HYPOTHETICAL REASONING IN DEDUCTIVE DATABASES

BY ANTHONY J. BONNER

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

L. Thorne McCarty and Tomasz Imielinski

and approved by

____

____

____

____

New Brunswick, New Jersey

October, 1991

ABSTRACT OF THE DISSERTATION

Hypothetical Reasoning in Deductive Databases

by Anthony J. Bonner, Ph.D.

Dissertation Director: L. Thorne McCarty and Tomasz Imielinski

This dissertation addresses a limitation of most deductive database systems: They cannot reason hypothetically. Although they reason effectively about the world as it is, they are poor at tasks such as planning and design, where one must infer the consequences of hypothetical actions and possibilities. For instance, with a typical database query language, a user can retrieve those students "who are eligible to graduate," but cannot retrieve those students "who *would* be eligible *if* they took one more course." To express such queries, the dissertation develops a logic programming language in which a user can create hypotheses and draw inferences from them. In addition, we show that the language has several important properties. First, it is more expressive than any database query language based on classical logic, since it can express some simple hypothetical queries that classical logic cannot. Second, it can describe large rulebases concisely, because as we show, hypothetical operations allow a user to specify new rulebases by reusing and modifying old ones. Finally, by imposing syntactic restrictions, the language expresses *exactly* the database queries in many well-known complexity classes, including polynomial space, exponential time, and the polynomial time hierarchy.

Acknowledgements

I would like to thank my two research advisors, Thorne McCarty and Tomasz Imielinski, for years of discussion, guidance and availability and for intense and intellectually-stimulating periods of research. They provided as much time as a graduate student could want, sometimes hours a day for weeks on end. They encouraged me to develop and publish my early ideas and provided prompt and thoughtful feedback to the many write-ups that I gave them. I learned a great deal from them both.

Thorne's work on clausal intuitionistic logic deserves special mention since this dissertation springs directly from my early attempts to understand that logic. Thorne also frequently challenged me to produce new results, and this contributed greatly to my productivity.

Tomasz provided a database perspective as well as many good seminars on logic programming and modal logic, which were crucial to much of my research. Tomasz also frequently challenged me to justify my work on more than purely technical grounds. The ideas of rulebase independence and predicate substitution owe a great deal to this challenge and to our many hours of discussion. He also provided some much needed financial and political support at crucial moments. He is also a great skiing partner.

Eric Allender provided an excellent course on computational complexity. Material learned in that course pervades much of this dissertation. As a member of my thesis committee, Eric met with me personally, answered all my email messages, and provided many helpful comments on my papers and on an early draft of this dissertation, comments which greatly improved the final version.

My external committee member, Ashok Chandra, provided valuable comments and contributed a great deal to my thesis defense. In addition, his work on alternating Turing machines and on database query languages was both a source of useful results and a model for much of my own research.

I would like to thank each member of my committee for taking the time to read and comment upon this dissertation.

My fellow graduate students at Rutgers contributed to a stimulating intellectual environment. This especially includes Jan Chomicki, Kumar Vadaparty, Ron van der Meyden, Mukesh Dalal, Armand Prieditis, Prem Devanbu, and Sridhar Mahadevan. Besides commenting on my papers, they challenged me to explain and justify my results.

I would also like to thank the more senior people in my field who have taken the time to study my work and comment on it. This includes Alex Borgida, Victor Vianu, Dale Miller, Serge Abiteboul, Alberto Mendelzon, Dirk Van Gucht, Larry Saxton, Ed Robertson and an anonymous reviewer from the journal of *Theoretical Computer Science*.

Finally, I would like to thank my parents for their confidence and financial support at crucial moments.

This work was supported, in part, by a post-graduate scholarship of the Natural Sciences and Engineering Research Council of Canada (NSERC) and by a Marion Johnson Award, issued by the Graduate School, Rutgers University. The Rutgers Department of Computer Science provided crucial support through teaching assistantships, summer stipends, grants for conference travel, and computing facilities.

Dedication

To my parents

Table of Contents

Chapter 1

Introduction

Researchers from several areas have recognized the need for computer systems that reason hypothetically. Decision support systems (DSS) are a good example, especially in domains like financial planning where many “what if” scenarios must be considered [31, 42]. A typical example is an analyst who must predict a company’s deficit for the upcoming year assuming that employee salaries are increased by a given percentage. Or he might want a table of deficit predictions for a number of hypothetical salary increases [50]. Similar problems occur in computer aided design (CAD). Here, one must evaluate the effect on the overall design of local design alternatives and of various external factors [23, 43]. For example, an engineer may need to know how much the price of an automobile would increase if supplier X raised his prices by Y percent [23]. The number of hypothetical scenarios multiplies quickly when several factors are varied simultaneously, such as prices, interest rates, tax rates, etc. One may also need to consider variations in more complex factors, such as government regulations, company policy, tax laws, etc.

The database community has begun to address these needs by developing systems that integrate query processing with hypothetical updates. Such systems allow a user to pose queries not only to the real database, but also to hypothetical databases. Hypothetical databases are derived from a real database by a series of hypothetical assumptions, or updates. Early work in this area was done by Stonebraker, who showed that hypothetical databases can be efficiently implemented by slight extensions to conventional database mechanisms [46, 45]. He also pointed out that hypothetical databases are useful for debugging purposes, for test data, and for a variety of simulations, and he argued that “there are advantages to making hypothetical databases central to the operation of a database management system" [45].

The logic-programming community has taken these ideas one step further, integrating hypothetical updates not just with query processing, but with logical inference as well. Since the premise of a logical rule is just a query, several researchers have developed *hypothetical rules*, in which the premise can query not just a real database, but hypothetical databases as well. Vieille, et al, for instance, have developed a deductive database along these lines [50], and Warren and Manchanda have used hypothetical rules to reason about database updates [51, 34]. In addition, Miller has shown that hypothetical rules can structure the runtime environment of logic programs, resulting in programs that are more elegant, more efficient, and easier to maintain [40].

Hypothetical rules have also been used to extend the capabilities of expert systems. Gabbay, for instance, has reported a need to augment Prolog with hypothetical rules in order to encode the British Nationality Act, because the act contains rules such as, "You are eligible for citizenship if your father *would* be eligible *if* he were still alive" [25, 24]. And McCarty, also motivated by legal applications, has developed a wide class of hypothetical rules for computer-based consultation systems, especially systems for reasoning about contract law and corporate tax law [37, 39].

Some theoretical work on hypothetical inference has also been carried out, largely by the logic-programming community. Most of this work focuses on hypothetical insertion, largely because these updates fit neatly into a well-known logical system, *intuitionistic logic* [22]. Most of this work has been semantic, first showing that hypothetical insertion is indeed intuitionistic, and then casting the semantics in terms of a least fixpoint theory, in the logic-programming tradition. Gabbay, for example, showed that in the propositional case, hypothetical insertion is intuitionistic [24]. Working independently, McCarty and Miller both extended this result to the predicate case and developed a fixpoint semantics [37, 40]. Earlier, Statman showed that intuitionistic logic is PSPACE-complete in the propositional case [44].

Thus, the semantics of hypothetical insertion is well-established, and an initial result on its complexity is also known. Furthermore, as described above, prototype systems have been built, both for hypothetical insertion [41] and for more general hypothetical updates [50], and implementation details have been work out for commercially-viable, data-intensive systems [46, 45]. What is lacking, however, is a comprehensive theory of hypothetical inference, one that goes beyond insertions and beyond semantics.

This dissertation provides that theory by developing a logic programming language that includes hypothetical insertion and deletion as well as a new hypothetical operator, and by establishing the main properties of this logic. The development takes place in stages: (*i*) developing the proof theory and semantics of the logic, (*ii*) establishing its data complexity, (*iii*) augmenting the logic with negation-as-failure, and (*iv*) characterizing the expressive power of the resulting logic. Furthermore, as part of this development, the dissertation introduces two new ideas: *predicate substitution* and *rulebase independence*.

The first idea, predicate substitution, is a new type of hypothetical update. Unlike insertion and deletion, which deal with individual tuples, predicate substitution deals with entire relations, substituting one relation for another. We show that predicate substitution has capabilities not enjoyed by insertion or deletion. The most important of these is allowing a user to specify new rulebases by reusing and modifying old ones. For example, a rulebase that sorts people by age can be reused, with slight modifications, to sort people by weight.

The second idea, rulebase independence, is a condition that we place on queries to a rulebased system. Intuitively, a user should be able to formulate queries without having a detailed knowledge of the rulebase. For example, if the rulebase encodes the British Nationality Act, then a user should be able to formulate queries, such as "Am I eligible for citizenship?", even if his understanding of the Act is rudimentary. Similar requirements exist in database systems, since a user must be able to formulate database queries without knowing the contents of the database.

Using the idea of rulebase independence, we show that classical logic is inadequate for hypothetical reasoning. In particular, there are some simple hypothetical queries that cannot be expressed in classical logic in a rulebase independent way. Thus, if hypothetical queries are to be expressed declaratively, then new logical mechanisms are needed.

This dissertation develops those mechanisms, in the form of a new logic programming language. This language combines logical inference with hypothetical database updates. The basic building-blocks of the language are hypothetical queries, which allow a user to pose queries not only to the real database, but to hypothetical databases as well. A hypothetical query updates the database, evaluates a query on the new database, and then returns the database to its original state. For example, to evaluate the query, "Can Tony graduate by taking History 250," the system would insert into the database the fact that Tony has taken History 250, then ask whether Tony can graduate, and finally remove the inserted fact.

Although this process might seem procedural, we show how to formulate it as a declarative language with a logical semantics. An advantage of the declarative approach is that the database is automatically restored to its original state, relieving the user of this task. For complex hypothetical queries, this can significantly reduce the user's programming burden as well as ensure that the database is reliably restored. In addition, because the queries are declarative, they can be used in the premises of rules. In this way, we extend the language of hypothetical queries to a logic for building rulebases.

To simplify the presentation, we develop several versions of the logic of increasing complexity. Successive versions are characterized by different classes of hypothetical updates: first insertions, then insertions and deletions, and finally, predicate substitution. For each version, we develop a semantics and determine its data complexity. For the simpler versions of the logic, we establish additional results: For insertions and deletions, we obtain strong results on expressibility; and for insertions alone, we identify syntactic restrictions that reduce the complexity of the logic.

We begin the development with hypothetical insertion, extending the work on intuitionistic logic programming described above [25, 37, 40]. After presenting a new and simpler proof that the logic is intuitionistic, we establish its data complexity, showing it to be PSPACE complete. We continue the development by adding negation-as-failure to the logic. Negation-as-failure is an essential addition to any practical, logic programming language, but until now its incorporation into intuitionistic logic has been considered problematic [24]. We resolve this problem, adding negation-as-failure to the logic in what we feel is a natural way. We then study the impact of this feature on the complexity and expressibility of the logic, showing that it has no effect on complexity, which remains PSPACE complete, but a dramatic effect on expressibility. For instance, without negation, there are many database queries that the logic cannot express, even some very simple ones; but with negation, the logic is expressively complete, expressing *every* database query computable in polynomial space.

Continuing the development, we identify syntactic restrictions that reduce the logic's complexity. The main restriction is *linear recursion.* Informally, a rule is linear iff recursion occurs through only one premise. Linear rules have played an important role in deductive databases, because "(i) there is a belief that most 'real life' recursive rules are linear, and (ii) algorithms have been developed to handle them efficiently" [6]. We show that linearity reduces the complexity of hypothetical insertion from PSPACE to NP. We then show that negation-as-failure has a subtle effect on the linearized logic, increasing both its complexity and its expressibility. With negation, the logic expresses *exactly* the queries in PHIER, the polynomial time hierarchy. With additional syntactic restrictions, we refine the logic into a hierarchy of sublogics that corresponds exactly to the levels of polynomial time hierarchy, from NP on up.

Having developed the logic of hypothetical insertion, we extend it to include hypothetical deletion. With deletion, a user can express queries such as, "Could I have graduated if I had *not* taken History 250?" Little research has been done on combining logical inference with hypothetical deletion, two exceptions being the work of Vieille, et al, [50] and Warren and Manchanda [51, 34]. As in their work, the form of deletion considered here is relatively simple in that it deletes base facts, not inferred facts. Nevertheless, we show that it increases both the complexity and expressibility of the logic, from PSPACE to EXPTIME. That is, without negation-as-failure, the logic is *data* complete for EXPTIME, and with negation, it is *expressively* complete for EXPTIME.

