# From Datalog Rules to Efficient Programs with Time and Space Guarantees*

Yanhong A. Liu Scott D. Stoller
Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794
liu@cs.sunysb.edu stoller@cs.sunysb.edu

## ABSTRACT

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. The associated space usage is optimal in that it is the minimum space needed for such consideration modulo scheduling optimizations that may 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

## Keywords

Complexity analysis, Datalog, data structure design, incremental computation, indexing, indexed representations, linked representations, program transformation, optimization, recursion, tabling

*This work was supported in part by NSF under grants CCR-0204280 and CCR-9876058 and ONR under grants N00014-01-1-0109 and N00014-02-1-0363.


*PPDP'03,* August 27–29, 2003, Uppsala, Sweden.


## 1. INTRODUCTION

Many computational problems are most clearly and easily specified using relational rules. Examples include database queries and problems in program analysis and model checking. Datalog [10, 2] 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.

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 implementation is typically for fast prototyping. 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 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. The associated space

usage is optimal in that it is the minimum space needed for such consideration modulo scheduling optimizations that may eliminate some summands in the space usage formula. For space complexity, our method separately analyzes the output space and the auxiliary space.

These two results are formally derived together using a systematic algorithm development method [31, 8, 30], 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

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

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 variables in $X_k$'s must be a subset of the variables in $X_{ij}$'s. If $h = 0$, then there are no $P_i$'s or $X_{ij}$'s, and $X_k$'s must be constants, 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 $h \geq 1$. 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*.

Such rules and facts are captured exactly by Datalog [10, 2], 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.

Variables occurring in exactly one hypothesis and not in the conclusion of a rule are called *wildcards*. Their names do not affect the meaning of the rule and can be replaced with underscores.

For ease of exposition, we give formal derivation for rules with at most two hypotheses, where wildcards 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 wildcards in rules with multiple hypotheses can be reduced to rules with at most two hypotheses and where wildcards 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:

```
form 2:  P1(X1s,Ys,C1s) ∧ P2(X2s,Ys,C2s)
           → Q2(X1s,X2s,Y's,C3s)
form 1:  P(Zs,As) → Q1(Z's,Bs)
form 0:  Q(Cs)
```
(2)

Each of `X1s`, `X2s`, `Ys`, `Y's`, `Zs`, and `Z's` abbreviates a group of variables, possibly including multiple occurrences of a variable. Each of `C1s`, `C2s`, `C3s`, `As`, `Bs`, and `Cs` abbreviates a group of constants, possibly including multiple occurrences of a constant. 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; if a shared variable occurs different times in the two hypotheses, we assume that the two `Ys`'s in the respective hypotheses capture the respective occurrences.

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 [34, 35] extended with a fixed-point operation [8]; 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. We use the notation below for pattern matching against tuples whose components may be constants or variables. It returns false if $X$ is not a tuple of length $n$ or if any $Y_i$ is a constant but the $i$th component of $X$ is not the same constant; otherwise, it binds each $Y_i$ that is a variable to the $i$th component of $X$.

$$X \textbf{ of } [Y_1 \ ... \ Y_n] \quad \text{matching } X \text{ against } [Y_1 \ ... \ Y_n]$$

We use the notation below for set comprehension. Each $Y_i$

| | |
|---|---|
| $\{X_1$ ... $X_n\}$ | a set with elements $X_1$,...,$X_n$ |
| $[X_1$ ... $X_n]$ | a tuple with elements $X_1$,...,$X_n$ in order |
| $\{[X_1\ Y_1]$ ... $[X_n\ Y_n]\}$ | a map that maps $X_1$ to $Y_1$, ..., $X_n$ to $Y_n$ |
| $\{\}$ | empty set |
| **exists** $X$ **in** $S$ | whether $S$ is empty and, if not, binding $X$ 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 $S$ is a subset of $T$ |
| $X$ **in** $S,\ X$ **notin** $S$ | whether or not, respectively, $X$ is an element of $S$ |
| $\#S$ | number of elements in 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** $M\}$ |

**Figure 1: Sets, tuples, maps, and operations on them.**

enumerates elements of $S_i$; for each combination 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. Each $Y_i$ can be a tuple, in which case an enumerated element of $S_i$ is first matched against it. If $Z$ is omitted, it is implicitly the constant *true*.

$$\{X : Y_1 \textbf{ in } S_1, ..., Y_n \textbf{ in } S_n \mid Z\} \quad \text{set former} \tag{3}$$

