.

On-demand computation and other optimizations

The programs our method generates compute all facts that can be inferred, which can then be used to answer queries of specific facts. If only facts of a particular relation P are needed, then we can first do a reachability analysis to include only rules on which P depends and then transform only those rules. If only the truth value of P on a particular set of arguments is needed, a more sophisticated on-demand (top-down) computation method is needed. The method described in this paper is bottom-up. How to achieve efficient top-down computation, or an efficient combination of bottom-up and top-down computations, completely by a transformational method is a topic that needs future study. We are developing methods to combine magic sets transformations [Bancilhon et al. 1986] and specialization [Tekle et al. 2008] with our transformations to achieve efficient on-demand computation with time and space guarantees.

Optimizations that schedule the order in which elements in worksets are considered also need to be studied, to achieve optimal space usage in a more absolute sense.

8. APPLICATIONS

We applied our transformation and complexity analysis to a number of nontrivial analysis problems and obtained improved algorithm complexities for some and greatly improved algorithm understanding and greatly simplified complexity analysis for all of them.

Table I summarizes the worst-case complexities of our derived algorithms for nine application problems: transitive closure (the running example), graph reachability [Cai and Paige 1988], pointer analysis [Andersen 1994], simplification of regular tree grammar based constraints [Liu et al. 2001], authorization in trust management [Ellison et al. 1999], and four kinds of regular path queries [Liu and Yu 2002; de Moor et al. 2003; Liu et al. 2004]. These problems and the complexities are explained below. Other problems to which we have applied our method include querying complex graphs [Liu and Stoller 2006], model checking push down systems [Hristova and Liu 2006], and information flow analysis [Hristova et al. 2007]. These and similar problems have applications in program analysis [Cousot and Cousot 1977; Heintze and Jaffar 1994; Reps 1998; Aiken 1999; Heintze and Tardieu 2001], model checking [Clarke et al. 1999], security frameworks [Li and Mitchell 2003; Stoller and Liu 2007], and queries of semi-structured data [Abiteboul 1997; Calvanese et al. 2000; Li and Moon 2001]. Many more analysis problems in these applications can be specified as Datalog rules and implemented with time and space guarantees using our method.

problem	running time	output space	auxiliary space
transitive closure	$O(\texttt{\#edge}\times\texttt{\#vertex})$	$O(\texttt{\#vertex}^2)$	$O(\texttt{\#edge})$
graph reachability	$O(\texttt{\#source}+\texttt{\#edge})$	$O(\texttt{\#vertex})$	$O(1)$
constraint simplification	$O(\texttt{\#node}^3)$	$O(\texttt{\#node}^2)$	$O(\texttt{\#node}^2)$
pointer analysis	$O(\texttt{\#node}^3)$	$O(\texttt{\#node}^2)$	$O(\texttt{\#node}^2)$
trust management	$O(\texttt{in}\times\texttt{\#key}^2)$	$O(\texttt{in}\times\texttt{\#key})$	$O(\texttt{\#key}^2\times\texttt{\#id}^2)$
existential & universal regular path queries (RPQ)	$O(\texttt{\#edge}\times\texttt{\#state}+\texttt{\#vertex}\times\texttt{\#transition})$	$O(\texttt{\#vertex}\times\texttt{\#state})$	$O(\texttt{\#label}\times\texttt{\#vertex}\times\texttt{\#state})$
existential & universal parametric RPQ	$O(\text{above}\times\texttt{\#subst}\times\texttt{\#param})$	$O(\text{above}\times\texttt{\#subst})$	$O(\text{above}\times\texttt{\#subst})$

Table I. Summary of worst-case complexities for example applications.

Graph reachability

Graph reachability finds all vertices reachable (`reach`) starting from a given set of source vertices (`source`) and following a given set of edges (`edge`). Table I gives the asymptotic complexities in terms of only input parameters `#source` and `#edge`.

More precisely, our derived algorithm uses no auxiliary maps; its time complexity is bound by $O(\texttt{\#source}+\texttt{min(\#reach}\times\texttt{\#edge.2/1,\#edge)})$ plus the time for reading input, and its output space is $O(\texttt{\#reach})$. These formulas give more information than the simplified formulas in Table I. For example, after reading in the set of edges, given any set of source vertices, the time complexity is bound by $O(\texttt{\#source}+\texttt{min(\#reach}\times\texttt{\#edge.2/1,\#edge)})$; when few vertices are reachable and the out-degrees of vertices are small, the running time is only $O(\texttt{\#source}+\texttt{\#reach})$, which is significantly better than $O(\texttt{\#source}+\texttt{\#edge})$.

