Stony Brook University

Efficient Datalog Queries with Time and Space Complexity Guarantees

A Dissertation Presented

by

Kazım Tuncay Tekle

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2010

Stony Brook University

The Graduate School

Kazım Tuncay Tekle

We, the dissertation committee for the above candidate for
the degree of Doctor of Philosophy,
hereby recommend the acceptance of this dissertation.

Yanhong A. Liu – Dissertation Advisor

Professor, Computer Science Department

David S. Warren – Chairperson of Defense

Professor, Computer Science Department

Michael Kifer – Committee Member

Professor, Computer Science Department

Patrick Cousot – External Committee Member

Professor, École Normale Supérieure, Paris

This dissertation is accepted by the Graduate School.

Lawrence Martin

Dean of the Graduate School

Abstract of the Dissertation

Efficient Datalog Queries with Time and Space Complexity Guarantees

by
Kazım Tuncay Tekle

Doctor of Philosophy
in
Computer Science
Stony Brook University
2010

Many complex analysis problems can be most clearly and easily specified as logic rules and queries, where rules specify how given facts can be combined to infer new facts, and queries select facts of interest to the analysis problem at hand. However, it has been extremely challenging to obtain efficient implementations from logic rules and understand their time and space complexities, particularly for answering queries of interest without inferring all facts.

This dissertation focuses on Datalog—an important class of rules for applications in databases, program analysis, security, semantic web, and many other areas. We describe a systematic method for precisely analyzing the time and space complexities of best known strategies for answering Datalog queries. We also characterize precise relationships among these strategies. Furthermore, we develop new transformations and combine them with existing transformations to drastically optimize the rules and queries for generating efficient implementations. Finally, we show the effectiveness of our analyses and transformations for solving important problems in program analysis, language processing, and semantic web, and for answering graph queries, which have many applications.

et sur ma thèse
j'écris ton nom

To liberty, equality, fraternity

Contents

List of Figures **vii**

1 Introduction **1**
1.1 Declarative languages and challenges 1
1.2 Analyses and optimizations for efficient implementations . . . 3

2 Datalog **5**
2.1 Syntax and semantics 5
2.2 Top-down evaluation 8
2.3 Bottom-up evaluation 10

3 Query-driven evaluations and their complexity analysis **13**
3.1 Complexity analysis for top-down evaluation with variant tabling 14
3.1.1 Binding annotation 17
3.1.2 Time complexity analysis 20
3.1.3 Space complexity analysis 21
3.2 Variant demand transformation for bottom-up evaluation . . 22
3.2.1 Variant demand transformation 23
3.2.2 Comparing with magic set transformation 24
3.3 Relating top-down evaluation with variant tabling and bottom-up evaluation after variant demand transformation 25
3.3.1 Time complexity comparison 25
3.3.2 Space complexity comparison 28
3.4 Experiments 29
3.5 Related work 31

4 Subsumptive tabling beats variant tabling and magic sets **33**
4.1 Complexity analysis for subsumptive tabling 35

4.1.1 Subsumptive binding annotation and complexity analysis . . . 36
4.1.2 Subsumptive beats variant and magic sets . . . 40
4.2 Subsumptive demand transformation for bottom-up evaluation . . . 43
4.2.1 Relationship to subsumptive tabling and magic set transformation . . . 47
4.3 Subsumption optimization . . . 49
4.4 Experiments . . . 52
4.5 Related work . . . 54

5 Specialization and recursion conversion 58
5.1 Static removal of redundancies using specialization . . . 59
5.2 Recursion conversion for chain queries . . . 62
5.2.1 Complexity comparison . . . 65
5.3 Related work . . . 67

6 Applications 68
6.1 Program pointer analysis . . . 69
6.2 Context-free grammar parsing . . . 74
6.3 Ontology queries . . . 76
6.4 Graph queries . . . 79
6.4.1 Graph query language . . . 80
6.4.2 Example graph queries for program analysis . . . 82
6.4.3 Generating efficient implementations . . . 83
6.4.4 Demand transformation and graph queries . . . 92
6.4.5 Effect of transformations on graph queries . . . 93
6.4.6 Related work . . . 97

7 Conclusion and future work 99

Bibliography 102

List of Figures

2.1 Prolog evaluation for the query and first set of rules in Example 2.2.1. 9
2.2 Prolog evaluation for the query and second set of rules in Example 2.2.1. 9
2.3 Exponential-time `edge`s for left-recursive transitive closure . . 10

3.1 V-topdown evaluation of query q, given a set of facts F and a set of rules R 18
3.2 Returns for v-topdown evaluation, firings for bottom-up evaluation, and space units for both, for benchmark *Join1*. The difference between returns and firings has been multiplied by 1000 for illustration. 30
3.3 Running time and memory usage of transitive closure in XSB for a query with both arguments bound. 30
3.4 Returns for v-topdown evaluation, firings for bottom-up evaluation, and space units for both, for the same generation benchmark. 31

4.1 Top-down evaluation of query q, given a set of facts F and a set of rules R, with subsumptive tabling and *local scheduling* 55
4.2 Top-down evaluation of query q, given a set of facts F and a set of rules R, with subsumptive tabling and *batched scheduling* 56
4.3 Firings/returns and space units for v-topdown, s-topdown, v-bottomup, and s-bottomup for the running example. 57
4.4 Firings/returns and space units for s-bottomup with and without subsumption optimization (SO) for the pointer analysis benchmark. 57

5.1 Algorithm for optimization of rules using specialization and recursion conversion. 64

5.2 A comparison of time complexities of computation using existing methods. 65

6.1 Running time of rules resulting from heuristics in XSB and YAP 78
6.2 Grammar for the graph query language. 81
6.3 Example queries for program analysis. 83
6.4 Time complexities for the original rules and the rules after demand transformation. 89
6.5 Running time of the implementation of rules in C++ at different implementation stages. 95
6.6 Memory usage of the implementation of rules in Python at different implementation stages. 96
6.7 Running time in milliseconds of implementations generated by our method, of the generated rules in XSB, and of the manually found best version of these rules in XSB, and similarly for bddbddb. - denotes incompletion in 10 minutes. . . . 96

Acknowledgements

Getting a Ph.D. is a long journey. Getting a Ph.D. away from home is even longer. At the end of this long journey, I believe that it is very hard to do justice to people involved in the process. I will try nonetheless.

First off, I could not have imagined a better advisor than my own advisor Prof. Annie Liu. During this long journey, her academic guidance was great, her personal guidance even better. In research, she showed me how to find great problems to do research on, how to solve the problems, how to present the solutions, and how to refine the presentations. In personal terms, she set an example of sincerity and compassion, and showed that one can do good research with these qualities. When I failed, she showed me the way through; when I succeeded, she showed me the way to succeed further. This dissertation would not have been possible without her.

My committee members, Prof. David Warren and Prof. Michael Kifer, continuously provided invaluable feedback for my research. Their guidance gave me direction on the problems to attack, and how to fit the solutions in context. My endless questions to retrieve information from their wide expertise were always welcome and answered in the friendliest and keenest way possible. Prof. Patrick Cousot kindly agreed to be on my thesis committee, and provided excellent feedback to shape my future research as well. I thank all of you.

Our lab provided a great environment for both intellectual stimulation and friendly conversation. Whether discussing research or chatting about a random subject, I was always glad spending time with my labmates. I thank you all in chronological order of meeting: Katia Hristova, Tom Rothamel, Michael Gorbovitski, Puneet Gupta, Jon Brandvein, Bo Lin.

Friendship in Stony Brook was precious, and the friends I found knew how to put a smile on my face. I cannot thank you enough: Ankush, Arzu, Asım, Basia, Celine, Costas, Güneş, İrem, José, Marija, Manas, Max, Nik, Tom. Friends from Turkey spread around the U.S. made the journey much better than it would be otherwise. I thank Emre, Osman, Mert and Gülfem

the most among these.

Leaving my family and friends in Turkey was the hardest part. My parents and my sister always supported me and inspired me to do better. My friends in Turkey were always supportive in times of desperation and fueled me with energy on each vacation. I will not enumerate your names here, but you know who you are. Without your support from 5000 miles away, this day would never come.

Most of the time, women are much less appreciated than they should be. To take a small step in changing this, I would like to conclude by paying homage to three women who taught me almost everything I know. My mom, Fatma Günal, was my first teacher; she taught me both my first words and how to be a good person even when most people are not. My first formal teacher in primary school, Meral Demiray, taught me both my first formal classes and that anything is possible if you work for it. My last formal teacher and Ph.D. advisor, Annie Liu, taught me both how to do great research and how to be a good person doing it. I will try very hard to succeed, because I know that my success will make you proud.

Chapter 1

Introduction

1.1 Declarative languages and challenges

Many complex analysis problems can be most effectively and easily described using a declarative language. The declarative specification makes it easy to understand the nature of the problem, without being distracted by implementation details. One way of writing a declarative specification is to write logic rules and queries.

Logic rules specify how given facts in a problem setting can be combined to infer new facts. For example, for program analysis, definitions of flow and dependence relations can be specified as rules; for model checking, definitions of system behaviors can be specified as rules; and for system security, access control policies can be specified as rules.

Once the specification of a problem is given by logic rules, queries can be used to select facts of interest to the analysis problem at hand. For program analysis, flow and dependence information involving particular program points of interest can be specified as queries; for model checking the properties to be checked can be specified as queries; and for system security, checking access to resources by users can be specified as queries. Queries can be used to filter the facts inferred by the rules, and moreover be a guide in the inference of the facts of interest. We use *query-driven evaluation* to refer to an evaluation that is expressed by a query, querying over facts that can be inferred from the rules.

Despite the clarity and ease-of-use of rules, efficient implementations of rules have been extremely challenging, as seen in the large amount of existing work in logic programming and deductive databases. Also, understanding

the time and space complexities of logic programs has been just as challenging, if not more. This is especially so for answering rule-based queries on demand, e.g. for top-down evaluation of rules, because the queries and given facts can be far apart, and can be connected via the rules in many ways.

Furthermore, the running times of implementations using these methods can vary dramatically depending on seemingly small changes in recursive rules, and the orders of hypotheses in rules, and even less is known about the space usage. Developing efficient implementations for answering queries on-demand for any given rules and queries with time and space guarantees is a nontrivial, recurring task.

In this work, we address these challenges for an important class of logic rules, *Datalog* [15], used in deductive databases [1], program analysis [77], security [25], and many other applications [44, 32, 63].

Given a set of Datalog rules, facts, and a query, answers to the query can be inferred using bottom-up evaluation starting with the facts or top-down evaluation starting with the query. Many evaluation methods have been studied [14, 7, 34, 71, 72], notably including top-down evaluation with tabling [66] that guarantees termination in polynomial time, and optimal bottom-up evaluation with complexity guarantees [51] after program transformations such as the magic set transformation [8].

Despite extensive research on improving Datalog evaluation methods, and on optimizing Datalog programs, e.g., [13, 16, 55, 65, 24], the performance of rule engines remains little understood [45, 46]. In particular, performance differences using different evaluation methods are most often drastic, and even using the same evaluation method, changing the order of hypotheses in rules most often yields dramatically different performance that is easily observed to be asymptotic. Recent work studied efficient bottom-up evaluation with precise complexity guarantees [51], but precise complexities for efficiently answering queries using top-down evaluation with tabling remain unknown. Significant research in relating various top-down and bottom-up evaluation methods exist, but a large gap remains in precisely relating top-down evaluation with tabling and demand-driven bottom-up evaluation.

1.2 Analyses and optimizations for efficient implementations

This work first describes precise time and space complexity analysis for efficiently answering Datalog queries, and precise relationships between top-down evaluation with the dominant tabling strategy, *variant tabling* and demand-driven bottom-up evaluation obtained by a novel transformation called *variant demand transformation*.

We present a systematic method for precisely calculating the worst-case time and space complexities of top-down evaluation with variant tabling. The calculation is based on possible binding patterns of arguments of predicates during the evaluation, and expresses the complexities in terms of parameters that characterize the actual number of facts used.

We then describe variant demand transformation (VDT), which transforms Datalog rules for efficiently answering queries using bottom-up evaluation of the transformed rules. The transformation is akin to the magic set transformation, but is simpler and produces simpler rules that yield exponentially smaller space in the number of arguments of predicates.

Additionally, we establish precise relationships between top-down evaluation with variant tabling and bottom-up evaluation after VDT, in terms of precise time and space complexities; and confirm our analysis results through experiments on benchmarks from OpenRuleBench [45].

This work then discusses another tabling strategy for top-down evaluation, called *subsumptive tabling* [58], performing more reuse of previously inferred answers than variant tabling, and its relationship to other methods. This work gives precise time and space complexity analysis for efficiently answering Datalog queries using subsumptive tabling, and precise relationships between different tabling strategies and magic set transformation. We show that subsumptive tabling is equal to or better than variant tabling in time and space complexities. Furthermore, we characterize a class of Datalog rules and queries, for which subsumptive tabling is guaranteed to be better than variant tabling in time and space complexities. We also show that subsumptive tabling is guaranteed to be better than MST and VDT for the identified class of rules in time and space complexities.

Despite extensive research on optimizing the evaluation of Datalog rules, given a set of rules, a query, and a fixed order of hypotheses for each rule, no source-level transformation that leads to guaranteed better time complexity than MST or mimicking subsumptive tabling is known to the best of our knowledge. This work describes a transformation, called *subsumptive*

demand transformation (SDT) such that bottom-up evaluation of rules produced by SDT achieves the complexity performance of subsumptive tabling. We show that for rules that have no more than two hypotheses and no wildcards, time complexities of bottom-up evaluation after SDT and subsumptive tabling are equal, and that bottom-up evaluation after SDT may be better otherwise. Therefore, we give the first transformation method for bottom-up evaluation that beats MST in time complexity.

Building on our analyses, we devise a transformational method for forcing queries to be subsumed when it is better to do so in time complexity. Using this method, we show how to systematically derive Heintze and Tardieu's demand-driven pointer analysis [33] from the definition of Andersen's pointer analysis that provide precise complexity guarantees.

Then, we describe powerful transformational methods for optimization of Datalog queries. These include recursion conversion to transform recursive rules into appropriate left or right linear recursive forms based on the kinds of queries, so that the connection between the queries and given facts can be established efficiently, and specialization to remove unnecessary predicates, rules, and constant arguments. The complexity analyses are employed for selecting the most efficient implementation of the query after the transformations are applied.

We show the effectiveness of our analyses and transformations for solving important problems. In program analysis, we show an extensive analysis for Andersen's pointer analysis. In language processing, we show the effectiveness of our methods on rules for parsing context-free grammars. In semantic web, we analyze and optimize ontology queries. In answering graph queries, which have many applications, we show a method for transforming graph queries to Datalog rules and queries, and we obtain improved complexities in contrast to the original implementations by using the powerful analyses and optimizations described.

The rest of this work is organized as follows. Chapter 2 describes Datalog and major evaluation strategies. Chapter 3 describes the complexity analyses for query-driven evaluation strategies, and describes transformations for obtaining query-driven bottom-up evaluation. Chapter 4 describes complexity analyses for subsumptive tabling, SDT and properties of these two strategies. Chapter 5 describes transformational methods for optimization. Chapter 6 describes applications. Chapter 7 concludes the thesis.

Chapter 2

Datalog

Datalog is a language for defining rules, facts, and queries, where rules can be used with facts to answer queries. In this chapter, we describe the syntax and semantics of Datalog, extensions to Datalog, the terminology used throughout the work, and two major evaluation strategies for Datalog queries.

2.1 Syntax and semantics

A Datalog rule is of the form:

$$p(a_1, ..., a_k) :- p_1(a_{11}, ..., a_{1k_1}), ..., \ p_h(a_{h1}, ..., a_{hk_h}).$$

where h is a finite natural number, each p_i (respectively p) is a predicate of finite number k_i (respectively k) arguments, each a_{ij} and a_i is either a constant or a variable, and each variable in the arguments of p must also be in the arguments of some p_i.

A predicate with arguments is called an *atom*. The $:-$ operator (read *if*) splits the rule into two parts: the left side and the right side. If the right side of a rule is empty, the atom on the left must have only constant arguments, and is called a *fact*; indicated with an ending dot. The left side of a rule cannot be empty. For the rest of this work, "rule" refers only to the case where both sides of the rule are not empty, where each atom on the right is called a *hypothesis*, and the atom on the left is called the *conclusion*. A Datalog *query* is an atom followed by a question mark.

For defining the semantics of Datalog rules, we give the following definitions.

- A *substitution* θ is a map from variables to constants or variables. A

substitution θ can be applied to an atom a, denoted $\theta(a)$, which results in an atom identical to a except each variable that appears in θ has been replaced with the corresponding value.

- A fact f *matches* an atom a, if there exists a substitution θ such that $f = \theta(a)$.

For a rule r whose conclusion is c, if there exists a substitution θ such that for each hypothesis h of r, $\theta(h)$ is a fact, then $\theta(c)$ can be *inferred* as a fact.

The meaning of a set of rules, facts, and a query is the set of facts that are given or can be inferred using the rules and that match the query.

Example 2.1.1. For a graph whose edges are given as the facts of a binary `edge` predicate, the transitive closure of edges can be specified in Datalog as follows, where `x`, `y`, and `z` are variables:

```
path(x,y) :- edge(x,y).              (B)
path(x,y) :- path(x,z), edge(z,y).   (L)
```

A query for this set of rules is `path(c,x)?`, where `c` is a constant, whose meaning is the set of facts of `path` whose first argument is `c`. The second argument of the set of facts in the meaning of the query will be vertices that are reachable from `c`.

Notation and other terminology. In examples, we use `c`, $\texttt{c}_1$, $\texttt{c}_2$, etc. for constants, and the other letters for variables.

Two atoms a and a' *unify* if there are two substitutions θ and θ' of variables such that $\theta(a) = \theta'(a')$.

An *IDB (intensional database) predicate* is a predicate defined by the rules, and an *EDB (extensional database) predicate* is a predicate for which facts are given. A hypothesis is called an *IDB hypothesis* if its predicate is an IDB predicate, and an *EDB hypothesis* otherwise.

For complexity calculation, we use the following notations.

- `#p`: the number of facts of predicate `p`, called the *size of* `p`.
- `#p.`$\texttt{i}_1$`,...,`$\texttt{i}_n$`/`$\texttt{j}_1$`,...,`$\texttt{j}_m$: the maximum number of combinations of different values of the $\texttt{i}_1$,...,$\texttt{i}_n$th arguments of the facts of predicate `p` (given or inferred), given any fixed value for the $\texttt{j}_1$,...,$\texttt{j}_m$th arguments.

- `#p.i`: *actual* number of values of the ith argument of a particular instance of `p`.
- `dom(p.i)`: the size of the domain of the `i`th argument of predicate `p`, i.e., the number of all possible values of that argument of `p`.

A set of rules is said to be in *minimal form*, if there exists no more than two hypotheses in each rule and each variable appears in either two hypotheses or one hypothesis and the conclusion. Any set of Datalog rules can be trivially transformed into minimal form.

Negation. Datalog may be extended by allowing negated hypotheses. Arbitrary negation leads to semantics issues as seen in the following example.

Example 2.1.2. Consider the following rules, where `not` denotes negation.

```
p(x) :- r(x), not q(x).
q(x) :- p(x).
r(c).
```

Is `q(c)` in the meaning of the rules and fact above? Since `r(c)` is a fact, and `q(c)` is not a fact, `p(c)` should be a fact due to the first rule. However, due to the second rule, since `p(c)` is a fact, `q(c)` should be a fact as well, which contradicts our previous assumptions. Therefore, arbitrary negation leads to contradictory facts.

There are various semantics proposed for Datalog with arbitrary negation, including well-founded semantics [29], stable-model semantics [30], and inflationary semantics [39].

There is a form of negation called *stratified negation* for which an intuitive semantics exists and can be efficiently evaluated. Stratified negation disallows negation on each hypothesis which is mutually recursive with the conclusion of its rule. Formally speaking, given a set of rules, we construct a graph G whose nodes are predicates. There is an edge from predicate p to predicate q in G if there is a rule whose conclusion's predicate is p and which contains a hypothesis of predicate q; and the edge is labeled $\neg$ if the hypothesis is negated. If there exists no cycle containing an edge with label $\neg$ in G, then we say that the rules are stratified.

If a set of rules is stratified, for rule r whose conclusion is c, if there exists a substitution θ such that for each *positive* (non-negated) hypothesis h of r, $\theta(h)$ is a fact, and for each negated hypothesis n of r, $\theta(n)$ cannot be inferred as a fact, then $\theta(c)$ can be *inferred* as a fact.

In this work, we will mainly consider rules without negation, but will explicitly state the type of negation and the semantics used when we use negation.

2.2 Top-down evaluation

To answer a query, standard Prolog evaluation [52] starts with the query, generates subqueries from hypotheses of rules whose conclusions match the query, considering rules in the order given, and considering hypotheses from left to right, and does so repeatedly until the subqueries match given facts.

For Datalog queries, Prolog evaluation may suffer from repeated subqueries or infinite recursion when recursive rules exist. For a given query, the time to answer the query is also highly dependent on the order of rules and the order of hypotheses within rules. The following example illustrates that the difference may even be from finite time to infinite time.

Example 2.2.1. Consider the following fact and rules:

```
q(c1,c2).
p(x,x).
p(x,z) :- q(x,y),p(y,z).
```

and the query `p(x,c2)?`.

We represent Prolog evaluation by a tree, where each branch represents using a rule to find facts for the next hypothesis. The tree in Figure 2.1 represents the evaluation of the query for the rules and fact above. The answers to the query are given in finite time as can be seen from the evaluation tree.

Given the same query, same fact, same first rule, but reordering the second rule's hypotheses to be:

```
p(x,z) :- p(y,z), q(x,y).
```

Prolog evaluation of the query no longer terminates in finite time. The evaluation tree is shown in Figure 2.2.

It may seem conceivable that there exists an ordering of hypotheses and rules for any set of rules, such that Prolog evaluation will terminate. However, the following example shows that it is not possible. Therefore, Prolog evaluation is inherently not suitable for Datalog queries, when recursive rules exist.

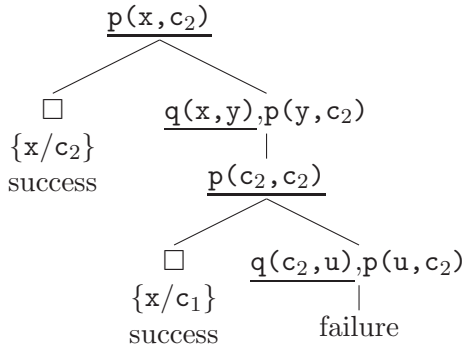

Figure 2.1: Prolog evaluation for the query and first set of rules in Example 2.2.1.

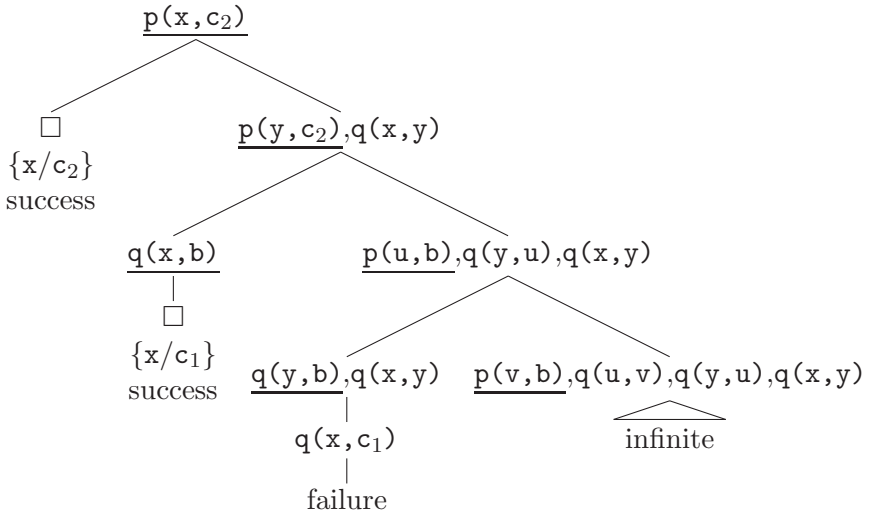

Figure 2.2: Prolog evaluation for the query and second set of rules in Example 2.2.1.

Example 2.2.2. Consider the following facts and rules:

```
p(c1,c2).
p(c2,c3).
p(x,y) :- p(y,x).
p(x,z) :- p(x,y),p(y,z).
```

and the query `p(c1,c3)?`. It is obvious that the query is a fact, due to the second rule and the given facts. However, there exists no ordering of rules

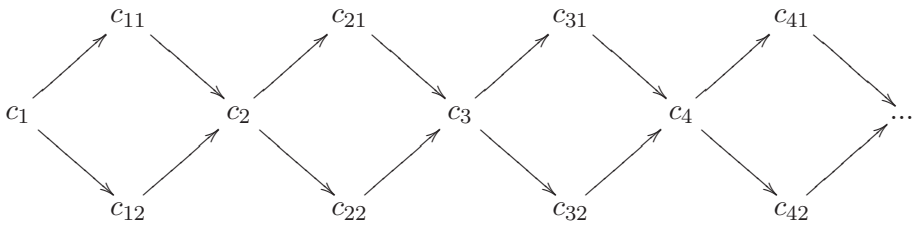

Figure 2.3: Exponential-time `edge`s for left-recursive transitive closure

and hypotheses such that Prolog evaluation terminates for this query.