Having developed both insertion and deletion, we go on to develop a new hypothetical operator, *predicate substitution.* Unlike insertion and deletion, which deal with individual tuples, predicate substitution deals with entire relations, substituting one relation for another. We show that predicate substitution has capabilities not enjoyed by insertion or deletion. The most important of these is allowing a user to specify new rulebases by reusing and modifying old ones. For example, a rulebase that sorts people by age can be reused, with slight modifications, to sort people by weight.

The ability to reuse and modify has important applications. In general this ability is needed when new rulebased applications differ only slightly from existing ones, or when an application is to be incrementally updated. Such techniques are particularly important when reprogramming is expensive or unreliable. These issues have not yet received significant attention in the deductive database literature. This dissertation attempts to address them.

Because of its ability to reuse and modify, predicate substitution provides a concise notation for describing large sets of rules. From the database perspective, it is much more concise than familiar query languages such as relational algebra or Datalog (i.e., function-free Horn logic). In fact, we show that a *finite* rulebase in this language can describe an *infinite* rulebase in Datalog. We develop a novel semantics reflecting this capability.

Chapter 2

Hypothetical Insertions

Much of the work on hypothetical inference has focused on a particular class of hypothetical updates: The insertion of single tuples into the database [25, 24, 37, 38, 40]. The main reason for this is that hypothetical insertion fits neatly into a well-known logical system: *intuitionistic logic* [22]. In intuitionistic logic, hypothetical queries are represented in a simple and direct way, as implications. For example, as a query, the expression $B \leftarrow C$ means, "Would B be true if C were inserted into the database?" The query, "Could I graduate if I took History 250?" is of this form. Once queries like this are allowed, it is natural to use them in the premises of rules. Thus, researchers in hypothetical inference have studied rules of the form $A \leftarrow (B \leftarrow C)$, which intuitively means, "A is true if adding C to the database would make B true." The following rule, taken from the British Nationality Act, is of this form: "You are eligible for British citizenship if your father *would* be eligible *if* he were alive" [25]. Rules like this are called *embedded implications* [37, 38].

Embedded implications are not classical and do not obey the rules of classical logic. Instead, they obey the rules of intuitionistic logic. This result was first established for the propositional case by Gabbay [24], and was extended to the predicate case by McCarty and Miller [37, 38, 40]. Working independently, McCarty and Miller both developed intuitionistic fixpoint semantics for embedded implications. McCarty also showed that embedded implications have interesting semantic properties analogous to the least minimal-model property of Horn logic [37], and he developed a practical, Prolog-style inference system for them based on unification [38].

This chapter[1] continues the research on embedded implications in two ways. First,

---

[1]Material in this chapter originally appeared in [7] and [13].

it introduces a new notion of expressibility, in which queries are required to be *rulebase independent*. Rulebase independence means that a user can formulate a query without having a detailed knowledge of the rulebase. For instance, if the rulebase encodes the British Nationality Act, then a user should be able to formulate queries, such as "Am I eligible for citizenship?", even if he does not completely understand the Act. Similar constraints exist in relational database systems, since a user must be able to formulate database queries without knowing the contents of the database.

Using the idea of rulebase independence, this chapter shows that classical logic is inadequate for hypothetical inference. In particular, there are some simple hypothetical queries that cannot be expressed in classical logic in a rulebase independent way. Thus, if hypothetical queries are to be expressed declaratively, then new logical mechanisms are needed. Intuitionistic logic is just such a mechanism.

This chapter also presents a new and simplified proof that embedded-implications are intuitionistic. Instead of developing a fixpoint semantics as in [37, 40], we use the techniques of modal logic to construct an intuitionistic structure that we call the *canonical model*. This structure provides a straightforward link between inference and semantics and leads to a succinct proof of the intuitionistic completeness of embedded implications.

2.1 Examples

A deductive database consists of a database of *facts*, and a set of logical *rules*. To answer queries, the rules are applied to the facts to infer more facts. Hypothetical systems extend this framework, allowing a user to ask about what *could* be inferred *if* the database were changed. This section gives examples in which the database is changed by hypothetically inserting facts into it.

The examples are based on a deductive database for a university. In this system, the database contains information about what courses the students have taken, and the rulebase infers which students are eligible to graduate. Facts are represented as ground atomic formulas. For instance, the formula $take(s, c)$ means that student s has taken course c, and the formula $grad(s)$ means that student s is eligible to graduate. The database thus contains facts such as $take(tony, cs250)$, and the rulebase contains rules such as

$$grad(s) \leftarrow take(s, his101), take(s, eng201).$$

This rule means that a student s can graduate by taking History 101 and English 201.

In the examples below, R denotes a rulebase, DB denotes a database, and $R + DB$ denotes the union (that is, the logical conjunction) of R and DB. Given a query ϕ, the expression $R + DB \vdash \phi$ means that ϕ can be inferred from the rules in R and the facts in DB. For example, the expression $R + DB \vdash grad(s)$ means that we can infer that student s is eligible to graduate.

2.1.1 Hypothetical Queries

In the following examples, each hypothetical query is described in three ways: (i) informally in English, (ii) formally at the meta-level, and (iii) formally, at the object-level.[2] In each example, a hypothetical query posed to the database DB is converted to an atomic query posed to another database, obtained by inserting tuples into DB.

Example 2.1 Consider the query, "If Tony took $cs452$, would he be eligible to graduate?" In other words, if $take(tony, cs452)$ were added to the database, could we infer $grad(tony)$? This query can be formalized at the meta-level as follows:

$$R + DB + take(tony, cs452) \ \vdash \ grad(tony) \tag{2.1}$$

At the object level, the implication $grad(tony) \leftarrow take(tony, cs452)$ represents this query. That is, $R + DB \ \vdash \ grad(tony) \leftarrow take(tony, cs452)$ iff condition (2.1) is satisfied. □

Example 2.2 "Retrieve those students who could graduate if they took (at most) one more course." At the meta-level, we want those students s such that for some course c,

[2] See [32] for a description of meta-level and object-level reasoning.

$$R + DB + take(s,c) \;\vdash\; grad(s)$$

At the object-level, the expression $\psi(s) \;=\; \exists_c\,[grad(s) \leftarrow take(s,c)]$ represents this query. That is, for each value of s, $R, DB \vdash \psi(s)$ iff the meta-level condition is satisfied. □

2.1.2 Hypothetical Rules

Having introduced hypothetical queries, we can use them in the premises of rules. Such rules have the form $A \leftarrow (B \leftarrow C)$, which intuitively means, "A can be inferred if inserting C into the database allows B to be inferred." These rules are called *embedded implications* [37, 38], and they turn the query language into a logic for building rulebases.

Example 2.3 Suppose the university has the following policy:

> "A student qualifies for a degree in *math and physics* if he is within one course of a degree in *math* and within one course of a degree in *physics*."

This policy is represented by the following two rules:

$$\begin{aligned} grad1(s,d) \;&\leftarrow\; \exists_c\,[grad(s,d) \leftarrow take(s,c)]. \\ grad(s, math\&phy) \;&\leftarrow\; grad1(s,math), grad1(s,phy). \end{aligned}$$

where the predicate $grad(s,d)$ means that student s is eligible for a degree in discipline d, and $grad1(s,d)$ means that s is within one course of a degree in discipline d. Note that the premise of the first rule is a hypothetical query similar to the one in example 2.2. If a rulebase R contains this rule, then for any student s, and any discipline d,

$$\begin{aligned} & R + DB \;\vdash\; grad1(s,d) \\ \textit{if}\quad & R + DB \;\vdash\; \exists_c\,[grad(s,d) \leftarrow take(s,c)] && (2.2) \\ \textit{if}\quad & \exists_c\,[R + DB \;\vdash\; grad(s,d) \leftarrow take(s,c)] && (2.3) \\ \textit{if}\quad & \exists_c\,[R + DB + take(s,c) \;\vdash\; grad(s,d)] \end{aligned}$$

□

In Example 2.3, notice that in going from line (2.2) to line (2.3), we have moved the existential quantifier outside the implication turnstile, $\vdash$. This operation is consistent with our intended interpretation of hypothetical queries, as given in example 2.2. Furthermore, this operation, which is not valid in classical logic, is the source of the intuitionistic semantics of hypothetical inference.

The next example shows that hypothetical rules can be cascaded, so that to answer one hypothetical query, we must first answer other hypothetical queries over other databases.

Example 2.4 We define the predicates $grad1(s)$, $grad2(s)$ and $grad3(s)$, which mean that student s is within 1, 2 and 3 courses of graduation, resp.

$$\begin{aligned}
grad3(s) &\leftarrow \exists_c\, [grad2(s) \leftarrow take(s,c)]. \\
grad2(s) &\leftarrow \exists_c\, [grad1(s) \leftarrow take(s,c)]. \\
grad1(s) &\leftarrow \exists_c\, [grad(s) \leftarrow take(s,c)].
\end{aligned}$$

To answer the query $grad3(s)$, the first rule poses the query $grad2(s)$ to a hypothetical database. To answer this query, the second rule poses the query $grad1(s)$ to a second hypothetical database. Finally, to answer this query, the third rule poses the query $grad(s)$ to a third hypothetical database. More precisely, if a rulebase R contains these three rules, then for any student s,

$$\begin{aligned}
& R + DB \;\vdash\; grad3(s) \\
\textit{if}\quad & \exists_{c_3}\, [R + DB + take(s,c_3) \;\vdash\; grad2(s)] \\
\textit{if}\quad & \exists_{c_3}\exists_{c_2}\, [R + DB + take(s,c_3) + take(s,c_2) \;\vdash\; grad1(s)] \\
\textit{if}\quad & \exists_{c_3}\exists_{c_2}\exists_{c_1}\, [R + DB + take(s,c_3) + take(s,c_2) + take(s,c_1) \;\vdash\; grad(s)]
\end{aligned}$$

In this way, to answer one query, we must first answer other queries over other databases.
□

Hypothetical rules can be recursive. The next example suggests that recursive rules can perform complex, hypothetical transactions on a database.

Example 2.5 The predicates $grad1(s)$, $grad2(s)$ and $grad3(s)$ from Example 2.4 can be generalized to the single predicate $grad^*(s,n)$, which means that student s is within

n courses of graduation. $grad^*(s,n)$ is true iff there is a set of at most n courses such that s could graduate if he took these courses. The following recursive rules define this predicate:

$$\begin{aligned} grad^*(s,n) &\leftarrow n > 0,\ \exists_c\, [grad^*(s,n-1) \leftarrow take(s,c)]. \\ grad^*(s,0) &\leftarrow grad(s). \end{aligned}$$

The first rule recursively chooses n courses, one at a time, and hypothetically asserts that student s has taken each of them. After this is done, the second rule determines whether the student is eligible to graduate in the hypothetical database. More precisely, if a rulebase R contains these two rules, then for any student s, and any non-negative integer n,

$$\begin{aligned} & R + DB \vdash grad^*(s,n) \\ \textit{if} \quad & \exists_{c_n} \cdots \exists_{c_1}\, [R + DB + take(s,c_n) + \cdots + take(s,c_1) \vdash grad(s)] \end{aligned}$$

The choice of the courses c_i is nondeterministic. Thus, if any sequence of choices leads to graduation, then $grad^*(s,n)$ is true. Since the same course may be chosen more than once, the rules choose at most n distinct courses, as we wished. $\Box$

2.2 Rulebase Independence

A database query is often treated as a mapping that takes a database as input and returns a relation as output [15, 14]. The expressibility of a database query language is therefore the set of mappings that the language represents. Many expressibility results based on this idea have appeared in the literature [3, 16, 18, 29, 1, 2, 28], and several more are established later in this dissertation. For the purpose of this section, however, we introduce a new notion of expressibility, one that is appropriate for rule-based systems. In this notion, a query can still be viewed as a mapping, but the input to the mapping is a rulebase, not just a database. Using this idea, we prove that classical logic is inadequate for hypothetical reasoning, and we show a simple hypothetical query that cannot be expressed in classical logic.

The principle underlying our new notion of expressibility is rulebase independence: *A user should be able to formulate queries to a deductive database without knowing the contents of the rulebase.* For instance, if the rulebase encodes the US-Canada Income Tax Treaty, then a user who is only slightly familiar with the treaty should still be able to pose questions to the system, such as, "Is my pension income from Canada taxable in the US?" Similar constraints exist in relational database systems, since a user must be able to formulate database queries without knowing the contents of the database.

Rulebase independence is important for several reasons. First, it permits casual use of the rulebase: One does not have to be an expert in the contents of the rulebase in order to formulate queries. Second, it increases reliability: Without rulebase independence, if a user's knowledge of the rulebase is inaccurate, then his queries may be formulated incorrectly. Third, rulebase independence increases flexibility by making changes to the rulebase less expensive: Without rulebase independence, application programs that query the rulebase, and libraries of commonly used queries, would have to be updated each time the rulebase is changed. Finally, rulebase independence is important not only for the formulation of queries but also for the construction of rulebases. Since the premise of a rule is itself a rulebase query, keeping rule premises independent of the rest of the rulebase has all the advantages just listed for queries: It makes rules easier to formulate and more reliable, and it prevents small changes from propagating throughout the whole system.