Constraint simplification

Simplification of regular tree grammar based constraints is used to analyze recursive data structures in programs [Liu et al. 2001]. It expresses seven kinds of constraints using seven kinds of relations. It uses ten rules to infer simplified forms of constraints from given constraints.

The cubic time complexity in Table I is the worst-case bound for the formula $\texttt{\#simp} \times \texttt{min(k} + \texttt{\#copy.1/2, k} \times \texttt{\#simp.\{2,3,...,k+2\}/1)}$ that is obtained using our method, where `simp` is the set of constraints of simplified forms, `copy` is the set of constraints of copy forms, and `k` is the maximum arity of data constructors in programs. This is exactly the running time analyzed in [Liu et al. 2001], which helped better understand the practical performance of the algorithm. The analysis based on Datalog rules is drastically simpler; it also improves over the analysis in [Liu et al. 2001] on some of the non-worst-case rules.

Pointer analysis

Andersen's pointer analysis for C programs [Andersen 1994] defines a points-to relation based on four kinds of assignment statements that involve pointers: taking address $x = \&y$ (`address`), making copy $x = y$ (`copy`), taking content $x = *y$ (`content`), and assigning content $*x = y$ (`assign`). The analysis can be easily specified using four Datalog rules, one for each kind of statement. In the resulting points-to relation, each pointer can point to a set of locations in the program.

The cubic time complexity in Table I is an upper bound of the more precise formula we derive, $O(\texttt{\#address} + \texttt{\#copy} \times \texttt{\#to} + \texttt{\#content} \times \texttt{\#to}^2 + \texttt{\#assign} \times \texttt{\#to}^2)$, where the four kinds of statements can add up to all the statements in the program, and `to` set, the largest set of locations pointed to by a single pointer, could in the worst case include all locations in the program. However, the `to` set is typically very small for well-behaved programs, so the actual running time could be much better than the worst case bound of $O(\texttt{\#node}^3)$, and could even be $O(\texttt{\#node})$ when all but a small number of pointers point to a small number of locations. The more precise formula also shows precisely how the running time depends on each kind of statement.

Andersen's analysis is well-known to have a worst-case cubic time complexity, but is much more efficient in practice and often appears to be quadratic or even linear. Our derived algorithms and complexities offer a simple and most precise explanation.

Trust management

SPKI/SDSI [Ellison et al. 1999] is a well-known trust management framework based on public keys that is designed to facilitate the development of secure and scalable distributed systems.

It has two types of certificates. A name certificate defines a local name in the issuer's local name space. An authorization certificate is issued by an issuer to grant a set of permissions to a subject and possibly allows the subject to delegate the permission. The heart of authorization checking is to compute the name-reduction closure given a set of certificates. It composes certificates to infer reduced certificates and can be expressed directly as a Datalog rule with a constraint that tests whether an id is the head of a sequence of ids, and with a call to an external function that returns the tail of a sequence in the conclusion, as shown in (18).

The time complexity in Table I is an upper bound of the more precise formula we derive, `#cert`×`#cert.3/{1,2,4=[]}`, where `cert` is the set of inferred certificates, and `#cert.3/{1,2,4=[]}` is the maximum number of keys a single local name reduces to. In the worst case, the latter could include all keys (`key`) that appear in the certificates, but it is much smaller on average in practice and is close to a constant in large systems. As noted in [Jha and Reps 2004], `#cert` is bounded by `in`×`#key`, where `in` is the size of the input, i.e., the sum of the sizes of the given certificates. These lead to the time complexity in Table I. The auxiliary space is used for mapping key-id pairs to key-id pairs. Our complexity analysis is more precise and informative than using only worst-case sizes [Clarke et al. 2001; Jha and Reps 2004].

Regular path queries

Four kinds of regular path queries are considered. Consider an edge-labeled directed graph G, a vertex $v0$ of G, and a regular expression P. An existential query computes all vertices v in G such that there is a path from $v0$ to v that matches P, where `state` and `transition` describe a non-deterministic finite automaton that corresponds to P. A universal query computes all vertices v in G such that all paths from $v0$ to v match P, where `state` and `transition` describe a deterministic finite automaton that corresponds to P. Parametric queries allow labels to have parameters (`param`) and compute substitutions (`subst`) of variables to constants together with the matched vertices. The rules infer a relation `match(v,s)` on vertex `v` and state `s`, so the output space has a factor of `#vertex`×`#state`.