Prolog evaluation may be terribly inefficient for Datalog queries, even when it does terminate. Vardi [69] showed that Datalog is P-complete for data complexity, therefore it should be possible to evaluate Datalog queries in polynomial time in the size of the given facts.

Given the rules in Example 2.1.1 (left-recursive definition of transitive closure) and query `path(`c_1`,y)?`, when the `edge` predicate is defined by the edges shown in Figure 2.3, Prolog evaluation takes exponential time in the number edges.

For efficiently answering a Datalog query with top-down evaluation, *tabling* [18] is used. Tabling is a strategy for storing and reusing facts inferred for subqueries to avoid repetitive evaluation. In the next chapters, we discuss different tabling strategies in detail.

2.3 Bottom-up evaluation

Bottom-up evaluation starts with given facts, infers new facts from conclusions of rules whose hypotheses match existing facts, and does so repeatedly until all facts are inferred.

The most basic strategy for bottom-up evaluation is *naive evaluation* [3, 67]. For a set of rules R, a set of facts F, and a query Q, the algorithm for naive evaluation is shown in Algorithm 0.

Naive evaluation always terminates finitely because inference for Datalog queries is monotonic, i.e., adding new facts can only result in the inference of more facts. However, it is inefficient. Without specifying any techniques for evaluating efficiency, consider the following. Suppose we call the computation of D_{i+1} a function of R and D_i, i.e., $D_{i+1} = f(R, D_i)$ where f infers facts using R given facts in D_i at one step. Therefore, at step i of the

Algorithm 0 Naive Evaluation

$D_0 = F$
$i = 0$
repeat
  $U =$ all possible facts inferable using R at one step, given facts in D_i.
  $D_{i+1} = D \cup U$
  $i = i + 1$
until $D_i = D_{i-1}$
Filter D_i wrt Q

loop in the algorithm, we compute $D_{i+1} = f(R, D_i)$. Since f is monotonic, $D_i \subseteq D_{i+1}$. Therefore, we can write

$$f(R, D_i) = D_i + df(R, D_i)$$

where *df* denotes the new facts inferred at step i. Step $i+1$ of the algorithm will compute

$$D_{i+2} = f(R, D_{i+1}) = f(R, D_i + df(R, D_i))$$

Therefore, at step $i+1$, the algorithm infers facts in $f(R, D_i)$ again, which were already inferred at step i. Thus, naive evaluation performs duplicate computations.

Instead of computing the entire set $f(I, D_i)$, one should only compute the effect of *df* if possible, since $f(I, D_{i-1})$ is already known. Using a definition of *df* and only inferring facts using the difference at each step in the loop of naive evaluation yields *semi-naive evaluation* [7]. However, finding *df* is hard in general, and particular *df* functions have been shown for some subclasses of rules [7].

The method of Liu et al. [51] takes the idea of differentiating even further, and generates bottom-up implementations that processes facts one by one, and maintains auxiliary maps for efficient retrieval of relevant facts at each step. For best time complexity, this method decomposes any rule that has more than two hypotheses into a set of rules of two hypotheses. In this work, we decompose the hypotheses from left to right. We call this method *left-optimal bottom-up evaluation*, because the time complexity of evaluating a set of rules using this method is optimal for the fixed left-to-right ordering of the hypotheses in a rule.

- The time complexity incurred by each rule for this method is the *number of firings* of the rule—the number of combinations of facts that make all hypotheses true.
- The space complexity of this method consists of the space used by the inferred facts, and the space used by auxiliary maps as indices for constant time retrieval of relevant facts.

For the rest of this work, *bottom-up evaluation* refers to left-optimal bottom-up evaluation.

Bottom-up evaluation infers all facts that can possibly be inferred without taking the given query into account, and thus may take asymptotically more time than necessary. To take the query into account, we perform *demand transformation*, which is discussed in the next chapter in detail.

Chapter 3

Query-driven evaluations and their complexity analysis

This chapter describes precise time and space complexity analysis for efficiently answering Datalog queries, and precise relationships between top-down evaluation with the dominant tabling strategy, *variant tabling*, and bottom-up evaluation after *variant demand transformation* (VDT), a novel transformation for making bottom-up evaluation query-driven.

We first present a systematic method for precisely calculating the worst-case time and space complexities of top-down evaluation with variant tabling. The calculation is based on possible binding patterns of arguments of predicates during the evaluation, and expresses the complexities in terms of parameters that characterize the actual number of facts used. We then describe *variant demand transformation*, which transforms Datalog rules for efficiently answering queries using bottom-up evaluation of the transformed rules. The transformation is akin to the magic set transformation, but is simpler and produces simpler rules that yield exponentially smaller space in the number of arguments of predicates.

Additionally, we establish precise relationships between top-down evaluation with variant tabling and bottom-up evaluation after VDT, in terms of precise time and space complexities. We show that the time complexity of bottom-up evaluation after VDT is better than or equal to top-down evaluation with variant tabling, and that for rules that have no more than two hypotheses and no wildcards (i.e., rules in minimal form), their complexities are equivalent. Then, we show that the space complexity of top-down evaluation with variant tabling is better than or equal to demand-driven

bottom-up evaluation, and that if the time complexity of bottom-up evaluation after VDT is better than top-down evaluation with variant tabling, then its space complexity must be worse. We confirm our analysis results through experiments on benchmarks from OpenRuleBench [45].

Notation. We refer to the different evaluation methods to be described as follows:

- *V-topdown*: Top-down evaluation with variant tabling
- *V-bottomup*: Bottom-up evaluation after variant demand transformation

The precise descriptions of these methods will be given in the consequent sections. The asymptotic time complexities of the above methods are denoted $T_{v-topdn}$ and $T_{v-botup}$. Similarly, asymptotic space complexities are denoted with S and the corresponding subscript. For asymptotic time complexity analysis, we assume perfect hashing, i.e., finding the value for a key in a hash map takes $O(1)$ time. For space complexities, we do not consider the stack space used by the methods, therefore we only consider the space taken by the subqueries generated and facts inferred.

3.1 Complexity analysis for top-down evaluation with variant tabling

To answer a query, top-down evaluation starts with the query, generates subqueries from hypotheses of rules whose conclusions match the query, considering rules in the order given, and considering hypotheses from left to right, and does so repeatedly until the subqueries match given facts. This may lead to repeated subqueries or infinite recursion when recursive rules exist. To address this problem, *tabling* memoizes answers to subqueries, and reuses them when possible.

In this chapter, we consider top-down evaluation using variant tabling with depth-first scheduling and without early completion.

- *Variant tabling* [18] is the dominant tabling strategy. It stores and reuses the answers to *variant*s of previously encountered subqueries, where a subquery is a variant of another if they are equal modulo variable renaming.

- *Depth-first scheduling* selects the next subqueries to evaluate in a depth-first manner. The two major scheduling strategies, local and batched [27], have the same asymptotic time and space complexities as depth-first scheduling. We describe complexity analysis using depth-first scheduling, because it is simpler.

- *Early completion* stops evaluation for a subquery whose arguments are all bound, once the subquery is evaluated to be true. *No early completion* means using all relevant rules to infer answers to a subquery even if it is a subquery whose arguments are all bound and has been evaluated to be true.

We also make the following two assumptions.

- All IDB predicates are tabled. This allows the best possible asymptotic time complexity; it may use unnecessarily large space, which is a problem that should be addressed, but is beyond the scope of this work.

- All predicates are perfectly indexed, so that it takes constant time to retrieve a fact of the predicate given fixed values for some of its arguments. In systems implementing variant tabling, perfect indexing can be manually specified, such as in XSB [61], or is automatically performed, such as in YAP [22].

For the rest of the chapter, *v-topdown evaluation* refers to evaluation using variant tabling, with depth-first scheduling, without early completion, and with the two assumptions above.

Figure 3.1 gives the algorithm for v-topdown evaluation. It recursively calls **invoke** as described below. Two global maps are used: $Table$ and $Suspension$. A map maps a key to a set of values, where each pair of key and set of values is called an entry. $Table$ maps each subquery encountered that is not a variant of a previously encountered subquery to a set of facts inferred for the subquery. The keys of $Suspension$ are pairs of atoms consisting of a key k of $Table$ and a hypothesis for which an answer for k can be used to resume computation. The values for each key are tuples of arguments to call **invoke** with when a fact for the hypothesis in the key is inferred.

In the algorithm, the following functions are used:

- **concl**(r) and **hypos**(r): the conclusion and the set of hypotheses of rule r, respectively.

- **unify**(a,b): a most general unifier of atoms a and b if it exists, $\emptyset$ otherwise.
- **subst**(a,θ): the atom a after substitution of variables using θ.
- **variant**(a,b): whether atoms a and b are equal modulo variable renaming.
- **keys**(m): keys of map m.

The algorithm starts from the given query, and calls procedure **invoke** for each rule whose conclusion matches the given query. The procedure takes four arguments:

1. a query q,
2. a rule r whose conclusion matches q,
3. an index i of the hypothesis of r to process,
4. a substitution θ from matching q against the conclusion of r, and matching facts against up to the ith hypothesis of r.

If the number of hypotheses of r is smaller than or equal to i, the procedure substitutes variables of the ith hypothesis of r using θ (called h_i), and performs the following on h_i:

- If h_i is not an IDB hypothesis, then find each fact that matches the hypothesis, and call **invoke** with i incremented for the next hypothesis, and with θ extended with the new match.
- If h_i is an IDB hypothesis and is a variant of an existing key of $Table$, then for each fact in the values for that key, match the fact against h_i, and call **invoke** with i incremented for the next hypothesis, and with θ extended with the new match. Also, record the current arguments of **invoke** for resuming computation after a new fact is added to the values of this table entry.
- If h_i is an IDB hypothesis and is not a variant of an existing table key, create a table entry whose key is h_i, and whose set of values is the empty set. For each rule r' whose conclusion matches h_i, call **invoke** with the arguments h_i, r', 1, and the substitution from the match. Also, record the current arguments of **invoke** for resuming computation after a new fact is added to the values for the new table entry.

If the number of hypotheses of r is smaller than i, then a fact is inferred and the substitution θ must contain all variables in q, because the rules are safe. The fact inferred is q after substitution using θ. The fact is added to the values for key q if it is not already in the values, and finally for each tuple of arguments that can resume computation with a new fact, **invoke** is called with the arguments after updating the substitution in the tuple to account for the inferred fact.

The time complexity is the number of calls to **invoke**, because all other operations are constant time in data size. We make the following observations for counting the number of calls to **invoke**: (1) The combination of the first two arguments of **invoke** are determined by the call to **invoke** whose index argument is 1, because other calls copy these two arguments from the enclosing call to **invoke**. (2) The calls to **invoke** whose index argument is 1 must be for queries that are not variants of the subqueries in $Table$, and match the conclusion of some rule. (3) For each pair of the first two arguments to **invoke**, rule r and query q that matches the conclusion of r, the combinations of the last two arguments, index i and substitution θ, are the combinations of facts that match the hypotheses of r.

The space complexity is the number of facts stored in the table entries. We do not consider stack space in this work.

For easier and more precise calculation, we first generate a query and rules annotated with the patterns of argument bindings based on the given query, but whose evaluation is otherwise the same as the given query and rules. Then, we calculate the complexity of evaluating the annotated query and rules. Annotations make complexity calculation easier by distributing the complexity to parts of the query and rules that contribute to it in simpler ways.

3.1.1 Binding annotation

To annotate a set of rules with respect to a query, we first determine the patterns of argument bindings during the evaluation of the query, called *variant demand patterns*, and then generate an annotated rule for each pattern determined.

Variant demand patterns. Given a set of rules and a query, each subquery $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ encountered during v-topdown evaluation yields a *variant demand pattern* $\langle \texttt{p}, \texttt{s} \rangle$, where `s` is a string, called the *pattern string*, of length k whose ith character is '`b`' if $\texttt{a}_i$ is bound, and '`f`' otherwise. For an

```
Suspension = new map
Table = new map
Table[q] = ∅
// Call invoke for each rule matching q
for r ∈ R | θ = unify(concl(r), q) ≠ ∅:
  invoke(q, r, 1, θ)
return Table[q]

procedure invoke(q, r, i, θ):
  // If there are still hypotheses of r to process
  if i ≤ |hypos(r)|:
    h_i = subst(the ith hypothesis of r, θ)
    if h_i is not an IDB hypothesis:
      // Call invoke for each matching fact
      for fact ∈ F | θ' = unify(h_i, fact) ≠ ∅:
        invoke(q, r, i+1, θ ∪ θ')
    // If h_i is a variant of an existing table key
    else if ∃k ∈ keys(Table) | variant(h_i, k):
      // Record current arguments
      // for resuming invoke later
      Suspension[⟨k, h_i⟩] ∪= {⟨q, r, θ, i⟩}
      // Call invoke for each fact in values for key k
      for fact ∈ Table[k]:
        θ' = unify(h_i, fact)
        invoke(q, r, i+1, θ ∪ θ')
    // If a variant does not exist in table keys
    else:
      Table[h_i] = ∅
      // Record current arguments
      // for resuming invoke later
      Suspension[⟨h_i, h_i⟩] ∪= {⟨q, r, θ, i⟩}
      // Call invoke for each r matching new query h_i
      for r' ∈ R | θ' = unify(concl(r'), h_i) ≠ ∅:
        invoke(h_i, r', 1, θ')
  // If no more hypothesis is left to process
  else:
    fact = subst(q, θ)
    // If the fact has not been inferred before
    if fact ∉ Table[q]:
      // Add the fact to the table
      Table[q] ∪= {fact}
      // Resume computations
      for ⟨k, h⟩ ∈ keys(Suspension) | k = q:
        for ⟨q', r', θ', i'⟩ ∈ Suspension[⟨q, h⟩]:
          θ'' = unify(h, fact)
          invoke(q', r', i'+1, θ' ∪ θ'')
endproc
```

Figure 3.1: V-topdown evaluation of query q, given a set of facts F and a set of rules R

atom $\mathtt{p(a_1,...,a_k)}$ and a pattern string $\mathtt{s}$ of length k, we say that $\mathtt{a}_i$ is *bound by* $\mathtt{s}$ if the ith character of $\mathtt{s}$ is '$\mathtt{b}$'.

Variant demand patterns are computed iteratively as follows until no new variant demand patterns can be added. The variant demand pattern of the given query $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ is $\langle \texttt{p}, \texttt{s} \rangle$, where the ith character of s is 'b' if $\texttt{a}_i$ is a constant, and 'f' otherwise. For each computed variant demand pattern $\langle \texttt{p}, \texttt{s} \rangle$, for each rule r that defines p, and for each IDB hypothesis h of r whose predicate is, say, q, add a variant demand pattern $\langle \texttt{q}, \texttt{t} \rangle$, where the ith character of t is 'b' if the ith argument of h is a constant, or appears in a hypothesis to the left of h in r, or is an argument of the conclusion of r bound by s; and 'f' otherwise.

Annotation. For each variant demand pattern $\langle \texttt{p}, \texttt{s} \rangle$ computed, and for each rule r that defines p, we generate an annotated rule that obeys the pattern string s, where the conclusion is annotated with s, and each hypothesis is annotated with the pattern string obtained as described above.

Formally, for each variant demand pattern $\langle \texttt{p}, \texttt{s} \rangle$, and each rule of the form

$\texttt{p}(...) :- \texttt{h}_1(...), ..., \texttt{h}_n(...).$

We generate the rule

$\texttt{p\_s}(...) :- \texttt{h}_1\texttt{\_s}_1(...), ..., \texttt{h}_n\texttt{\_s}_n(...).$

where for each $1 \leq k \leq n$, the ith character of $\texttt{s}_k$ is 'b' if the ith argument of $\texttt{h}_k$ is a constant, or appears in a hypothesis to the left of $\texttt{h}_k$, or is an argument of the conclusion bound by s, and 'f' otherwise.

For the given query p(...)?, the annotated query p_s(...)? is generated, where the ith character of s is 'b' if the ith argument of the given query is a constant; and 'f' otherwise.

Example. For rules (B) and (L), and target query path(c,y)?, the set of variant demand patterns is $\{\langle \texttt{path}, \texttt{'bf'} \rangle\}$, and annotation results in the annotated query path_bf(c,y)? and two annotated rules:

```
path_bf(x,y) :- edge_bf(x,y).                  (B')
path_bf(x,y) :- path_bf(x,z), edge_bf(z,y).    (L')
```

For rules (B) and (R), and the same target query, the set of variant demand patterns is the same, and annotation results in the same annotated query, rule (B'), and the following rule:

```
path_bf(x,y) :- edge_bf(x,z), path_bf(z,y).    (R')
```

Annotation is the same as predicate splitting [68], except we annotate all hypotheses, in contrast to only IDB, for ease of complexity analyses.

3.1.2 Time complexity analysis

For an annotated rule, the asymptotic time complexity it incurs is the product of: (1) *local complexity*—the number of different values that the free variables in the rule can take, and (2) *number of invocations*—the number of different values that the bound arguments of the conclusion can take. We give a method to calculate an upper bound for each factor. Summing the complexities incurred by all rules gives the overall complexity.

The local complexity of a rule is the product of complexity factors incurred by all hypotheses of the rule. Each hypothesis, say $\texttt{p\_s(a}_1\texttt{,...,a}_n\texttt{)}$, of r incurs the complexity factor $O(\texttt{\#p.f}_1\texttt{,...,f}_k\texttt{/b}_1\texttt{,...,b}_l)$, where $\texttt{f}_i$ is the index of the ith '`f`' in `s`, and $\texttt{b}_i$ is the index of the ith '`b`' in `s`.

For example, for rule `(L')`, the first hypothesis incurs the complexity factor $O(\texttt{\#path.2/1})$, and the second hypothesis incurs the complexity factor $O(\texttt{\#edge.2/1})$. Therefore, the local complexity is
$O(\texttt{\#path.2/1}\times\texttt{\#edge.2/1})$.

For computing the number of invocations of a rule r, three steps are performed. First, among all hypotheses of all rules and the given query, find those whose predicate is the same as the predicate of the conclusion of r. Second, for each one found, say called h, calculate the number of different values its bound arguments can take. If a bound argument is a constant, then it can take only that one value. If a bound argument is a variable, say `x`, then the minimum of the following is taken: (1) If `x` is the ith argument of a hypothesis to the left of h whose predicate is `p`, then `x` may take $O(\texttt{\#p.i})$ different values. (2) If `x` appears in the conclusion c, there are two cases: if c is a variant of h, and the bound arguments of c and h are the same, then `x` may take one value; otherwise it may take $O(\texttt{dom(p.i)})$ values, where `p` is the predicate of c, and `x` is the ith argument of c. The product of the numbers of different values that the bound arguments can take in h is the total number of invocations of r due to h. Third, the sum of the products due to all h's is the number of invocations to r.

For example, the predicate of the conclusion of rule `(R')` appears in the query, and in the second hypothesis of rule `(R')` itself. The first argument of the query is constant, so it takes only one value. The first argument of the second hypothesis is a variable `z`, which appears as the second argument of the first hypothesis, and thus takes $O(\texttt{\#edge.2})$ different values. Therefore, the number of invocations of rule `(R')` is $O(1+\texttt{\#edge.2})$, which is $O(\texttt{\#edge.2})$.

The calculated complexities for rules `(B')`, `(L')`, and `(R')` are respectively:

- $O($`#edge.2/1`$)$
- $O($`#path.2/1` $\times$ `#edge.2/1`$)$
- $O($`#edge.2/1` $\times$`#path.2/1` $\times$ `#edge.2`$)$

Therefore, the time complexity of the target query using left-recursion is $O($`#path.2/1` $\times$ `#edge.2/1`$)$, and using right recursion is $O($`#edge.2/1` $\times$ `#path.2/1` $\times$ `#edge.2`$)$.

3.1.3 Space complexity analysis

The asymptotic space complexity of v-topdown evaluation is bounded by the space for table entries. Each table entry is keyed on an annotated predicate and values for the bound arguments. For an annotated predicate, the space it takes is the product of: (1) *number of table entries created*—the number of values that the bound arguments can take in subqueries of the annotated predicate, and (2) *size of each table entry*—the number of different values that the free arguments can take in the facts inferred for the annotated predicate. We give a method to calculate an upper bound for each factor. Summing the space used for all predicates gives the total space.

The number of table entries created for an annotated predicate `p` is calculated as follows. First, among all hypotheses of all rules and the given query, find those whose predicate is `p`. Then, for each such hypothesis, perform the second and third step of the method for computing the number of invocations in the previous subsection.

For example, for the left-recursive version of transitive closure and target query `path(c,y)?`, the number of table entries created for the predicate `path_bf` is $O(1)$ due to the given query and rule `(L')`, since the first hypothesis of `(L')` is a variant of its conclusion. For the right-recursive version and the same query, the number of table entries created for `path_bf` is $O($`#edge.2`$)$ due to the query and rule `(R')`.

The size of each table entry for each annotated predicate `p` is calculated as follows. For each rule r that defines `p`, we calculate the number of values that the free variables of the conclusion can take. Each of these free variables can take $O($`#q.i`$)$ different values if it is the ith argument of a hypothesis of r whose predicate is `q`; if there are multiple such hypotheses, the minimum of these is taken. The product of the numbers of different values that the free variables of the conclusion can take in r is the size of each table entry for facts inferred by r. The sum over all rules gives the final size of each table entry.

For example, consider the left-recursive version of transitive closure, and the target query. For the size of each table entry of `path_bf`, rule `(B')` incurs $O($`#edge.2`$)$ due to the first hypothesis, and rule `(L')` incurs $O($`#edge.2`$)$ due to the second hypothesis. Therefore, the total size of each table entry is $O($`#edge.2`$)$. The number of table entries created for `path_bf` is $O(1)$ as shown above. Therefore, the total space complexity is $O(1\times$`#edge.2`$)$. Using this analysis, the space complexity for the right-recursive version for the target query is $O($`#edge.2`$\times($`#edge.2`$+$`#path.2`$))$.

Besides estimating memory usage, space complexity analysis can also help compare the actual running time for queries that have the same asymptotic time complexity. Creating a table entry is more expensive than adding a fact to a table entry in implementations such as XSB [61]. Therefore, the query that creates fewer table entries uses a constant factor less memory, and runs a constant factor faster.

For example, consider the left-recursive and right-recursive version of transitive closure. Given a query where both arguments are bound, the time complexities of both versions are the same. The left-recursive version contains rule `(L')`, for which the number of table entries is analyzed above. However, annotated rules for the right-recursive version contains the rule `(R'')`.

```
path_bb(x,y) :- edge_bf(x,z), path_bb(z,y). (R'')
```

The second hypothesis of `(R'')` creates $O($`#edge.2`$)$ table entries, in contrast to $O(1)$ table entries created by `(L')`. Therefore, we conclude that the right-recursive version should run a constant factor slower. The experiments in Section 3.4 confirm this.

3.2 Variant demand transformation for bottom-up evaluation

Bottom-up evaluation, as discussed in the previous chapter, infers all facts that can possibly be inferred without taking the given query into account, and thus may take asymptotically more time than necessary. To take the query into account, we perform *variant demand transformation*.

- Variant demand transformation transforms the given set of rules and query into a new set of rules and a fact, so that bottom-up evaluation using the new rules and fact, for any given set of facts, infers only useful facts for answering the query. It achieves this by mimicking top-down

evaluation of the given query q so that for predicates in the given rules, only facts that would be inferred during v-topdown evaluation of q are inferred in a bottom-up evaluation of the transformed rules.

- We also show that variant demand transformation can be obtained by simplifying the output of the well-known magic set transformation (MST). Annotations in MST are not necessary using bottom-up evaluation, because the indices corresponding to the annotations are generated automatically. Therefore, the output of our transformation is simpler.

For the rest of this section, *v-bottomup evaluation* refers to performing bottom-up evaluation on the rules generated by variant demand transformation.