**LFP**$(S_0, F)$ denotes the minimum element $S$, with respect to the subset ordering $\subseteq$, that satisfies the condition $S_0 \subseteq S$ and $F(S) = S$. We use standard control constructs **while**, **for**, **if**, and **case**, and we use indentation to indicate scope. We abbreviate $X := X$ **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 forms 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`, and 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`. 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] :
           [P1 X1s Ys C1s] in R and
           [P2 X2s Ys C2s] in R}
```
(4)

The meaning of the given set of rules and facts is

`LFP(e0,F)`, where `F(R) = R+e1(R)+e2(R)`. (5)

Note 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.

[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.

**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 (5) is equivalent to

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

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
  R with:= x
```
(6)

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 [8], finite differencing [31, 29], and real-time simulation [30, 7], respectively, by Paige et al. Step 2 is the driving force; Liu [18] gives references to much work that exploits similar ideas. Step 3 is the enabling mechanism; the main difficulty here is that linked structures using based representations [30, 7] do not suffice, and sophisticated indexed structures must 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 each of the three steps, as proven by Paige et al. in [8, 31, 29, 30, 7], and the correctness of our extensions to Step 3. For Step 3, determining 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 (6) 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 (3) or involves high-level set operations such as union and difference. Therefore, the expensive expressions in (6) 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 (4)
E1 = e1(R) = as in (4)
E2 = e2(R) = as in (4)
W = e0+e1(R)+e2(R)-R = E0+E1+E2-R
```
(7)

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 `{}` with the initialization `R:={}`. `E1` can also be updated easily together with the update `R with:=x`: 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 `E2P2YsX2s`, indicating that it is used for maintaining `E2` and built from `P2` and maps variables in `Ys` to variables in `X2s`:

```
E2P2YsX2s = {[Ys X2s] : [P2 X2s Ys C2s] in R}
```
(8)

Note that we overloaded `Ys` and `X2s` in the auxiliary map to represent *sets* of variables, i.e., without duplicates and the order of elements does not matter. We don't care how many times these variables occur in either of the two hypothesis or the conclusion. Note also that, if arguments of the hypothesis `P2(X2s,Ys,C2s)` start with unduplicated variables in `Ys`, followed by unduplicated variables in `X2s`, and possibly followed by constants, then `E2P2YsX2s` is not needed and facts of `P2` that are in `R` can be used directly; 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

```
E2P1YsX1s = {[Ys X1s] : [P1 X1s Ys C1s] in R}
```
(9)

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 `E2edgewu = {[w u] : [edge u w]in R}`, which is an inverse of `edge`. No auxiliary map `E2pathwv` is needed, since facts of `path` that are in `R` can be used directly.

**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 (7), (8) and (9), when `R` is `{}`, we have:

```
E0 = {[Q Cs] : Q(Cs) in givenFacts}
E1 = {}
E2 = {}
W = E0 = {[Q Cs] : Q(Cs) in givenFacts}
E2P2YsX2s = {}
E2P1YsX1s = {}
```
(10)

and when `x` is added to `R` in the loop body, these variables can be updated as follows:

```
case x of [P Zs As]
  E1 with:= [Q1 Z's Bs]
  if [Q1 Z's Bs] notin R then W with:= [Q1 Z's Bs]
case x of [P1 X1s Ys C1s]
  E2 +:= {[Q2 X1s X2s Y's C3s]: X2s in E2P2YsX2s{Ys}}
  W  +:= {[Q2 X1s X2s Y's C3s]: X2s in E2P2YsX2s{Ys}
            | [Q2 X1s X2s Y's C3s] notin R}
  E2P1YsX1s with:= [Ys X1s]
case x of [P2 X2s Ys C2s]
  E2 +:= {[Q2 X1s X2s Y's C3s]: X1s in E2P1YsX1s{Ys}}
  W  +:= {[Q2 X1s X2s Y's C3s]: X1s in E2P1YsX1s{Ys}
            | [Q2 X1s X2s Y's C3s] notin R}
  E2P2YsX2s with:= [Ys X2s]
W less:= x
```
(11)

Adding these initializations and updates and using `W` in place of `e0+e1(R)+e2(R)-R` in (6), we obtain the following complete code. It is easy to see that `W` serves as the workset.