All four kinds of queries naturally have a rule with three hypotheses. Decomposing it into two rules, each with two hypotheses, yields three alternatives. For non-parametric queries, two of the alternatives have the time and space complexities given in Table I, and the third has running time O(`#edge`×`#transition`) and auxiliary space O(`#vertex`^{2}×`#state`^{2}). Considering all three hypotheses together without decomposition yields an algorithm with running time O(`#edge`×`#transition`) and auxiliary space O(`#label`×`(#vertex`+`#state)`), exactly as in [Liu and Yu 2002]. The time complexity in Table I improves over O(`#edge`×`#transition`) in [Liu and Yu 2002; de Moor et al. 2003]. The trade-off is the worse auxiliary space compared with O(`#label`×`(#vertex`+`#state)`) in [Liu and Yu 2002], though `#state` is very small in practice. For parametric queries, similar complexities can be obtained, except with an additional factor of `#subst`×`#param` in the time complexity and `#subst` in the space complexities.

For all four kinds of queries, the factor `#edge`×`#state` +`#vertex`×`#transition` in running time is an asymptotic improvement over `#edge`×`#transition` in previous algorithms [Liu and Yu 2002; de Moor et al. 2003; Liu et al. 2004], because in the worst case, `#edge` can be quadratic in `#vertex`, and `#transition` can be quadratic in `#state`.

Experiments

We have developed a prototype implementation of the method and have used it in generating efficient algorithms for a number of problems in program analysis, model checking, and security, including most of the examples discussed in this paper, and some others. We discuss here measurements that characterize the number of hypotheses in rules, the number of firings for rules with one or two hypotheses, and the number of firings for rules with more than two hypotheses. Experiments were also conducted previously to confirm the analyzed time complexities, for model checking push down system [Hristova and Liu 2006], information flow analysis [Hristova et al. 2007], and trust management policy analysis [Hristova et al. 2007].

First, we counted the number of hypotheses of rules for all the example applications in Table I. Table II shows, for each example, the total number of rules and the number of rules with each different number of hypotheses. For the constraint simplification example, `#con` and `#sel` denote the numbers of constructors and selectors, respectively, in the program being analyzed. For the trust management example, * indicates that the rule has an additional constraint. None of the examples has a rule with more than three hypotheses. This helps support our observation that the number of hypotheses of a rule is typically a very small constant, most often no more than two.

problem	#rules total	#rules with 1 hypothesis	#rules with 2 hypotheses	#rules with 3 hypotheses
transitive closure	2	1	1	0
graph reachability	2	1	1	0
constraint simplification	6+3×`#con` +`#sel`	0	6+3×`#con` +`#sel`	0
pointer analysis	4	1	1	2
trust management	2	1	1*	0
existential & universal regular path queries (RPQ)	3	0	2	1
existential & universal parametric RPQ	1+`#subst` +`#label`	0	1+`#subst`	`#label`

Table II. Summary of worst-case complexities for example applications.

Second, for the transitive closure example, we counted the number of firings and measured the running times of the generated program on randomly generated graphs of different sizes. Table III shows the number of edges, number of vertices, number of rule firings, analyzed upper bound on combinations of facts that make all hypotheses of rules true, and the running time for each graph. Comparing columns 3 and 4 clearly shows that the number of firings are bounded by the analyzed upper bound on combinations. The running times are for generated Python programs not optimized for constant factors. They are averages over five runs, taken on a Sun Blade 1500 with 1GHz UltraSPARC IIIi CPU and .5GB RAM, running Solaris 8 and Python 2.5. Figures 6 and 7 show that the running time shown in Table III is linear in the number of edges when the number of vertices is fixed, and linear in the number of vertices when the number of edges is fixed, as analyzed, and that the number of firings well reflects the running time.

Note that the running time in Figure 7 appears to increase very slightly faster than linear, but that is because of the relatively more constant-factor time spent on graphs with fewer vertices. The reason is, when `#edge` is fixed, graphs with fewer vertices are denser and thus produce relatively more `path` facts, and processing a `path` fact in a firing has on average a slightly larger constant cost than processing an `edge` fact in a firing, because the first rule uses `edge` in a simpler way. For the same reason but used in a symmetric way, in Figure 6, relatively slightly less time is spent on graphs with fewer edges when `#vertex` is fixed

#edge	#vertex	#firings total	analyzed bound #edge×#vertex	running time in seconds
1000	2000	1,337,878	2,000,000	13.3579
1000	4000	2,904,172	4,000,000	20.5445
1000	6000	4,375,843	6,000,000	26.1082
1000	8000	5,844,170	8,000,000	31.5518
1000	10000	7,431,620	10,000,000	37.5729
200	10000	1,502,830	2,000,000	5.7458
400	10000	2,925,208	4,000,000	11.9971
600	10000	4,435,026	6,000,000	19.5471
800	10000	5,893,465	8,000,000	27.8315
1000	10000	7,431,620	10,000,000	37.6039

Table III. Number of firings and running time for the transitive closure example.