3.2.1 Variant demand transformation

To perform variant demand transformation, we first compute variant demand patterns as shown in Section 3.1.1. Then, for each variant demand pattern $\langle \mathtt{p}, \mathtt{s} \rangle$, and for each rule

`p(...)` `:-` $\mathtt{h}_1$`,...,` $\mathtt{h}_n$`.`

the following rule is generated

`p(...)` `:- d_p_s(`$\mathtt{a}_1$`,...,`$\mathtt{a}_k$`),` $\mathtt{h}_1$`,...,` $\mathtt{h}_n$`.`

where $\mathtt{a}_1$,...,$\mathtt{a}_k$ are the arguments of the conclusion bound by `s`. The new hypothesis is added to ensure that only facts that would be inferred in v-topdown evaluation are inferred. Then, a fact and rules that define the facts of each predicate `d_p_s` are generated. For the given query, `p(`$\mathtt{a}_1$`,...,`$\mathtt{a}_k$`)?`, the following fact is generated

`d_p_s(`$\mathtt{a}_{b1}$`,...,`$\mathtt{a}_{bl}$`).`

where $\mathtt{a}_{b1}$,...,$\mathtt{a}_{bl}$ are the constant arguments of the query, and `s` is the pattern string of the query. For each rule r generated, `c :-` $\mathtt{h}_0$`,...,` $\mathtt{h}_n$`.`, and for each $\mathtt{h}_i$ whose predicate is an IDB predicate `p`, the following rule is generated

`d_p_s(`$\mathtt{a}_1$`,...,`$\mathtt{a}_k$`) :-` $\mathtt{h}_0$`,...,` $\mathtt{h}_{i-1}$`.`

where $\mathtt{a}_1$,..., $\mathtt{a}_k$ are the bound arguments of $\mathtt{h}_i$, and `s` is the pattern string of $\mathtt{h}_i$.

For example, for rules `(B)` and `(L)`, and target query `path(c,y)?`, the set of variant demand patterns is $\{\langle$`path`,`'bf'`$\rangle\}$. Variant demand transformation generates the following fact and rules.

```
d_path_bf(c).                                (F)
path(x,y) :- d_path_bf(x), edge(x,y).        (Bd)
path(x,y) :- d_path_bf(x), path(x,z),        (Ld)
             edge(z,y).
d_path_bf(x) :- d_path_bf(x).                (D)
```

Fact `(F)` corresponds to the given query. Rules `(Bd)` and `(Ld)` correspond to the variant demand pattern ⟨`path`,‘`bf`’⟩. Rule `(D)` is for the second hypothesis of rule `(Ld)`. Bottom-up evaluation using the generated rules has smaller time complexity, because in the given rule `(L)`, the variable `x` could take an arbitrary value, whereas in rule `(Ld)`, its value is restricted by the new hypothesis, `d_path_bf`, for which only one fact, `(F)`, exists, so `x` can only be `c`.

Note that variant demand transformation does not necessarily reduce the asymptotic time complexity. Consider rules `(B)` and `(L)`, and source query `path(x,c)?`, instead of target query. The set of variant demand patterns is {⟨`path`,‘`fb`’⟩, ⟨`path`,‘`ff`’⟩}. Demand transformation generates the following fact and rules.

```
d_path_fb(c).
path(x,y) :- d_path_fb(y), edge(x,y).
path(x,y) :- d_path_fb(y),
             path(x,z), edge(z,y).
path(x,y) :- d_path_ff(), edge(x,y).
path(x,y) :- d_path_ff(),
             path(x,z), edge(z,y).
d_path_ff() :- d_path_fb(y).
d_path_ff() :- d_path_ff().
```

The time complexity of bottom-up evaluation using the generated rules is not better than the original rules if the underlying graph is connected, since no variable is restricted analogous to `x` in the previous example. For variant demand transformation to improve the complexity for target query as it did for source query, the left-recursive rule needs to be transformed into a right-recursive rule using recursion conversion as we show in Chapter 5.

3.2.2 Comparing with magic set transformation

Magic set transformation (MST) has the same goal as variant demand transformation. MST has three similar steps: binding annotation, generating rules and adding hypotheses for reflecting the demand of computation, and

generating a fact for the demand by the query. A detailed description of the MST algorithm can be found in [68].

The disadvantage of MST, in contrast to variant demand transformation, is that it annotates the IDB predicates in the generated rules, and this may result in exponentially increased space complexity in program size, as shown below.

Consider the last example in the previous subsection. MST yields the following fact and rules.

```
d_path_fb(c).
path_fb(x,y) :- d_path_fb(y), edge(x,y).
path_fb(x,y) :- d_path_fb(y),
                path_fb(x,z), edge(z,y).
path_ff(x,y) :- d_path_ff(), edge(x,y).
path_ff(x,y) :- d_path_ff(),
                path_ff(x,z), edge(z,y).
d_path_ff() :- d_path_fb(y).
d_path_ff() :- d_path_ff().
```

The difference is the extra annotations in the annotated `path` predicates in the rules generated by MST. These rules may infer some same facts of `path` for two differently annotated predicates, `path_fb` and `path_ff`.

In general, annotating IDB hypotheses with their pattern strings is not necessary, because using bottom-up evaluation, indices for matching hypotheses are created automatically. Removing these annotations yields simpler rules, and reduces space taken by the same fact duplicated for multiple new predicates generated. The extra space from keeping the annotations is exponential in the number of arguments of differently annotated predicates.

Removing annotations of IDB predicates in the generated rules by MST yields the rules generated by variant demand transformation.

3.3 Relating top-down evaluation with variant tabling and bottom-up evaluation after variant demand transformation

3.3.1 Time complexity comparison

We establish the time complexity relationship between v-topdown and v-bottomup evaluations. First, we show the relationship in the general case,

then identify a subset of Datalog for which the two evaluations are equivalent, and finally show that adding early completion may improve v-topdown evaluation.

Theorem 3.3.1 states that v-bottomup evaluation is faster than or equal to v-topdown evaluation in time complexity.

Theorem 3.3.1 (V-bottomup is no slower than v-topdown)**.**
$T_{v-botup} \leq T_{v-topdn}$.

Proof. For a set of rules and query P, let P_a be the set of rules and query after annotating P.

$T_{v-topdn}$ is the sum of the complexities incurred by each rule in P_a. For each rule r in P_a of the form `p(...) :- body.`, there is a rule r' of the form `p(...) :- d(...), body.` in P', where `d(...)` is the new demand hypothesis. The complexity incurred by r for $T_{v-topdn}$ is $i \times l$, where i is *the number of invocations* to r, and l is *the local complexity*, and l is the product of the sizes of hypotheses. Since facts of `d` are obtained from all of the call sites to `p` with the same binding pattern as top-down evaluation, `#d`$= i$. For $T_{v-botup}$, the complexity incurred by a rule is the number of times the rule fires. Therefore, the complexity incurred by r' has an upper bound `#d`$\times l = i \times l$.

The only rules in P' that do not correspond to a rule in P_a are the rules that infer facts of the predicates added for demand. The additional complexity incurred for $T_{v-botup}$ by each such rule is already dominated by a component of the complexity in $T_{v-topdn}$, because this complexity equals the number of invocations for the rule that the demand hypothesis would be added to, and the number of invocations is used as a factor in a summand of $T_{v-topdn}$.

Hence, $T_{v-botup} \leq T_{v-topdn}$. □

We show that for Datalog rules in minimal form, i.e., rules with no more than two hypotheses, and no singleton variables, the time complexities of v-topdown evaluation and v-bottomup evaluation are equal.

Lemma 3.3.2. *In bottom-up evaluation, if all variables in the hypotheses of a rule r are also in the conclusion of r, then the number of facts inferred using r equals the number of firings of r.*

Proof. The number of facts inferred using r cannot be larger than the number of firings of r, since a fact is inferred only in a firing of r.

Let f_1 and f_2 be two different firings of a rule r. There is at least one variable whose value is different between f_1 and f_2. Since all variables in

the hypotheses also appear in the conclusion, the facts inferred from f_1 and f_2 must be different.

Therefore, the number of facts inferred using r equals the number of firings of r. □

Theorem 3.3.3 (Equivalence for a subset of Datalog)**.** *For rules in minimal form,* $T_{v-botup} = T_{v-topdn}$.

Proof. Let P be a set of rules in minimal form and a query. Let P_a be the set of rules after annotating the rules in P. Each rule r in P_a is of one of two forms:

(i) r has one hypothesis, so has the form `c :- h.` In P', there is a rule r' corresponding to r, and is of the form `c :- d, h.`, where `d` is the new demand hypothesis. The complexity incurred by r' to $T_{v-botup}$ and by r to $T_{v-topdn}$ are both dominated by the size of the predicate of `h`, since `h` contains all variables in `d`.

(ii) r has two hypotheses, so has the form `c :-` $\mathtt{h}_1$`,` $\mathtt{h}_2$`.` In P', there is a rule r' corresponding to r, and is of the form `c :- d,` $\mathtt{h}_1$`,` $\mathtt{h}_2$`.`, where `d` is the demand hypothesis added. As before, the complexity incurred by r to $T_{v-topdn}$, denoted $T_{v-topdn}(r)$, equals the product of the sizes of the predicates `d`, $\mathtt{h}_1$, and $\mathtt{h}_2$. However, bottom-up computation can decompose the rules to possibly improve performance. In this case, it would obtain the following two rules: `new :- d,` $\mathtt{h}_1$`.` and `c :- new,` $\mathtt{h}_2$. The complexity of the first rule is less than $T_{v-topdn}(r)$. Since there are no singleton variables, the variables of `d` and $\mathtt{h}_1$ must appear in `new`. Then, by Lemma 3.3.2, the size of the predicate of `new` equals the running time of the rule that generates it, and hence the complexity incurred by the second rule obtained from r' equals $T_{v-topdn}(r)$.

Therefore, for each complexity summand incurred by rules in P_a for $T_{v-topdn}$, there is a rule in P' that incurs the same complexity summand for $T_{v-botup}$. Combining this with Theorem 3.3.1, which states that $T_{v-botup} \leq T_{v-topdn}$, we obtain $T_{v-botup} = T_{v-topdn}$. □

Early completion is an optimization for v-topdown evaluation. It stops backtracking for queries with all arguments bound immediately after they are proven to be true. Theorem 3.3.4 states that adding early completion to v-topdown evaluation may make it asymptotically faster than v-bottomup evaluation.

Theorem 3.3.4 (V-topdown with early completion may be faster)**.** *If early completion is used by v-topdown evaluation,* $T_{v-topdn}$ *may be smaller than* $T_{v-botup}$.

Proof. With early completion, v-topdown evaluation stops backtracking when it proves that a subquery with all arguments bound is true, whereas bottom-up evaluation always exhausts all possible ways of proving facts. Therefore, with early completion, the time complexity of v-topdown evaluation can be smaller than v-bottomup evaluation. □

3.3.2 Space complexity comparison

We establish the space complexity relationship between v-topdown and v-bottomup evaluation. We first show the relationship in the general case. We then show that if v-bottomup evaluation has better time complexity, then its space complexity must be worse.

Theorem 3.3.6 states that v-topdown evaluation does not use asymptotically more space than v-bottomup evaluation. We prove it by showing the components of space used for v-bottomup evaluation and their correspondence with the space usage in v-topdown evaluation.

Lemma 3.3.5. *Let P be a set of Datalog rules and a query. For each predicate p in P, let $BU(p)$ be the set of facts of p inferred using v-bottomup evaluation of P, and let $TD(p)$ be the set of facts of p inferred during v-topdown evaluation of P. Then, $BU(p) = TD(p)$.*

Proof. We showed in Theorem 3.3.1 that the bound argument values of subqueries for which each rule will be invoked in v-topdown computation is a fact for the demand hypothesis added in bottom-up computation. Therefore, for each predicate p, the same facts will be inferred by each rule that defines p using either method. Hence, $BU(p) = TD(p)$. □

Theorem 3.3.6 (V-topdown uses no more space than v-bottomup)**.**
$S_{v-topdn} \leq S_{v-botup}$.

Proof. $S_{v-botup}$ consists of the sums of each of the following items: (i) the number of facts of each predicate defined by rules, (ii) the number of facts of each demand predicate, (iii) the number of facts of each predicate defined for decomposing rules into rules with at most two hypotheses, and (iv) the size of the auxiliary maps maintained for fact retrieval. $S_{v-topdn}$ consists only of the sum of the sizes of table entries for each predicate defined by rules.

For a predicate p, by Lemma 3.3.5, the set of facts of p inferred by either evaluation method is the same. Each fact of p is stored only once in the bottom-up method, but they can be stored in 2^k auxiliary maps, where k is the number of arguments of p. For v-topdown evaluation, each fact of p may

be stored in at most 2^k tables, where the number of table entries correspond exactly to the auxiliary maps. Therefore, $S_{v-topdn} \leq S_{v-botup}$. □

Theorem 3.3.7 states that improvement in time complexity for v-bottomup evaluation is only possible by using more space. We prove it by using the fact that such improvement is only possible by using more space in decomposed rules.

Theorem 3.3.7 (V-bottomup is faster only when it uses more space)**.** *For a set of rules and a query, if* $T_{v-botup} < T_{v-topdn}$*, then* $S_{v-topdn} < S_{v-botup}$*.*

Proof. Let P be a set of rules and query for which $T_{v-botup} < T_{v-topdn}$. Then, in the bottom-up evaluation of P, there is a rule which is decomposed into multiple rules for bottom-up evaluation. This implies that the third component of $S_{v-botup}$ shown in Theorem 3.3.6 is nonzero. Since $S_{v-topdn}$ only consists of the first and fourth item of $S_{v-botup}$, $S_{v-topdn} < S_{v-botup}$. □

3.4 Experiments

We support our complexity analyses and comparisons by experiments. For top-down evaluation, we use XSB [75]. For bottom-up evaluation, we use the implementation method of [51] to generate Python code from the rules.

We examined all benchmarks in OpenRuleBench [45]. All benchmark rules can be classified as pure joins (no recursion), transitive closure, and same generation (whether two nodes are in the same generation in trees). We show experimental results for three benchmarks, one for each class.

We instantiate the complexity parameters in predicted complexities with their values computed from the data. We use *space units* to mean number of unique table inserts for v-topdown, and the number of facts inferred plus the number of elements in auxiliary maps for v-bottomup evaluation. We use *returns* to mean the number of total facts returned from rules for v-topdown evaluation, and *firings* to mean the number of firings for v-bottomup evaluation.

In all three benchmarks, the predicates have two arguments. For experiments, we fix `#p` and `#p.1/2` for each input predicate `p` to generate a set of data such that the size of each predicate is maximal, i.e., the worst-case behavior is exhibited. Then, we increase `#p` and `#p.1/2` to generate the next set of data, and repeat.

For pure joins, we show results for the benchmark *Join1*, which contains 4 rules that join 5 predicates, with a query with all arguments free. Figure

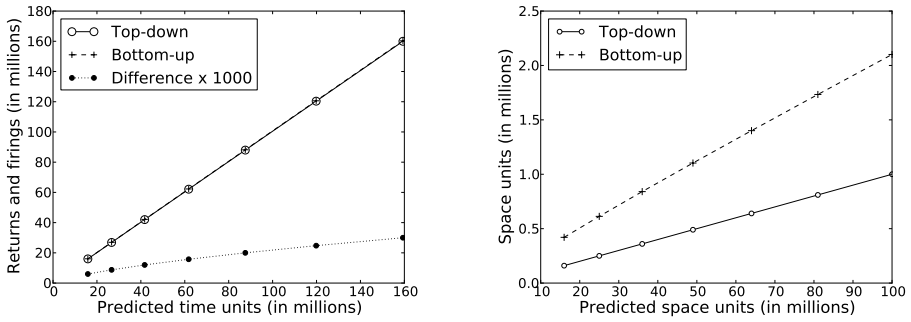

Figure 3.2: Returns for v-topdown evaluation, firings for bottom-up evaluation, and space units for both, for benchmark *Join1*. The difference between returns and firings has been multiplied by 1000 for illustration.

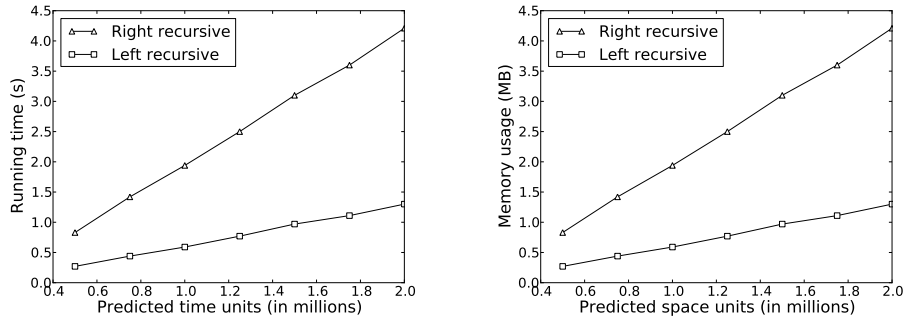

Figure 3.3: Running time and memory usage of transitive closure in XSB for a query with both arguments bound.

3.2 shows that returns for v-topdown evaluation and firings for v-bottomup evaluation are linear in predicated time units. It also shows that the space units for both is linear in predicted space units. These confirm our analyses. The time difference between v-topdown and v-bottomup evaluations arise from the rules that infer demand. The space difference between them arise from demand predicates and auxiliary maps.

For transitive closure, we analyze the time complexity, and by using the space complexity analysis for v-topdown evaluation, give a comparison of actual running times when the asymptotic time complexity is the same. We showed in Section 3.3 that the left- and right-recursive versions of transitive closure have the same asymptotic time and space complexities for a query

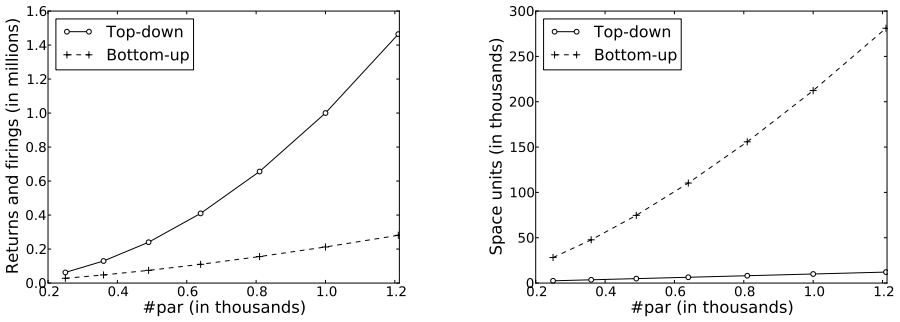

Figure 3.4: Returns for v-topdown evaluation, firings for bottom-up evaluation, and space units for both, for the same generation benchmark.

with both arguments bound, but the right-recursive version creates asymptotically more table entries. Therefore, the right-recursive version will run slower by a constant factor, and use a constant factor more space. Figure 3.3 confirms that their complexities are the same, since the actual time and space are linear in the predictions. It also confirms that the right-recursive version uses a constant factor more time and space.

The same generation benchmark contains a rule with three hypotheses: `sg(x,y) :- par(x,`z_1`), sg(`z_1`,`z_2`), par(y,`z_2`).` We use the query `sg(c,y)?`, and show the time and space tradeoff. Bottom-up evaluation eliminates variable z_1 after decomposing the rules into rules with two hypotheses, and has better time complexity than v-topdown evaluation. Therefore, v-topdown evaluation has better space complexity. Figure 3.4 confirms our analysis: the returns of v-topdown evaluation increases faster asymptotically, and the space units of v-topdown evaluation increases slower asymptotically.

3.5 Related work

Top-down evaluation with variant tabling was introduced in [66], and an implementation of it is described in [18]. Optimal bottom-up evaluation, on which our left-optimal bottom-up evaluation is based, is described in [51].

For top-down evaluation of Datalog with variant tabling, the only known bound on the time complexity is $O(k^v)$, where k is the number of constants in the input data, and v is the maximum number of variables in a rule [75], and there is no complexity analysis studied for space. Our method calculates

worst-case time complexity much more precisely, and is the first to calculate worst-case space complexity and calculates it precisely.

For bottom-up evaluation, time and space complexities have been analyzed before, using prefix-firing by Ganzinger et al. [28] and optimal bottom-up evaluation by Liu et al. [51]. Bottom-up evaluation was used to mimic top-down evaluation after program transformations, mostly notably magic set transformation [8]. Our variant demand transformation is simpler and produces simpler rules that have the same time and space in terms of data complexity and exponentially smaller space in terms of program complexity.

The relationship between top-down and bottom-up evaluation has been studied [57]. Ullman [68] shows that bottom-up evaluation after magic set transformation has better than or equal time complexity with a breadth-first top-down strategy called QRGT without tabling. Ramakrishnan et al. [56] describe magic set transformation with tail recursion optimization that is better than or equal to than top-down evaluation with tail recursion optimization. Bry [12] shows that top-down evaluation with variant tabling and bottom-up evaluation after magic set transformation can be implemented in a unified framework, and that they infer the same facts for the given predicates, but does not study time and space complexities. Our work is the first to establish precise relationships between top-down evaluation with variant tabling and bottom-up evaluation after a demand-driven transformation in terms of precise time and space complexities.

The complexity results can be used for optimizations by comparing the complexity formulas of different rules with the same semantics. However, comparison of complexity formulas may be difficult in general, in which case estimations of size parameters [47] can be used to help.

Chapter 4

Subsumptive tabling beats variant tabling and magic sets

In the last chapter, we have seen that for a subquery encountered during top-down with variant tabling, only answers from identical subqueries are reused. However, there may exist another subquery already encountered which is guaranteed to contain all answers to the current subquery, i.e., *subsumes* the current subquery. Using this, top-down evaluation can be coupled with a tabling strategy that performs more reuse of previously inferred answers, called *subsumptive tabling* [58].

Despite extensive research on optimization of Datalog rules [20, 60, 54, 43], given a set of rules, a query, and a fixed order of hypotheses for each rule, no transformations that yield better time complexity than MST are known. In the last chapter, we introduced *demand transformation* with better space complexity in program size, but the same time complexity. There exists no transformation such that the bottom-up evaluation of transformed rules achieves the performance of subsumptive tabling.

This chapter describes precise time and space complexity analysis for efficiently answering Datalog queries that uses subsumptive tabling, and precise relationships between top-down evaluations with variant and subsumptive tabling, and their relationship to bottom-up evaluation after MST. We give complexity analyses for top-down evaluation with subsumptive tabling by determining the binding patterns of arguments for queries, and the subqueries that are guaranteed to reuse answers from subsuming subqueries, and then extending the analysis in the previous chapter for subsumptive

tabling. We show that top-down evaluation using subsumptive tabling is equal to or better than using variant tabling in both time and space complexities. We also characterize a class of Datalog rules, for which subsumptive tabling is guaranteed to be better than variant tabling in both time and space complexities. Using this result, and the relationships established in the last chapter, we show that subsumptive tabling beats MST in time and space complexities as well. We also show that subsumptive tabling is guaranteed to be better than MST for the previously identified class of rules in time and space complexities.

Additionally, we describe a transformation, called *subsumptive demand transformation* (SDT), such that the bottom-up evaluation of the rules produced by SDT achieves the performance of subsumptive tabling. We modify bottom-up evaluation slightly, and couple it with SDT to obtain *subsumptive bottom-up evaluation*. We show that for a set of rules where each rule has no more than two hypotheses and no wildcards, the time and space complexities of subsumptive bottom-up evaluation and subsumptive tabling are equal, and for other rules the time complexity of subsumptive bottom-up evaluation may be better than subsumptive tabling. By extension, we show that using SDT is equal to or better than using MST. We show that for rules for which subsumptive tabling outperforms variant tabling, SDT outperforms MST.

Building on our analyses, we devise a transformation for making sure that a query that subsumes another in subsumptive tabling is queried first when it is better to do so in time complexity. Using this method, we show how to systematically derive Heintze and Tardieu's demand-driven pointer analysis [33] from the definition of Andersen's pointer analysis.

We show experimental results on an illustrative set of Datalog rules, rules for Andersen's pointer analysis [4], and ontology queries for semantic web from OpenRuleBench [45]; and confirm when subsumptive tabling and SDT are necessary for efficient evaluation of queries, confirm our complexity analyses when the best evaluation methods are chosen.

Notation. We refer to the different evaluation methods described in this and the previous chapter as follows:

- *V-topdown*: Top-down evaluation with variant tabling
- *S-topdown*: Top-down evaluation with subsumptive tabling
- *V-bottomup*: Bottom-up evaluation after variant demand transformation

- *S-bottomup*: Bottom-up evaluation after subsumptive demand transformation

The asymptotic time complexities of the above methods are denoted $T_{v-topdn}$, $T_{s-topdn}$, $T_{v-botup}$, $T_{s-botup}$. Similarly, asymptotic space complexities are denoted with S and the corresponding subscript. For space complexities, we do not consider the stack space used by the methods. As in the last chapter, for asymptotic time complexity analysis, we assume perfect hashing, i.e., finding the value for a key in a hash map takes $O(1)$ time; and for space complexities, we do not consider the stack space used by the methods, therefore we only consider the space taken by the subqueries generated and facts inferred.

Running example. For the running example, we define a predicate of being *related*. Two people x and y are said to be related if they are in an immediate family, or if there are two other people u and v in an immediate family, where the x is related to u, and y is related to v. This relation can be defined in Datalog using the following rules:

```
rel(x,y) :- imm(x,y).                        (1)
rel(x,y) :- imm(u,v), rel(u,x), rel(v,y). (2)
```

4.1 Complexity analysis for subsumptive tabling

Subsumptive tabling. To answer a query, top-down evaluation starts with the query, generates subqueries from hypotheses of rules whose conclusions match the query, considering rules in the order given, and considering hypotheses from left to right, and does so repeatedly until the subqueries match given facts. This may lead to repeated subqueries or infinite recursion when recursive rules exist. To address this problem, *tabling* memoizes answers to subqueries, and reuses them when possible.