```
initialize using (10)
R := {}
while exists x in W                (12)
  update using (11)
  R with:= x
```

**Eliminating dead code and cleaning up.** To compute the result `R`, only `W`, `E2P2YsX2s`, and `E2P1YsX1s` are needed. So `E0`, `E1`, and `E2` are dead. Eliminating them from (12), we

obtain the following algorithm:

```
W := {[Q Cs] : Q(Cs) in givenFacts}
E2P2YsX2s := {}
E2P1YsX1s := {}
R := {}
while exists x in W
  case x of [P Zs As]
    if [Q1 Z's Bs] notin R then W with:= [Q1 Z's Bs]
  case x of [P1 X1s Ys C1s]
    W +:= {[Q2 X1s X2s Y's C3s]: X2s in E2P2YsX2s{Ys}
              | [Q2 X1s X2s Y's C3s] notin R}
    E2P1YsX1s with:= [Ys X1s]
  case x of [P2 X2s Ys C2s]
    W +:= {[Q2 X1s X2s Y's C3s]: X1s in E2P1YsX1s{Ys}
              | [Q2 X1s X2s Y's C3s] notin R}
    E2P2YsX2s with:= [Ys X2s]
  W less:= x
  R with:= x
```

(13)

Finally, the code is cleaned up to contain only uniform element-level operations for data structure design. That is, 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 **case**-clauses appropriately. Then, we do the following three sets of transformations.

(i) Straightforwardly transform set-level operations (unions here) into loops that use element-level operations. In particular, replace an assignment or addition of a set $\{X : Y \textbf{ in } S \mid Z\}$ to `W` with a **for**-loop that adds elements one at a time: **for** $Y$ **in** $S$ **if** $Z$ **then** `W` **with** $:= X$.

(ii) Replace tuples and tuple operations with maps and map operations. In particular, 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

$$\textbf{while exists } Z \textbf{ in } M \,...\, Z...$$

with

$$\begin{array}{l}\textbf{while exists } X \textbf{ in } \textbf{dom}(M)\\ \quad \textbf{while exists } Y \textbf{ in } M\{X\}\\ \quad\quad ...\,[X\,Y]...,\end{array}$$

replace **for**-loops in a similar way, and replace $M \neq \{\}$ with $\textbf{dom}(M) \neq \{\}$; finally, replace $[X\,Y]$ **notin** $M$ with $Y$ **notin** $M\{X\}$, and replace $M$ **with** $:= [X\,Y]$ with $M\{X\}$ **with** $:= Y$.

(iii) Make all element-level updates easy by testing membership first. In particular, replace $S$ **with** $:= X$ with **if** $X$ **notin** $S$ **then** $S$ **with** $:= X$. Note that `W less:=x` does not need the test `x in W`, since `x` is retrieved from `W`. Also, `R with:=x` does not need the test `x in R` since elements are moved from `W` to `R` one at a time and each element is put into `W` and thus `R` only once.

**Example.** For the transitive closure example, after representing `R` as `Redge` and `Rpath` and representing `W` as `Wedge` and `Wpath` but before applying the three sets of transformations, we obtain the algorithm in Figure 2. Note that the first two cases in (13) 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 `E2pathwv`.

# 5. DATA STRUCTURE DESIGN

We describe how to guarantee that each set operation in the cleaned-up version of (13) takes worst-case $O(1)$ time. The operations are of the following kinds: set initialization $S := \{\}$, computing domain set $\textbf{dom}(M)$, computing image set $M\{X\}$, 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 $\textbf{dom}(M)$).

**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 [4] 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. [30, 7] 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

> **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 $S$ after it has been located in $W$. The idea is to use a set $B$, called a *base*, to store values for both $W$ and $S$, such that retrieval from $W$ also locates the value in $S$. Base $B$ is represented as a set (this set is only conceptual) of records, with a $K$ field storing the key (i.e., value). Set $S$ is represented using a $S$ 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 $B$. This representation allows an arbitrary number of weakly based sets but only a constant number of