Third, for the even-length path example in Section 7, we counted the number of firings for rules decomposed using the third way, where an auxiliary relation is used for the intermediate results, and the number of pairs of facts that need to be considered when no auxiliary relation is used. Table IV and Figure 8 show that these two numbers are the same when graph edges are so sparse that is there is at most one path between any two vertices, but otherwise using auxiliary relations reduces the number significantly, even asymptotically.

9. RELATED WORK AND CONCLUSION

Datalog and optimization methods for Datalog have been studied extensively in logic programming and database areas. What distinguishes our results is (i) the direct transformation of any set of Datalog rules into a complete algorithm and data structures specialized for those rules, and (ii) precise analysis of the worst-case time and space complexities supported by the algorithm and the data structures.

Optimization methods for Datalog and more general logic programs include smart evaluation methods and rewriting methods [Ceri et al. 1990; Abiteboul et al. 1995]. Examples of the former include semi-naive evaluation for bottom-up evaluation [Ceri et al. 1990; Naughton and Ramakrishnan 1991], top-down evaluation

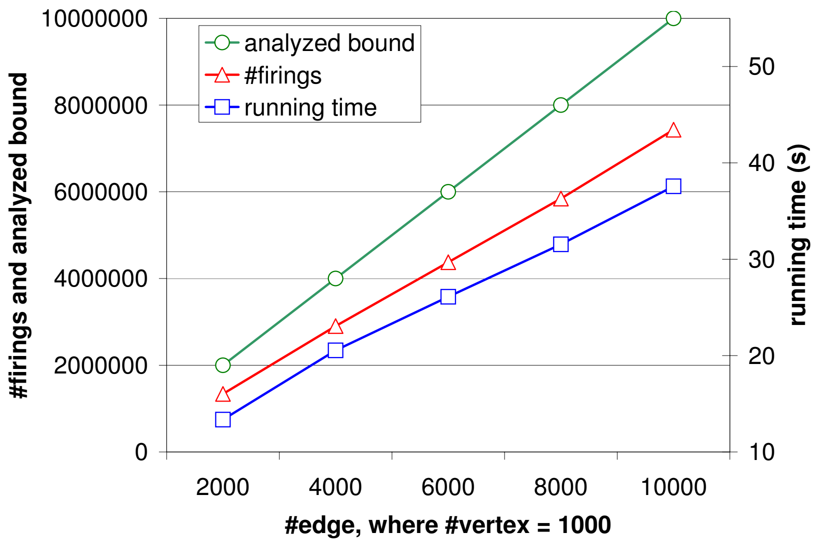

Fig. 6. Number of firings and running time for the transitive closure example with varying `#edge`.

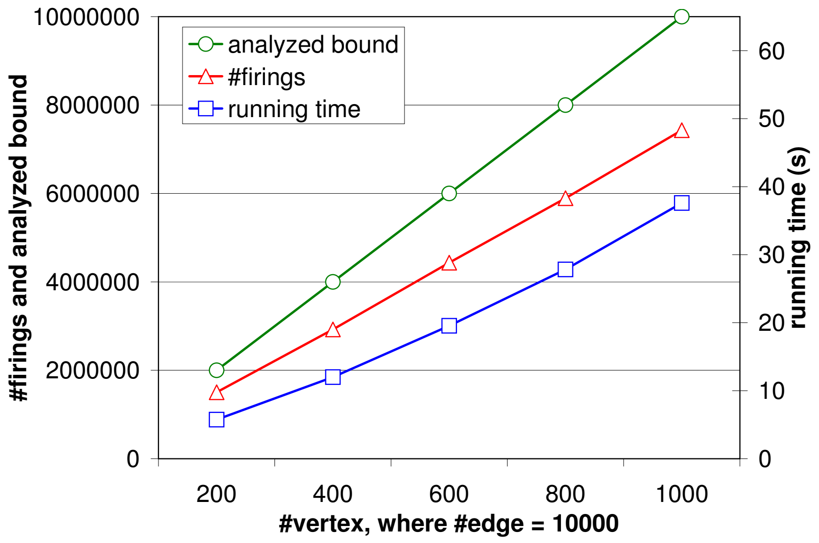

Fig. 7. Number of firings and running time for the transitive closure example with varying `#vertex`.