Subsumptive tabling [58] reuses more answers by considering previous subqueries that *subsume* a new query; in contrast to *variant tabling*, which only considers previous queries that are equivalent to the new query. A subquery q_1 *subsumes* another subquery q_2 if there is a substitution θ of variables such that $\theta(q_1) = q_2$.

For example, given the rules in the running example and a query `rel(x,y)?`, a subquery `rel(c,x)?` for some constant `c` will be generated from the second hypothesis of the second rule. In variant tabling, this subquery will be used to generate more subqueries. In subsumptive tabling, since the given

query subsumes this subquery, the answers to the subquery will be looked up in the table entry for the given query.

For analyzing the time and space complexities of subsumptive tabling for top-down evaluation, we make the following assumptions:

- *Depth-first scheduling* is used. This selects the next subqueries to evaluate in a depth-first manner.
- *No early completion* is used. This uses all relevant rules to infer answers to a subquery, even when it is a subquery whose arguments are all bound and has been evaluated to be true.
- All IDB predicates are tabled. This allows for the best asymptotic time complexity.
- All predicates are perfectly indexed. So, it takes constant time to retrieve a fact of a predicate given fixed values for some of its arguments.

For v-topdown, the selection of the scheduling strategy does not change complexities, since each distinct subquery is guaranteed to be processed, regardless of when. But, for s-topdown, the order of evaluation of subqueries may change whether a subquery is processed or not, since a subsuming one may have been encountered before in an order, and may not have been encountered in another. We show complexity analyses for *depth-first scheduling*, and then describe the changes necessary for other dominant scheduling strategies. The algorithms for top-down with subsumptive tabling using *local scheduling* and *batched scheduling* are given in Figure 4.1 and 4.2.

The time complexity of s-tabling is the sum of the number of facts that match the hypotheses in the body of each rule for each subquery that is not looked up in the table. This is impossible to determine statically, since it is not possible to determine if a subquery will be looked up or evaluated. The space complexity is the number of facts stored in the table entries, and similarly is impossible to determine statically.

4.1.1 Subsumptive binding annotation and complexity analysis

In this section, we show the calculation of time and space complexities for s-topdown, that it beats v-topdown, and that it beats MST for a form of rules that is always possible to obtain, and identified a class of rules for which s-topdown is guaranteed to outperform the other methods.

The time complexity of s-topdown is the sum of the number of facts that match the hypotheses in the body of each rule for each subquery that is not looked up in the table. The space complexity is the number of facts stored in the table entries. These are impossible to determine statically, since it is not possible to determine if a subquery will be looked up or evaluated.

We give a method for obtaining an upper bound on the complexities that is as close as possible to the actual complexities. For easier and more precise calculation of the complexities, we first generate a query and rules annotated with the patterns of argument bindings based on the given query, but whose evaluation using s-topdown is otherwise the same as the given query and rules. Then, we calculate the complexity of evaluating the annotated query and rules.

To annotate a set of rules with respect to a query, we first determine the patterns of argument bindings during the evaluation of the query, called *subsumptive demand patterns*, and then generate an annotated rule for each pattern determined. This method is a generalization of the binding annotation method shown in the last chapter that is used for v-topdown.

Subsumptive demand patterns. For each subquery, we determine if it is *guaranteed to be evaluated* during s-topdown, where a subquery is guaranteed to be evaluated if it will be evaluated during s-topdown regardless of the given or inferred facts. Given a set of rules and a query, each subquery $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ encountered during s-topdown yields an *s-demand pattern* $\langle \texttt{p}, n, r, g, s\rangle$, where the subquery is the nth hypothesis of rule r, s is a string, called the *pattern string*, of length k whose ith character is '`b`' if $\texttt{a}_i$ is bound, and '`f`' otherwise, and g is a boolean value that determines if this subquery is guaranteed to be evaluated. For an atom $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ and a pattern string `s` of length k, we say that $\texttt{a}_i$ is *bound by* `s` if the ith character of `s` is '`b`'.

An s-demand pattern d_1 is said to subsume another s-demand pattern d_2 if d_2 is guaranteed to be evaluated and any subquery with the pattern d_1 subsumes subqueries with the pattern d_2. Formally, an s-demand pattern $\langle \texttt{p}_1, n_1, r_1, g_1, s_1\rangle$ subsumes an s-demand pattern $\langle \texttt{p}_2, n_2, r_2, g_2, s_2\rangle$ if g_1 is `true`, and (i) s_1 contains no '`b`'s, or (ii) for each j such that the jth character of s_1 is '`b`', the jth character of s_2 is '`b`', and the hypotheses of r_1 to the left of its n_1th hypothesis is a subset of the hypotheses of r to the left of its n_2th hypothesis.

S-demand patterns are computed iteratively as follows until no new s-demand patterns can be added. The s-demand pattern of the given query

$\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ is $\langle \texttt{p}, 1, \emptyset, \texttt{true}, s\rangle$, where the ith character of s is '`b`' if $\texttt{a}_i$ is a constant, and '`f`' otherwise. For each computed s-demand pattern $\langle \texttt{p}, n, r, g, s\rangle$, for each rule r_2 that defines `p`, and for each IDB hypothesis h_j of r_2 whose predicate is, say, `q`, add an s-demand pattern $\langle \texttt{q}, n_2, r_2, g_2, s_2\rangle$, if this s-demand pattern is not subsumed by an s-demand pattern already computed, where n_2 is j, g_2 is `true` if g is `true` and n_2 is 1, the ith character of s_2 is '`b`' if the ith argument of h is a constant, or appears in a hypothesis to the left of h in r, or is an argument of the conclusion of r bound by s; and '`f`' otherwise.

After s-demand patterns are computed as above, we take the projection of the first and last element of each tuple to obtain a set of predicate-annotation pairs.

Annotation. For each predicate-annotation pair $\langle \texttt{p}, \texttt{s}\rangle$ computed, and for each rule r that defines `p`, we generate an annotated rule that obeys the pattern string `s`, where the conclusion is annotated with `s`, and each hypothesis is annotated with the pattern string obtained as described above.

Formally, for each demand pattern $\langle \texttt{p}, \texttt{s}\rangle$, and each rule of the form

$\texttt{p}(...) :- \texttt{h}_1(...), ..., \texttt{h}_n(...).$

We generate the rule

$\texttt{p\_s}(...) :- \texttt{h}_1\texttt{\_s}_1(...), ..., \texttt{h}_n\texttt{\_s}_n(...).$

where for each $1 \leq k \leq n$, the ith character of $\texttt{s}_k$ is '`b`' if the ith argument of $\texttt{h}_k$ is a constant, or appears in a hypothesis to the left of $\texttt{h}_k$, or is an argument of the conclusion bound by `s`, and '`f`' otherwise.

For the given query `p(...)?`, the annotated query `p_s(...)?` is generated, where the ith character of `s` is '`b`' if the ith argument of the given query is a constant; and '`f`' otherwise.

Example. For the rules in the running example and the query `rel(x,y)?`, we show the difference between the annotated rules for v-topdown and s-topdown. For s-topdown, the computed set of s-demand patterns is $\{\langle$`rel`,'`ff`'$\rangle\}$, since the given query with both arguments free subsumes all other subsequent subqueries for `rel`, and hence the following query `rel_ff(x,y)?` and two annotated rules are generated.

```
rel_ff(x,y) :- imm(x,y).                (1a)
rel_ff(x,y) :- imm(u,v), rel_bf(u,x),   (2a)
               rel_bf(v,y).
```

For v-topdown, since subsumption is not used, the set of demand patterns would be $\{\langle$`rel`,'`ff`'$\rangle, \langle$`rel`,'`bf`'$\rangle, \langle$`rel`,'`bb`'$\rangle\}$, and annotation results in the same annotated query, the two rules above and the following four annotated rules:

```
rel_bf(x,y) :- imm(x,y).                (1b)
rel_bf(x,y) :- imm(u,v), rel_bb(u,x),   (2b)
               rel_bf(v,y).
rel_bb(x,y) :- imm(x,y).                (1c)
rel_bb(x,y) :- imm(u,v), rel_bb(u,x),   (2c)
               rel_bb(v,y).
```

To the best of our knowledge, subsumptive binding annotation is the first annotation method that considers subsuming queries, and is distinct from previous methods used for v-topdown in the last chapter and predicate splitting [68].

Other scheduling strategies. For local scheduling, batched-scheduling or other scheduling strategies, the method for obtaining the demand patterns need to be modified. The idea is that for each scheduling strategy, one needs to determine a heuristic that identifies subqueries guaranteed to be evaluated. For example, for batched scheduling, which retrieves one answer from a subquery and then continues to other hypothesis, only the given query and the first hypothesis of the first rule that defines the given query is guaranteed to be evaluated. For local scheduling, which retrieves all answers from a subquery before continuing to other hypotheses, a subquery is guaranteed to be evaluated if it is the first hypothesis of a rule whose conclusion is guaranteed to be evaluated.

Using these heuristics, the demand pattern identification method can be modified to obtain the relevant demand patterns, and the same annotation method can be used afterwards.

Time and space complexity analyses. Once we perform subsumptive binding annotation, we use the method in the last chapter on the annotated rules obtained for computing time and space complexity.

The time complexity is the sum of asymptotic complexities incurred by all annotated rules. For an annotated rule, the asymptotic time complexity it incurs is the product of: (1) *local complexity*—the number of different values that the free variables in the rule can take, and (2) *number of invocations*—the number of different values that the bound arguments of the conclusion can take.

The local complexity of a rule is the product of complexity factors incurred by all hypotheses of the rule. Each hypothesis, say $\texttt{p\_s(a}_1\texttt{,...,a}_n\texttt{)}$, of r incurs the complexity factor $O(\texttt{\#p.f}_1\texttt{,...,f}_k\texttt{/b}_1\texttt{,...,b}_l)$, where $\texttt{f}_i$ is the index of the ith '`f`' in `s`, and $\texttt{b}_i$ is the index of the ith '`b`' in `s`.

The number of invocations of a rule is the sum of all values possibly taken by its bound arguments in all possible subqueries. This is calculated by looking at the hypotheses with the same annotation as the conclusion of the rule. For each hypothesis in each rule whose annotation is the same as the conclusion of the analyzed rule, we determine how many values the bound arguments take by looking at the origin of the binding of those bound arguments.

For the running example, the local time complexity of `(2a)` is $O(\texttt{\#imm}\times\texttt{\#rel.2/1}\times\texttt{\#rel.2/1})$, and the number of invocations is $O(1)$ since the annotation of the conclusion is '`ff`', and hence can only take one value. Therefore, the time complexity incurred by `(2a)` is $O(\texttt{\#imm}\times\texttt{\#rel.2/1}\times\texttt{\#rel.2/1})$.

The space complexity of s-topdown consists of the facts stored. This is the sum of the product of the number of distinct table entries for an annotation and the number of facts for each table entry over all s-demand patterns.

The number of distinct table entries is the number of values that the bound arguments in the conclusion take for an annotated predicate. It can be calculated as shown for the number of invocations in time complexity.

The number of facts for each table entry is the number of values that the free arguments in the conclusion take for rules defining that annotated predicate. It is calculated by analyzing which hypotheses bind the free arguments in the conclusion.

For the running example, the only s-demand pattern is $\langle\texttt{rel}\text{,`}\texttt{ff}\text{'}\rangle$, therefore there is $O(1)$ table entries, and the number of facts for that table entry is $O(\texttt{\#rel})$. Hence, the space complexity is $O(\texttt{\#rel})$.

4.1.2 Subsumptive beats variant and magic sets

We show that s-topdown always beats v-topdown, and therefore beats the magic set transformation (MST) for a form of Datalog rules, which all Datalog rules can be reduced to.

The following lemma shows that the check for subsuming used in s-topdown is more expensive than the check for being variant in v-topdown.

Lemma 4.1.1. *Let q be an IDB subquery of k arguments. The first time q is encountered in top-down evaluation, the time complexity of table lookup for q is $O(k)$ in variant tabling, and $O(2^k)$ in subsumptive tabling.*

Proof. In variant tabling, the evaluation looks up whether there is a previous table entry with the same label up to variable renaming, and create it otherwise. This lookup can be trivially done in $O(k)$ time. Table entry creation can also be done in $O(k)$ time.

In subsumptive tabling, the evaluation needs to look up for every possible subsuming query, whether there is a table with that label. If the query has b bound arguments, there are 2^b queries that subsume q, and b is $O(k)$, therefore table lookup takes $O(2^k)$ time. Table entry creation can be done in $O(k)$ time. Therefore, the total time complexity is $O(2^k)$. □

However, the difference is asymptotic only in the number of arguments. From now on, we assume that the number of arguments of any predicate is bounded by a constant.

We show that the time complexity of s-topdown is no worse than the time complexity of v-topdown. For the theorems below, the time and space complexities are compared for a set of rules and a given query.

Theorem 4.1.2 (Subsumptive beats variant in time)**.**
$T_{s-topdn} \leq T_{v-topdn}$.

Proof. A subquery generated during s-topdown is guaranteed to be generated during v-topdown, since they follow the same algorithm, except that s-topdown may avoid generating some subqueries. Therefore, the facts inferred during s-topdown is a subset, not necessarily proper, of the facts inferred during v-topdown. Table lookup is more costly in s-topdown as shown in Lemma 1. However, since the number of arguments of a predicate is constant, we obtain $T_{s-topdn} \leq T_{v-topdn}$. □

Using a similar argument, we show that s-topdown uses no more space than v-topdown asymptotically.

Theorem 4.1.3 (Subsumptive beats variant in space)**.**
$S_{s-topdn} \leq S_{v-topdn}$.

Proof. As described in the last proof, the subqueries generated during s-topdown is a subset of the subqueries generated during v-topdown. Therefore, the number of tables created during s-topdown is no more than during v-topdown, and the number of answers stored in the tables that exist in both are equivalent since they both need to be correct. Therefore, $S_{s-topdn} \leq S_{v-topdn}$. □

We have established that s-topdown is at worst equal to v-topdown. The following theorem shows that it can in fact be be better both in time and space complexity.

Theorem 4.1.4 (Subsumptive properly beats variant)**.** *There exists a set of rules and a query for which* $T_{s-topdn} < T_{v-topdn}$ *and* $S_{s-topdn} < S_{v-topdn}$.

Proof. We prove this theorem using the running example. Recall that subsumptive binding annotation of the rules for the query with both arguments free results in rules `(1a)` and `(2a)`, whereas annotation for v-topdown results in the same two rules plus four extra rules.

We have shown that the time complexity incurred by `(2a)` is $O(\texttt{\#imm} \times (\texttt{\#rel.2/1})^2)$. However, for v-topdown, consider the following annotated rules.

```
rel_bf(x,y) :- imm(u,v), rel_bb(u,x), rel_bf(v,y).
rel_bb(x,y) :- imm(u,v), rel_bb(u,x), rel_bb(v,y).
```

The local complexity for `(2c)` is $O(\texttt{\#imm})$, however, there are at least $O((\texttt{\#imm.2})^2)$ invocations to `(2c)` due to the last hypothesis of `(2c)`. Therefore, the time complexity of v-topdown is at least $O(\texttt{\#imm} \times (\texttt{\#imm.2})^2)$.

Now, notice that values for the second argument of `rel` always come from the second argument of `imm`, therefore $O(\texttt{\#imm.2})$ may asymptotically be larger than $O(\texttt{\#rel.2/1})$, but not vice versa. Therefore, $O(\texttt{\#imm} \times (\texttt{\#rel.2/1})^2)$ is asymptotically smaller than $O(\texttt{\#imm} \times (\texttt{\#imm.2})^2)$. Therefore, we have proven our theorem for time complexity.

For space complexity, we have shown that the space complexity of these rules for s-topdown is $O(\texttt{\#rel})$, which is optimal since it is only as large as the output. For v-topdown, there are $O((\texttt{\#imm.2})^2)$ table entries created for `rel_bb`, which is asymptotically larger than $O(\texttt{\#rel})$ in the worst-case. Therefore, we have proven our theorem for space complexity as well. □

Using Theorems 4.1.2, 4.1.3, and 4.1.4 and theorems about v-topdown in the last chapter, we establish that s-topdown beats MST for Datalog rules in *minimal form*. First, we recall the following theorem from the last chapter.

Theorem 4.1.5 (Variant vs. MST for minimal form)**.** *For rules in minimal form,* $T_{v-botup} = T_{v-topdn}$ *and* $S_{v-topdn} \leq S_{v-botup}$*.*

Combining this theorem with the results above, we obtain that for rules in minimal form, s-topdown beats v-bottomup.

Corollary 4.1.6 (Subsumptive beats magic sets for minimal form)**.** *For rules in minimal form,* $T_{s-topdn} \leq T_{v-botup}$ *and* $S_{s-topdn} \leq S_{v-botup}$*.*

We finally would like to give a class of rules for which the time complexity of s-topdown is better than v-topdown and v-bottomup. The property of this class is that the free variables in hypotheses are bound without considering the bound arguments from the conclusions.

Theorem 4.1.7. *Let P be a set of Datalog rules and a query, such that there exists a rule r in P which contains a free variable v such that (i) in the leftmost hypothesis h that v appears in, all arguments of h are free, (ii) v appears in another hypothesis in r that has bound arguments, (iii) v appears in an IDB hypothesis in r that will be subsumed in s-topdown, then* $T_{s-topdn} \leq T_{v-topdn}$*.*

Proof. We analyze a rule r that satisfies the given properties. If there exists such a v with properties (i) and (ii), then that means that v may take asymptotically more values in a hypothesis than it can after all hypotheses are considered, hence, those initial values are irrelevant. If an IDB hypothesis as given in (iii) exists, there will be subqueries generated from that hypothesis using those irrelevant values for v. If those subqueries are subsumed in s-topdown, then v-topdown will have to process asymptotically more facts than s-topdown; since the irrelevant values for v will be looked up in the table for s-topdown, and discarded, whereas in v-topdown, they contribute to the complexity. Therefore, a set of rules with the given properties are guaranteed to be worse in time complexity in v-topdown. □

4.2 Subsumptive demand transformation for bottom-up evaluation

We have shown that s-topdown beats v-topdown and the analogous transformation MST for bottom-up. However, there exists no transformation analogous to subsumptive tabling for bottom-up evaluation. In this section, we develop *subsumptive demand transformation* (SDT) for which the bottom-up evaluation of resulting rules have no worse performance than s-topdown.

Subsumptive demand transformation transforms a set of rules and a query into a new set of rules, such that all the facts that can be inferred from the new set of rules contain only facts that would be inferred during v-topdown of the original rules. It adds new rules that define needed facts for each hypothesis in each rule, adds hypotheses to the original rules to restrict computation to infer only needed facts, and adds negated hypotheses to the rules that define needed facts to make use of subsumption. For each subsumptive demand pattern $\langle \mathtt{p}, \mathtt{s} \rangle$, and for each rule

`p(...)  :-` $\mathtt{h}_1$`,...,` $\mathtt{h}_n$`.`

it generates

`p(...)  :- d_p_s(`$\mathtt{a}_1$`,...,`$\mathtt{a}_k$`),` $\mathtt{h}_1$`,...,` $\mathtt{h}_n$`.`

where $\mathtt{a}_1$,...,$\mathtt{a}_k$ are arguments of the conclusion that are bound by `s`. The added hypotheses restrict the rules to infer only facts necessary. The predicates of those hypotheses are defined as follows. For the given query, `p(`$\mathtt{a}_1$`,...,`$\mathtt{a}_k$`)?`, the following fact is generated

`d_p_s(`$\mathtt{a}_{b1}$`,...,`$\mathtt{a}_{bl}$`).`

where $\mathtt{a}_{b1}$,...,$\mathtt{a}_{bl}$ are the constant arguments of the query, and `s` is the pattern string of the query. For each rule r generated, `c :-` $\mathtt{h}_0$`,...,` $\mathtt{h}_n$`.`, and for each $\mathtt{h}_i$ whose predicate is an IDB predicate `p`, the following rule is generated

`d_p_s(`$\mathtt{a}_1$`,...,`$\mathtt{a}_k$`) :-` $\mathtt{h}_0$`,...,` $\mathtt{h}_{i-1}$`,`
`not d_p_s`$_1$`(`$\mathtt{a}_{b_1}$`,...,`$\mathtt{a}_{b_k}$`), ...`

where $\mathtt{a}_1$,..., $\mathtt{a}_k$ are the bound arguments of $\mathtt{h}_i$, and `s` is the pattern string of $\mathtt{h}_i$, and there exists a negated hypothesis `not d_p_s`$_i$`(`$\mathtt{a}_{b_1}$`,...,`$\mathtt{a}_{b_k}$`)` for each pattern string $\mathtt{s}_i$ which *subsumes* s, and the arguments of those hypotheses correspond to the arguments of the conclusion bound with respect to $\mathtt{s}_i$. A pattern string $\mathtt{s}_1$ of length n subsumes a pattern string $\mathtt{s}_2$ of length n iff $\mathtt{s}_1$ and $\mathtt{s}_2$ are different, and for each i less than n, the ith character of $\mathtt{s}_1$ is either '`f`' or the same as the ith character of $\mathtt{s}_2$. Semantically, the negated hypotheses ensure that no demand has been inferred that would correspond to a subsuming query in s-topdown.

For the rules in the running example, and the query `rel(c,y)?`, subsumptive demand transformation results in the following rules:

```
rel(x,y) :- d_rel_bf(x), imm(x,y).          (1bf)
rel(x,y) :- d_rel_bf(x), imm(u,v),
            rel(u,x), rel(v,y).             (2bf)
rel(x,y) :- d_rel_bb(x,y), imm(x,y).        (1bb)
rel(x,y) :- d_rel_bb(x,y), imm(u,v),        (2bb)
```

```
            rel(u,x), rel(v,y).
d_rel_bf(c)                                    (Q)
d_rel_bb(u,x) :- d_rel_bf(x), imm(u,v),        (2bf.3)
                 not d_rel_bf(u),
                 not d_rel_fb(x),
                 not d_rel_ff.
d_rel_bf(v) :- d_rel_bf(x), imm(u,v),          (2bf.4)
               rel(u,x),
               not d_rel_ff.
d_rel_bb(u,x) :- d_rel_bb(x,y), imm(u,v). (2bb.3)
                 not d_rel_bf(u),
                 not d_rel_fb(x),
                 not d_rel_ff.
d_rel_bb(u,x) :- d_rel_bb(x,y), imm(u,v), (2bb.4)
                 rel(u,x).
                 not d_rel_bf(u),
                 not d_rel_fb(x),
                 not d_rel_ff.
```

where each rule (`Ns`$_1$`s`$_2$) is generated from rule (`N`) for the pattern string `s`$_1$`s`$_2$, each rule (`Ns`$_1$`s`$_2$`.M`) captures the demand due to the `M`th hypothesis of rule (`Ns`$_1$`s`$_2$), and the fact (`Q`) corresponds to the given query.

Finally, each resulting rule is split into rules of two hypotheses from left to right. This transformation coupled with the bottom-up evaluation provides an implementation method with same time complexity as and better space complexity than v-bottomup.

This transformation follows the same method as variant demand transformation, except that it uses subsumptive demand patterns and inserts the negated hypotheses for subsumption of demand facts.

Handling negation. SDT introduces negated hypotheses to rules, and negation in Datalog may be interpreted under several different semantics. To match the behavior of subsumptive tabling, we use *inflationary semantics* [39] for negation. Inflationary semantics is a temporal semantics, and it checks whether a fact exists at the time when the negation is encountered. In other words, when there exists a substitution of variables in a rule such that all non-negated hypotheses of a rule are facts, then the negated hypotheses under that subsitution are checked whether any of them has currently been inferred as a fact. If none has been inferred as a fact, the rule is used to infer the conclusion under the substitution as a fact. We

call bottom-up evaluation extended with inflationary semantics *inflationary bottom-up evaluation*.

During inflationary bottom-up evaluation, a fact may be considered false at one point, and true at a later point. We show that the implementation of the rules obtained by SDT with inflationary semantics infers the same facts of the given predicates as the rules obtained by variant demand transformation, therefore SDT preserves correctness.

Theorem 4.2.1 (Correctness of SDT)**.** *Let P be a set of Datalog rules and a query. Let P_v be the rules obtained by variant demand transformation from P, and let P_s be the rules obtained by subsumptive demand transformation from P. Then, for each predicate p in P, the bottom-up evaluation of P_v and inflationary bottom-up evaluation of P_s infer the same facts of p.*

Proof. The rules that define given predicates are the same in P_v and P_s, only demand predicates from SDT have additional negated hypotheses. Suppose a fact d of a demand predicate is inferred in bottom-up evaluation of P_v and not in the bottom-up evaluation of P_s, and that d is used to infer a fact f of a given predicate. Then, for the rule used to infer d in P_v, during bottom-up evaluation of P_s, all positive hypotheses were satisfied, but at least one of the negated hypothesis was a fact d'. By definition, d' is a demand fact that corresponds to a query subsuming the query corresponding to d, therefore there exists a rule that infers f using d'. Hence, the evaluation of P_v does not infer more facts for given predicates than the evaluation of P_s.