```
Wedge := {[u v] : edge(u,v) in givenFacts}
Wpath := {}
E2edgewu := {} //inverse map for edge
Redge := {}
Rpath := {}
while Wedge != {} or Wpath != {}
  while exists [u,w] in Wedge
    if [u w] notin Rpath then Wpath with:= [u w]         //rule 1
    Wpath +:= {[u v]: v in Rpath{w} | [u v] notin Rpath} //rule 2, and use of Rpath
    E2edgewu with:= [w u]                                //update of inverse map
    Wedge less:= x
    Redge with:= x
  while exists [w,v] in Wpath
    Wpath +:= {[u v]: u in E2edgewu{w} | [u v] notin Rpath} //rule 2, and use of inverse map
    Wpath less:= x
    Rpath with:= x
```

**Figure 2: Transitive closure algorithm after decomposing relations `R` and `W` and before other clean-up transformations.**

strongly based sets. Essentially, base $B$ provides a kind of indexing to elements of $S$ starting from elements of $W$.

However, often a non-constant number of sets must be strongly based for constant-time associative access [19, 24], and this is particularly the case here for compiling general forms of rules. Specifically, for the cleaned-up version of (13), 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, by tests of whether a fact of `Qi` to be added to `WQi` is already in `RQi` or `WQi`,

(ii) anchors of the auxiliary maps `E2PiYsXis`'s, by the image set operations in `E2PiYsXis{Ys}`'s, and

(iii) auxiliary maps `E2PiYsXis`'s, by tests of whether a tuple `[Ys Xis]` built from `Pi(Xis,Ys,Cis)` and to be added to `E2PiYsXis` 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 `E2PiYsXis` is built does not appear in any conclusion, because in that case, every tuple `[Ys Xis]` added to `E2PiYsXis` 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 (13), and

(ii) non-anchors of the auxiliary maps `E2PiYsXis`'s, by the nested **for**-loops in the cleaned-up version of (13) 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 `E2PiYsXis`'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. One or more records are used to put each array, linked list, and tail pointer together. 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 `E2PiYsXis` such that `Pi` appears in the conclusion of some rule 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. Associative access of kind (iii) above is not needed for other `E2PiYsXis`'s, so each other

E2PiYsXis 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.

Note that we did not discuss representations for relations Ri's that do not occur in the conclusion of any rule. 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 results 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 E2PiYsXis'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 (13), and E2P2YsX2s and E2P1YsX1s in each of the last two cases of (13). 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 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. Elements of the base are stored in an array indexed by the vertices, for efficient access of the first component of Rpath, Wpath, and E2edgewu. Each element u of the base array is a record.

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 the pair of u and v is an element of Rpath and null otherwise.

A similar WpathArray field is used for Wpath. A linked list 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; it is null if WpathArray is null.

A linked list with tail pointer is used to link vertices in the first component of Wedge. A WedgeQueue field of u is a linked list of successor vertices v for all tuples [u v] in Wedge.

An E2edgewuList field of u is used for the inverse map E2edgewu; it is a linked list of vertices v for all predecessors v of u.

**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 [4, 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 the rules 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 = ∅, 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 ≠ 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 ∩ J = ∅. For I = ∅, we take #P.I/J = 1. For J = ∅, we take #P.I/J = #P.I.

**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; and #edge.1/2 is the maximum number of predecessors of a vertex, i.e., the maximum in-degree of vertices.

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

#P = #P.{1,...,a} for relation P of a arguments
#P.i ≤ #D(P.i)
#P.I ≤ #P.J for I ⊆ J
#P.(I ∪ J) ≤ #P.I × #P.J/I and #P.J/I ≤ #P.J for I ∩ J = ∅

These imply commonly used constraints, including in particular #P ≤ #D(P.1) × ... × #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

#path.2/1 ≤ #path.2 ≤ #D(path.2) = #vertex
#path ≤ #D(path.1) × #D(path.2) = #vertex$^2$
#edge.1/2 ≤ #edge.1 ≤ #D(edge.1) = #vertex
#edge ≤ #D(edge.1) × #D(edge.2) = #vertex$^2$

**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

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

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

$$\texttt{r.\#firedTimes} \leq \texttt{min}(\texttt{\#P1} \times \texttt{\#P2.IX2s/IYs}, \texttt{\#P2} \times \texttt{\#P1.IX1s/IYs}).$$

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 `#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 $\texttt{\#edge} \leq \texttt{\#vertex}^2$, we obtain the worst-case time complexity $O(\texttt{\#edge}\times\texttt{\#vertex})$.

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. Basically, we can bound, for each rule, the number of instances of the conclusion based on the number of instances of the hypotheses combined, and we can bound the number of instances of a hypothesis by summing all the facts that are instances of the hypothesis based on the given facts and on the rules that can conclude these instances. These constraints are relatively straightforward to formalize based on the size characteristics defined above. We omit the details here.

**Space complexity and optimality.** 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 `E2PiYsXis`'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

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

  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.

- `E2PiYsXis`'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 `E2PiYsXis` 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 `E2PiYsXis` is

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