with tabling [Tamaki and Sato 1986; Chen and Warren 1996], and static and dynamic filtering [Kifer and Lozinskii 1986; 1990]. Examples of the latter include magic sets transformation [Bancilhon et al. 1986] and partial evaluation [Lloyd and Shepherdson 1991; Leuschel 1998]. Our method is not an evaluation method because it transforms the rules rather than evaluating them; our method is not a rewriting method in that it does not transform within the frameworks of rules or

#vertex	#firings with aux. relation	considered pairs no aux. relation
200	62	62
400	236	236
600	978	978
800	3,300	3,300
1000	22,673	22,869
2000	2,675,518	4,027,540
4000	5,816,141	14,390,936
6000	8,775,245	30,287,659
8000	11,735,884	52,231,705
10000	14,944,301	82,523,036

Table IV. Number of firings with and without using auxiliary relations for the even-length path example.

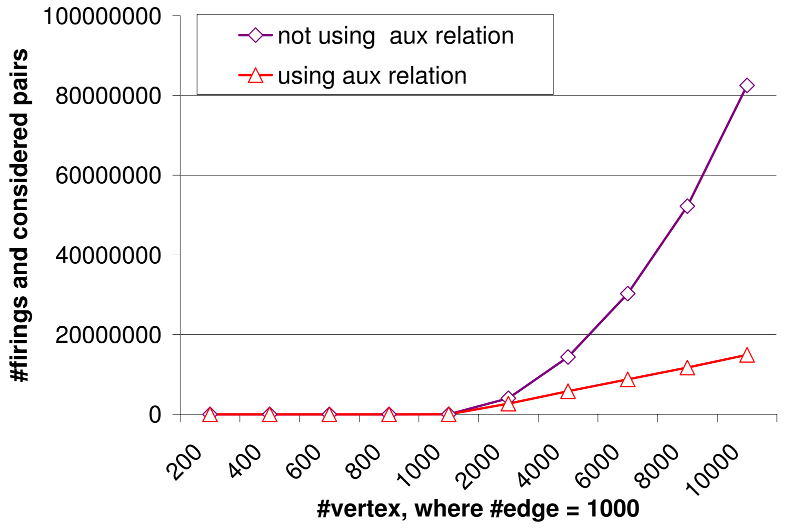

Fig. 8. Number of firings when an auxiliary relation is used vs. number of considered pairs when no auxiliary relation is used, for the even-length path example.

some algebras. Instead, it compiles the rules directly into an implementation in a standard imperative programming language. The generated implementation performs a kind of bottom-up computation based on careful incremental updates with data structure support.

Previous methods for evaluating or rewriting Datalog rules mostly do not provide complexity analysis [Ceri et al. 1990], even though complexity measures and complexity classes for various query languages have been studied [Vardi 1982; Kolaitis and Vardi 1995; Dantsin et al. 2001; Gottlob and Papadimitriou 2003; Gottlob et al.

2006]. In fact, such analysis can be very difficult. For example, for top-down evaluation with tabling and indexing [Tamaki and Sato 1986; Sagonas et al. 1994; Chen and Warren 1996; Ramakrishnan et al. 2001], a graph reachability program may have several different time complexities between linear and quadratic, depending on the order of the rules, the order of the hypotheses in a rule, the indexing used, etc. It is well known that a Datalog program runs in $O(\mathtt{n}^{\mathtt{k}})$ time where k is the largest number of variables in any single rule, and n is the number of constants in the facts and rules, but such a bound is too loose for most problems. There are also methods for efficient evaluation of Datalog queries using binary decision diagrams [Whaley and Lam 2004; Lam et al. 2005] and relational databases [Avgustinov et al. 2007], but these methods do not provide time and space complexity guarantees.

McAllester [McAllester 1999] introduced a method for capturing precise time complexities of logic programs, by counting the number of *prefix firings*, i.e., combinations of facts that make all prefixes of the hypotheses of a rule true. Ganzinger and McAllester generalized it to include priorities and deletion of rules [Ganzinger and McAllester 2001] and furthermore priorities for instances of the same rule [Ganzinger and McAllester 2002]. Datalog is a subclass of their programming models, albeit an important subclass. The main difference is that we generate specialized algorithms and data structures from the rules, while they use an evaluation method that is interpretive and uses extensive hashing; such a method incurs extra time overhead, often consumes unnecessarily large space, and gives no tight worst-case guarantees on time and space. Also, they do not discuss the implication of changing the order of hypotheses, and do not discuss space complexity. Finally, they did not attempt to automate the complexity analysis. A follow-up [Nielson et al. 2002] discusses how to automate the complexity analysis, but does not address the other limitations.