For the other direction, since the evaluation is monotonic, the evaluation of P_s cannot infer more facts that the evaluation of P_d. □

Scheduling in bottom-up. We have shown that the order of subqueries are important for s-topdown as discussed in Section 3. Analogously, the order of demand facts inferred and considered is important in s-bottomup. In inflationary bottom-up evaluation, facts are processed in an undefined order. We modify that by considering facts for the demand predicates first, and in the order they are inferred. If no demand facts are left, then we consider facts for given predicates. We call this evaluation method *demand-first inflationary bottom-up evaluation*.

By considering demand facts first and in order, demand-first inflationary bottom-up evaluation mimicks s-topdown. The only difference that remains is the retrieval order of facts of given predicates. We have not defined such an order for s-topdown either, and in practice, different systems may opt for different orders. From now on, we assume that a particular retrieval

order is used for facts of given predicates, and the same order is used during demand-first inflationary bottom-up evaluation.

For a set of rules and a query P, we call the demand-first inflationary bottom-up evaluation of the rules resulting from SDT of P, *s-bottomup* of P.

4.2.1 Relationship to subsumptive tabling and magic set transformation

By using the analogy in the inferred demand facts and encountered subqueries, we first show that s-bottomup beats s-topdown in time complexity.

Theorem 4.2.2 (S-bottomup beats s-topdown)**.**
$T_{s-botup} \leq T_{s-topdn}$.

Proof. Let P be a set of rules and a query. Let P_a be the set of rules and query after subsumptive binding annotation of P, and P_s be the set of rules obtained by SDT of P.

$T_{s-topdn}$ is the sum of the complexities incurred by each rule in P_a. For each rule r in P_a of the form `p(...) :- body.`, there is a rule r' of the form `p(...) :- d(...), body, negated_hypos.` in P_s, where `d(...)` is the new demand hypothesis. The complexity incurred by r for $T_{s-topdn}$ is $i \times l$, where i is *the number of invocations* to r, and l is *the local complexity*, and l is the product of the sizes of hypotheses.

Facts of `d` are obtained from all of the call sites to `p` with the same binding pattern as s-topdown, and new facts for `d` are no longer inferred in the same asymptotic time as it would take for s-topdown to reach a subsuming subquery, since the scheduling for s-bottomup is analogous to s-topdown, `#d`$= i$. For $T_{s-botup}$, the complexity incurred by a rule is the number of times the rule fires. Therefore, the complexity incurred by r' has an upper bound `#d`$\times l = i \times l$. Note that the negated hypotheses do not incur additional time complexity since they require constant-time lookups in the set of facts inferred.

The only rules in P' that do not correspond to a rule in P_a are the rules that infer facts of the predicates added for demand. The additional complexity incurred for $T_{s-botup}$ by each such rule is already dominated by a component of the complexity in $T_{s-topdn}$, because this complexity equals the number of invocations for the rule that the demand hypothesis would be added to, and the number of invocations is used as a factor in a summand of $T_{s-topdn}$.

Hence, $T_{s-botup} \leq T_{s-topdn}$. □

As is the case between v-topdown and v-bottomup, we show that s-bottomup and s-topdown have the same time complexity for rules in minimal form. For this purpose, we reuse the below lemma from the last chapter.

Lemma 4.2.3. *In bottom-up evaluation, if all variables in the hypotheses of a rule r are also in the conclusion of r, then the number of facts inferred using r equals the number of firings of r.*

Theorem 4.2.4 (S-bottomup equals s-topdown for minimal form)**.** *For rules in minimal form, $T_{s-botup} = T_{s-topdn}$.*

Proof. Let P be a set of rules and a query. Let P_a be the set of rules and query after subsumptive binding annotation of P, and P_s be the set of rules obtained by SDT of P. Each rule r in P_a is of one of two forms:

(i) r has one hypothesis, so has the form `c :- h.` In P_s, there is a rule r' corresponding to r, and is of the form `c :- d, h.`, where `d` is the new demand hypothesis. The complexity incurred by r' to $T_{s-botup}$ and by r to $T_{s-topdn}$ are both dominated by the size of the predicate of `h`, since `h` contains all variables in `d`.

(ii) r has two hypotheses, so has the form `c :-` $\mathtt{h}_1$`,` $\mathtt{h}_2$`.` In P_s, there is a rule r' corresponding to r, and is of the form `c :- d,` $\mathtt{h}_1$`,` $\mathtt{h}_2$`.`, where `d` is the demand hypothesis added. As before, the complexity incurred by r to $T_{s-topdn}$, denoted $T_{s-topdn}(r)$, equals the product of the sizes of the predicates `d`, $\mathtt{h}_1$, and $\mathtt{h}_2$, and the number of facts of `d` and the number of invocations to the rule is the same as argued in the previous theorem. However, bottom-up computation can decompose the rules to possibly improve performance. In this case, it would obtain the following two rules: `new :-` `d,` $\mathtt{h}_1$`.` and `c :- new,` $\mathtt{h}_2$. The complexity of the first rule is less than $T_{s-topdn}(r)$. Since there are no singleton variables, the variables of `d` and $\mathtt{h}_1$ must appear in `new`. Then, by the lemma above, the size of the predicate of `new` equals the running time of the rule that generates it, and hence the complexity incurred by the second rule obtained from r' equals $T_{s-topdn}(r)$.

Therefore, for each complexity summand incurred by rules in P_a for $T_{s-topdn}$, there is a rule in P_s that incurs the same complexity summand for $T_{s-botup}$. Combining this with Theorem 4.2.2, which states that $T_{s-botup} \leq T_{s-topdn}$, we obtain $T_{s-botup} = T_{s-topdn}$. □

To compare s-bottomup and v-bottomup, note that we have already shown that s-bottomup beats v-bottomup in space complexity in Theorem 4.2.1, since both infer the same facts for given predicates and s-bottomup may infer less facts for demand predicates.

For time complexity, s-bottomup beats v-bottomup due to Theorem 4.2.4 and Corollary 4.1.6, and is asymptotically faster for the class of rules and queries described in Theorem 4.1.7 due to the aforementioned results. Therefore, we obtain the following corollary.

Corollary 4.2.5. *$T_{s-botup} \leq T_{v-botup}$ and $S_{s-botup} \leq S_{v-botup}$. For the rules described in Theorem 4.1.7, $T_{s-botup} < T_{v-botup}$ and $S_{s-botup} < S_{v-botup}$.*

4.3 Subsumption optimization

We have shown that s-topdown beats v-topdown, and s-bottomup beats v-bottomup. However, there are cases when the subsumptive methods are not effective because subqueries that subsume others may appear later during evaluation, and the complexity may have been reduced if they had appeared earlier. A class of rules and queries for which this is guaranteed to happen has been shown in Theorem 4.1.7. In this section, we show a transformation method to ensure that subqueries that subsume others for s-topdown and s-bottomup are processed first. We first show the effectiveness of the method for s-topdown, and show that it works for s-bottomup as well. We call our transformation *subsumption optimization*.

The method first identifies demand patterns that should be subsumed, and then transforms the rules so that any subquery with that demand pattern is subsumed during s-topdown.

(i) Identification of demand patterns to subsume. For a set s of demand patterns of a predicate determined for the given rules and query, the method generates all sets of subsuming demand patterns, and generates annotated rules for each such set. Then, for each set of the resulting rules, we compare the time complexity of the resulting annotated rules with the original annotated rules. If it can be proven that the asymptotic time complexity of the resulting rules due to a set s_2 of subsuming demand patterns is lower than the original annotated rules, then we say that the demand patterns in s should be subsumed by the demand patterns in $s2$.

For rules `(1)` and `(2)` in the running example, and the query `pt(c,y)?`, the demand patterns are ⟨`rel`,'`bf`'⟩ and ⟨`rel`,'`bb`'⟩. The rules obtained by subsumptive binding annotation are:

```
rel_bf(x,y) :- imm(x,y).
rel_bf(x,y) :- imm(u,v), rel_bb(u,x), rel_bf(v,y).
rel_bb(x,y) :- imm(x,y).
```

```
rel_bb(x,y) :- imm(u,v), rel_bb(u,x), rel_bb(v,y).
```

It would be asymptotically better in time complexity if the calls to `rel_bb` were subsumed by previously encountered `rel_bf` subqueries. The intuition is that if the outdegree of the `rel` relation is constant, then asymptotically many more queries to `rel_bb` may be made than the ones that would be relevant due to the last rule. Therefore, ⟨`rel`,'`bb`'⟩ should be subsumed by ⟨`rel`,'`bf`'⟩.

(ii) Transformation. For each demand pattern ⟨`p`, `s`⟩ that should be subsumed by ⟨`p`, `s2`⟩, for each rule r whose ith hypothesis' is `p(`$\mathtt{a}_1$`,...,`$\mathtt{a}_n$`)` and pattern string is `s`, we generate the following rule:

`q(`$\mathtt{a}_{b1}$`,...,`$\mathtt{a}_{bk}$`) :- p(`$\mathtt{a}_1$`,...,`$\mathtt{a}_n$`).`

where `q` is a fresh predicate name, and $b1$,...,bk are the indices of the 'b' characters in `s2`. Then, before the ith hypothesis of r, we insert `q(`$\mathtt{a}_{b1}$`,...,`$\mathtt{a}_{bk}$`)` as a hypothesis. Therefore, after this transformation, a subquery that subsumes the subquery of the original ith hypothesis will be guaranteed to be made during s-topdown, so all subqueries corresponding to the demand pattern ⟨`p`, `s`⟩ will be avoided.

For the second rule in the running example, the second hypothesis is the only hypothesis whose pattern string should be avoided ('`bb`'). Thus, this rule is transformed to the following two rules.

```
rel_bf(x,y) :- imm(u,v), i(u), rel(u,x), rel(v,y).
i(u) :-  rel(u,x).
```

We prove that the subsumption optimization preserves the semantics of the original set of rules.

Theorem 4.3.1. *Let P be a set of Datalog rules and a query. For each given predicate p of P, s-topdown of P and s-topdown of the rules resulting from subsumption optimization infer the same facts for p.*

Proof. The rules after subsumption optimization are more restricted due to added hypotheses, so s-topdown of those rules could only infer the same facts or less than s-topdown of P. However, they could not infer less either, since the added hypotheses are defined by rules whose hypothesis subsume existing hypotheses, therefore they could not be false while all other hypotheses are true. Hence, s-topdown of P and s-topdown of the rules after subsumption optimization infer the same facts for given predicates. □

We show that this transformation achieves the same effect for s-bottomup, by showing that the introduced hypotheses and rules ensure that demand facts cannot be inferred for demand patterns that should be subsumed.

Theorem 4.3.2. *Let P be a set of Datalog rules and a query. The s-bottomup of P after subsumption optimization does not infer any fact for the demand predicates that correspond to the demand patterns that should be subsumed.*

Proof. For any hypothesis h corresponding to a demand pattern that should be subsumed, the transformation introduces a new hypothesis to its left. For that hypothesis to be true, the demand fact that corresponds to a subsuming demand pattern must be inferred by construction of the transformation. Thus, when all of the hypotheses to the left of h is true, a demand fact corresponding to a demand pattern subsuming the demand pattern for h must have been inferred. Therefore, no fact can be inferred for demand predicates of demand patterns that should be subsumed. □

Application to demand-driven pointer analysis. We show that by applying subsumption optimization to the specification of Andersen's pointer analysis for C [4], we automatically derive Heintze and Tardieu's algorithm for demand driven pointer analysis [33], and our complexity analysis can be used to obtain a precise time and space complexity analysis.

Given a C program, statements in a program relevant to pointer analysis can be reduced to four kinds, which can be represented directly as Datalog facts:

- `p = &q` is represented by `bare_addr(p,q)`.
- `p = q` is represented by `bare_bare(p,q)`.
- `p = *q` is represented by `bare_star(p,q)`.
- `*p = q` is represented by `star_bare(p,q)`.

Andersen's pointer analysis can be specified directly as four Datalog rules, where `pt(p,q)` denotes `p` points to `q`:

```
pt(p,q) :- bare_addr(p,q).                    (A1)
pt(p,q) :- bare_bare(p,r), pt(r,q).           (A2)
pt(p,q) :- bare_star(p,s), pt(s,r), pt(r,q). (A3)
pt(p,q) :- star_bare(r,s), pt(r,p), pt(s,q). (A4)
```

Given a query `pt(c,q)?` with the first argument bound, the demand patterns are ⟨`pt`,'`bf`'⟩ and ⟨`pt`,'`bb`'⟩. Subsumptive binding annotation yields:

```
pt_bf(p,q) :- bare_addr_bf(p,q).
pt_bf(p,q) :- bare_bare_bf(p,r), pt_bf(r,q).
pt_bf(p,q) :- bare_star_bf(p,s), pt_bf(s,r), pt_bf(r,q).
pt_bf(p,q) :- star_bare_ff(r,s), pt_bb(r,p), pt_bf(s,q).

pt_bb(p,q) :- bare_addr_bb(p,q).
pt_bb(p,q) :- bare_bare_bf(p,r), pt_bb(r,q).
pt_bb(p,q) :- bare_star_bf(p,s), pt_bf(s,r), pt_bb(r,q).
pt_bb(p,q) :- star_bare_ff(r,s), pt_bb(r,p), pt_bb(s,q).
```

Our complexity analysis can be used to show that asymptotically many more queries to `pt_bb` may be made than the ones that would be relevant due to the last two rules generated. The number of calls to `pt_bb` may be as many as v^2 where v is the number of variables in the C program, however the `pt` relation may not be as dense as $O(v^2)$. Due to this fact, if one carries on the complexity analysis for the set of rules, it can be inferred that ⟨`pt`,'`bb`'⟩ should be subsumed by ⟨`pt`,'`bf`'⟩. Subsumption optimization results in the rules `(A1)`, `(A2)`, `(A3)`, and the following two rules due to `(A4)`:

```
pt(p,q) :- star_bare(r,s), i(r), pt(r,p), pt(s,q).
i(r) :- p(r,p).
```

S-bottomup of the resulting rules corresponds precisely to Heintze and Tardieu's algorithm, and performing precise complexity analysis on the result gives the explanation for why and when the algorithm is efficient.

4.4 Experiments

We support our complexity analyses and comparisons by experiments. For s-topdown and v-topdown, we use XSB [75]. For v-bottomup and s-bottomup, we use the implementation method of [51] modified with the described extensions when necessary to generate Python code from the rules.

We show experimental results for the examples we have discussed, the running example and the demand-driven pointer analysis. Our experiments on the rules were conducted on a 3.0 GHz Intel Q9650 with 4 GB of memory, running SuSE Linux, and using XSB 3.2.

We instantiate the complexity parameters in predicted complexities with their values computed from the data. We use *space units* to mean number of

unique table inserts for v- and s-topdown, and the number of facts inferred plus the number of elements in auxiliary maps [51] for v- and s-bottomup. We use *returns* to mean the number of total facts returned from rule invocations for tabled top-down evaluation, and *firings* to mean the number of firings for demand-driven bottom-up evaluation.

In our benchmarks, predicates have two arguments. For experiments, we fix `#p` and `#p.1/2` for each input predicate `p` to generate a set of data such that the size of each predicate is maximal, i.e., the worst-case behavior is exhibited. Then, we increase `#p` and `#p.1/2` to generate the next set of data, and repeat.

For a query with both arguments free for the running example, we confirm the asymptotic time and space complexities. The left figure in Figure 4.3 shows that s-bottomup and s-topdown are asymptotically faster than v-bottomup and v-topdown. It also shows that due to splitting rules into two hypotheses s-bottomup is asymptotically faster than s-topdown. The right figure in Figure 4.3 shows that the space usage of v- and s-topdown is smaller than v-bottomup and s-bottomup, respectively. It shows that the space usage of s-topdown and s-bottomup is smaller than v-topdown and v-bottomup, respectively.

For demand-driven pointer analysis, and a query with the first argument bound, we apply subsumption optimization to the rules and confirm the asymptotic improvement in time complexity against s-bottomup, and show the difference in space usage. Figure 4.4 confirms that subsumption optimization improves time complexity and reduces space usage.

We also performed experiments on a well-known benchmark in semantic web [45] that has 961 rules and 654 facts; and we also performed the pointer analysis on real data from a C program with 2430 facts. Both queries run out of memory in XSB using v-topdown, but the ontology query runs in 12 seconds, and the pointer analysis runs in under 0.1 seconds when s-topdown is used.

Finally, to compare our results with existing SQL database implementations, we converted the rules of the running example into SQL queries and performed experiments on MSSQL, one of the few SQL databases that support recursion. The running example timed out (more than 10 minutes) for all of our data points except the first two. The variant and subsumptive demand transformations cannot be used on SQL databases since no mainstream SQL database implementation supports mutual recursion to the best of our knowledge, and demand transformations result in mutually recursive rules.

4.5 Related work

Datalog has been extensively studied [15, 1]. Variant tabling was introduced in [66], and has been widely studied. It has been implemented in top-down evaluation engines such as XSB [18] and YAP [22]. Subsumptive tabling [58] was introduced more recently, but has not been studied widely, and only implemented in XSB.

The time and space complexity of variant tabling for Datalog was given an imprecise upper bound in [75], and a method for precise calculation of time and space complexities of variant tabling for Datalog is given in the last chapter. There is no prior work on complexity analysis of subsumptive tabling to the best of our knowledge. The analyses in this chapter build on the methods in the last chapter to present the first method for precise calculation of worst-case time and space complexities of subsumptive tabling, and establishes that it beats variant tabling.

For bottom-up evaluation, transformations for demand-driven evaluation have been studied, including the well-known magic set transformation (MST) [8] and variant demand transformation that our work extends. It has been shown that both these transformations are equivalent to variant tabling both operationally [12] and in terms of complexity in the last chapter. We show that subsumptive tabling beats MST, and then give the first transformation for bottom-up evaluation, for which the evaluation of resulting rules is analogous to subsumptive tabling. We show that the complexity of the resulting rules from our transformation beats subsumptive tabling and MST in time complexity.

The relationship between top-down and bottom-up evaluation has been studied in a variety of contexts with different flavors of rules [57, 68, 56, 12]. Our work is the first to establish precise relationships between variant and subsumptive tabling, and MST and the novel subsumptive demand transformation.

By characterizing improvement using the complexity analysis, we also introduce a transformation called subsumption optimization, to reuse as many facts as possible. We have shown that our methods can be used to systematically derive a well-known demand driven pointer analysis algorithm [33].

Additionally, we have implemented our method and confirmed our analysis results through experiments on well-studied benchmarks.

```
... same top code ...

procedure invoke(q,r,i,θ,res=false):
  // If there are still hypotheses of r to process
  if i ≤ |hypos(r)|:
    h_i = subst(the ith hypothesis of r,θ)
    if h_i is not an IDB hypothesis:
      // Call invoke for each matching fact
      for fact ∈ F | θ' = unify(h_i,fact) ≠ ∅:
        invoke(q,r,i+1,θ∪θ')
    // If h_i is a variant of an existing table key
    else if ∃k ∈ keys(Table) | variant(h_i,k):
      // Record current arguments for resuming invoke later
      Suspension[⟨k,h_i⟩] ∪= {⟨q,r,θ,i⟩}
      // Call invoke for each fact in values for key k
      for fact ∈ Table[k]:
        θ' = unify(h_i,fact)
        invoke(q,r,i+1,θ∪θ')
    // If a variant does not exist in table keys
    else:
      Table[h_i] = ∅
      // Call invoke for each r matching new query h_i
      for r' ∈ R | θ' = unify(concl(r'),h_i) ≠ ∅:
         invoke(h_i,r',1,θ')
      for fact ∈ Table[h_i]:
        θ' = unify(h_i,fact)
        invoke(q,r,i+1,θ∪θ')
        resume(h_i,fact)
      // Record current arguments for resuming invoke later
      Suspension[⟨h_i,h_i⟩] ∪= {⟨q,r,θ,i⟩}
  // If no more hypothesis is left to process
  else:
    fact = subst(q,θ)
    // If the fact has not been inferred before
    if fact ∉ Table[q]:
      // Add the fact to the table
      Table[q] ∪= {fact}
      if res:
        resume(q,fact)
endproc

procedure resume(q,fact):
  // Resume computations
  for ⟨k,h⟩ ∈ keys(Suspension) | k = q:
    for ⟨q',r',θ',i'⟩ ∈ Suspension[⟨q,h⟩]:
      θ'' = unify(h,fact)
      invoke(q',r',i'+1,θ'∪θ'',true)
```

Figure 4.1: Top-down evaluation of query q, given a set of facts F and a set of rules R, with subsumptive tabling and *local scheduling*

```
... same top code ...

procedure invoke(q,r,i,θ,res=false):
  // If there are still hypotheses of r to process
  if i ≤ |hypos(r)|:
    h_i = subst(the ith hypothesis of r,θ)
    if h_i is not an IDB hypothesis:
      // Call invoke for each matching fact
      for fact ∈ F | θ' = unify(h_i,fact) ≠ ∅:
        invoke(q,r,i+1,θ∪θ')
    // If h_i is a variant of an existing table key
    else if ∃k ∈ keys(Table) | variant(h_i,k):
      // Record current arguments for resuming invoke later
      Suspension[⟨k,h_i⟩] ∪= {⟨q,r,θ,i⟩}
      // Call invoke for each fact in values for key k
      for fact ∈ Table[k]:
        θ' = unify(h_i,fact)
        invoke(q,r,i+1,θ∪θ')
    // If a variant does not exist in table keys
    else:
      Table[h_i] = ∅
      caller[h_i] = ⟨q,r,θ,i⟩
      // Call invoke for each r matching new query h_i
      for r' ∈ R | θ' = unify(concl(r'),h_i) ≠ ∅:
          invoke(h_i,r',1,θ')
      for fact ∈ Table[h_i]:
        θ' = unify(h_i,fact)
        resume(h_i,fact)
      // Record current arguments for resuming invoke later
      Suspension[⟨h_i,h_i⟩] ∪= {⟨q,r,θ,i⟩}
  // If no more hypothesis is left to process
  else:
    fact = subst(q,θ)
    // If the fact has not been inferred before
    if fact ∉ Table[q]:
      // Add the fact to the table
      Table[q] ∪= {fact}
      if res:
        resume(q,fact)
      else:
        ⟨q',r',θ',i⟩ = caller[q]
        invoke(q',r',i'+1,θ∪θ')
endproc

procedure resume(q,fact):
  // Resume computations
  for ⟨k,h⟩ ∈ keys(Suspension) | k = q:
    for ⟨q',r',θ',i'⟩ ∈ Suspension[⟨q,h⟩]:
      θ'' = unify(h,fact)
      invoke(q',r',i'+1,θ'∪θ'',true)
```

Figure 4.2: Top-down evaluation of query q, given a set of facts F and a set of rules R, with subsumptive tabling and *batched scheduling*

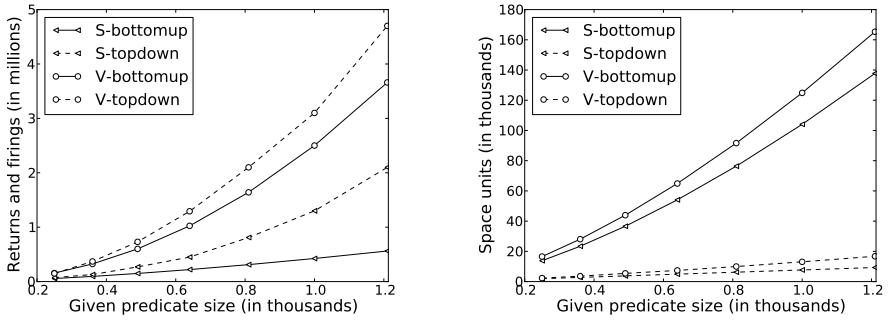

Figure 4.3: Firings/returns and space units for v-topdown, s-topdown, v-bottomup, and s-bottomup for the running example.

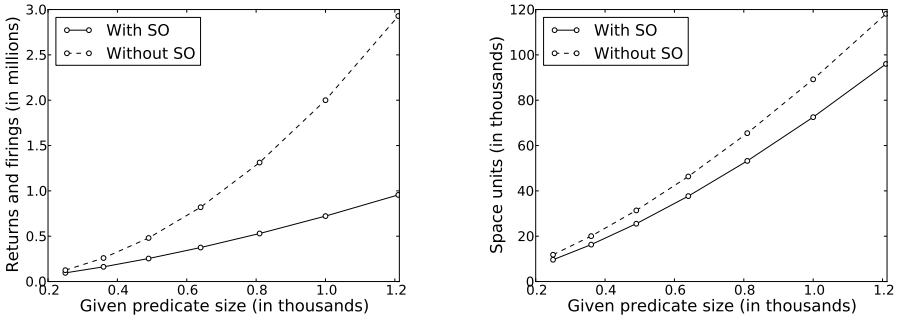

Figure 4.4: Firings/returns and space units for s-bottomup with and without subsumption optimization (SO) for the pointer analysis benchmark.