We illustrate the idea of rulebase independence with a negative example, that is, by an expression which is rulebase *dependent.* Consider the rulebase R_1 consisting of the following, single rule:

$$grad(s) \leftarrow take(s, his101), take(s, eng201).$$

which says that a student s can graduate by taking History 101 and English 201. Now suppose we wish to ask the query, "Is student s within one course of graduation?" Knowing the contents of R_1, we could represent this query by the expression $\psi = take(s, eng101) \vee take(s, cs250)$. This expression is *database independent*, because it assumes nothing about the contents of the database of students and courses: When the database is updated, ψ will continue to give correct answers. However, ψ is not *rulebase independent*, since when the rulebase changes, ψ may return incorrect answers. For example, ψ will *not* return those students who are within one course of graduation when given the rulebase R_2 consisting of the following, single rule:

$$grad(s) \leftarrow take(s, his101), take(eng201), take(s, math250).$$

The following principle generalizes this example and captures the idea of rulebase independence:

> *An expression ψ expresses a query in a rulebase independent way iff it returns the correct answer for all databases DB and all rulebases R.*

2.2.1 The Limitations of Classical Logic

Classical logic can express some simple hypothetical queries in a rulebase independent way. For instance, the expression $grad(tony) \leftarrow take(tony, phy250)$ expresses the query, "Could tony graduate if he took Physics 450?" That is, by the deduction theorem for classical logic,

$$\begin{array}{rl} & R + DB \ \vdash \ grad(tony) \leftarrow take(tony, phy450) \\ \textit{iff} & R + DB + take(tony, phy450) \ \vdash \ grad(tony) \end{array}$$

for any database DB and any rulebase R. However, classical logic is not very good at expressing more-complex hypothetical queries. For instance, suppose that Tony has taken many courses in mathematics and in physics. Tony might qualify for a physics degree by taking the senior physics course, Physics 450, or he might qualify for a math degree by taking the senior math course, Math 452. We want to know if Tony is in one of these situations. That is, we want to ask the query, "Can Tony graduate by taking Physics 450 or by taking Math 452?" This section shows that this query cannot be expressed in classical logic in a rulebase independent way.

To express this query formally, let G denote the proposition that Tony can graduate, let A denote the proposition that he has taken Physics 450, and let B denote the proposition that he has taken Math 452. We therefore want an expression ψ such that for all databases DB and all rulebases R,

$$R + DB \vdash \psi \quad \textit{iff} \quad R + DB + A \vdash G \quad \textit{or} \quad R + DB + B \vdash G \tag{2.4}$$

We show that in classical logic, no such ψ exists. Before proving this, however, we consider two promising candidates for ψ, and show why they do not work.

The first candidate is $\psi = G \leftarrow (A \vee B)$, which we might read as, "G is true if A is true or B is true." This expression does not work because of the following classical equivalence:

$$G \leftarrow (A \vee B) \equiv (G \leftarrow A) \wedge (G \leftarrow B)$$

and hence,

$$R + DB \vdash \psi \quad \textit{iff} \quad R + DB + A \vdash G \quad \textit{and} \quad R + DB + B \vdash G. \tag{2.5}$$

But this is not what we had in mind, since if (2.5) is true, and if we choose R and DB so that $R + DB + A \vdash G$ and $R + DB + B \not\vdash G$, then $R + DB \not\vdash \psi$, which violates condition (2.4).

Another promising candidate for ψ is the expression $(G \leftarrow A) \vee (G \leftarrow B)$, which we might read as, "G is true if A is true *or* G is true if B is true." But this expression does not work either, because of the following classical equivalences:

$$\begin{aligned} (G \leftarrow A) \vee (G \leftarrow B) &\equiv (G \vee \sim A) \vee (C \vee \sim B) \\ &\equiv G \vee \sim A \vee \sim B \\ &\equiv G \leftarrow A, B \end{aligned} \tag{2.6}$$

Thus, $$R + DB \vdash \psi \quad \textit{iff} \quad R + DB + A + B \vdash G.$$

Again, this is not what we had in mind.

We have thus eliminated two promising candidates for ψ. Neither candidate represents query (2.4) in a rulebase independent way. Is there some other expression that does? The following lemma assures us that there is not.

Lemma 2.1 *In classical logic, there is no expression ψ such that for any database DB and any set R of Horn rules,*

$$R + DB \vdash \psi \quad \textit{iff} \quad R + DB + A \vdash G \quad \textit{or} \quad R + DB + B \vdash G$$

Proof: (by contradiction) Suppose that such a ψ exists. We show that when the database is empty, ψ cannot give correct answers for all three of the following rulebases:

$$R_1 = \{G \leftarrow A\} \qquad R_2 = \{G \leftarrow B\} \qquad R_3 = \{G \leftarrow A, B\}.$$

In particular, we show that if ψ returns correct answers for R_1 and R_2, then it must return incorrect answers for R_3. The proof exploits the equivalences of statement (2.6) above, i.e., $R_1 \vee R_2 \equiv R_3$.

Clearly $R_1 + A \vdash G$ and $R_2 + B \vdash G$

thus $R_1 \vdash \psi$ and $R_2 \vdash \psi$ by the definition of ψ,

so $(R_1 \vee R_2) \vdash \psi$

$R_3 \vdash \psi$ since $R_1 \vee R_2 \equiv R_3$,

$R_3 + A \vdash G$ or $R_3 + B \vdash G$ by the definition of ψ,

$\{G \leftarrow A, B\} + A \vdash G$ or $\{G \leftarrow A, B\} + B \vdash G$

which is a contradiction.

□

Although this lemma is about queries, it can be extended to the construction of rulebases in a straightforward way. Suppose, for instance, that we wish to construct a set of rules R_o that implement the query of Lemma 2.1. That is, we want a set of rules R_o which define an atom C so that

$$R + R_o + DB \vdash C \quad \textit{iff} \quad R + DB + A \vdash G \quad \textit{or} \quad R + DB + B \vdash G \tag{2.7}$$

for any database DB and any rulebase R not containing C. In classical logic, there is no set of formulas R_o (Horn or otherwise) that satisfies this condition. To see this, note that if R_o existed, then we could contradict Lemma 2.1 using $\psi = C \leftarrow R_o$. Classically, therefore, we cannot define the query of Lemma 2.1 either as a formula ψ or as a rulebase R_o.

2.3 Hypothetical Inference

Section 2.1 gave examples of hypothetical inference, and Section 2.2 showed that classical logic is inadequate for this kind of inference. Hence, if hypothetical inference is to be expressed declaratively, then new logical mechanisms are needed, i.e., proof procedures for embedded implications. Such procedures have been developed by several researchers [25, 38, 40, 35]. This section defines a simplified version of these systems, which retains many of the essential properties of the more elaborate systems while admitting a clean theoretical analysis. This inference system is an extension of Horn logic, both syntactically and proof theoretically, and much of the terminology is borrowed from first-order predicate logic. All logical expressions are function-free.

Definition 2.1 *A Horn rule is an expression of the form* $B \leftarrow B_1, B_2 ... B_k$ *where* $k \geq 0$ *and* B *and each* B_i *are atomic.*

Definition 2.2 *An embedded implication is an expression of the form* $B \leftarrow \phi_1, \phi_2 ... \phi_k$ *where* $k \geq 0$*,* B *is atomic, and each* ϕ_i *is a Horn rule.*

Note that embedded implications include Horn rules as a special case, and Horn rules include atomic formulas as a special case.

Definition 2.3 *A hypothetical rulebased system is a triple* $\mathcal{R} = [R, dom, pred]$ *where* R *is a finite set of embedded implications,* dom *is a finite set of constant symbols, and* $pred$ *is a finite set of predicate symbols with associated arities.* dom *must include all the constant symbols appearing in* R*, and* $pred$*, all the predicates symbols.*

If $\mathcal{R} = [R, dom, pred]$ is a hypothetical rulebased system, then $\mathcal{DB}(\mathcal{R})$ shall denote the set of all ground atomic formulas constructible from the predicate symbols in *pred* and the constant symbols in *dom*. Any database constructible from these symbols is a subset of $\mathcal{DB}(\mathcal{R})$, and during hypothetical inference, only those formulas in $\mathcal{DB}(\mathcal{R})$ are inserted or removed from a database. When $\mathcal{R}$ is understood, we write $\mathcal{DB}$ for short.

Definition 2.4 *(Inference) If* $[R, dom, pred]$ *is a hypothetical rulebased system, then for any database* $DB \subseteq \mathcal{DB}$*, and for all ground atomic formulas* $A, B_1, B_2 ... B_k$ *in* $\mathcal{DB}$*,*

1. $R + DB \vdash A$ *if* $A \in DB$

2. $R + DB \vdash A$ *if there is a rule* $A' \leftarrow \phi_1, ..., \phi_k$ *in* R *and a ground substitution* θ *such that* $A = A'\theta$ *and* $R + DB \vdash \phi_i\theta$ *for each* i.

3. $R + DB \ \vdash \ A \leftarrow B_1...B_k$ *if* $R + DB + \{B_1, ..., B_k\} \ \vdash \ A$

Note that hypothetical inference is defined with respect to a particular rulebased system. Thus, whenever we write $R + DB \vdash \phi$, some system $\mathcal{R} = [R, dom, pred]$ is understood.

The following lemma is a basic result about hypothetical inference.

Lemma 2.2 *(Monotonicity) If* $DB_1 \subseteq DB_2$ *and* $R + DB_1 \vdash \phi$ *then* $R + DB_2 \vdash \phi$

Proof: By a straightforward induction over the length of derivations. □

It is implicit in Definition 2.4 that the variables in a rule are universally quantified. As in Horn logic, however, if a variable appears in the body of a rule but not the head, then the universal quantifier can be moved inside the rule body and converted to an existential. For example, in the rule $A(x) \leftarrow [B(x, y) \leftarrow C(x, y)]$, the variable y does not appear in the head. If this rule appears in a rulebase R, then the inference rules above imply that for any ground term x,

$$R + DB \ \vdash \ A(x) \quad \textit{if} \quad R + DB \ \vdash \ B(x, y) \leftarrow C(x, y) \quad \textit{for some } y.$$

Consequently, this rule can be read in two equivalent ways:

$$\begin{aligned} & \forall_x \forall_y \ A(x) \leftarrow [B(x, y) \leftarrow C(x, y)] \\ \textit{and} \quad & \forall_x \ A(x) \leftarrow \exists_y \ [B(x, y) \leftarrow C(x, y)] \end{aligned}$$

It is the latter interpretation that enables the logic to represent the examples given in Section 2.1. In particular, the rule $grad1(s) \leftarrow [grad(s) \leftarrow take(s, c)]$ defines those students who are "within one course of graduation."

2.3.1 Examples

This section provides examples of hypothetical inference. The first example shows how embedded implications can express the hypothetical query studied in Section 2.2.1, which cannot be expressed in classical logic.

Example 2.6 Let $R_o = \{C \leftarrow (G \leftarrow A), \quad C \leftarrow (G \leftarrow B)\}$. Then for any rulebase R and any database DB,

$$R + R_o + DB \vdash C \quad \textit{if} \quad R + DB + A \vdash G \quad \textit{or} \quad R + DB + B \vdash G$$

Furthermore, if neither R nor DB mentions the atom C, then the *if* becomes *iff*. □

The next example shows how simple, hypothetical updates can be combined to produce more-complex, hypothetical transactions.

Example 2.7 Suppose R consists of rules defining a predicate D plus the following $n+1$ rules:

$$\begin{aligned}
&A_1 \leftarrow (A_2 \leftarrow B_1) \\
&A_2 \leftarrow (A_3 \leftarrow B_2) \\
&\quad \cdots \\
&A_n \leftarrow (A_{n+1} \leftarrow B_n) \\
&A_{n+1} \leftarrow D
\end{aligned}$$

Then $R + DB \vdash A_i$ *if* $R + DB + \{B_i, ..., B_n\} \vdash D$. □

The next two examples not only illustrate hypothetical inference, they also introduce, in simplified form, the kind of rules used in Chapter 4 to simulate Turing machines. In these two examples, the database DB is assumed to include the following atomic formulas, which define a finite sequence $\langle a_1, a_2, ..., a_n, a_{n+1}\rangle$:

$$FIRST(a_1),\ NEXT(a_1, a_2),\ NEXT(a_2, a_3)\ ...\ NEXT(a_n, a_{n+1}),\ LAST(a_{n+1}).$$

The first example is a recursive version of Example 2.7.