The idea of considering one new fact at a time and finding other facts using indexing to form firings of rules is quite straightforward, and is used in many implementations. For example, in compilation of constraint handling rules [Holzbaur et al. 2005; Schrijvers 2005], the active constraint corresponds to the new fact considered, although the constraints are more general. The idea of reducing the time complexity to the number of firings of rules is used at least as early as the 1970s by Beeri and Bernstein [Beeri and Bernstein 1979], where they give a linear-time algorithm for solving the attribute closure problem that is expressed using rules. Again, what distinguishes our work is the generation of precise algorithms and data structure specialized for any given set of Datalog rules, and furthermore the generation of precise time and space complexity formulas.

Our derivation of complete algorithms and data structures from Datalog rules uses and extends Paige's method [Paige 1981; Paige and Koenig 1982; Paige 1986; Cai and Paige 1988; Paige 1989; Cai et al. 1991] for fixed-points specifications built on set-based languages like SETL [Schwartz et al. 1986; Snyder 1990]. There are four main advancements. First, we start with Datalog rules, which are easier and clearer than fixed-point specifications. Second, our method for deriving incremental maintenance handles sets of tuples of any length and sets with different types of tuples, while Paige's method handles sets and maps, represented as sets of pairs, of uniform element types. Third, our data-structure design method handles general sets of tuples, and derives more general and sophisticated combinations of arrays, linked lists, and records than allowed by the based representation [Paige 1989; Cai et al. 1991] in Paige's method. Fourth, the advancements above allow our method to generate precise complexity formulas from the Datalog rules, while Paige's method does not generate them from fixed-point specifications. The precise complexity analysis for both time and space as well as the trade-offs, using detailed size characterizations of the given facts, can help better understand the practical performance of the generated algorithms.

There are many other program analysis and model-checking methods that use equations, constraints, automata, and formal languages [Cousot and Cousot 1977; Heintze and Jaffar 1994; Reps 1998; Aiken 1999; Esparza et al. 2000], and there are other query languages, but using rules is typically more direct and more general. Furthermore, the algorithms and implementations our method generates are formally derived using a systematic method, in contrast to the ad hoc development of other analysis algorithms and query evaluation methods; this helps assure the correctness and complexity guarantees for the generated algorithms and implementations. We have developed a prototype implementation of the method, used it in generating efficient algorithms for a number of application problems in program analysis, model checking, and security, including most of the examples discussed in this paper; we have also conducted a number of experiments to confirm the analyzed time complexities [Hristova and Liu 2006; Hristova et al. 2007; Hristova et al. 2007]. We are developing techniques to handle more extensions of Datalog.

Acknowledgment

Deepak Goyal provided very helpful comments on the first draft of this paper. Discussions with many colleagues, especially Nevin Heintze, Fritz Henglein, Michael Kifer, David McAllester, Anil Nerode, C.R. Ramakrishnan, Ganesan Ramalingam, and David Warren, helped greatly in understanding related work.

REFERENCES

Abiteboul, S. 1997. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory.* 1–18.

Abiteboul, S., Hull, R., and Vianu, V. 1995. *Foundations of Databases.* Addison-Wesley, Reading, Mass.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1983. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass.

Aiken, A. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming 35,* 2-3, 79–111.

Andersen, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen.

Avgustinov, P., Hajiyev, E., Ongkingco, N., de Moor, O., Sereni, D., Tibble, J., and Verbaere, M. 2007. Semantics of static pointcuts in AspectJ. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages.* ACM Press, New York, NY, USA, 11–23.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. 1986. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems.* 1–16.

Beeri, C. and Bernstein, P. A. 1979. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst 4,* 1, 30–59.

Cai, J., Facon, P., Henglein, F., Paige, R., and Schonberg, E. 1991. Type analysis and data structure selection. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 126–164.

Cai, J. and Paige, R. 1988. Program derivation by fixed point computation. *Science of Computer Programming 11*, 197–261.

Calvanese, D., DeGiacomo, G., Lenzerini, M., and Vardi, M. 2000. Answering regular path queries using views. In *Proceedings of the 16th IEEE International Conference on Data Engineering*. 389–398.