Chapter 5

Specialization and recursion conversion

Given a set of rules and a kind of query, i.e., a query predicate with indications of which arguments will be bound, we show methods to generate a set of rules that is specialized for the kind of query, and produces complexity formulas for the time and space complexities of the generated rules.

We first describe *static removal of redundancies* that specializes the transformed rules with respect to the kinds of query, so that bound parameters of the query predicate are used to restrict possible instantiations of the rules as much as possible. This is a simplified form of partial evaluation [42] and may yield asymptotic improvements in running time.

Then, we describe *recursion conversion* that transforms recursive rules into appropriate left or right linear recursive forms based on the kinds of queries, so that the connection between the queries and given facts can be established efficiently. Queries can then be answered equally efficiently for equivalent but slightly different recursive rules, which could otherwise differ asymptotically in running times.

In the methods, we use versions of transitive closure as our running example as below.

Doubly recursive:
```
path(x,y) :- edge(x,y).
path(x,y) :- path(x,z), path(z,y) .
```
(5.1)

Right recursive:
```
path(x,y) :- edge(x,y).
path(x,y) :- edge(x,z), path(z,y) .
```
(5.2)

Left recursive:
```
path(x,y) :- edge(x,y).
path(x,y) :- path(x,z), edge(z,y) .
```
(5.3)

These three sets of rules can be proven by induction to infer the same `path` facts. The right- and left-recursive versions of the transitive closure concatenate edges from the vertex on the left, respectively right, with paths to the vertex on the right, respectively left.

For these rules, there are 4 possible queries: `path(x,y)?` returns all pairs of vertices that have a path between them. `path(c,y)?` returns all vertices that are reachable from `c`. `path(x,c)?` returns all vertices that can reach `c`. `path(`$\mathtt{c}_1$`,`$\mathtt{c}_2$`)?` returns whether $\mathtt{c}_2$ is reachable from $\mathtt{c}_1$.

We use the following notation for complexity analysis. For queries regarding transitive closure, if the first argument is bound, it is denoted by $\mathtt{c}_1$, and if the second argument is bound, it is denoted by $\mathtt{c}_2$.

- V : number of vertices, P: number of paths, E: number of edges.
- $E(\mathtt{c})$: number of edges that are on any path from `c` to any vertex.
  $IE(\mathtt{c})$: number of edges that are on any path from any vertex to `c`.
- $o(\mathtt{c})$: outdegree of `c`, o: maximum outdegree of vertices.
  $i(\mathtt{c})$: indegree of `c`, i: maximum indegree of vertices.
- $R(\mathtt{c})$: number of vertices reachable from `c`, R: maximum number of vertices reachable from any vertex.
  $IR(\mathtt{c})$: number of vertices that reach `c`.

5.1 Static removal of redundancies using specialization

Constants in the arguments of a query are called *static inputs*. For example, in the query `path(c,x)?`, `c` is a static input. Specialization uses static inputs to restrict the number of inferred facts by transforming the rules. Program specialization is also known as partial evaluation, and has been studied in logic programming [42], where it is sometimes called partial deduction.

Specialization for a set of Datalog rules S, and a query Q is obtaining another set of rules S' and a query Q' that satisfy the following: Every fact inferred as an answer to Q' during the evaluation of S' is a *projection* of a fact inferred as an answer to Q during the evaluation of S, where a *projection* of a fact is a selection of zero or more arguments from that fact up to a renaming of the predicate.

As an example, consider S being (5.3), and `path(c,y)?` being Q. Let S' be:

$$\begin{aligned} &\texttt{path}_{1\texttt{c}}\texttt{(y) :- edge(c,y).} \\ &\texttt{path}_{1\texttt{c}}\texttt{(y) :- path}_{1\texttt{c}}\texttt{(z), edge(z,y).} \end{aligned} \tag{5.4}$$

and Q' be $\texttt{path}_{1\texttt{c}}\texttt{(y)?}$. The original query finds all vertices that are reachable from `c` by selecting the `path` facts whose first argument is `c`. Q' and S' do exactly that, and the answers to Q' are the vertices that are reachable from `c`. By inserting `c` as the first argument in the answers of Q', one trivially reconstructs the answers of Q.

To describe specialization, we need to define substitution. For a set of rules S, we denote the set of hypotheses of all rules by $\texttt{h}(S)$. We denote the conclusion of a rule r by $\texttt{c}(r)$. A *substitution* is a map from variables to constants. A substitution θ applied to a rule r, denoted $r\theta$, replaces the variables in r with constants according to θ. We say that an atom a' is an *instance* of an atom a if there is a substitution θ such that $a\theta = a'$; in case such a substitution exists, it is denoted $\texttt{subst}(a, a')$.

We specialize a set of Datalog rules with respect to a query via the fixpoint of a function f, which takes a set S of rules and a set A of atoms, and returns both of them with new elements added. At each step of computation, if there is an atom a in A, and a rule r in S for which a is an instance of the conclusion of r, then a new rule r', which is r updated with the substitution that makes a and the conclusion of r identical, is added to S and all hypotheses of r' are added to A. That is:

$$f(\langle S, A\rangle) = \langle S \cup S', A \cup \texttt{h}(S')\rangle$$

$$\text{where } S' = \{r\theta | a \in A, r \in R, \theta = \texttt{subst}(\texttt{c}(r), a) \neq undef\}.$$

Given a set of rules S, and a query Q, specialization computes the fixpoint of $f(S, Q)$ and returns the first component of the output pair as the desired set of specialized rules. The output of the function also has the original rules in the specialized set, therefore we need to remove them if they are not needed for the evaluation of the specialized query. An original rule r in the output is not needed, unless a hypothesis of a specialized rule is identical to the conclusion of r up to variable renaming. Once these rules are removed, we rewrite all atoms that have constant arguments to remove constants, and assign names based on the original predicate names and the places and values of bound arguments. We only rewrite the atoms whose predicates appear in the conclusion of some rule.

Specialization of (5.3) with respect to the query `path(a,y)?` yields:

```
path_1c(y) :- edge(c,y).
path_1c(y) :- path_1c(z), edge(z,y).
```
(5.5)

and the query `path_1c(y)?`. Given the same query, if one applies specialization to (5.1), the original unspecialized rules remain since the `path(z,y)` hypothesis of the second rule is identical to the conclusion of the original rules up to variable renaming. The original rules of (5.2) also remain after specialization for the same reason.

To make specialization independent of the values of the static input, we perform the following: For any query Q with n distinct static inputs, we generate n fresh constants: say `c1`, ... ,`cn`, and replace the constants in Q with these fresh constants in order (i.e. the first distinct constant by `c1`, the second by `c2`, and so on). Next, we do specialization as described above for the given rules and rewritten Q. Note that, at this point, constants occur in the specialized rules only in the atoms for which no facts are derived by the rules. For any rule in the given set of rules, if a constant `ci` occurs in the rule, we replace it with a variable, say `x`, that does not occur in the rule, add `ci(x)` as a new hypothesis, where `ci` is a fresh predicate name to be used with `ci`, and add the fact `ci(oci)` to the set of rules, where `oci` is the ith original constant in the query. With this result, if another query Q' whose bound arguments are in the same places as Q is given, and Q''s ith constant is different than Q's, we retract the fact related to `ci`, and add a fact of `ci` that represents the new constant. For example, specialization of (5.3) with respect to the query `path(c,x)?` yields:

```
c(c).
path_1c(y) :- c(x), edge(x,y).
path_1c(y) :- path_1c(z), edge(z,y).
```
(5.6)

and the query `path_1c(y)?`. If one wants to change the original query to `path(c_2,x)?`, it is not necessary to re-perform specialization, but just replace the fact `c(c)`, with `c(c_2)`.

Note that, for any set of rules, specialization does not result in different time complexities of the generated rules when the rule order within the set or the hypothesis order inside the rules is changed.

We have shown that specialization may result in a set with more specialized rules, however it may include unspecialized rules as well. Evaluating a purely specialized set of rules should be more advantageous. The purely specialized rules derived from (5.3), and the query `path(c,x)?` can be evaluated in linear time in the number of edges. Since the time is proportional

to the combination of facts that make the hypotheses true, and `z` can only be assigned the vertices that can be reached from `c` as values, the evaluation takes time proportional to $E(\texttt{c})$. Specialization of the programs (5.1) and (5.2) with respect to the same query is evaluated in asymptotically worse time since they include the original rules. Therefore, programs with the same semantics might have different execution times with respect to the same queries, even after specialization.

Differences in time complexity of the specialized programs can only result from the combination of the bound arguments in the query and the version of program that is being specialized, so we show such cases. If the left-recursive version is given and the left argument of the query is bound, or symmetrically if the right-recursive version is given and the right argument of the query is bound, the specialized versions have cost $O(E)$. For the doubly recursive version, no matter which arguments are bound, the complexity is $O(R \times P)$. The following are the complexities of evaluating programs with respect to queries with different bound arguments:

Bound argument	Time complexity		
	Left-rec.	Right-rec.	Doubly-rec.
None	$O(R \times E)$	$O(R \times E)$	$O(R \times P)$
First	$O(E(c))$	$O(R \times E)$	$O(R \times P)$
Second	$O(R \times E)$	$O(IE(c_2))$	$O(R \times P)$
Both	$O(E(c))$	$O(IE(c_2))$	$O(R \times P)$

5.2 Recursion conversion for chain queries

In the previous section, we showed that specialization might not obtain a more specialized set of rules for a given query. In general, for any set of unspecializable rules, another set of rules that infers the same set of facts may be specializable. For transitive closure, one needs to convert a particular form of recursion into another for the specialization to work. We give a general transformation which is applicable to transitive closure. Given the following set of rules:

$$\texttt{r}(\vec{x})\ \texttt{:-}\ \texttt{p}_1(\vec{x}_1),\ \ldots\ ,\ \texttt{p}_n(\vec{x}_n).$$
$$\texttt{r}(\vec{x})\ \texttt{:-}\ \texttt{r}(\vec{y}),\ \texttt{r}(\vec{z}).$$

where $\vec{x}$, $\vec{x}_n$, $\vec{y}$, $\vec{z}$ each denote one or more variables, $\vec{y}$ and $\vec{z}$ have common variables $\vec{t}$, the uncommon ones are in different places in $\vec{y}$ than in $\vec{z}$, and at the same place in $\vec{x}$ as in $\vec{y}$ or $\vec{z}$, and the variables in $\vec{t}$ do not appear in $\vec{x}$. Also $\texttt{p}_i$ is not mutually recursive with `r`. Then the above rules are equivalent to both sets of rules below:

```
r(x⃗) :- p1(x⃗1),..,pn(x⃗n).          r(x⃗) :- p1(x⃗1),..,pn(x⃗n).
r(x⃗) :- p1(y⃗1),..,pn(y⃗n),r(z⃗).    r(x⃗) :- r(y⃗),p1(z⃗1),..,pn(z⃗n).
```

where each $\vec{y}_i$ (and $\vec{z}_i$) is obtained by substituting the variables of $\vec{x}_i$ with the substitution that makes $\vec{x}$ and $\vec{y}$ (respectively $\vec{z}$) identical.

All versions of transitive closure are instances of one of these schemas. Since they are all shown to be equivalent and there is a transformation method to transform from one to another, we exploit this fact before specialization.

We give a detailed complexity analysis of specialization extended with recursion conversion for transitive closure. Recursion conversion is also insensitive to hypothesis order or rule order. We just need to consider the main three versions of the transitive closure.

After applying the described transformations to any version of transitive closure, if any of the arguments is bound in the query, the program can be evaluated in $O(E)$ time, and if both arguments are free then the program can be evaluated in $O(R \times E)$ time. One can revise the $O(E)$ bound by more precise bounds as follows:

Bound argument	Time complexity for all three programs
None	$O(R \times E)$
First	$O(E(c))$
Second	$O(IE(c_2))$
Both	$O(\min(E(c_1), IE(c_2)))$

Recursion conversion as described is possible only for the given schema, i.e., doubly-recursive or linear Datalog programs, so it is of significance to convert a Datalog program into a linear one if possible. The question whether it is possible to perform such a transformation has been answered negatively in general, and a subset of Datalog programs have been shown to be convertible to linear ones [2].

For our purposes, any linearization procedure for a subset of Datalog is useful. If we obtain a program which obeys the schema for recursion conversion, we apply the recursion conversion to obtain different versions of the same program. We then apply our specialization algorithm to these different versions. After these steps, we can generate the program as in [51] and automatically analyze the time complexity of the bottom-up evaluation of each resulting program and choose the best one. In any of the steps if the transformation is not possible, we skip that step. The whole method can be summarized as: linearize (if possible), apply recursion conversion (if possible), specialize all versions, generate program, calculate complexity and choose the best. The algorithm is presented in Figure 5.1.

Algorithm *Optimization of rules*
Input: A set of Datalog rules S and a query Q
Output: A sequential program for the generation of answers to Q, with time complexity guarantees

1. **if** any rule in S is linearizable
2. **then** $S = \text{Linearize}(S)$
3. $RS \leftarrow \{S\}$
4. **for** each predicate p in S that fits the recursion conversion schema
5. **do** $S' = p$'s recursion type converted in S
6. $RS \leftarrow RS \cup \{S'\}$
7. $RSC = \{\}$: to keep rule sets with complexities
8. **for** each set R of rules in RS
9. **do** $R' \leftarrow R$ specialized for Q
10. $C \leftarrow$ Time complexity of evaluating R'
11. $RSC \leftarrow RSC \cup \{(R', C)\}$
12. Among all pairs in RSC, remove the ones that are provably worse in complexity than at least one pair.
13. **for** each pair (R, C) in RSC
14. **do** generate program from R
15. output C as the time complexity associated with it

Figure 5.1: Algorithm for optimization of rules using specialization and recursion conversion.

The time complexity of the method is dominated by the specialization step, which has a super-exponential upper bound in the maximum arity of the predicates. In practice, the arity of the predicates is relatively small, 2-3 in many realistic Datalog programs and almost never exceeds 10. Thus, assuming a small constant for the maximum arity of predicates, the transformation takes linear time in the size of the set of rules, since for each rule, there is a constant number of different atoms that can unify with its conclusion, and specialization of a rule with respect to an atom takes time proportional to its size.

There are Datalog programs for which recursion conversion is not possible; and specialization cannot succeed in obtaining better running time. In this case, a transformation method such as magic sets may obtain asymptotic speedup with tighter complexity bounds, but the worst-case running times of programs transformed by both our method and magic sets are the same.

5.2.1 Complexity comparison

We show the complexity of evaluating all versions with respect to various other strategies. We consider 12 versions of the transitive closure: the left, right and doubly-recursive programs, and for each program, different order of the two rules, and different order of hypotheses in the recursive rule. We denote the versions by three fields, the first being the recursion type (right, left, or doubly), the second being the order of rules (base-first or recursion-first), the third being the order of hypotheses (regular or inverse). Then for each version, we ask 4 different kind of queries: both arguments bound, only the first argument bound, only the second argument bound, and both arguments free. All results for left- and right-recursive rules are summarized in Figure 5.2. For the doubly-recursive rule, regardless of the version, Prolog takes infinite time, tabling and magic set takes $O(V^3)$, and static filtering takes $O(R \times P)$ time.

In this figure, we omit the order of rules, because the complexities and inferred facts remain the same for static filtering and magic sets, since they are bottom-up methods. For tabling, since termination is guaranteed, the complexities and inferred facts also remain the same. However, for Prolog evaluation, if the program does not terminate, there will be no inferred facts if the recursive rule is first, otherwise the evaluation will infer some facts, before it gets stuck in an infinite loop.

Method	Bound arg.	Time complexity			
		Left-rec.		Right-rec.	
		Regular	Inverse	Regular	Inverse
Prolog, cyclic gr	Any	Infinite			
Prolog, acyclic gr	Any	Infinite	Exponential	Exponential	Infinite
Tabling	None	$O(V^3)$	$O(V \times E)$	$O(V^3)$	$O(V \times E)$
	First	$O(E)$	$O(V \times E)$	$O(V^2)$	$O(V \times E)$
	Second	$O(V^3)$	$O(V^2)$	$O(V^3)$	$O(E)$
	Both	$O(E)$	$O(V^2)$	$O(V^2)$	$O(E)$
Static filtering	None	$O(V \times E)$		$O(V \times E)$	
	First	$O(R(c) \times o)$		$O(R \times E)$	
	Second	$O(R \times E)$		$O(IR(c_2) \times i)$	
	Both	$O(R(c) \times o)$		$O(IR(c_2) \times i)$	
Magic set	None	$O(V \times E)$		$O(V \times E)$	
	First	$O(R(c) \times o)$	$O(E)$	$O(V \times R(c) \times o)$	$O(V \times E)$
	Second	$O(V \times E)$	$O(V \times IR(c_2) \times i)$	$O(E)$	$O(IR(c_2) \times i)$
	Both	$O(R(c) \times o)$	$O(E)$	$O(E)$	$O(IR(c_2) \times i)$

Figure 5.2: A comparison of time complexities of computation using existing methods.

Prolog. Prolog evaluation resolves subgoals in a top-down fashion. It has the general vulnerability that for any version of the transitive closure, for cyclic graphs, it will not terminate once it enters a cycle, because it will be doomed to resolve the same subgoals infinitely many times. Even when the input is restricted to acyclic graphs, it may still not terminate or it may terminate in exponential time. Prolog does not keep track of discovered vertices and discovers a vertex through all possible paths, which is exponential in the worst case. For versions whose first hypothesis is recursive in the recursive rule, the evaluation will be infinite with respect to all queries regardless of the graph structure. The doubly-recursive versions are always infinite; what differs is the generated facts due to the order of rules and hypotheses.

Tabling. Tabling adds memoization to Prolog evaluation to avoid repeating subgoals. It is guaranteed to be finite and be bounded by $O(V^3)$ for any version and query. If during tabled execution, one ever encounters a `path` call with both arguments free, the time complexity bound will be either $O(V \times E)$ or $O(V^3)$. If one encounters calls to `path` with both or one of the arguments bound, but bound to different values during the execution, then the time is $O(V \times E)$ or $O(V^3)$. If one only encounters calls to `path` with one of the arguments bound to the same value and the other argument free, then the time is $O(E)$ or $O(V^2)$. The criterion on obtaining the bounds in Figure 5.2 is the amount of data kept for each tabled predicate.

Static filtering and off-line partial evaluation. These are bottom-up procedures, and are not affected by the order of rules and hypotheses. Static filtering and partial evaluation work in essence as the specialization procedure described. Static filtering restricts, i.e. *filters*, the facts used during the evaluation using constants in the query. It is vulnerable to changes of the recursion type in the definition. For example, the method will be able to impose filters on the first argument for the rules in case the left-recursive version is used and the first argument is bound in the query, but will not be able to impose any filters on rules if such a query is asked to the right-recursive version. The doubly-recursive version is not *filterable*.

If static filtering yields linear time evaluation, it does so using less than all edges (except the time to read in all facts); more precisely speaking it only looks at edges reachable from c, which is bound by $R(c) \times o$. Symmetrically, using the right-recursive program with the second argument bound, the evaluation only considers edges that can reach b, which is bound by

$IR(c_2) \times\ i$.

Dynamic filtering. Dynamic filtering is a version of filtering where the filters are set according to the underlying database during the evaluation. It is not easy to analyze, because the complexity measure may drastically change from one data set to another. As a simplistic overview, we can say that for dense graphs, dynamic filtering behaves exactly the same as static filtering; in contrast, for sparse graphs the filters imposed may remain fairly strict and the evaluation may be better than static filtering, although even for sparse graphs, the filters may reduce to those imposed by static filtering.

5.3 Related work

Various methods for efficiently evaluating Datalog programs such as static filtering [37] and dynamic filtering [36] have been proposed. These methods use special data structures for evaluating Datalog programs rather than using traditional evaluation engines. For static filtering, the computational complexity of the evaluation can be analyzed easily from the rules. For dynamic filtering, however, the computational complexity depends on input data therefore cannot be determined statically.

Using static filtering for the evaluation a Datalog program can be shown to be the same as using partial evaluation combined with the program generation method described. Partial evaluation for logic programming [42] is a general framework for taking static inputs into account for general logic programs. The specialization method described in this chapter is a form of partial evaluation for Datalog programs.

Borrowing ideas from the theory of grammars for logic programming is natural since the evaluation of both involve similar components. We have incorporated one such idea [11] for our conversion between left-recursive and right-recursive programs. Grammar related ideas for Datalog programs can also be found in, e.g., [31]. Forms of recursion conversion have been discussed in other contexts as well. The conversion from doubly-recursive rules to rules with only one recursive hypothesis is a specific instance of linearization [78, 53].

Chapter 6

Applications

In this chapter, we show the application of our complexity analyses and optimizations on several important applications.

We first show two variants of Andersen's pointer analysis expressed as Datalog rules, and show complexity analyses on both; and show that subsumptive tabling is asymptotically faster than variant tabling for it. We also illustrate the subsumption optimization and its effectiveness on this example. The application of subsumption optimization results in a systematic derivation of Heintze and Tardieu's demand-driven pointer analysis algorithm [33].

Secondly, we show that context-free grammars can be expressed as Datalog rules, show that the complexity of tabled top-down evaluation as analyzed by our method is more precise than the manually derived and analyzed algorithm of Earley [26]. Using our analyses, we show the complexity of different subsets of context-free grammars.

Thirdly, we describe the analyses and optimizations for an example from ontology queries. We show that the complexity analyses help determine that subsumptive tabling will perform asymptotically better for the example, and the actual running times are aligned with that observation. We also use our complexity analyses to reorder hypotheses in the rules so that the performance is improved for variant tabling.

Finally, we describe the use of a powerful graph query language for querying programs, and show that the combination of transformations we have described so far are an effective method for efficient implementations of the queries. Our implementation method combines transformation to Datalog, recursion conversion, demand transformation, and specialization, and

finally generates efficient analysis programs with precise complexity guarantees. This combination improves an $O(VE)$ time complexity factor using previous methods to $O(E)$, where V and E are the numbers of graph vertices and edges, respectively.

6.1 Program pointer analysis

Andersen's pointer analysis is a flow-insensitive pointer analysis. We first show the analysis for C, analyze the complexity of evaluation strategies, and apply our optimization methods for obtaining efficient implementations of the analysis with complexity guarantees.

Given a C program, the C statements and their corresponding representation as Datalog facts are as follows.

- `p = &q` is represented by `bare_addr(p,q)`.
- `p = q` is represented by `bare_bare(p,q)`.
- `p = *q` is represented by `bare_star(p,q)`.
- `*p = q` is represented by `star_bare(p,q)`.

Then, Andersen's pointer analysis can be defined using Datalog rules as follows, where `pt(p,q)` denotes `p` points to `q`:

```
pt(p,q) :- bare_addr(p,q).                    (A1)
pt(p,q) :- bare_bare(p,r), pt(r,q).           (A2)
pt(p,q) :- bare_star(p,s), pt(s,r), pt(r,q). (A3)
pt(p,q) :- star_bare(r,s), pt(r,p), pt(s,q). (A4)
```

Complexity analysis and comparison of evaluation methods. We first discuss the complexity for variant and subsumptive tabling for a query where both arguments are free. For the query `pt(p,q)?`, subsumptive binding annotation obtains the query `pt_ff(p,q)?`, and the rules

```
pt_ff(p,q) :- bare_addr_ff(p,q).                          (1FF)
pt_ff(p,q) :- bare_bare_ff(p,r), pt_bf(r,q).              (2FF)
pt_ff(p,q) :- bare_star_ff(p,s), pt_bf(s,r), pt_bf(r,q). (3FF)
pt_ff(p,q) :- star_bare_ff(r,s), pt_bf(r,p), pt_bf(s,q). (4FF)
```

No annotated rules are generated from subqueries whose predicates are `pt_bf`, since a more general query, whose predicate is `pt_ff`, is guaranteed to be evaluated.