  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. Recall that `Ys` and `Xis` in auxiliary maps are *sets* of variables, and each variable may occur multiple times in arguments of `Pi`, so in general, the minimum space over all possible indices `IXis` for `Xis` and `IYs` for `Ys` is taken, and the total space is

  $$\texttt{\#DY1} \times ... \times \texttt{\#DYj} + \texttt{min}\{\texttt{\#Pi.(IYs}\cup\texttt{IXis)} - \texttt{\#Pi.IYs}: \text{ all possible } \texttt{IXis} \text{ and } \texttt{IYs}\}.$$

We call the space taken by result sets `RQi`'s *output space*, and the space taken by auxiliary maps `E2PiYsXis`'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 is optimal in that it is the minimum space needed to support all the operations in the derived algorithms modulo scheduling optimizations that may eliminate some summands in the space usage formula.

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

# 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 **case**-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 hypotheses.** For rules with more than two hypotheses, we can both transform them into rules with two hypotheses and generalize our derivation 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. 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.

Transforming rules is higher-level, more declarative, simpler, and clearer than treatment in the derivation process, but the space taken by the auxiliary relations may be unnecessary. Treatment in the derivation process considers all hypotheses directly, by trying all possible decompositions into fewer but more than two hypotheses and, for each part, following all possible sequences of adding one hypothesis at a time. The best running time this approach can achieve is the same as the transformational method. However, it 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.

**Datalog rules with wildcards in rules with multiple hypotheses.** We can either eliminate wildcards by simple transformations or handle them easily in the derivation process. Both approaches result in exactly 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 contains wildcards, we introduce an auxiliary relation to hold only the non-wildcard arguments of the hypothesis. Precisely, for every hypothesis of a rule that contains wildcards, 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 non-wildcard arguments of $P$, and (ii) add a new rule $P(X_1, ..., X_a) \rightarrow Q(X'_1, ..., X'_{a'})$. The only effect of this transformation 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 input space plus output space for the original given problem. In our derived algorithms for the transformed rules, all instances of a hypothesis that differ only in the wildcard components are considered only once together for different values of the wildcard components.

**Extension of Datalog rules with negation.** Datalog rules do not contain negated hypotheses, but negation is useful for expressing some analysis problems. The two most well-known semantics for Datalog with negation are stratified semantics and well-founded semantics [2].

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 other least fixed-point semantics, such as well-founded 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 [22, 18, 23], 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 [21, 20]. 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 describe 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 currently developing methods to combine magic sets transformations [6] 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.

We summarize the worst-case complexities of our derived algorithms for seven example problems in Table 1: transitive closure (the running example), graph reachability [8], four kinds of regular path queries [24, 14], and simplification of regular tree grammar based constraints [19].

Graph reachability finds all vertices reachable (`reach(v)`) starting from a given set of source vertices (`source(v)`) and following a given set of edges (`edge(u,v)`). Our derived algorithm uses no auxiliary maps; its time complexity is `#source`+`min(#reach`×`#edge.2/1,#edge)`, and its output space is `#reach`. These formulas give more information than the simplified formulas in Table 1. For example, when few vertices are reachable and the out-degrees of vertices are small, the running time is approximately `#source`+`#reach`, which is significantly better than `#source`+`#edge`.

Regular path queries of four kinds 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 and compute substitutions (`subst`) of variables to constants together with the matched vertices. `#param` is the number of parameters in the pattern. The output space has a factor of `#state`×`#vertex` because the rules infer a relation `match(v,s)` on vertex `v` and state `s`. All four kinds of queries naturally have a rule with three hypotheses. Decomposing it into two rules, each with two hypotheses, yields two possibilities: one has the complexities given in Table 1, and the other has running time $O($`#edge`×`#transition`$)$ and auxiliary space $O($`#vertex`$^2$+`#state`$^2)$, for non-parametric queries. Considering all three hypotheses together gives better auxiliary space, $O($`#label`×`#vertex`+`#label`×`#state`$)$, with running time $O($`#edge`×`#transition`$)$; these are the same as in [24]. The time complexity in Table 1 improves over [24, 14]; the trade-off is the additional factor `#label`, though very small in practice, in total space. For parametric queries, similar results hold, except with an additional factor of `#subst` in the complexities.

Simplification of regular tree grammar based constraints is used to analyze recursive data structures in programs [19]. 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 1 is the worst-case bound for `#simp` × `min(k` + `#copy.1/2,` `k` × `#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 [19], which helped better understand the practical performance of the algorithm. The analysis here is drastically simpler; it also improves over the analysis in [19] on some of the non-worst-case rules.

These and similar problems have applications in program analysis [13, 16, 33, 5], model checking [12], and queries of semi-structured databases [1, 3, 9, 17]. Many more analysis problems in these applications can easily be specified as Datalog rules and implemented with time and space guarantees using our method.