Ceri, S., Gottlob, G., and Tanca, L. 1990. *Logic Programming and Databases*. Springer-Verlag.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1 (Jan.), 20–74.

Clarke, D. E., Elien, J.-E., Ellison, C. M., Fredette, M., Morcos, A., and Rivest, R. L. 2001. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security 9,* 4, 285–322.

Clarke, Jr., E. M., Grumberg, O., and Peled, D. A. 1999. *Model Checking*. MIT Press.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*. 238–252.

Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33,* 3, 374–425.

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

Ellison, C., Frantz, B., Lampson, B., Rivest, R. L., Thomas, B., and Ylonen, T. 1999. RFC 2693: SPKI Certificate Theory.

Esparza, J., Hansel, D., Rossmanith, P., and Schwoon, S. 2000. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th International Conference on Computer-Aided Verification*. Lecture Notes in Computer Science, vol. 1855. Springer-Verlag.

Ganzinger, H. and McAllester, D. A. 2001. A new meta-complexity theorem for bottom-up logic programs. In *Proceedings of the 1st International Joint Conference on Automated Reasoning*. Springer-Verlag, Berlin, 514–528.

Ganzinger, H. and McAllester, D. A. 2002. Logical algorithms. In *ICLP '02: Proceedings of the 18th International Conference on Logic Programming*. Springer-Verlag, 209–223.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, R. A. Kowalski and K. Bowen, Eds. The MIT Press, Cambridge, Massachusetts, 1070–1080.

Gottlob, G., Koch, C., and Schulz, K. U. 2006. Conjunctive queries over trees. *Journal of the ACM 53,* 2, 238–272.

Gottlob, G. and Papadimitriou, C. 2003. On the complexity of single-rule datalog queries. *Inf. Comput. 183,* 1, 104–122.

Heintze, N. and Jaffar, J. 1994. Set constraints and set-based analysis. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science, vol. 874. Springer-Verlag, Berlin, 281–298.

Heintze, N. and Tardieu, O. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*. 254–263.

Holzbaur, C., Banda, M. G. D. L., Stuckey, P. J., and Duck, G. J. 2005. Optimizing compilation of constraint handling rules in hal. *Theory and Practice of Logic Programming 5,* 4-5, 503–531.

Hristova, K. and Liu, Y. A. 2006. Improved algorithm complexities for linear temporal logic model checking of push down systems. In *Proceedings of the 7th International Conference on Verification, Model Checking and Abstract Interpretation*. Springer-Verlag, Berlin, 190–206.

Hristova, K., Rothamel, T., Liu, Y. A., and Stoller, S. D. 2007. Efficient type inference for secure information flow. Technical Report DAR 07-35, Computer Science Department, SUNY Stony Brook. May. A preliminary version of this work appeared in *PLAS'06: Proceedings of the 2006 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*.

HRISTOVA, K., TEKLE, K. T., AND LIU, Y. A. 2007. Efficient trust management policy analysis from rules. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. 211–220.

JAFFAR, J. AND MAHER, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

JHA, S. AND REPS, T. W. 2004. Model checking SPKI/SDSI. *Journal of Computer Security 12,* 3-4, 317–353.

KIFER, M. AND LOZINSKII, E. L. 1986. A framework for an efficient implementation of deductive databases. In *Proceedings of the 6th Advanced Database Symposium*. IPS Japan, 109–116.

KIFER, M. AND LOZINSKII, E. L. 1990. On compile-time query optimization in deductive databases by means of static filtering. *ACM Trans. Database Syst 15,* 3, 385–426.

KOLAITIS, P. G. AND VARDI, M. Y. 1995. On the expressive power of datalog: tools and a case study. *Journal of Computer and System Sciences 51,* 1 (Aug.), 110–134.

LAM, M. S., WHALEY, J., LIVSHITS, V. B., MARTIN, M. C., AVOTS, D., CARBIN, M., AND UNKEL, C. 2005. Context-sensitive program analysis as database queries. In *PODS'05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM Press, New York, NY, USA, 1–12.

LEUSCHEL, M. 1998. Logic program specialisation. In *Partial Evaluation*, J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1706. Springer, 155–188.

LI, N. AND MITCHELL, J. C. 2003. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, V. Dahl and P. Wadler, Eds. Lecture Notes in Computer Science, vol. 2562. Springer-Verlag, 58–73.

LI, Q. AND MOON, B. 2001. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Databases*. 361–370.

LIU, Y. A. 2000. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation 13,* 4 (Dec.), 289–313.