Example 2.8 Suppose R consists of rules defining the predicate D plus the following three rules:

$$E \leftarrow FIRST(x),\ A(x).$$
$$A(x) \leftarrow NEXT(x,y),\ [A(y) \leftarrow B(x)].$$
$$A(x) \leftarrow LAST(x),\ D.$$

Then $\quad R + DB \vdash E \quad \mathit{if} \quad R + DB + \{B(a_1), ...B(a_n)\} \ \vdash\ D.$ $\square$

Example 2.9 Suppose R consists of rules defining the predicate D plus the following four rules:

$$E \leftarrow FIRST(x),\ A(x).$$
$$A(x) \leftarrow NEXT(x,y),\ [A(y) \leftarrow B(x)],\ [A(y) \leftarrow C(x)].$$
$$A(x) \leftarrow LAST(x),\ D.$$

$$\begin{array}{lllll} \text{Then} & R + DB \vdash E & \mathit{if} & R + DB + \{B(a_1), B(a_2), ..., B(a_n)\} & \vdash\ D \\ & & \mathit{and} & R + DB + \{C(a_1), B(a_2), ..., B(a_n)\} & \vdash\ D \\ & & \mathit{and} & R + DB + \{B(a_1), C(a_2), ..., B(a_n)\} & \vdash\ D \\ & & \mathit{and} & R + DB + \{C(a_1), C(a_2), ..., B(a_n)\} & \vdash\ D \\ & & & \ldots & \\ & & \mathit{and} & R + DB + \{C(a_1), C(a_2), ..., C(a_n)\} & \vdash\ D \end{array}$$

$\square$

Example 2.9 illustrates how a fixed set of hypothetical rules can express $O(2^n)$ meta-level queries, where n is the size of the data domain. The next section exploits this capability to construct a rulebase whose data complexity is coNP complete.

2.3.2 The Power of Hypothetical Inference

This section shows that hypothetical inference is more powerful than well-known database query languages such as relational algebra and Datalog (function-free Horn logic). These languages can only express queries that are computable in polynomial time. In contrast, this section shows that that hypothetical inference can solve problems that are coNP-hard. In particular, we construct a rulebase R that solves the validity problem for propositional formulas. This problem is impossible to solve in Datalog, unless P = NP.

To solve the validity problem, we encode a propositional formula as a database DB. We also construct a rulebase R such that $R + DB \vdash VALID$ iff the formula is valid, where $VALID$ is a predicate of arity zero. Since the rulebase R is fixed (i.e., is independent of the propositional formula), the data complexity of hypothetical inference is coNP-hard.

It is convenient to think of the propositional formula as a circuit. From this point of view, the database DB represents a circuit, and $R + DB \vdash VALID$ iff the output of the circuit is 1 for every possible set of inputs. Without loss of generality, we can assume the circuit is constructed entirely of nor gates. We use constant symbols to represent wires and atomic formulas to represent circuit interconnections. Thus, the formula $NOR(a, b, c)$ means that wire a is the output of a nor gate whose input wires are b and c. Similarly, the formulas $ZERO(a)$ and $ONE(a)$ mean that the value of wire a is 0 or 1, resp. Finally, the formula $OUPUT(b)$ means that wire b is the output of the entire circuit.

If the database DB represents a circuit plus a set of input values, then the following rules evaluate the circuit. If these rules are denoted R_0, then $R_0 + DB \vdash EVAL$ iff the output of the circuit is 1, where $EVAL$ is a zero-ary predicate symbol. Notice that the rulebase is entirely Horn.

$$EVAL \leftarrow OUPUT(x),\ ONE(x).$$
$$ONE(x) \leftarrow NOR(x, y, z),\ ZERO(y),\ ZERO(z).$$
$$ZERO(x) \leftarrow NOR(x, y, z),\ ONE(y).$$
$$ZERO(x) \leftarrow NOR(x, y, z),\ ONE(z).$$

To determine whether the circuit is valid, we construct a rulebase R that hypothetically inserts every possible set of input values into the database, one set at a time, and evaluates the circuit for each set. To set the input values, the rules repeatedly select a wire and hypothetically set its value to 0 or 1. So that the rulebase can select input wires one after another, we assume that the input wires are linearly ordered. That is, if the input wires are $a_1, ..., a_n$, then we put the following atomic formulas into the database:

$$FIRST(a_1),\ NEXT(a_1, a_2),\ NEXT(a_2, a_3),\ ...\ NEXT(a_n, a_{n+1}),\ LAST(a_{n+1})$$

where a_{n+1} is an extra constant symbol, introduced to simplify the construction of the rulebase. Let DB be the database that encodes the circuit and orders the inputs in this way.

Given such a database, we form a rulebase R from the four rules above plus the following three rules:

$$VALID \leftarrow FIRST(x),\ SET(x).$$
$$SET(x) \leftarrow NEXT(x, y),\ [SET(y) \leftarrow ONE(x)],\ [SET(y) \leftarrow ZERO(x)].$$
$$SET(x) \leftarrow LAST(x),\ EVAL.$$

These rules are best understood from a top-down perspective. The first rule begins the inference process by selecting the first element in the linear order. The second rule then recurses down the linear order, selecting one new element after another. Each element in the linear order corresponds to an input wire, and after each selection, the second rule hypothetically sets the input wire to 0 or 1. Specifically, the input wire is set to 1 in the left branch of the proof tree, and to 0 in the right branch. In this way, while selecting the n input wires, the second rule generates a proof tree with 2^n branches, each branch corresponding to a different set of input values. In each of these branches, the database contains different hypothetical values for the input wires, as in Example 2.9.

After all the input wires have been assigned a value, the third rule invokes the predicate $EVAL$ to evaluate the circuit, i.e., to determine whether the output of the circuit is 1. $EVAL$ is invoked in each of the 2^n branches of the proof tree, that is, for each possible set of input values. If $EVAL$ is inferred in every branch, then the three rules above infer the predicate $VALID$, and only then. That is, $R + DB \vdash VALID$ iff DB represents a circuit whose output is 1 for every set of input values.

The rulebase R therefore solves a problem whose data complexity is coNP-hard.

2.4 Semantics

The inference system defined in Section 2.3 is not classical. This was demonstrated by Example 2.6, which showed how to express a hypothetical query that classical logic cannot. What then is the semantics of the system? Several researchers have shown that it is intuitionistic [37, 40]. This section reviews intuitionistic semantics and develops a new and simpler proof that the inference system is sound and complete intuitionistically.

Informally, the reason that hypothetical inference is intuitionistic, and not classical, is that classical logic has too many equivalences. For instance, in classical logic, the formulas $A \leftarrow B$ and $A \vee \sim B$ are equivalent, whereas in intuitionistic logic they are not. In this sense, intuitionistic logic makes distinctions that classical logic does not. Likewise, intuitionistic logic distinguishes between the formulas $G \leftarrow A, B$ and $(G \leftarrow A) \vee (G \leftarrow B)$, whereas classical logic does not, as shown by statement (2.6) of Section 2.2. The inability of classical logic to make this particular distinction is at the heart of Lemma 2.1, which captures the limited power of classical logic to express hypothetical queries.

It is also worth noting that every intuitionistic theorem is a classical theorem. Intuitionistic reasoning is therefore like classical reasoning except that certain inferences are forbidden, such as the equivalences mentioned above. This property is evident in the inference system of Definition 2.4, since each inference rule in this system is classically sound. Thus, if $R + DB \vdash \phi$ then $R + DB \models_c \phi$, where $\models_c$ denotes classical entailment. The converse is not true however. To see this, consider the following set of three rules:

$$R \;=\; \{A \leftarrow (B \leftarrow C), \quad D \leftarrow A, \quad D \leftarrow C\}$$

From Definition 2.4, it follows that $R \not\vdash D$. However, if this rulebase is interpreted classically, then $R \models_c D$. To see this, note that from the classical definition of implication, the first rule in R is equivalent to $(A \vee \sim B) \wedge (A \vee C)$ and the last two are equivalent to $D \leftarrow (A \vee C)$. Classically, therefore, we can infer $A \vee C$, and hence D. Thus, although the inference system of Definition 2.4 is classically sound, it is not classically complete.

2.4.1 Intuitionistic Logic

This section defines the semantics of first-order, intuitionistic logic in the function-free case. A more extensive treatment may be found in [22].

Intuitionistic logic has a modal-like semantics that can be interpreted as a semantics of incomplete knowledge. For instance, an intuitionistic structure is a set of substates, each representing a state of our "knowledge." The substates are organized into a partial order, and as we "climb" the partial order from one substate to another, our knowledge increases, that is, we know about more objects, and we know more true formulas. This description is made precise by the definitions below.

The syntax of the logic is first-order and includes three infinite sets: a set of variables $x, y, z, ...$, a set of constant symbols $a, b, c, ...$, and a set of predicate symbols $A, B, C, ...$. We call each of these sets a *universe*.

Definition 2.5 *An intuitionistic structure is a quadruple $M = \langle S, \leq, \pi, d \rangle$, where*

- *S is a non-empty set,*
- *$\leq$ is a partial order on S,*
- *π is a monotonic mapping from elements of S to sets of ground atomic formulas; i.e., if $s_1 \leq s_2$ then $\pi(s_1) \subseteq \pi(s_2)$.*
- *d is a mapping from elements of S to sets of constant symbols. Furthermore, for any element s in S, $d(s)$ must contain all the constant symbols appearing in $\pi(s)$.*

Each element s of S is called a *substate* of M, and $d(s)$ is called the *domain* of s. Notice that as we "climb" the partial order from one substate s to another, both the domain $d(s)$ and the set of atoms $\pi(s)$ increase. Informally, these two quantities represent our knowledge of what objects exist and of what facts are true. In this sense, substates higher in a structure represent states of greater knowledge.

Truth in an intuitionistic structure is defined relative to its substates. That is, one can ask whether a formula ψ is true at a particular substate s of some intuitionistic structure M, written $s, M \models_i \psi$. In general, this expression is undefined if ψ contains constant symbols not in the domain $d(s)$ (that is, if ψ mentions objects that are not known to exist.) In this case, we say that ψ is not defined at s. The following definitions make these ideas precise.

Definition 2.6 *If dom is a set of constant symbols, then $\mathcal{F}(dom)$ denotes the set of first-order formulas containing only those constant symbols in dom. If s is a substate, then $\mathcal{F}(s)$ means $\mathcal{F}[d(s)]$.*

$\mathcal{F}(s)$ is thus the set of all formulas defined at the substate s. Note that if $s_1 \leq s_2$ then $\mathcal{F}(s_1) \subseteq \mathcal{F}(s_2)$.

Definition 2.7 *(Satisfaction) Suppose M is an intuitionistic structure and s is a substate of M. Then,*

$$
\begin{array}{lll}
s, M \models_i A & \textit{iff} & A \in \pi(s) \quad \textit{when } A \textit{ is atomic.} \\
s, M \models_i \psi_1 \wedge \psi_2 & \textit{iff} & s, M \models_i \psi_1 \quad \textit{and} \quad s, M \models_i \psi_2 \\
 & \textit{and} & \psi_1 \wedge \psi_2 \textit{ is in } \mathcal{F}(s). \\
s, M \models_i \psi_1 \vee \psi_2 & \textit{iff} & s, M \models_i \psi_1 \quad \textit{or} \quad s, M \models_i \psi_2 \\
 & \textit{and} & \psi_1 \vee \psi_2 \textit{ is in } \mathcal{F}(s). \\
s, M \models_i \sim \psi & \textit{iff} & r, M \not\models_i \psi \quad \textit{for all } r \geq s \\
 & \textit{and} & \sim \psi \textit{ is in } \mathcal{F}(s). \\
s, M \models_i \psi_2 \leftarrow \psi_1 & \textit{iff} & r, M \models_i \psi_2 \quad \textit{whenever} \quad r, M \models_i \psi_1, \quad \textit{for all } r \geq s \\
 & \textit{and} & \psi_2 \leftarrow \psi_1 \textit{ is in } \mathcal{F}(s). \\
s, M \models_i \forall_x \psi(x) & \textit{iff} & r, M \models_i \psi(b) \quad \textit{for all } b \in d(r) \textit{ and all } r \geq s \\
 & \textit{and} & \forall_x \psi(x) \textit{ is in } \mathcal{F}(s). \\
s, M \models_i \exists_x \psi(x) & \textit{iff} & s, M \models_i \psi(b) \quad \textit{for some } b \in d(s) \\
 & \textit{and} & \exists_x \psi(x) \textit{ is in } \mathcal{F}(s).
\end{array}
$$

An intuitive interpretation of this semantics may be found in [33] and [22]. Note that unlike classical logic, intuitionistic implication is not defined in terms of disjunction and negation. Rather, it has an independent semantic definition. It is this definition that captures hypothetical insertion.