For variant tabling, the same query, the four rules above and the following rules are obtained:

```
pt_bf(p,q) :- bare_addr_bf(p,q).                          (1BF)
pt_bf(p,q) :- bare_bare_bf(p,r), pt_bf(r,q).              (2BF)
pt_bf(p,q) :- bare_star_bf(p,s), pt_bf(s,r), pt_bf(r,q). (3BF)
pt_bf(p,q) :- star_bare_ff(r,s), pt_bb(r,p), pt_bf(s,q). (4BF)

pt_bb(p,q) :- bare_addr_bb(p,q).                          (1BB)
pt_bb(p,q) :- bare_bare_bf(p,r), pt_bb(r,q).              (2BB)
pt_bb(p,q) :- bare_star_bf(p,s), pt_bf(s,r), pt_bb(r,q). (3BB)
pt_bb(p,q) :- star_bare_ff(r,s), pt_bb(r,p), pt_bb(s,q). (4BB)
```

Both for variant and subsumptive tabling, the complexity of the rules `(1FF)-(4FF)` are as follows:

- `(1FF)`: $O($`#bare_addr`$)$
- `(2FF)`: $O($`#bare_bare` $\times$ `#pt.2/1`$)$
- `(3FF)`: $O($`#bare_star` $\times$ `#pt.2/1` $\times$ `#pt.2/1`$)$
- `(4FF)`: $O($`#star_bare` $\times$ `#pt.2/1` $\times$ `#pt.2/1`$)$

For the rules generated only for variant tabling, we first show their local complexity as follows:

- `(1BF)`: $O($`#bare_addr.2/1`$)$
- `(2BF)`: $O($`#bare_bare.2/1` $\times$ `#pt.2/1`$)$
- `(3BF)`: $O($`#bare_star.2/1` $\times$ `#pt.2/1` $\times$ `#pt.2/1`$)$
- `(4BF)`: $O($`#star_bare` $\times$ `#pt.2/1`$)$
- `(1BF)`: $O(1)$
- `(2BF)`: $O($`#bare_bare.2/1`$)$
- `(3BF)`: $O($`#bare_star.2/1` $\times$ `#pt.2/1`$)$

- `(4BF)`: $O($`#star_bare`$)$

The number of invocations for these rules is more involved to analyze. Using our systematic analysis, we first identify the number of invocations to `path_bf`. There are many calls to `path_bf`, we show the lower bound for a few of these. For example, due to the second hypothesis of `(2FF)`, there are $O($`#bare_bare.2`$)$ invocations; due to the third hypothesis of `(3FF)`, there are $O($`#pt.2`$)$ invocations. The sum of all such invocations form a lower bound for the number of invocations to `path_bf`. Now, we identify a lower bound on the number of invocations to `path_bb`. For example, due to the third hypothesis of `(3BB)`, the number of invocations to `path_bb` is $O($`#pt.2`$\times$`#pt.2`$)$. The complexity of each rule in the above list is the product of the local complexity as shown and the number of invocations as identified.

We show that subsumptive tabling has better time and space complexity than variant tabling for this query. The time complexity of `(3FF)` and `(4FF)` dominate the time complexity for subsumptive tabling. Therefore, we need to show there exists a rule in the extra rules for variant tabling that incurs more time complexity than one of these rules. The complexity incurred by `(4FF)` is $O($`#star_bare`$\times($`#pt.2/1`$)^2)$. For `(4BB)`, the local complexity is $O($`#star_bare`$)$, and the number of invocations is at least $O(($`#pt.2`$)^2)$ as shown; this is asymptotically worse than $O(($`#pt.2/1`$)^2)$. Therefore, the time complexity of top-down evaluation with variant tabling is worse than top-down evaluation with subsumptive tabling for this example.

The space complexity of top-down evaluation with subsumptive tabling is $O($`#pt`$)$, since there is only one table for the given query, and all answers are stored in that table. However, variant tabling uses the mentioned space, plus creates furthermore tables, such as $O(($`#pt.2`$)^2)$ tables due to the last hypothesis of `(3BB)` as shown in the analysis for the number of invocations, which is asymptotically worse than the space used by subsumptive tabling. Therefore, the space complexity of top-down evaluation with variant tabling is worse than top-down evaluation with subsumptive tabling for this example.

Deriving Sridharan and Fink's complexity analyses systematically. Using our analyses and optimizations, we have already shown how to obtain Heintze and Tardieu's demand driven pointer analysis. We now show the application of precise complexity analyses. The complexity of Andersen's analysis has been obtained manually in previous work with regard to different constraints. Sridharan and Fink [64] give such analysis for Andersen's

analysis for Java. We show that the bounds manually obtained are looser than our bounds by systematically deriving their bounds from our analysis.

We represent relevant Java statements in Datalog as follows:

- `x = new T()` is represented by `create(x,o)`, where `o` is a unique identifier for the object created with this statement.
- `x = y` is represented by `assign(x,y)`.
- `x = y.f` is represented by `deref(x,y,f)`.
- `x.f = y` is represented by `ref(x,f,y)`.

Given this representation, Andersen's analysis for Java as defined by [64] can be written in Datalog as follows:

```
    pt(x,o) :- create(x,o).                          (R1)
    pt(x,t) :- assign(x,y), pt(y,t).                 (R2)
    pt(x,t) :- deref(x,y,f), pt(y,o), pt_f(o,f,t). (R3)
pt_f(o,f,t) :- ref(x,f,y), pt(x,o), pt(y,t).         (R4)
```

Any fact `pt(x,o)` that can be inferred by the above rules means that the variable `x` may point to object `o`, and `pt_f(x,f,y)` means that the field reference `x.f` may point to object `o`.

The alias analysis algorithm in [64] uses the following graph and definitions, and performs a dynamic transitive closure on the graph.

A graph G is defined whose nodes are variables and field references in the program, and whose edges are as follows:

- an edge from `x` to `o`, for each statement `create(x,o)`.
- an edge from `x` to `y`, for each statement `assign(x,y)`.
- an edge from `x` to `o.f` for each `o` that `y` points to, for each statement `deref(x,y,f)`.
- an edge from `o.f` to `y` for each `o` that `x` points to, for each statement `ref(x,f,y)`.

The meaning of an edge `x` $\rightarrow$ `y` in G is: if `y` is an object, then `x` may point to `y`, otherwise the objects that `x` may point to is a subset of the objects that `y` may point to.

The complexity of the algorithm in [64] is $O(N^2 \times \max_x D(x) + N \times E)$, where E is the number of edges, N is the number of variables and `new` statements, and $D(x)$ is the number of statements that dereference a variable x. We first express their algorithm in Datalog rules, and show that it is equivalent to our rules above. The edges are defined by rules `(S1)-(S4)`, and the `pt` relation is defined by the transitive closure of the edges (rules `(S5)-(S6)`).

```
edge_d(x,o,f) :- deref(x,y,f), pt(y,o). (S1)
edge_r(o,f,y) :- ref(x,f,y), pt(x,o).   (S2)
pt(x,y) :- create(x,y).                 (S3)
pt(x,y) :- assign(x,z), pt(z,y).        (S4)
pt(x,y) :- edge_d(x,z,f), pt_f(z,f,y).  (S5)
pt_f(x,f,y) :- edge_r(x,f,z), pt(z,y).  (S6)
```

It is trivial to see that unfolding `edge_d` and `edge_r`, we obtain an equivalent set of rules as `(R1)-(R4)`, therefore these sets of rules compute the same `pt` relation.

We show that even imprecise systematic complexity analysis can achieve the complexity in [64]. The complexity of `(S3)` is discarded since it has only one hypothesis, and is bound by the input or output size. The complexity of `(S1)` is $O(N^2 \times \max_y D(y))$, since `y` and `o` can take N values, and there are only $D($`y`$)$ `deref` statements per `y`. The complexity of `(S2)` is $O(N^2 \times \max_x D(x))$, since `x` and `o` can take N values, and there are only $D($`x`$)$ `ref` statements per `x`. The complexity of `(S5)` and `(S6)` is $O(E \times N)$, since there are E edges and `y` can take N values. Therefore, an upper bound on the time complexity of the rules is $O(N^2 \times \max_x D(x) + N \times E)$, which is the complexity of the algorithm given in [64]. Using this complexity, a subset of programs called k-sparse are defined with the following properties:

- $max_x D(x) < k$, for a constant k
- $E < kN$, for a constant k

Using the complexity formula above, and substituting the constraints, the complexity of the algorithm in [64] is $O(N^2)$ for k-sparse programs. On the other hand, we have a more precise complexity analysis for such programs. The time complexity of each rule using subsumptive tabling for the query where both arguments are free is as follows.

- `(S1)`: $O($`#deref`$\times$`#pt.2/1`$)$

- (S2): $O($#ref$\times$#pt.2/1$)$
- (S3): $O($#create$)$
- (S4): $O($#assign$\times$#pt.2/1$)$
- (S5): $O($#edge_d$\times$#pt.2/1$)$
- (S6): $O($#edge_r$\times$#pt.2/1$)$

The total complexity is the sum of the precise complexities for each rule, which is more precise than the complexity bound in [64].

6.2 Context-free grammar parsing

A context-free grammar is a grammar whose rules are of the form:

$$V \rightarrow w$$

where V is a single nonterminal symbol, and w is a sequence of terminal and non-terminal symbols.

Earley[26] proposed a top-down parsing algorithm for context-free grammars. The worst-case running time of the algorithm is $O(n^3)$, where n is the length of the input string. Earley showed that the algorithm runs in $O(n^2)$ for unambiguous grammars, and in $O(n)$ for LR(k) grammars for any constant k. We show a method for generating Datalog rules from a context-free grammar so that top-down evaluation with tabling achieves at least the same time complexity in general, and for the applicable classes of context-free grammars. We show that the complexity of the generated rules is more precise.

Given any grammar rule, $V \rightarrow a_1 a_2 ... a_n$, where each a_i is either a terminal or a nonterminal, we generate $n-2$ Datalog rules of the form:

V(k$_0$,k$_n$) :- a$_1$(k$_0$,k$_1$), a$_2$(k$_1$,k$_2$), ..., a$_n$(k$_{n-1}$,k$_n$).

If $n \geq 3$, we decompose each such Datalog rule to generate multiple rules, so that they contain two hypotheses each, by joining two leftmost hypotheses first in an intermediate rule, and the conclusion of this intermediate rule with third hypothesis in another intermediate rule, and so on. For example, for the above rule we obtain:

int$_1$(k$_0$,k$_2$) :- a$_1$(k$_0$,k$_1$), a$_2$(k$_1$,k$_2$).
int$_2$(k$_0$,k$_3$) :- int$_1$(k$_0$,k$_2$), a$_3$(k$_2$,k$_3$).
...

$\texttt{V(k}_0\texttt{,k}_n\texttt{) :- int}_{n-1}\texttt{(k}_0\texttt{,k}_{n-1}\texttt{), a}_n\texttt{(k}_{n-1}\texttt{,k}_n\texttt{).}$

The input string s of the form $s_1 ... s_n$ is converted to a set of facts of the form, $\texttt{s}_i\texttt{(}i\texttt{,}i+1\texttt{)}$ for each $1 \leq i \leq n$, and we generate a query $\texttt{R(1,}n+1\texttt{)?}$ where `R` is the root nonterminal of the grammar.

Complexity analysis. The complexity of either tabled top-down or perfect bottom-up evaluation of the generated rules is trivially $O(n^3)$, since each rule contains at most 3 variables in the body, and each variable can take values in the domain $[1, n+1]$. Now, we consider the precise complexity. We consider each rule is of one of the following forms:

```
p(i,j) :- q(i,j).           (R1)
p(i,j) :- q(i,k), r(k,j).   (R2)
```

We perform binding annotation to the rules with respect to the query. The only possible annotations are `bf` or `bb`, because (i) the first argument of the query is bound, (ii) the first argument of the first hypothesis in each rule is bound if the first argument of the conclusion is bound, (iii) the first argument of the second hypothesis in each rule is bound since it appears in the first hypothesis. Notice that if the second argument is bound, then it is always bound to $n+1$, because the second argument of the query is bound to $n+1$, and the only way that the second argument of a hypothesis is bound is propagation from the second argument of the conclusion. We specialize the hypotheses annotated with `bb` so that we remove the second argument. We get rules of either of the four forms:

```
p_bf(i,j) :- q_bf(i,j).            (R1')
p_b(i) :- q_b(i).                  (R1'')
p_bf(i,j) :- q_bf(i,k), r_bf(k,j). (R2')
p_b(i) :- q_bf(i,k), r_b(k).       (R2'')
```

From rules of this form, we can generate the precise complexities as we have shown before.

Unambiguous context-free grammars. An *unambiguous context-free grammar* is one for which there is no string s that has more than one parse tree. In [26], it is shown that the algorithm runs in $O(n^2)$ for unambiguous context-free grammars. We show the same holds for the generated rules.

The only rule form that can take more than $O(n^2)$ among the forms above is `(R2')`, since it has 3 variables. Note that every time a rule of

the form `(R2')` generates a fact of the form `p_bf(i,j)`, the generation is due to a substitution of the variables `i`,`j`,`k`, where `i` is substituted by the conclusion, and `j`,`k` by the hypotheses.

Lemma 6.2.1. *If the grammar is unambiguous, for each rule of the form* `(R2')`*, for each* `i`,`j` *pair, there can be at most one* `k`*.*

Proof. Suppose there is a rule r for which there is more than one `k` for an `i`,`j` pair for some input string s. This would imply that the nonterminal that corresponds to the conclusion of r has multiple parse trees. This would in turn imply that there exists a string s that has multiple parse trees, which can be trivially constructed using s. Hence, we arrive at a contradiction, and the lemma must be true. □

Due to the lemma above, there can be at most n^2 values for `i`,`j`,`k`, hence the complexity of the rules for unambiguous grammars is $O(n^2)$.

LR(k) grammars. An *LR(k) grammar* [38] is a grammar that intuitively corresponds to a grammar for which while looking at the nth character of the string, the grammar rule to use next can be determined by looking at until the $n + k$th character. Earley's algorithm parses LR(k) grammars in linear-time.

We do not have a mechanism for lookahead, so we only consider LR(0) grammars. Intuitively, an LR(0) grammar is a grammar for which the grammar rule to be applied while looking at the nth character can be determined by looking at it only. More technically speaking, LR(0) grammars have no shift-reduce or reduce-reduce conflict. Since there are no shift-reduce or reduce-reduce conflicts, there is one and only one fact for each subquery with a `bf` annotation in the rules of the form above. Therefore, rules of any form can take at most linear time. Hence, the complexity of the generated rules for LR(0) grammars is $O(n)$.

6.3 Ontology queries

Ontology queries are an important class of queries that are becoming increasingly important as more progress is made in the semantic web efforts. The OWL Web Ontology Language guide by the World Wide Web consortium[1] contains an example ontology for a wine portal[2]. This ontology is

[1] http://www.w3.org/TR/owl-guide/

[2] http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf

converted to Datalog rules and facts by the KAON2 tool from the Semantic Web organization[3]. Our experiments on the rules were conducted on a 3.0 GHz Intel Q9650 with 4 GB of memory, running SuSE Linux, and using XSB 3.2 and YAP 6.2.

The set of rules and facts resulting from the wine portal ontology contains 961 rules defining 225 predicates, 654 facts for 113 predicates. Although the number of facts is small, we will observe that bad asymptotic time and space complexity characteristics may make evaluation infeasible even for such a small set of data, depending on the rules and query.

First, we discuss our complexity analysis to determine if subsumptive tabling will outperform variant tabling. Since the number of rules is high, we do not show all of the results of our complexity analyses, but we provide an insight to the analyses via a few of the rules. One of the resulting rules is the following:

```
kaon2equal(x,z) :- kaon2hu(x),kaon2hu(y),kaon2hu(z),
                   kaon2equal(x,y),kaon2equal(y,z).
```

For a query that finds all wines from California (`californiawine(x)?`), our analysis determines that there exists a query for `kaon2equal` with both arguments free. Then, we get the following annotated rules for variant tabling:

```
kaon2equal_ff(x,z) :- kaon2hu_f(x),kaon2hu_f(y),kaon2hu_f(z),
                      kaon2equal_bb(x,y),kaon2equal_bb(y,z).
kaon2equal_bb(x,z) :- kaon2hu_b(x),kaon2hu_f(y),kaon2hu_b(z),
                      kaon2equal_bb(x,y),kaon2equal_bb(y,z).
```

and only the first of the above rules for subsumptive tabling. Although the time complexity of these two rules are the same in the worst case; the space used by variant and subsumptive tabling is different. For subsumptive tabling, since all the `bb` calls to `kaon2equal` are subsumed, no table is created for them, and the space complexity is $O(\texttt{\#kaon2equal})$. However, in variant tabling, all such calls will be evaluated, and therefore there will be table entries for each such call. Notice that the values for the bound arguments come from `kaon2hu` hypotheses without restriction, therefore the total space complexity is $O(\texttt{\#kaon2equal}+\texttt{\#kaon2hu}^2)$.

In a real implementation such as XSB or YAP, the time to create tables, i.e. allocate space, is constant but significant in contrast to many

[3] http://kaon2.semanticweb.org/

operations. We have already discussed in previous chapters that rules with the same complexity can be separated in running time by their space complexity. There are many predicates in the rules that are defined similar to `kaon2equal`, therefore the space complexity is significant in determining actual performance.

We performed the first set of experiments on the given set of rules in XSB and YAP, with a query that finds all wines from California as above. The experiment results were aligned with our complexity analyses. XSB and YAP both ran out of memory after several minutes of computation when variant tabling is used. On the other hand, XSB with subsumptive tabling returned all answers to the query in 12 seconds with local scheduling and 20 seconds with batched scheduling.

To improve performance of variant tabling, reordering hypotheses is the first optimization in mind. However, precise complexity comparison over all of the rules for different versions is not feasible due to the number of rules and predicates. Therefore, we apply the following three heuristics for reordering hypotheses in rules to improve the performance of variant tabling.

1. Minimize the number of free variables in a rule, preserve original order if multiple such alternatives exist.

2. Among the alternatives for minimal number of free variables, choose one that avoids calls to unary predicates with a variable argument if possible.

3. Among the alternatives for minimal number of free variables, choose one that avoids calls to binary predicates with both variable arguments if possible.

We performed experiments in XSB with local and batched scheduling, and in YAP for the resulting rules from the application of these heuristics, and all resulting rules terminate in contrast to the original set of rules. The results are in Figure 6.1.

Engine	Heuristics		
	1	2	3
XSB-local	268 s	329 s	273 s
XSB-batched	19 s	23 s	16 s
YAP	12 s	14 s	11 s

Figure 6.1: Running time of rules resulting from heuristics in XSB and YAP

Note that there is a dramatic difference between the different scheduling strategies in XSB, and this is an interesting topic for investigation but beyond the scope of this thesis.

After the heuristics are applied, using subsumptive tabling does not help, because there are no subqueries that are subsumed. We have observed that in such cases, subsumptive tabling is on average 8% slower than variant tabling.

To sum up, we showed that our analyses can be effectively used to determine when subsumptive tabling outperforms variant tabling. We also showed that even when complexity comparison is not feasible for different versions of the rules with the same semantics, heuristics educated by the understanding of analyses can be used for obtaining versions that have better running time and memory usage.

6.4 Graph queries

Graph queries can be used to express many problems from different areas, including program analysis in particular. Such queries can help find bugs [35], detect malicious virus patterns [19], report security violations [62], check temporal safety properties [6], etc. Efficient hand-written implementations for program analyses are difficult to develop, verify, and maintain, and query languages for specifying such analysis problems are desirable for ensuring the correctness of analyses while reducing the effort of implementations. However, higher-level query languages often lack efficient implementations or complexity guarantees. An automated approach to generating efficient implementations with complexity guarantees is needed for practical uses of such languages.

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

This combination of transformations improves an $O(VE)$ time complexity factor using previous methods to $O(E)$, where V and E are the numbers of graph vertices and edges, respectively. We show precise time complexity analysis for an example program analysis problem using this method. We also describe implementations of the example analysis problem and experiments for analyzing a set of programs of varying sizes, and confirm the calculated complexities. Additionally, we compare our results against XSB [61], a state-of-the-art top-down evaluation engine that employs tabling strategies, and bddbddb [41], a bottom-up Datalog evaluation engine that employs binary decision diagrams for storing and manipulating relations. For both systems, tight complexity guarantees are not available, and we demonstrate that they show different behavior with respect to different forms of rules without a method of predetermining the best form.

Even though the graph query language and most of the transformations to Datalog were proposed before [50], and similar languages have been used for program transformations [70], this work presents the first substantial use of the language to give precise complexity analyses for program analyses that are more complex than analyses possible without the power of the language [49]. We also show that our novel combination of transformations is necessary for generating efficient implementations that are both asymptotically better, and with better constants than possible before [50]. Finally, we conduct the first substantial experimental evaluation of different combinations of transformations that our method uses, and compare them with implementations of rules in XSB and bddbddb.

6.4.1 Graph query language

We use a graph query language that can specify the existence of paths with various properties proposed in [50]. For our examples, we use the language to query the control-flow graphs of programs. We consider control-flow graphs whose vertices correspond to program points, and labeled edges correspond to operations. We use edge labels that reflect only information relevant to the analysis of interest. Consider an assignment statement $a = 5$ in a program. If we are interested in analyzing reaching definitions, then this statement may be represented by the label `def(a)`, indicating a definition of (i.e., assignment to) a. One may use several abstractions to represent one statement, therefore multiple edges between two vertices are possible. For the statement $x = y + 1$, edges can correspond to the definition of x, and the usage of y. We denote the entry point of a program as `start`.

Many analyses can be performed on a control-flow graph. For example,

the use of an uninitialized variable in a program can be determined by finding a path starting from the entry point in the program such that a variable is never initialized but eventually used on the path.

The graph query language supports graph path expressions that allow convenient use of both vertices and edges of arbitrary kind as well as additional global and local parameters in graph paths. Figure 6.2 gives a grammar of the language.

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

Figure 6.2: Grammar for the graph query language.

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

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

Since we transform our queries to Datalog, we need to represent the

graph data as Datalog facts. For each edge label `p(a`$_1$`,...,a`$_k$`)` between vertices x and y, the corresponding Datalog fact is `p(x,y,a`$_1$`,...,a`$_k$`)`.

6.4.2 Example graph queries for program analysis

We show a variety of program analysis problems specified as graph queries, and illustrate the power of the language with queries that use different language features. The queries are shown in Figure 6.3 and explained below.

(i) Uninitialized variables. We use this example as our running example. An edge corresponding to the definition of a variable x is labeled `def(`x`)`, and an edge corresponding to the use of a variable x is labeled `use(`x`)`. The query shown in Figure 6.3 returns the set of pairs of program point w and variable x such that x is not defined or used before w, and used for the first time at w.

(ii) Hash values in a map. In Java, it is illegal to change the hash value of an object while it is in a HashMap [10]. We use `add/rem_map`(x,y) to denote adding/removing y to/from map x, and `change_hash`(x) to denote changing the hash value of x. The query shown in Figure 6.3 returns the set of program points w at which an object's hash is changed after it has been added to a HashMap and not removed subsequently.

(iii) Expensive loops. Concatenation to a string is an expensive computation if done repeatedly. We use `concat(`x`,`y`)` to represent the operation that concatenates y to x. The query shown in Figure 6.3 returns the set of pairs of program point w and variable x, such that a string is concatenated to x after program point w, and there is a loop containing the program point w.

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

(iv) Live branches. The semantics of MATLAB implies that an if-branch with a set s as the condition is taken if s is nonempty and all elements of s are positive numbers. Dead-code elimination for if-branches is possible if the branch can never be taken. The query shown in Figure 6.3 returns the set of program points w such that the if-branch at w is not removable

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

Figure 6.3: Example queries for program analysis.

by dead-code elimination. This is done by finding a path in the program such that all elements added to a set x are guaranteed to be positive. We use $\texttt{add}(x,y)$ to denote the addition of y to set x, and $\texttt{if}(x)$ to denote an if-branch with condition x.

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

Many more examples can be shown, but we do not show them here since they are not conceptually different. Such examples include the specification of malicious virus patterns [19], security violations in programs and operating systems [62, 5, 17], and temporal safety properties [6].

6.4.3 Generating efficient implementations

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

guarantees. We demonstrate the steps on our running example, the uninitialized variables query.

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

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

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

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

- For a constraint c, a fresh predicate name $\texttt{p}_c$ is used. $f(c) = \texttt{p}_c\texttt{(v}_1\texttt{,...,v}_n\texttt{)}$, where $\texttt{v}_1\texttt{,...,v}_n$ are the variables in c.