LIU, Y. A., LI, N., AND STOLLER, S. D. 2001. Solving regular tree grammar based constraints. In *Proceedings of the 8th International Static Analysis Symposium*. Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, Berlin, 213–233.

LIU, Y. A., ROTHAMEL, T., YU, F., STOLLER, S., AND HU, N. 2004. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. Washington, DC, 219–230.

LIU, Y. A. AND STOLLER, S. D. 2002. Program optimization using indexed and recursive data structures. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. 108–118.

LIU, Y. A. AND STOLLER, S. D. 2003. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation 16,* 1-2 (Mar.-June), 37–62. Special issue in memory of Bob Paige.

LIU, Y. A. AND STOLLER, S. D. 2006. Querying complex graphs. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages*. Lecture Notes in Computer Science, vol. 3819. Springer-Verlag, Berlin, 199–214.

LIU, Y. A., STOLLER, S. D., AND TEITELBAUM, T. 1998. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst. 20,* 3 (May), 546–585.

LIU, Y. A., STOLLER, S. D., AND TEITELBAUM, T. 2001. Strengthening invariants for efficient computation. *Science of Computer Programming 41,* 2 (Oct.), 139–172.

LIU, Y. A. AND YU, F. 2002. Solving regular path queries. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 2386. Springer-Verlag, Berlin, 195–208.

LLOYD, J. W. AND SHEPHERDSON, J. C. 1991. Partial evaluation in logic programming. *J. Log. Program. 11,* 3&4, 217–242.

McAllester, D. A. 1999. On the complexity analysis of static analyses. In *Proceedings of the 6th International Static Analysis Symposium*. Lecture Notes in Computer Science, vol. 1694. Springer-Verlag, Berlin, 312–329.

Naughton, J. F. and Ramakrishnan, R. 1991. Bottom-up evaluation of logic programs. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. The MIT Press, Cambridge, Mass., 640–700.

Nielson, F., Nielson, H. R., and Seidl, H. 2002. Automatic complexity analysis. In *Proceedings of the 11th European Symposium on Programming*. Lecture Notes in Computer Science, vol. 2305. Springer-Verlag, Berlin, 243–261.

Paige, R. 1981. *Formal Differentiation: A Program Synthesis Technique*. Computer Science and Artificial Intelligence, vol. 6. UMI Research Press, Ann Arbor, Michigan. Revision of Ph.D. dissertation, New York University, 1979.

Paige, R. 1986. Programming with invariants. *IEEE Software 3,* 1 (Jan.), 56–69.

Paige, R. 1989. Real-time simulation of a set machine on a RAM. In *Computing and Information, Vol. II*. Canadian Scholars Press, 69–73. Proceedings of ICCI '89: The International Conference on Computing and Information, Toronto, Canada, May 23-27, 1989.

Paige, R. and Koenig, S. 1982. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst. 4,* 3 (July), 402–454.

Ramakrishnan, I. V., Sekar, R. C., and Voronkov, A. 2001. Term indexing. In *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds. Elsevier and MIT Press, Chapter 26, 1853–1964.

Reps, T. 1998. Program analysis via graph reachability. *Information and Software Technology 40,* 11-12 (Nov.), 701–726. Special issue on program slicing.

Sagonas, K., Swift, T., and Warren, D. S. 1994. XSB as a deductive database. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*.

Schrijvers, T. 2005. Analyses, optimizations and extensions of constraint handling rules. Ph.D. thesis, Katholieke Universiteit Leuven, Belgium.

Schwartz, J. T., Dewar, R. B. K., Dubinsky, E., and Schonberg, E. 1986. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York.

Snyder, W. K. 1990. The SETL2 Programming Language. Technical report 490, Courant Institute of Mathematical Sciences, New York University. Sept.

Stoller, S. D. and Liu, Y. A. 2007. Generating efficient security software from policies. In *Department of Defense Sponsored Information Security Research: New Methods for Protecting Against Cyber Threats*. John Wiley & Sons, New York, 416–424.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Proceedings of the 3rd International Conference on Logic Programming*, E. Shapiro, Ed. Springer-Verlag, Berlin, 84–98.

Tekle, K. T., Hristova, K., and Liu, Y. A. 2008. Generating specialized rules and programs for demand-driven analysis. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*. Springer-Verlag, Berlin, Urbana, Illinois.

Vardi, M. Y. 1982. The complexity of relational query languages (extended abstract). In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*. ACM Press, New York, NY, USA, 137–146.

Whaley, J. and Lam, M. S. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. 131–144.

Received Month Year; revised Month Year; accepted Month Year