# 9. RELATED WORK AND CONCLUSION

Datalog and optimization methods for Datalog have been studied extensively in logic programming and database areas [10, 2]. 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 include smart evaluation methods and rewriting methods [10]. The former includes bottom-up evaluation [10, 26], semi-naive evaluation [10], and top-down evaluation with tabling [36, 11]. The latter includes magic sets transformation [6], among others [10]. 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 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 [10]. In fact, such analysis can be very difficult. For example, for top-down evaluation with tabling and indexing [36, 11, 32], 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 $O(\mathtt{n}^k)$ 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. This has been refined significantly by McAllester [25], using prefix firings to capture time com-

| problem | running time | output space | auxiliary space |
|---|---|---|---|
| transitive closure | $O(\texttt{\#edge}\times\texttt{\#vertex})$ | $O(\texttt{\#vertex}^2)$ | $O(\texttt{\#edge})$ |
| graph reachability | $O(\texttt{\#source}+\texttt{\#edge})$ | $O(\texttt{\#vertex})$ | $O(1)$ |
| existential and 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 and universal parametric RPQ | $O(\text{above}\times\texttt{\#subst}\times\texttt{\#param})$ | $O(\text{above}\times\texttt{\#subst})$ | $O(\text{above}\times\texttt{\#subst})$ |
| constraint simplification | $O(\texttt{\#node}^3)$ | $O(\texttt{\#node}^2)$ | $O(\texttt{\#node}^2)$ |

**Table 1: Summary of worst-case complexities for example applications.**

plexity. However, the time complexity there is sensitive to the order of hypotheses in the rules. Space complexity is not discussed. The proposed implementation method is interpretive, and uses space liberally for extensive hashing, so space consumption is often unnecessarily large, and there are no tight worst-case guarantees on time and space. Also, there was no attempt to automate the complexity analysis. A recent follow-up [27] discusses how to automate the complexity analysis but does not address the other limitations.

Our derivation of complete algorithms and data structures from fixed-point specifications uses Paige's method [28, 31, 29, 8, 30, 7] for languages like SETL [34, 35], except that based representations [30, 7] in his method do not apply and a more general and sophisticated combination of arrays, linked lists, and records is needed. Paige's method also did not handle union types, i.e., sets with different element types. Our precise complexity analysis for both time and space as well as the trade-offs, and 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 [13, 16, 33, 5, 15], 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 are currently implementing the method and developing techniques to handle 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, David McAllester, Anil Nerode, CR Ramakrishnan, Ganesan Ramalingam, and David Warren, helped greatly in understanding related work.

## 10. REFERENCES

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

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

[3] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 122–133. ACM, New York, 1997.

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

[5] A. Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, 1999.

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

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

[8] J. Cai and R. Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11:197–261, Sept. 1988/89.

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

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

[11] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, Jan. 1996.

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

[13] P. Cousot and R. Cousot. 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*, pages 238–252. ACM, New York, 1977.

[14] O. de Moor, D. Lacey, and E. V. Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1-2), 2003. To appear.

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

[16] N. Heintze and J. Jaffar. Set constraints and set-based

analysis. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 281–298. Springer-Verlag, Berlin, 1994.

[17] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 361–370, 2001.

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

[19] Y. A. Liu, N. Li, and S. D. Stoller. Solving regular tree grammar based constraints. In *Proceedings of the 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 213–233. Springer-Verlag, Berlin, 2001.

[20] Y. A. Liu and S. D. Stoller. Program optimization using indexed and recursive data structures. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 108–118. ACM, New York, 2002.

[21] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1-2):37–62, Mar.-June 2003. An earlier version appeared in *Proceedings of the 8th European Symposium on Programming*, 1999.

[22] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998. An earlier version appeared in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1995.

[23] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Strengthening invariants for efficient computation. *Sci. Comput. Program.*, 41(2):139–172, Oct. 2001. An earlier version appeared in *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, 1996.

[24] Y. A. Liu and F. Yu. Solving regular path queries. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*, volume 2386 of *Lecture Notes in Computer Science*, pages 195–208. Springer-Verlag, Berlin, 2002.

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

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

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

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

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

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

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

[32] I. V. Ramakrishnan, R. C. Sekar, and A. Voronkov. Term indexing. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 26, pages 1853–1964. Elsevier and MIT Press, 2001.

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

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

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

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