The following basic result is an immediate consequence of the above definitions [22]. It reflects the idea that knowledge increases monotonically as we climb from one substate to a higher one.

Lemma 2.3 *$s, M \models_i \phi$ iff $r, M \models_i \phi$ for all $r \geq s$.*

Definition 2.8 *(Models) $M \models_i \psi$ iff $s, M \models_i \psi$ for all substates s of M such that $\psi \in \mathcal{F}(s)$. In this case, we say that M is a model of ψ, or that ψ is true in M.*

Definition 2.9 *(Validity) A formula ψ is valid iff $M \models_i \psi$ for all intuitionistic structures M.*

Definition 2.10 *(Entailment) Suppose ψ_1 and ψ_2 are formulas. Then $\psi_1 \models_i \psi_2$ iff the formula $\psi_2 \leftarrow \psi_1$ is valid. In this case, we say that ψ_1 entails ψ_2.*

2.4.2 Soundness and Completeness

This section shows that the inference system defined in Section 2.3 is sound and complete with respect to intuitionistic semantics.

This result assumes several conventions about how to interpret rules and rulebases as first-order formulas, conventions that are common in the logic-programming literature. First, we interpret a rule $A \leftarrow \phi_1 ... \phi_k$ as an abbreviation for the first-order formula $A \leftarrow (\phi_1 \wedge ... \wedge \phi_k)$. Second, we interpret free variables in a rule as universally quantified variables. With these two conventions, the rule $A(x) \leftarrow B(x), C(x)$ is interpreted as the formula $\forall_x [A(x) \leftarrow (B(x) \wedge C(x))]$. Third, we treat a finite set of formulas, such as a rulebase or a database, as a conjunction of the formulas. With these three conventions, we show that hypothetical inference is equivalent to intuitionistic entailment.

Recall that hypothetical inference is defined with respect to a particular rulebased system. A rulebased system is a triple $[R, dom, pred]$, where R is a set of embedded implications, *dom* is a set of constant symbols, and *pred* is a set of predicate symbols. Given such a rulebased system, $\mathcal{DB}$ denotes the set of ground atomic formulas constructible from the constant symbols in *dom* and the predicate symbols in *pred*. Any database constructible from these symbols is a subset of $\mathcal{DB}$.

We can now state the following theorem, which is the main result of this section.

Theorem 2.4 *Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system, and let ϕ be a ground Horn rule constructed from the symbols in dom and pred. Then, for any database $DB \subseteq \mathcal{DB}$,*

$$R + DB \models_i \phi \quad \textit{iff} \quad R + DB \vdash \phi$$

The *if* direction of Theorem 2.4 (i.e., soundness) follows immediately because the rules of hypothetical inference given in Definition 2.4 are intuitionistically valid. In particular, inference rule *1* is trivially valid, and inference rules *2* and *3* correspond to modus ponens and the deduction theorem, respectively.

The rest of this section proves the *only if* direction (i.e., completeness). The approach taken here is inspired by the Henkin constructions of modal logic [21, 27]: Given a rulebased system $\mathcal{R}$, we define an intuitionistic structure $M_{\mathcal{R}}$ called the *canonical model* of $\mathcal{R}$. This structure, defined proof-theoretically, provides the necessary link between inference and semantics.

Definition 2.11 *Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system. The canonical model of $\mathcal{R}$ is the intuitionistic structure $M_{\mathcal{R}} = \langle S, \leq, \pi, d \rangle$ where*

$$\begin{aligned}
S &= \{DB \mid DB \subseteq \mathcal{DB}\} = \textit{the set of all databases},\\
d(DB) &= dom, \textit{ the universe of constant symbols},\\
\pi(DB) &= \{A \in \mathcal{DB} \mid R + DB \vdash A\},\\
DB_1 \leq DB_2 &\quad \textit{iff} \quad DB_1 \subseteq DB_2
\end{aligned}$$

In other words, each substate of the canonical model is a database, the atoms that are true at a substate DB are just those atoms that can be inferred from DB and R, and the substates are ordered by set containment.

The canonical model $M_{\mathcal{R}}$ is a legitimate intuitionistic structure. That is, the mappings π and d satisfy the conditions of Definition 2.5. First, the mapping π is monotonic, since by Lemma 2.3, hypothetical inference is monotonic. Second, every constant symbol appearing in $\pi(DB)$ is in the domain $d(DB)$, since $d(DB)$ is the universe of all constant symbols.

The following basic result about canonical models is immediate, and is the starting point for our proof of completeness.

Lemma 2.5 *Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system, and let A be an atom in $\mathcal{DB}$. Then, for any database $DB \subseteq \mathcal{DB}$,*

$$DB, M_{\mathcal{R}} \models_i A \quad \textit{iff} \quad R + DB \vdash A$$

We now prove a series of results about canonical models, results that lead directly to a proof of completeness. The first result extends the *only if* direction of Lemma 2.5 from atomic queries A to Horn queries ϕ.

Lemma 2.6 *Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system, and let ϕ be a ground Horn rule constructed from the symbols in dom and pred. Then, for any database $DB \subseteq \mathcal{DB}$,*

$$\textit{if} \quad DB, M_{\mathcal{R}} \models_i \phi \quad \textit{then} \quad R + DB \vdash \phi$$

Proof: ϕ has the form $A \leftarrow B_1 ... B_k$ where A and each B_i are atomic formulas in $\mathcal{DB}$. Hence, if $DB, M_{\mathcal{R}} \models_i A \leftarrow B_1 ... B_k$ then,

$DB', M_{\mathcal{R}} \models_i B_1 ... B_k \;\Rightarrow\; DB', M_{\mathcal{R}} \models_i A$	for all $DB' \geq DB$, by Definition 2.7,
$R + DB' \vdash B_1 ... B_k \;\Rightarrow\; R + DB' \vdash A$	for all $DB' \geq DB$, by Lemma 2.5,
$R + DB + \{B_1, ..., B_k\} \vdash A$	using $DB' = DB + \{B_1, ..., B_k\}$,
$R + DB \vdash A \leftarrow B_1 ... B_k$	by rule 3 of Definition 2.4.

□

The next result exploits the fact that in the canonical model, each substate has the same domain, *dom*. Thus, a formula is defined on one substate iff it is defined on all substates.

Lemma 2.7 *Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system, and let $\psi(x)$ be a first-order formula whose constant symbols are in dom. Then, for any database $DB \subseteq \mathcal{DB}$,*

$$DB, M_{\mathcal{R}} \models_i \forall_x \psi(x) \quad \textit{iff} \quad DB, M_{\mathcal{R}} \models_i \psi(b) \;\; \textit{for all } b \in dom$$

Proof: First, note that if $b \in dom$, then the formulas $\psi(b)$ and $\forall_x \psi(x)$ contain only those constant symbols in *dom*. These formulas are therefore defined at every substate of $M_{\mathcal{R}}$, since *dom* is the domain of each of these substates. Thus, in Definition 2.7, the conditions "$\forall_x \psi(x)$ *is in* $\mathcal{F}(s)$" and "$\psi(b)$ *is in* $\mathcal{F}(s)$" are always satisfied, and can therefore be ignored. Keeping this in mind,

$$
\begin{array}{llll}
 & DB, M_{\mathcal{R}} \models_i \forall_x \psi(x) & & \\
\textit{iff} & DB', M_{\mathcal{R}} \models_i \psi(b) & \text{for all } b \in d(DB') \text{ and all } DB' \geq DB, & \text{by definition,} \\
\textit{iff} & DB', M_{\mathcal{R}} \models_i \psi(b) & \text{for all } b \in dom \text{ and all } DB' \geq DB, & \\
\textit{iff} & DB, M_{\mathcal{R}} \models_i \psi(b) & \text{for all } b \in dom, & \text{by Lemma 2.3.}
\end{array}
$$

□

Finally, the next result shows that the canonical model $M_{\mathcal{R}}$ is a model of the rulebase R.

Lemma 2.8 *Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system, and let $DB \subseteq \mathcal{DB}$ be a database. Then $DB, M_{\mathcal{R}} \models_i (R + DB)$.*

Proof: $DB, M_{\mathcal{R}} \models_i DB$ trivially, so we need only show that $DB, M_{\mathcal{R}} \models_i R$. We show that $DB, M_{\mathcal{R}} \models_i \psi$ for any rule ψ in R. By convention, the free variables in ψ are universally quantified. Furthermore, because ψ is in R, it contains only those constant symbols in *dom*. Lemma 2.7 is therefore applicable. That is, to show that $DB, M_{\mathcal{R}} \models_i \psi$ it is sufficient to show that $DB, M_{\mathcal{R}} \models_i \psi'$ for every ground instance ψ' of ψ (where each instance of ψ is generated by replacing the variables in ψ by constants from *dom*). Suppose, therefore, that $A \leftarrow \phi_1 ... \phi_k$ is a ground instance of ψ. Then, for any database $DB' \geq DB$, if $DB', M_{\mathcal{R}} \models_i \phi_1 \wedge ... \wedge \phi_k$ then,

$$
\begin{array}{ll}
DB', M_{\mathcal{R}} \models_i \phi_j & \text{for each } j, \text{ by Definition 2.7,} \\
R + DB' \vdash \phi_j & \text{for each } j, \text{ by Lemma 2.6,} \\
R + DB' \vdash A & \text{by rule 2 of Definition 2.4,} \\
 & \text{since } A \leftarrow \phi_1 ... \phi_k \text{ is a ground instance of a rule in } R, \\
DB', M_{\mathcal{R}} \models_i A & \text{by Lemma 2.5.}
\end{array}
$$

Thus $DB, M_{\mathcal{R}} \models_i A \leftarrow \phi_1 ... \phi_k$ by Definition 2.7. Since this this is true for any ground instance of any rule in R, it follows that $DB, M_{\mathcal{R}} \models_i R$. □

Completeness follows in a straightforward way.

Corollary 2.9 *(Completeness) Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system, and let ϕ be a ground Horn rule constructed from the symbols in dom and pred. Then for any database $DB \subseteq \mathcal{DB}$,*

$$\textit{if} \quad R + DB \models_i \phi \quad \textit{then} \quad R + DB \vdash \phi$$

Proof: First note that R, DB and ϕ are finite and are constructed from the constant symbols in *dom* and the predicate symbols in *pred*. They are therefore defined on every substate of $M_{\mathcal{R}}$. Keeping this in mind, suppose that $R + DB \models_i \phi$. Then,

$\phi \leftarrow (R + DB)$ is valid,	by Definition 2.10,
$M \models_i \phi \leftarrow (R + DB)$	for all structures M, by Definition 2.9,
$M_{\mathcal{R}} \models_i \phi \leftarrow (R + DB)$	since $M_{\mathcal{R}}$ is an intuitionistic structure,
$DB', M_{\mathcal{R}} \models_i \phi \leftarrow (R + DB)$	for all databases $DB' \subseteq \mathcal{DB}$, by Definition 2.8,
	since $\phi \leftarrow (R + DB)$ is defined on all substates,
$DB, M_{\mathcal{R}} \models_i \phi \leftarrow (R + DB)$	using $DB' = DB$,
$DB, M_{\mathcal{R}} \models_i \phi$	since $DB, M_{\mathcal{R}} \models_i (R + DB)$, by Lemma 2.8,
$R + DB \vdash \phi$	by Lemma 2.6.

□

Although this proves the main result of this section, we include one more result for completeness sake. This result shows that the canonical model $M_{\mathcal{R}}$ characterizes all the inferences sanctioned by the rulebased system $\mathcal{R}$. $M_{\mathcal{R}}$ is therefore a canonical model in the logic-programming tradition.

Corollary 2.10 *Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system, and let ϕ be a Horn rule constructed from the symbols in dom and pred. Then, for any database $DB \subseteq \mathcal{DB}$,*

$$R + DB \vdash \phi \quad \textit{iff} \quad DB, M_{\mathcal{R}} \models_i \phi$$

Proof: The *only if* direction is just Lemma 2.6. The *if* direction follows from soundness: If $R + DB \vdash \phi$ then $R + DB \models_i \phi$ by soundness, and so $DB, M_{\mathcal{R}} \models_i \phi$ as shown in the proof of Corollary 2.9. $\Box$

At this point, it is worth mentioning two properties of the canonical model M_R: (i) The domain is constant, i.e., does not vary from one substate to another; and (ii) every formula is defined at every substate. These properties helped simplify the proof of completeness. In addition, it follows from corollary 2.10 that we can restrict the semantics of embedded implications to structures having these two properties. For instance, if R is a rulebase and ϕ is an embedded implication, then $R \models \phi$ iff $R \leftarrow \phi$ is true in all structures having *constant* domain. This is a more restrictive notion of entailment than that given by Definition 2.10. Other restrictions are also possible. For instance, we need consider only those structures in which the domain of every substate is the universal domain. This restriction is particularly convenient as it allows us to ignore the conditions "ψ *is in* $\mathcal{F}(s)$" in Definition 2.7, since they are always satisfied. Thus, embedded implications can be viewed as having a kind of simplified intuitionistic semantics. This is true even for larger classes of embedded implications, such as those defined in [13].