- For an edge path e,

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

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

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

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

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

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

– if e is of form $\neg e_1$, and $f(e_1) =$ `p1(v11,...,v1k1)`, then $f(e) =$ `p(v1,...,vk)`, where `p` is a fresh predicate name, and `v1`,...,`vk` is the subsequence of `v11`,...,`v1k1` that are variables and appear anywhere else in the query except e. In this case, the following rule is added to R:

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

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

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

– if e is of form e_1*, and $f(e_1) =$ `p1(v11,...,v1k1)`, then $f(e) =$ `p(vs,vt,v1,...,vk)`, where `p` is a fresh predicate name, and `v1`,...,`vk` is the subsequence of `v13`,...,`v1k1` that are variables and appear anywhere else in the query except e. In this case, the following two rules are added to R, where `vf` is a fresh variable:

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

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

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

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

$$\begin{aligned}\mathtt{p}(\mathtt{v}_s,\mathtt{v}_t,\mathtt{v}_1,...,\mathtt{v}_k) \texttt{ :- } & \mathtt{p}_1(\mathtt{v}_s,\mathtt{v}_{f2},\mathtt{v}_{13},...,\mathtt{v}_{1k_1}),\\ & \mathtt{p}_2(\mathtt{v}_{f2},\mathtt{v}_{f3},\mathtt{v}_{23},...,\mathtt{v}_{2k_2}),\\ & ...,\\ & \mathtt{p}_n(\mathtt{v}_{fn},\mathtt{v}_t,\mathtt{v}_{n3},...,\mathtt{v}_{nk_n}).\end{aligned}$$

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

$$\mathtt{p}(\mathtt{v}_1,...,\mathtt{v}_k) \texttt{ :- } \mathtt{p}_1(\mathtt{v}_{11},...,\mathtt{v}_{1k_1}).$$

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

$$\begin{aligned}\mathtt{p}(\mathtt{v}_1,...,\mathtt{v}_k) \texttt{ :- } & \mathtt{p}_1(\mathtt{ov}_1,\mathtt{ov}_2,\mathtt{v}_{13},...,\mathtt{v}_{1k_1}),\ ...,\\ & \mathtt{p}_n(\mathtt{ov}_n,\mathtt{ov}_{n+1},\mathtt{v}_{n3},...,\mathtt{v}_{nk_n}).\end{aligned}$$

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

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

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

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

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

The time complexity of computation using each rule is given in the left column of Figure 6.4. The bottleneck is the complexity for `(R5)`, $O(V \times$`#ndu`$)$; since `#ndu` is bounded by $O(E \times$`#any`$)$ based on `(R3)`, this complexity is $O(V \times E \times$`#any`$)$.

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

This step first performs recursion conversion to obtain both left- and right-recursive forms of recursive rules. This uses the transformations in

Chapter 5. For example, for `(R5)`, an alternative rule with the same semantics is:

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

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

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

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

1. For a recursively defined predicate p, if there is a hypothesis whose predicate is p and its first argument is a constant, then we only generate the left-recursive form for the recursive rule that defines p, and respectively if the second argument is a constant, then we only generate the right-recursive form. These forms are asymptotically better to use, since after demand transformation, the chosen recursive form will be asymptotically faster than the alternative form.

2. Among two permutations in each rule, if the predicate of one of the hypotheses h_1 is a predicate for which facts are given, and the predicate of the other hypothesis h_2 is a predicate defined by rules, then we always order the hypotheses so that h_1 is first and h_2 is second. This reduces the time complexity, since after demand transformation, the demand for h_2 will be stricter.

3. If the positive hypotheses of the original rule do not share any variables, then we use the given order. This is due to the fact that the join of these hypotheses costs the same in either direction when no variables are shared, so we can ignore the alternative order.

For the running example, there are two choices of recursion forms, and 16 hypothesis orders for each form. Thus, 32 different versions exist. Using heuristic 1 above, we obtain only the left-recursive form `(R5)`, not `(R5')` since the predicate of the first hypothesis of `(R6)` is `ndus` and its first argument is a constant (`start`). Using heuristic 2, we obtain only the reversed

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

Figure 6.4: Time complexities for the original rules and the rules after demand transformation.

hypothesis order for (R6), i.e., (R6'), since use is a predicate for which facts are given, and ndus is a predicate defined by rules. Using heuristic 3, we use the given orders for (R3) and (R4) since their positive hypotheses do not share any variables. Therefore, we are left with only one ordering for each rule.

Step 3: Demand transformation. This step performs, for each rule set obtained from Step 2, demand transformation as shown in the previous chapter.

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

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

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

The time complexity of the resulting rules is given in the right column of Figure 6.4. It is reduced asymptotically, including dropping an $O(V)$ factor from `(R5)`, and the reduction of all $O($`#any`$)$ factors to tighter factors. The bottleneck complexity is reduced to $O(E\times$`#dem2.2/1`$)$ from $O(V\times E\times$`#any`$)$.

Step 4: Specialization. This step applies specialization and deterministic unfolding to the result from Step 3, to remove unnecessary predicates, arguments, and rules. Specialization uses a simplified version of partial evaluation, as described in Chapter 5.

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

$$\theta(\texttt{c})\ \texttt{:-}\ f(\theta(\texttt{h}_1))\texttt{,}\ \ldots\texttt{,}\ f(\theta(\texttt{h}_n))\texttt{.} \qquad \texttt{(Rs)}$$

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

rule, because it guarantees improvements in both time and space. This unfolding scheme is called deterministic unfolding [42].

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

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

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

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

Step 5: Program generation. This step generates efficient implementations with specialized data structures for the set of rules from Step 4. This uses the method by Liu and Stoller [51]. It guarantees that the generated implementation has the analyzed complexity. For the running example, the generated program in Python is 171 lines, and the generated program in C++ is 3534 lines.

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

6.4.4 Demand transformation and graph queries

We discuss the properties of the rules generated by applying demand transformation to the rules generated from graph queries, and why it is an effective method for efficient implementation of graph queries.

In the following theorem, we show that for rules generated for graph queries, the rules obtained after demand transformation contain only stratified negation.

Theorem 6.4.1. *Let R and q be the set of rules and query obtained from a graph query as described, and let R' be the set of rules obtained after demand transformation of R with respect to q. Both R and R' contain only stratified negation.*

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

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

Note that specialization may be applied without applying demand transformation first. There are cases when demand transformation without specialization obtains better asymptotic complexity than specialization without demand transformation. However, if specialization alone provides the same complexity as demand transformation, then it is more preferable to obtain the set of rules from specialization since the rules become simpler

and smaller. In our running example, applying specialization directly to `(R5)` would yield `(R5d1s)` and `(R5d2s)`, but demand transformation and specialization applied in order also yields the same rules.

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

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

Theorem 6.4.2. *For any specialization method S, if S obtains a set of rules R_S that is simpler than the original set of rules R, and is in no-copy normal form, then there is a form of recursion and an ordering of hypotheses of R, say R', such that demand transformation, S, and unfolding applied in sequence to R' produces R_S.*

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

6.4.5 Effect of transformations on graph queries

We have implemented the method described in Python. As the final output of our method, the implementation emits both Python code, and Patton [59] code that is transformed automatically to C++ code by Patton, which is finally compiled by GCC.

We show the results of experiments using the running example on the control-flow graphs of six benchmark programs of varying size written in

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

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

The complexity of the set of rules obtained after the transformations is $O(E \times \texttt{\#dem2.2/1})$. A systematic manual analysis of the rules reveals that `#dem2.2/1` is bounded by the number of variables in scope at a program point, since the first argument of `#dem2` is a program point, and the second argument only takes the variables that can reach that program point via edges. We computed the average number of variables in scope (s) at each point using static analysis for each program, and the line with plus markers in Figure 6.5 shows the running times of set of rules with respect to $E \times s$. The resulting plot is almost linear as expected; we think that the deviations from linearity are due to the fact that the benchmarks do not exhibit worst-case time complexity.

Specialization after demand transformation reduces running time and memory usage by a constant factor, and the decision for when to stop unfolding affects the running time and memory usage. In Figures 6.5 and 6.6, *unfolding 1* denotes only unfolding predicates defined by one rule, and *unfolding 2* denotes unfolding predicates defined by possibly multiple rules with only one hypothesis.

Figure 6.5 shows the running time of the set of rules at different implementation stages. Unfolding 1 has the best running time, since it avoids duplicate inference for predicates defined by only one rule. On average, compared to the rules after demand transformation, specialization with unfolding 1 reduces running time by 17%.

Figure 6.6 shows the memory usage of the implementations at different implementation stages. We obtain the memory usage of generated Python implementations, since memory profiling for Python is very precise using

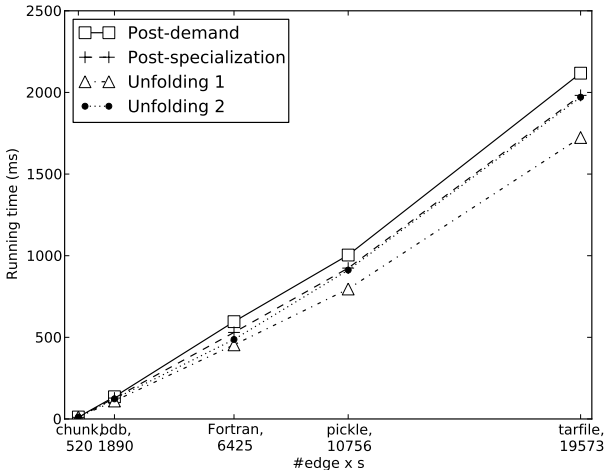

Figure 6.5: Running time of the implementation of rules in C++ at different implementation stages.

Heapy[4]. As expected, all steps show a constant decrease in memory usage, and unfolding 2 uses the least memory, since it removes the most rules. On average, compared to the rules after demand transformation, specialization with unfolding 2 reduces memory usage by 26%.

Comparison with state-of-the-art top-down and bottom-up systems. We have shown that recursion conversion and hypothesis permutation are important steps before demand transformation and specialization are applied for bottom-up computation. This is also true for top-down systems. A prominent top-down evaluation engine with tabling is XSB [61]. There is no known systematic analysis to find the best combination of form of recursion and hypothesis order for top-down evaluation in the literature, and we confirmed this by consulting the main developer of XSB [76].

We generated all recursion forms and hypothesis permutations for the running example, and manually ran and timed all benchmarks for all combinations in XSB. In general, the number of such combinations is exponential in program size. Among 32 possible combinations for our running example, 16 versions do not complete the benchmarks in less than 10 minutes, and all versions are slower than our generated code in C++. Note that our code generators are implemented for proof-of-concept, and they are not optimized

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

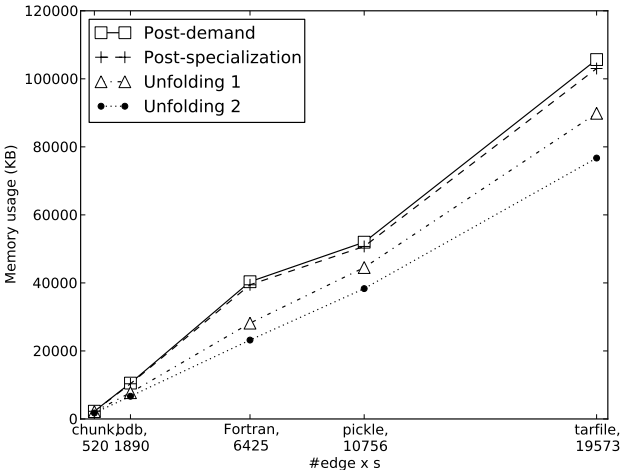

Figure 6.6: Memory usage of the implementation of rules in Python at different implementation stages.

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

Figure 6.7: Running time in milliseconds of implementations generated by our method, of the generated rules in XSB, and of the manually found best version of these rules in XSB, and similarly for bddbddb. - denotes incompletion in 10 minutes.

for constants in contrast to the effort put into the development of a mature system like XSB. Figure 6.7 shows the running time of our generated code, the running time of the rules and query generated in Step 1 of Section 4 in XSB, and the running time of the manually found best version for XSB.

A bottom-up evaluation engine that has been used to solve large problems is bddbddb [41]. bddbddb does not employ any transformations for efficiency, but employs binary decision diagrams (BDDs) to store and manipulate the relations. It does not provide any complexity guarantees for a

set of Datalog rules. We conducted experiments on bddbddb using both the original set of rules generated, and the rules yielded by our combination of transformations.

As expected, bddbddb shows asymptotically worse behavior on the original set of rules. On the final set of rules, we observed that the performance of bddbddb is highly dependent on the several options provided, especially the ordering of variables in the BDDs. We used the provided option of using machine learning to automatically find the best variable order, and also manually tried all 13 variable orderings possible. There is a large discrepancy between the results, using the worst variable ordering is up to 6 times slower, and using the automated variable ordering is up to 1.8 times slower than the manually obtained best result. The running times are shown in Figure 6.7.

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

6.4.6 Related work

The design and implementation of graph query languages for program analysis has been studied extensively. These include languages for both static analysis and runtime monitoring. The study of programs as relational data has been first proposed by Linton [48] with a language called QUEL. However, early query languages such as QUEL did not allow recursion and showed poor performance due to lack of optimization. Several other languages such as JQuery [74] and ASTLOG [21] have demonstrated better performance, but they lack support for specifying path properties in forms of regular expressions with parameters. We compare our work with several languages and implementations below.

Manual implementations. One of the most popular program analysis tools is FindBugs [35], which is used to find bugs in Java programs. FindBugs only supports the specification of bug patterns via manual implementations. However, the 355 different types of bugs that can be found by the tool are well-documented, and out of the 17 common bugs described, we can

express 16 of them in the language we use. The only one that cannot be expressed is for a bug that involves counting the number of occurrences of an edge; such aggregation operators are currently missing in the query language we use. Integrating the language we use and our method into a tool such as FindBugs would make it easier to add new analyses to the tool. Such analyses could then be clearly specified, and the efficient implementation can be automatically provided by our method.

Path queries. Regular path queries have been used in program analysis, e.g. in [23]. Parametric regular path queries [49] are regular-expression-like queries that allow the use of parameters, but do not support vertices and local parameters. Therefore, the language of parametric regular path queries is a strict subset of the query language proposed in [50] and used in this work. The language we use also strictly contains Condate [73]. The query language of Blast [9] is also a path query language for software verification, however it operates only on a particular kind of graph generated from the program, whereas the language we use can work with different graphs generated from the same program.

More powerful languages. PQL [40] is a more powerful program query language and is also transformed into Datalog rule. However, its implementation does not perform rule transformations as we do in this work, or provide complexity guarantees. The resulting Datalog rules from a PQL query are evaluated using bddbddb, a BDD-based implementation of Datalog. However, as shown in Section 6, transformations affect the running time of the resulting rules significantly, and the BDD-based implementation of Datalog does not provide any complexity bounds and shows irregular behavior.

PQL is more powerful in the sense that it allows arbitrary query declarations which are Datalog-like, rather than only graph expressions. It is less expressive in the sense that it does not allow arbitrary variables on vertices for return or reuse. Since Datalog rules are generated from PQL, our combination of transformations for Datalog can be used in conjunction to provide better complexity with precise complexity guarantees. This also applies to systems that use Datalog directly to query source code such as CodeQuest [32]. Additionally, since our implementation first transforms graph queries into Datalog, we can easily add support for Datalog in the graph query language too.

Chapter 7

Conclusion and future work

In this work, we focused on Datalog, an important rule-based language with the expressive power of polynomial-time algorithms. We have addressed the following aspects:

1. Precise time and space complexity analysis of the implementation of rules using different methods,
2. Source-level optimizations to improve time and space complexities of the implementation of queries,
3. Specifications of problems from different application areas as rules, and the complexity analysis of the implementation of rules.
4. Reduction of other high-level languages to rules for efficient implementation of those languages,

All of these aspects have been identified as research directions in previous work, however, not much progress had been made on the precise complexity analyses of the evaluation of Datalog queries, especially for evaluation methods that are known to be effective and used widely in practice. Inadequate complexity analyses often lead to ineffective optimization, since assessing the effectiveness of an optimization is much harder without a precise metric.

In this work, we first develop complexity analyses for important query evaluation methods. Then, utilizing these analyses, for the aspects above, we accomplish the following:

1. Precise time and space complexity analyses of the most effective methods for top-down evaluation, *variant* and *subsumptive tabling*, development of methods for bottom-up evaluation that have at least the same

complexity as these strategies, and the first demand-driven transformational method that beats the well known magic set transformation.

2. Use of specialization and rewriting of rules in different recursive forms for efficient implementation.

3. The precise complexity analyses of transitive closure, pointer analysis for C and Java, and context-free grammar parsing.

4. Reduction of powerful graph queries to Datalog and efficient implementation of them.

5. Implementation of all the methods and integration of the methods with XSB, a state-of-the-art top-down evaluation engine, when relevant.

Future work. The research in all four aspects is far from complete. In the short term, the following research problems are within grasp.

- The complexity analysis can be improved by approximation of the symbolic factors in the complexities, which will in turn improve the comparison of the complexities of different rules.

- There are many directions for the optimization of rules, such as heuristics for join-order optimization complemented by the complexity analysis, analysis for constant factor differences, optimizations for bottom-up evaluation to improve space complexities, etc.

- There are various high-level languages designed for the semantic web, database, security, and provenance communities, which can be transformed into rules, and effectiveness of optimizations for rules could be investigated.

In the long term, there are various directions to pursue, all of which are interesting and highly challenging. The following are a few of these directions.

- Extension of rules: The rule language that is analyzed and optimized can be extended with various features, such as aggregates, function symbols, and arbitrary negation. The analysis and subsequently optimization of a wider range of programs need to be studied.

- The design of a framework for expressing textbook algorithms as rules: This involves the expression of all standard textbook algorithms (greedy, dynamic programming, divide-and-conquer) as rules with automatically generated implementations guaranteeing matching (or even better!) complexities. Apart from the obvious benefits such as reducing implementation effort for algorithms, this can make teaching algorithms easier, by separating the logic of the algorithm from the implementation.

- Distributed computation: As networks grow larger and become ubiquitous, so does the necessity of algorithm specification in high-level languages for them. All of the current and proposed work can be extended to a rule language that supports distributed computation. This will pave the way for easier specification of algorithms over distributed networks. The complexity analyses can be extended for such specifications.

- Concurrent algorithms: The hardest algorithms to design are concurrent algorithms, and even harder is to systematically derive them. Even simplest concurrent algorithms are designed manually, and termination and complexity properties are mostly shown via complicated manually devised arguments,. A language for high-level specification of concurrent algorithms, and an implementation of the language with automated termination and complexity analysis is highly desirable, even for understanding the nature of the basic concurrent algorithms.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Foto N. Afrati, Manolis Gergatsoulis, and Francesca Toni. Linearisability on Datalog programs. *Theoretical Computer Science*, 308(1-3):199–226, 2003.

[3] Alfred V. Aho and Jeffrey D. Ullman. The universality of data retrieval languages. In *Conf. Rec. of the Sixth Annual ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[4] Lars Ole Andersen. Program analysis and specialization for the C programming language. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1994.

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

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

[7] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. of the 1986 ACM SIGMOD Intl. Conf. on Management of Data*, pages 16–52, 1986.

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

[9] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Blast query language for software verification. In *Proc. of the 11th Intl. Static Analysis Symp.*, pages 2–18, 2004.

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

[11] Derek R. Brough and Christopher J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9(2):115–134, 1991.

[12] Francois Bry. Query evaluation in deductive databases: Bottom-up and top-down reconciled. *Data Knowledge Engineering*, 5:289–312, 1990.

[13] Stefano Ceri, Georg Gottlob, and Luigi Lavazza. Translation and optimization of logic queries: The algebraic approach. In *Proc. of the 12th Intl. Conf. on Very Large Data Bases*, pages 395–402, 1986.

[14] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowledge and Data Engineering*, 1(1):146–166, 1989.

[15] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer, 1990.

[16] Stefano Ceri and Letizia Tanca. Optimization of systems of algebraic equations for evaluating datalog queries. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *Proc. of the 13th Intl. Conf. on Very Large Data Bases*, pages 31–41. Morgan Kaufmann, 1987.

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

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

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

[20] Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs (preliminary report). In *Proc. of the 20th Annual ACM Symp. on Theory of Computing*, pages 477–490, 1988.

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

[22] Anderson Faustino da Silva and Vítor Santos Costa. The design of the YAP compiler: An optimizing compiler for logic programming languages. *J. of Universal Computer Science*, 12(7):764–787, 2006.

[23] Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1-2):15–35, 2003.

[24] Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type inference for datalog and its application to query optimisation. In *Proc. of the 27th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 291–300, 2008.

[25] John DeTreville. Binder, a logic-based security language. In *Proc. of the 2002 IEEE Symp. on Security and Privacy*, pages 105–113, 2002.

[26] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[27] Juliana Freire, Terrance Swift, and David S. Warren. Beyond depth-first strategies: Improving tabled logic programs through alternative scheduling. *J. of Functional and Logic Programming*, 1998(3), 1998.

[28] Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proc. of the 1st Intl. Joint Conf. on Automated Reasoning*, pages 514–528, 2001.

[29] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[30] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proc. of the Fifth Intl. Conf. and Symp. on Logic Programming*, pages 1070–1080, 1988.

[31] Sergio Greco, Domenico Saccà, and Carlo Zaniolo. Grammars and automata to optimize chain logic queries. *Int. J. Foundations of Computer Science*, 10(3):349–, 1999.

[32] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *CodeQuest:* Scalable source code queries with Datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, pages 2–27, 2006.

[33] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 24–34, 2001.

[34] Lawrence J. Henschen and Shamim A. Naqvi. On compiling queries in recursive first-order databases. *J. ACM*, 31(1):47–85, 1984.

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

[36] Michael Kifer and Eliezer L. Lozinskii. A framework for an efficient implementation of deductive database systems. In *Proc. of the Advanced Database Symp.*, 1986.

[37] Michael Kifer and Eliezer L. Lozinskii. On compile-time query optimization in deductive databases by means of static filtering. *ACM Trans. Database Systems*, 15(3):385–426, 1990.

[38] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[39] Phokion G. Kolaitis and Christos H. Papadimitriou. Why not negation by fixpoint? *J. Computer and System Sciences*, 43(1):125–144, 1991.

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

[41] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proc. of the 24th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–12, 2005.

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

[43] Alon Y. Levy and Yehoshua Sagiv. Semantic query optimization in Datalog programs. In *Proc. of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 163–173, 1995.

[44] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *5th Intl. Symp. on Practical Aspects of Declarative Languages*, pages 58–73, 2003.

[45] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. OpenRuleBench: An analysis of the performance of rule engines. In *Proc. of the 18th Intl. Conf. on World Wide Web*, pages 601–610, 2009.

[46] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. OpenRuleBench: Detailed report. Technical report, Department of Computer Science, Stony Brook University, 2009. Available at http://semwebcentral.org/docman/view.php/158/69/report.pdf.

[47] Senlin Liang and Michael Kifer. Deriving predicate statistics in Datalog. In *Proc. of the 12th Intl. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 2010.

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

[49] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation*, pages 219–230, 2004.

[50] Yanhong A. Liu and Scott D. Stoller. Querying complex graphs. In *Proc. of the 7th Int. Symp. on Practical Aspects of Declarative Languages*, pages 199–214, 2006.

[51] Yanhong A. Liu and Scott D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Trans. on Programming Languages and Systems*, 29(1), 2009.

[52] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[53] R. Ramakrishnan, Y. Sagiv, J. D. Ullman, and Vardi. Proof-tree transformation theorems and their applications. In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 172–181, 1989.

[54] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proc. of the Fifth Intl. Conf. and Symp. on Logic Programming*, pages 140–159, 1988.

[55] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Y. Vardi. Logical query optimization by proof-tree transformation. *J. of Computer and System Sciences*, 47(1):222 – 248, 1993.

[56] Raghu Ramakrishnan and S. Sudarshan. Top-down versus bottom-up revisited. In *Proc. of the 1991 Intl. Symp. on Logic Programming*, pages 321–336, 1991.

[57] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *J. Logic Programming*, 23(2):125–149, 1995.

[58] Prasad Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. A thread in time saves tabling time. In *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, pages 112–126, 1996.

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

[60] Yehoshua Sagiv. Optimizing datalog programs. In *Foundations of Deductive Databases and Logic Programming.*, pages 659–698. Morgan Kaufmann, 1988.

[61] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as a deductive database. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, page 512, 1994.

[62] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, and Jacob West. Model checking an entire Linux distribution for security violations. In *Proc. of the 21st Annual Computer Security Applications Conf.*, pages 13–22, 2005.

[63] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, pages 1033–1044, 2007.

[64] Manu Sridharan and Stephen J. Fink. The complexity of andersen's analysis in practice. In *16th Intl. Symp. on Static Analysis*, pages 205–221, 2009.

[65] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, August 1997.

[66] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In *Proc. of the 3rd Intl. Conf. on Logic Programming*, pages 84–98, 1986.

[67] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Systems*, 10(3):289–321, 1985.

[68] Jeffrey D. Ullman. Bottom-up beats top-down for Datalog. In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 140–149, 1989.

[69] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proc. of the 14th Annual ACM Symp. on Theory of Computing*, pages 137–146, 1982.

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

[71] Laurent Vieille. A database-complete proof procedure based on SLD-resolution. In *Proc. of the 4th International Conference on Logic Programming*, pages 74–103, 1987.

[72] Laurent Vieille. From QSQ towards QoSaQ: Global optimization of recursive queries. In *Proc. from the 2nd Intl. Conf. on Expert Database Systems*, pages 743–778, 1988.

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

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

[75] David S. Warren. Programming in tabled prolog. Early draft available at http://www.cs.sunysb.edu/ warren/xsbbook/, 1999.

[76] David S. Warren. Personal communication, 2009.

[77] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Third Asian Symp. on Programming Languages and Systems*, pages 97–118, 2005.

[78] Weining Zhang, Clement T. Yu, and Daniel Troy. Necessary and sufficient conditions to linearize double recursive programs in logic databases. *ACM Trans. on Database Systems*, 15(3):459–482, 1990.