Chapter 3

Hypothetical Insertions and Deletions

Chapter 2 developed a language that combines logical inference with hypothetical database insertion. This chapter extends that work by allowing hypothetical deletion as well. Deletion allows a user to express queries such as, "Could I have graduated if I had *not* taken History 250?" Unlike insertion, little research has been done on combining logical inference with hypothetical deletion, two exceptions being the work of Vieille, et al [50] and Warren and Manchanda [51, 34]. As in their work, this chapter focuses on the deletion of base facts, not inferred facts. Although such deletions are relatively simple, they increase the complexity and expressive power of hypothetical reasoning, as shown in Chapters 4 and 6. To express hypothetical deletion, we shall depart from the strictly first-order syntax of Chapter 2 and introduce a new, modal-like notation. To justify this new device, we first re-examine the notion of a deductive database.

A deductive database consists of a database DB and a rulebase R. In Chapter 2, these rules and data were combined into a single set $R + DB$, where $+$ denotes set union. This union has the effect of blurring the distinction between rules and data, since it is no longer clear whether an atomic formula belongs to the database or to the rulebase. In the case of hypothetical insertion, this distinction does not matter; but in the case of deletion, it matters a great deal.

If the rulebase and the database are not kept distinct, then rulebase independence is lost (as defined in Section 2.2). In fact, even the simplest queries involving hypothetical deletion cannot be expressed in a rulebase independent way. For example, consider the query, "Could $B(a)$ be inferred if $A(a)$ were deleted from the database?" Suppose this query can be expressed by some formula ψ. Then ψ has the following property:

$$R + DB \vdash \psi \quad \textit{iff} \quad R + [DB \sim B(a)] \vdash A(a) \tag{3.1}$$

for any rulebase R and any database DB. This property leads immediately to a contradiction. To see this, consider the following two rulebases:

$$R_1 = \{ A(x) \leftarrow B(x) \} \qquad\qquad R_2 = \left\{ \begin{array}{l} A(x) \leftarrow B(x) \\ A(a) \end{array} \right\}$$

If $DB = \{B(a)\}$, then $R_1 + DB$ is logically equivalent to $R_2 + DB$. This is true both in classical logic and in intuitionistic logic. In fact, it is true in any logic obeying the laws of modus ponens and variable instantiation, i.e., in any reasonable logic. Thus $R_1 + DB$ and $R_2 + DB$ are logically indistinguishable. Hence, it should be the case that $R_1 + DB \vdash \psi$ iff $R_2 + DB \vdash \psi$. But this is a contradiction, since according to condition 3.1, $R_1 + DB \not\vdash \psi$ and $R_2 + DB \vdash \psi$. This contradiction shows that even the simplest query involving hypothetical deletion cannot be expressed in a logical framework unless the rulebase and the database are kept distinct.

For this reason, we represent a deductive database not as the union $R + DB$, but as an ordered pair $[R, DB]$. In particular, we use the expression $R, DB \vdash \psi$ to mean that ψ can be inferred from the rules in R and the data in DB. Note that since we have split the rules and data in this way, we are no longer working in a first-order language.

To represent hypothetical queries in this new language, we use a modal-like notation. For example, the expression $A[del : B]$ means, "If B were *deleted* from the database, then A would be true." Similarly, the expression $A[add : B]$ means, "If B were *added* to the database, then A would be true."

Example 3.1 Consider the query, "If Tony had not taken History 250, could he have graduated?" The expression $grad(tony)[del : take(tony, his260)]$ represents this query and has the following property:

$$\begin{array}{rl} & R, DB \ \vdash \ grad(tony)[del : take(tony, his260)] \\ \textit{iff} & R, DB \sim take(tony, his250) \ \vdash \ grad(tony) \end{array}$$

for any rulebase R and any database DB, where $\sim$ denotes set difference. □

Example 3.2 Consider the query, "Retrieve those students who took one more course than necessary to graduate." That is, we want those students who could have graduated by taking one less course. Formally, we want those students s such that for some course c,

$$\begin{aligned} & R, DB \vdash take(s,c) \\ and \quad & R, DB \sim take(s,c) \vdash grad(s) \end{aligned} \tag{3.2}$$

The expression $take(s,c) \wedge grad(s)[del : take(s,c)]$ represents this query. That is, condition (3.2) is satisfied iff $R, DB \vdash take(s,c)$ *and* $R, DB \vdash grad(s)[del : take(s,c)]$. $\square$

3.1 Hypothetical Inference

This section develops an inference system for hypothetical insertion and deletion. The system is an extension of Horn logic, both syntactically and proof-theoretically. It is a minimal extension in that it augments the proof theory of Horn logic with just two rules of inference: one for hypothetical insertion, and one for hypothetical deletion. In this inference system, all logical expressions are function-free.

The following two definitions generalize the notions of hypothetical query and hypothetical rule introduced in Section 2.3.

Definition 3.1 *A hypothetical query is an expression having one of the following forms:*

- A, *where A is atomic.*
- $A[add : B_1 ... B_k]$, *where $A, B_1 ... B_k$ are all atomic.*
- $A[del : B_1 ... B_k]$, *where $A, B_1 ... B_k$ are all atomic.*

Definition 3.2 *A hypothetical rule is an expression of the form $A \leftarrow \phi_1, \phi_2 ... \phi_k$ where $k \geq 0$, A is atomic, and each ϕ_i is a hypothetical query.*

The following definition generalizes the notion of a rulebased system introduced in Section 2.3.

Definition 3.3 *A hypothetical rulebased system is a triple $\mathcal{R} = [R, dom, pred]$ where R is a finite set of hypothetical rules, dom is a finite set of constant symbols, and* *pred is a finite set of predicate symbols with associated arities. dom must include all the constant symbols appearing in R, and pred must include all the predicate symbols appearing in R.*

If $\mathcal{R} = [R, dom, pred]$ is a hypothetical rulebased system, then $\mathcal{DB}(\mathcal{R})$ shall denote the set of all ground atomic formulas constructible from the predicate symbols in *pred* and the constant symbols in *dom*. Any database constructible from these symbols is a subset of $\mathcal{DB}(\mathcal{R})$, and during hypothetical inference, only those formulas in $\mathcal{DB}(\mathcal{R})$ are inserted or removed from a database. When $\mathcal{R}$ is understood, we write $\mathcal{DB}$ for short.

The following definition generalizes the notion of hypothetical inference given in Section 2.3.

Definition 3.4 *(Inference) If $[R, dom, pred]$ is a hypothetical rulebased system, then for any database $DB \subseteq \mathcal{DB}$, and for all ground atomic formulas $A, B_1, B_2...B_k$ in $\mathcal{DB}$,*

1. $R, DB \vdash A \quad$ *if* $\quad A \in DB$
2. $R, DB \vdash A \quad$ *if there is a rule* $A' \leftarrow \phi_1...\phi_k$ *in R and a ground substitution* θ *such that* $A = A'\theta$ *and* $R, DB \vdash \phi_i\theta$ *for each i.*
3. $R, DB \vdash A[add : B_1...B_k] \quad$ *if* $\quad R, DB + \{B_1, ..., B_k\} \vdash A$
4. $R, DB \vdash A[del : B_1...B_k] \quad$ *if* $\quad R, DB \sim \{B_1, ..., B_k\} \vdash A$

Note that hypothetical inference is defined with respect to a particular rulebased system. Thus, whenever we write $R, DB \vdash \phi$, some system $\mathcal{R} = [R, dom, pred]$ is understood.

3.2 Recursion to Exponential Depth

From the perspective of top-down inference, recursion in Datalog can only go to polynomial depth before goals begin to repeat. This is because a goal in Datalog is a ground atomic formula, and thus for a given rulebase, there are only polynomially many different goals (polynomial in the size of the data domain). However, in hypothetical the system defined above, the database changes during inference, so the same atomic goal may repeat many times while the database changes. In hypothetical inference, therefore, the entire expression $R, DB \vdash \psi$ should be viewed as a goal. Since there are exponentially many different databases, there are also exponentially many different goals. Recursion may therefore go to exponential depth before goals begin repeating. This capacity for exponential recursion is crucial to the proof of EXPTIME completeness given in Chapter 4.

This section shows that the exponential bound on recursion is tight. That is, we construct a rulebase R and a database DB such that the expression $R, DB \vdash INC$ can be derived only after exponentially many levels of recursion (exponential in the size of DB). In particular, to infer the atom INC, R must first count from 0 to $2^n - 1$ in binary, where each increment of the counter requires at least one level of recursion.

At each step in the derivation, the database represents an n-bit binary number. Initially, the database represents the number 0, and as inference proceeds, the database is hypothetically updated to represent larger and larger numbers. The initial database DB consists of the following entries:

$$FIRST(a_1),\ NEXT(a_1, a_2),\ NEXT(a_2, a_3),\ \dots\ NEXT(a_{n_1}, a_n),\ LAST(a_n),$$
$$ZERO(a_1),\ ZERO(a_2),\ \dots\ ZERO(a_n).$$

The first line represents a linear order $\langle a_1, a_2, ..., a_n \rangle$. We use this order to encode n-bit binary numbers. Each constant a_i represents a binary digit: The formula $ZERO(a_i)$ means that the i^{th} digit is 0, and the formula $ONE(a_i)$ means that the i^{th} digit is 1. In this way, any number between 0 and $2^n - 1$ can be represented as a database. For example, the second line above encodes the n-bit binary representation of the number 0.

Let DB_i be a database that represents the number i as an n-bit binary number. Thus $DB_0 = DB$. We construct a rulebase R such that $R, DB_i \vdash INC$ for each i. In particular, we construct R so that

- The expression $R, DB_i \vdash INC$ is derived by recursively deriving the expression $R, DB_{i+1} \vdash INC$.

- The expression $R, DB_{2^{n-1}} \vdash INC$ is derived in one step.

Thus, deriving the expression $R, DB_0 \vdash INC$ requires at least 2^{n-1} recursive calls.

The second item above is easy to satisfy. The database represents an n-bit binary number, so we need only check the highest-order bit, to see if its value is 1. The following rule does exactly this:

$$INC \leftarrow LAST(x), ONE(x).$$

Thus $R, DB_i \vdash INC$ can be derived in one step for $i \geq 2^{n-1}$.

To satisfy the first item above, we construct rules which effectively add 1 to DB_i to produce DB_{i+1}. The idea is to add 1 to the first digit of the number, and then to propagate a carry bit to higher order digits. The following rules implement this process by performing hypothetical database updates:

$INC \leftarrow FIRST(x),\ CARRY(x).$

$CARRY(x) \leftarrow ONE(x),\ NEXT(x,y),\ CARRY(y)[del : ONE(x)][add : ZERO(x)]$

$CARRY(x) \leftarrow ZERO(x),\ INC[del : ZERO(x)][add : ONE(x)]$

The predicate $CARRY(x)$ represents carry bits. During top-down inference, the goal $CARRY(a_i)$ intuitively means "carry a 1 into the i^{th} digit". The first rule starts the process by creating the subgoal $CARRY(a_1)$, which carries a 1 into the first digit. The second rule then recursively generates the subgoals $CARRY(a_2)$, $CARRY(a_3)$ ..., propagating the carry bit to higher order digits.

Given the subgoal $CARRY(a_i)$, either the second or the third rule is invoked, depending on the value of the i^{th} digit. When the i^{th} digit is 1, the second rule is used. This rule changes the digit to 0 by updating the database hypothetically. The rule then generates the subgoal $CARRY(a_{i+1})$, thus propagating the carry bit to the next higher digit. The second rule thus simulates the process of adding 1 to a digit whose value is already 1.

When the i^{th} digit is 0, the third rule is used. This rule changes the digit to 1 by updating the database hypothetically. At this point, the carry bit has propagated as far as it can, and the process of incrementing the number in the database is finished.

The third rule then starts the process all over again by invoking the predicate INC. In this way, once the number in the database has been incremented, it is incremented again, recursively.

Together, the four rules above define a rulebase R for which the expression $R, DB_0 \vdash INC$ is derivable, but only after exponentially many levels of recursion.

Note that the rulebase R uses both hypothetical addition and deletion. This is crucial to its ability to recurse to exponential depth, since as we shall see in Section 4.2, rulebases that use just hypothetical addition can recurse only to polynomial depth. As we shall see in Chapter 4, this difference in recursion depths accounts for the difference in complexity between the two logics, that is, between the PSPACE completeness for hypothetical addition, and the EXPTIME completeness for hypothetical addition and deletion.

3.3 Semantics

This section develops a semantics for the inference system introduced in Section 3.1 and shows that the inference system is sound and complete with respect to this semantics.

Before considering the semantics of hypothetical inference in general, first consider the special case in which all rules are Horn and all queries are atomic. In this case, the inference system of Section 3.1 is classical, and we can appeal to the classical model theory of Horn rules [5, 48]. According to this theory, a Horn rulebase R and a database DB have a unique minimal model, or least fixpoint, denoted $lfp(R+DB)$. Furthermore $R, DB \vdash B$ *iff* $B \in lfp(R+DB)$. Posing an atomic query to the rulebase is therefore equivalent to asking if the atom is in the least fixpoint. For this reason, R and DB can be regarded as a description of the much larger database $lfp(R+DB)$, a database which we could explicitly construct if we had the time, the space, and the inclination.

The situation is more complex when we ask hypothetical queries, even when the rulebase is Horn. For instance, to answer the query $R, DB \vdash B[add : C]$, we cannot simply inquire of the fixpoint $lfp(R+DB)$. The difficulty is that the query $B[add : C]$ changes the underlying database DB *before* the rules in R are applied. To answer this query, we need the fixpoint $\mathit{lfp}(R + DB + C)$, not the fixpoint $\mathit{lfp}(R + DB)$. For other hypothetical queries, we will need other fixpoints. To answer the query $B[del : C]$, for instance, we must determine whether $B \in \mathit{lfp}[R + (DB \sim C)]$. In general, given a hypothetical query ϕ and a database DB, we first generate a new database DB', and then we pose an atomic query to the fixpoint $\mathit{lfp}(R + DB')$.

To answer hypothetical queries, it is convenient to think of a Horn rulebase R not as a description of a single database $\mathit{lfp}(R + DB)$, but as a description of a mapping that takes a database DB' as input and returns a larger database $\mathit{lfp}(R+DB')$ as output. To answer atomic queries, we need access to just one value of the mapping, $\mathit{lfp}(R + DB)$, but to answer hypothetical queries, we may need to access many of its values.

This point of view generalizes from Horn rules to hypothetical rules in a straightforward way: A hypothetical rulebase R can be thought of as specifying a mapping M. This mapping takes a database DB as input and returns a larger database $M(DB)$ as output. The rules in R impose restrictions on the mapping. For instance, the Horn rule $A \leftarrow B$ says that for every database DB, if $B \in M(DB)$ then $A \in M(DB)$. This restriction is "context free" in that the value of M at one point is independent of the values of M at other points. In contrast, the restrictions imposed by hypothetical rules are "context sensitive". For instance, the rule $A \leftarrow B[add : C]$ says that for all databases DB, if $B \in M(DB + C)$ then $A \in M(DB)$. In this way, the value of a mapping at one point is constrained by the values of the mapping at other points. Hypothetical rules can therefore impose requirements on a mapping that are impossible to express with Horn rules.

In general, there will be many mappings M that satisfy the constraints imposed by a rulebase R. We say that each such mapping is a *model* of R. This leads in a natural way to a model-theoretic definition of entailment. The rest of this section formalizes this idea and shows that the inference system of Section 3.1 is sound and complete with respect to this semantics.

Before developing this semantics, however, it is worth noting the relationship between the mappings of this chapter and the intuitionistic structures of Section 2.4. According to Definition 2.5, the essential component of any intuitionistic structure is a mapping π, which returns a (possibly infinite) database as output. The mappings developed in this chapter are both a restriction and a generalization of these intuitionistic mappings. On the one hand, the mappings of this chapter are more restrictive in that they only take a database as input, whereas intuitionistic mappings can take arbitrary inputs. On the other hand, they are more general in that they are not required to be monotonic, as intuitionistic mappings are. Because of this non-monotonicity, the semantics developed in this section can be extended in a straightforward way to a semantics for hypothetical reasoning with negation-as-failure (Chapter 5). In contrast, adding negation-as-failure to intuitionistic logic is problematic, as pointed out in [24] and [12]. Finally, it is worth pointing out, that like the models of this chapter, the canonical model developed in Section 2.4.2 is essentially a mapping from databases to databases.

3.3.1 Models and Entailment

To make the preceding discussion precise, let $\mathcal{R} = [R, dom, pred]$ be a hypothetical rulebased system. Recall that $\mathcal{DB}$ is the set of ground atomic formulas constructible from the predicate symbols in *pred* and the constant symbols in *dom*. Any database constructible from these symbols is a subset of $\mathcal{DB}$. A mapping $M : 2^{\mathcal{DB}} \rightarrow 2^{\mathcal{DB}}$ thus takes a database DB as input and returns another database $M(DB)$ as output. These mappings are the basic structures of our model theory.

Not all such mappings can be a model of the rulebase R however. In fact there are some mappings that are not a model of any rulebase. For instance, according to the inference system of Section 3.1, $R, DB \vdash A$ for any atom A in the database DB. We should therefore expect that $DB \subseteq M(DB)$ for every model M of R. The semantic development is simplified somewhat if we impose this restriction on mappings at the outset. With this in mind, we make the following definitions:

Definition 3.5 *A mapping $M : 2^{\mathcal{DB}} \rightarrow 2^{\mathcal{DB}}$ is inflationary iff $DB \subseteq M(DB)$ for all databases DB.*

Definition 3.6 *(Structures)* *A hypothetical structure is an inflationary mapping* $M : 2^{\mathcal{DB}} \rightarrow 2^{\mathcal{DB}}$.

In order to simplify the presentation, *dom* and *pred* will be implicit in the rest of this section. This fixes $\mathcal{DB}$ and the set of all hypothetical structures. All databases will be subsets of $\mathcal{DB}$, all ground atomic formulas will be elements of $\mathcal{DB}$, and all ground hypothetical queries will be constructed from atoms in $\mathcal{DB}$. Furthermore, a rulebase system $[R, dom, pred]$ will be specified solely by the rulebase R. We state these assumptions here so that we do not have to continually repeat them. One should realize, however, that the model theory of this section, like the proof theory of Section 3.1, depends on *dom* and *pred*, the domains of constant and predicate symbols, resp. Keeping this in mind, we make the following definitions:

Definition 3.7 *(Satisfaction) Let M be a hypothetical structure, let DB be a database, let $A, B_1, ..., B_k$ be ground atomic formulas, and let $\phi_1, ..., \phi_k$ be ground hypothetical queries. Then,*

$$
\begin{array}{lll}
M, DB \models_h A & \textit{iff} & A \in M(DB) \\
M, DB \models_h A[add : B_1 ... B_k] & \textit{iff} & M, DB + \{B_1, ..., B_k\} \models_h A \\
M, DB \models_h A[del : B_1 ... B_k] & \textit{iff} & M, DB \sim \{B_1, ..., B_k\} \models_h A \\
M, DB \models_h \phi_1 ... \phi_k & \textit{iff} & M, DB \models_h \phi_i \quad \textit{for each } 1 \leq i \leq k.
\end{array}
$$

If $M, DB \models_h \phi$, *then we say that M satisfies ϕ at DB.*

Definition 3.8 *Let R be a hypothetical rulebase. Then inst(R) is the set of all ground instances of the rules in R with respect to dom, that is, the result of replacing, in every possible way, the variables in R by constant symbols from dom.*

Definition 3.9 *(Models) Let R be a hypothetical rulebase, and let M be a hypothetical structure. Then M is a model of R iff $M, DB \models_h \phi_1 ... \phi_k \;\Rightarrow\; M, DB \models_h A$ for all databases DB and all rules $A \leftarrow \phi_1 ... \phi_k$ in inst(R).*

Definition 3.10 *(Entailment) Let R be a hypothetical rulebase, let DB be a database, and let ϕ be a ground hypothetical query . Then $R, DB \models_h \phi$ iff $M, DB \models_h \phi$ for all models M of R.*

Note that since *dom* and *pred* are implicit, when we write $R, DB \models_h \phi$, some rulebased system $[R, dom, pred]$ is understood.

3.3.2 Soundness and Completeness

This section establishes the soundness and completeness of the inference system of Section 3.1 with respect to the semantics developed above. That is, we prove the following two theorems:

Theorem 3.1 *(Soundness) If* $R, DB \vdash \phi$ *then* $R, DB \models_h \phi$ *for any ground hypothetical query* ϕ *and any database* DB*.*

Theorem 3.2 *(Completeness) If* $R, DB \models_h \phi$ *then* $R, DB \vdash \phi$ *for any ground hypothetical query* ϕ *and any database* DB*.*

In these two theorems, as in the rest of this section, a rulebase system $[R, dom, pred]$ is understood. Note that the proof theory and the model theory are both defined with respect to this system.

Soundness is not difficult and is an immediate consequence of the following result:

Lemma 3.3 *Let* M *be a model of a hypothetical rulebase* R*. Then, if* $R, DB \vdash \phi$ *then* $M, DB \models_h \phi$*, for any database* DB *and any ground hypothetical query* ϕ*.*

Proof: M satisfies a set of inference rules parallel to those of the inference system of Definition 3.4. In particular,

1. $M, DB \models_h A$ *if* $A \in DB$,
2. $M, DB \models_h A$ *if* $M, DB \vdash \phi_1 ... \phi_k$ *for some rule* $A \leftarrow \phi_1 ... \phi_k$ *in* $inst(R)$,
3. $M, DB \models_h A[add : B_1 ... B_k]$ *if* $M, DB + \{B_1, ..., B_k\} \models_h A$,
4. $M, DB \models_h A[del : B_1 ... B_k]$ *if* $M, DB \sim \{B_1, ..., B_k\} \models_h A$,
5. $M, DB \models_h \phi_1 ... \phi_n$ *if* $M, DB \models_h \phi_i$ *for each* $1 \leq i \leq n$.

The first inference rule follows because M is inflationary, and the second follows because M is a model of R. The remaining three inference rules follow immediately from the definition of satisfaction. □

To prove completeness, we define a hypothetical structure M_R which we call the *canonical model* of the rulebase R. This model, defined proof-theoretically, provides the necessary link between inference and semantics.

Definition 3.11 *Let R be a hypothetical rulebase. Then M_R is the hypothetical structure such that $M_R(DB) = \{A \in \mathcal{DB} \mid R, DB \vdash A\}$ for every database DB.*

Clearly, M_R is a mapping from $2^{\mathcal{DB}}$ to $2^{\mathcal{DB}}$. In addition, it is inflationary, since $R, DB \vdash A$ for all $A \in DB$. Thus, the canonical model is indeed a hypothetical structure. The following basic result says that satisfaction in this model is equivalent to inference from the rulebase R.

Lemma 3.4 *$M_R, DB \models_h \phi$ iff $R, DB \vdash \phi$ for any hypothetical query ϕ.*

Proof: There are three cases depending on whether ϕ is atomic, a hypothetical addition, or a hypothetical deletion. In each case, the proof is an immediate consequence of the definition of M_R. For instance, when ϕ is a hypothetical addition,

$$
\begin{array}{lll}
 & M_R, DB \models_h A[add: B_1 ... B_k] & \\
\textit{iff} & M_R, DB + \{B_1 ... B_k\} \models_h A & \text{by Definition 3.7,} \\
\textit{iff} & R, DB + \{B_1 ... B_k\} \vdash A & \text{by Definition 3.11,} \\
\textit{iff} & R, DB \vdash A[add: B_1 ... B_k] & \text{by Definition 3.7.}
\end{array}
$$

□

The next basic result says that the canonical model M_R is in fact a model of R.

Lemma 3.5 *M_R is a model of R.*

Proof: Let $A \leftarrow \phi_1 ... \phi_k$ be a ground instance of a rule in R. Then,

if	$M_R, DB \models_h \phi_1 ... \phi_k$	
then	$R, DB \vdash \phi_1 ... \phi_k$	by Lemma 3.4,
	$R, DB \vdash A$	by hypothetical inference, since $A \leftarrow \phi_1 ... \phi_k$ is an instance of a rule in R,
	$A \in M_R(DB)$	by Definition 3.11,
	$M_R, DB \models_h A$	by Definition 3.7.

Since this is true for every instance of every rule in R, M_R is a model of R, by Definition 3.9. □

Completeness follows immediately from these two lemmas. i.e., If $R, DB \models_h \phi$ then $M, DB \models_h \phi$ for every model M of R, by definition. In particular $M_R, DB \models_h \phi$, since M_R is a model of R. Thus $R, DB \vdash \phi$ by Lemma 3.4.

3.4 Fixpoint Semantics

Section 3.3 developed a model-theoretic semantics for hypothetical rulebases. This section complements the model theory with a fixpoint theory in the tradition of logic programming. In particular, we show that every rulebase has a unique minimal model, which can be constructed in a bottom-up fashion by repeated application of a T operator. These results are crucial to the upper complexity bounds established in Chapter 4 and to the semantics of negation developed in Chapter 5.

The development in this section follows the approach originally introduce by Apt, Van Emden and Kowalski [48, 5] and that is now standard in logic programming: We first define a lattice of structures; we then define an operator on this lattice, show that it is monotonic and continuous, and finally invoke the Tarski fixpoint theorem. This approach shows that the results are a consequence, not of the details of the logic, but of its abstract, lattice-theoretic properties. Furthermore, although we have defined hypothetical inference with respect to a finite domain, the lattice-theoretic approach does not rely on this assumption. The results of this section therefore extend to an infinite domain in a straightforward way. An approach that does rely on the finiteness of the domain can be developed using the induction techniques presented in [10]. That approach, however, is more tedious than the one presented here and does not abstract away the details of hypothetical inference.

The fixpoint theory of Horn logic is based on a lattice of first-order structures. Likewise, the fixpoint theory of hypothetical inference developed here is based on a lattice of hypothetical structures. The next definition presents the basic lattice operations. As in previous sections, we assume that the set of constant symbols *dom* and the set of predicate symbols *pred* are fixed and that all formulas are constructed from these symbols.

Definition 3.12 *Let M_1 and M_2 be hypothetical structures, and let $\mathcal{M}$ be a set of hypothetical structures. Then,*

- $M_1 \leq M_2$ *iff* $M_1(DB) \subseteq M_2(DB)$ *for every database DB.*
- $\sqcup\mathcal{M}$ *is the hypothetical structure such that* $\sqcup\mathcal{M}(DB) = \cup_{M\in\mathcal{M}}\, M(DB)$ *for every database DB.*
- $\sqcap\mathcal{M}$ *is the hypothetical structure such that* $\sqcap\mathcal{M}(DB) = \cap_{M\in\mathcal{M}}\, M(DB)$ *for every database DB.*

The operators $\sqcup$ and $\sqcap$ are called the *join* and *meet*, respectively. Note that $\leq$ is a partial order. The following basic result is straightforward.

Lemma 3.6 *The set of all hypothetical structures is a complete lattice under the partial order $\leq$ and the operations $\sqcup$ and $\sqcap$.*

Because a hypothetical structure is an inflationary mapping, the minimal element of this lattice is the identity mapping $\mathcal{I}$, where $\mathcal{I}(DB) = DB$ for all databases DB. The maximal element is the structure $M_\top$ where $M_\top(DB) = \mathcal{DB}$, the set of all ground atomic formulas. Note that $M_\top$ is a model of every hypothetical rulebase, since the head of every instance of every rule is true in this structure.

The following two results about the lattice of hypothetical structures are straightforward.

Lemma 3.7 *If* $M_1 \leq M_2$ *and* $M_1, DB \models_h \phi$ *then* $M_2, DB \models_h \phi$, *for any hypothetical query* ϕ *and any database* DB.

For each hypothetical rulebase R, we define an operator T_R. This operator takes a hypothetical structure M as input and returns another hypothetical structure $T_R(M)$ as output.

Definition 3.13 *Let* R *be a set of hypothetical rules. Then* $T_R(M)$ *is a hypothetical structure where for each database* DB, *the value of* $[T_R(M)](DB)$ *is*

$$DB \cup \{A \mid \ M, DB \models_h \phi_1 ... \phi_k \ \textit{ for some rule } A \leftarrow \phi_1 ... \phi_k \textit{ in } inst(R)\}$$

The union with the database DB is included in this definition to ensure that $T_R(M)$ is inflationary, and thus a well-defined hypothetical structure (See Definition 3.6). The following basic results about the operator T_R are straightforward.

Lemma 3.8 *Let* R *be a hypothetical rulebase, and let* M *be a hypothetical structure. Then* M *is a model of* R *iff it is a fixpoint of* T_R, *i.e., iff* $T_R(M) \leq M$.

Lemma 3.9 T_R *is monotonic; that is, if* $M_1 \leq M_2$ *then* $T_R(M_1) \leq T_R(M_2)$.

Proof: Follows immediately from Lemma 3.7. □

Since T_R is a monotonic operator on a complete lattice, the meet of any of its fixpoints is also a fixpoint. This follows immediately by the Tarski Fixpoint theorem [47], whose proof we reproduce below.

Lemma 3.10 *If* $\mathcal{M}$ *is a set of fixpoints of* T_R, *then* $\sqcap\mathcal{M}$ *is also a fixpoint of* T_R.

Proof: Note that $\sqcap\mathcal{M} \leq M$ for every M in $\mathcal{M}$. Therefore $T_R(\sqcap\mathcal{M}) \leq T_R(M) \leq M$, where the first inequality follows from the monotonicity of T_R, and the second follows because M is a fixpoint of T_R. Thus $T_R(\sqcap\mathcal{M}) \leq M$ for every M in $\mathcal{M}$, and hence $\mathrm{T}_R(\sqcap\mathcal{M}) \leq \sqcap\mathcal{M}$. □

As a special case of Lemma 3.10, the intersection of all the fixpoints of T_R is itself a fixpoint of T_R, the least fixpoint. We thus have the following result.

Theorem 3.11 *The operator T_R has a least fixpoint, denoted $lfp(T_R)$.*

Since the fixpoints of T_R are exactly the models of R, Lemma 3.10 and Theorem 3.11 can be phrased in terms of models: The meet of any collection of models of a rulebase is itself a model of the rulebase. (This generalizes the model intersection property of Horn clause logic [48, 5].) Every hypothetical rulebase also has a unique minimal model, which is just $lfp(T_R)$, the least fixpoint. As the following corollary shows, the minimal model is special, because satisfaction in this model is equivalent to logical entailment.

Corollary 3.12 $R, DB \models_h \phi$ *iff* $lfp(T_R), DB \models_h \phi$ *for any hypothetical query ϕ.*

Proof: If $R, DB \models_h \phi$ then $M, DB \models_h \phi$ for every model M of R, by definition. In particular, $lfp(T_R), DB \models_h \phi$, since $lfp(T_R)$ is a model of R. In the other direction, if $lfp(T_R), DB \models_h \phi$, then $M, DB \models_h \phi$ for any model M of R, by Lemma 3.7, since $lfp(T_R) \leq M$. Thus $R, DB \models_h \phi$. □

The next result is included for completeness. It shows that the minimal model $lfp(T_R)$ is equal to the canonical model M_R, introduced in Section 3.3.2.

Lemma 3.13 *The minimal model of R is identical to the canonical model of R.*

Proof: For any ground atomic formula A and any database DB,

	$A \in M_R(DB)$	
iff	$M_R, DB \models_h A$	by Definition 3.7,
iff	$R + DB \vdash A$	by Definition 3.11,
iff	$R + DB \models_h A$	by soundness and completeness,
iff	$lfp(T_R), DB \models_h A$	by Corollary 3.12,
iff	$A \in [lfp(T_R)](DB)$	by Definition 3.7.

Thus $[lfp(T_R)](DB) = M_R(DB)$ for all databases DB. Hence $lfp(T_R) = M_R$. □

We now show that the minimal model can be constructed in a bottom-up fashion by repeatedly applying the operator T_R to the identity mapping $\mathcal{I}$. This is not difficult to prove when the domain of constant symbols *dom* is finite. So far, however, none of the results in this chapter have used the finiteness assumption. This assumption is used extensively in the next chapter, on complexity, but we would like to avoid using it in this chapter in order to keep the semantics as general as possible. For one thing, such generality suggests that hypothetical inference can be extended to include function symbols, which results in an infinite Herbrand universe. We therefore go out of our way to use the Tarski Fixpoint theorem in proving our remaining results. The main step in this approach is showing that the operator T_R is continuous. This is the task of the next three lemmas.

Lemma 3.14 *Let $\mathcal{M}$ be a set of hypothetical structures. Then, for any database DB, and any ground hypothetical query ϕ,*

$$\sqcup\mathcal{M}, DB \models_h \phi \quad \textit{iff} \quad M, DB \models_h \phi \ \ \textit{for some } M \textit{ in } \mathcal{M}.$$

Proof: There are three cases depending on whether ϕ is atomic, a hypothetical addition, or a hypothetical deletion. In each case, the proof is straightforward. For instance, when ϕ is a hypothetical addition,

$$\begin{array}{lll}
 & \sqcup\mathcal{M}, DB \models_h A[add : B_1 ... B_k] & \\
\textit{iff} & \sqcup\mathcal{M}, DB + \{B_1, ..., B_k\} \models_h A & \text{by Definition 3.7,} \\
\textit{iff} & A \in [\sqcup\mathcal{M}](DB + \{B_1, ..., B_k\}) & \text{by Definition 3.7,} \\
\textit{iff} & A \in \cup_{M \in \mathcal{M}} \ M(DB + \{B_1, ..., B_k\}) & \text{by Definition 3.12,} \\
\textit{iff} & A \in M(DB + \{B_1, ..., B_k\}) & \text{for some } M \text{ in } \mathcal{M}, \\
\textit{iff} & M, DB + \{B_1, ..., B_k\} \models_h A & \text{by Definition 3.7,} \\
\textit{iff} & M, DB \models_h A[add : B_1 ... B_k] & \text{by Definition 3.7.}
\end{array}$$

□

Lemma 3.15 *Let $M_1 \leq M_2 \leq M_3 ...$ be an increasing sequence of hypothetical structures. Then, for any database DB, and any ground hypothetical queries $\phi_1 ... \phi_k$,*

$$\textit{If} \quad \sqcup_{i \geq 1} M_i, DB \models_h \phi_1 ... \phi_k \quad \textit{then} \quad M_{i_0}, DB \models_h \phi_1 ... \phi_k \quad \textit{for some } i_0.$$

Proof: Suppose that $\sqcup_{i \geq 1} M_i, DB \models_h \phi_1 ... \phi_k$. Then, for all $1 \leq j \leq k$,

$$\begin{array}{ll}
\sqcup_{i\geq 1} M_i, DB \models_h \phi_j & \text{by Definition 3.7,} \\
M_{i_j}, DB \models_h \phi_j & \text{for some } i_j, \text{ by Lemma 3.14,} \\
M_i, DB \models_h \phi_j & \text{for all } i \geq i_j, \text{ by Lemma 3.7, since } M_i \geq Mi_j, \\
M_{i_0}, DB \models_h \phi_j & \text{for } i_0 = max(i_1 ... i_k).
\end{array}$$

Hence $M_{i_0}, DB \models_h \phi_1 ... \phi_k$. □

Lemma 3.16 *T_R is continuous; that is, if $M_1 \leq M_2 \leq M_3 \leq \cdots$ is an increasing sequence of hypothetical structures, then $T_R(\sqcup_{i\geq 1} M_i) = \sqcup_{i\geq 1} T_R(M_i)$.*

Proof: For clarity, we use the expression $\sqcup_i$ to denote $\sqcup_{i\geq 1}$. Clearly $M_j \leq \sqcup_i M_i$ for any $j \geq 1$. Thus $T_R(M_j) \leq T_R(\sqcup_i M_i)$ by the monotonicity of T_R. Thus $\sqcup_j T_R(M_j) \leq T_R(\sqcup_i M_i)$. We must therefore prove the converse, that $T_R(\sqcup_i M_i) \leq \sqcup_j T_R(M_j)$. To do this, choose a database DB and an atom $A \in [T_R(\sqcup_i M_i)](DB)$. We must show that $A \in [\sqcup_i T_i(M_i)](DB)$. There are two cases. First, if $A \in DB$, then $A \in [T_i(M_i)](DB)$ for every i, by Definition 3.13. Hence $A \in \cup_i [T_i(M_i)(DB)] = [\sqcup_i T_i(M_i)](DB)$, by Definition 3.12. Second, if $A \notin DB$, then

$$\begin{array}{ll}
\sqcup_i M_i, DB \models_h \phi_1 ... \phi_k & \text{for some rule } A \leftarrow \phi_1 ... \phi_k \text{ in } inst(R) \text{, by Definition 3.13,} \\
M_i, DB \models_h \phi & \text{for some } i \text{, by Lemma 3.15,} \\
A \in [T_R(M_i)](DB) & \text{by the Definition 3.13,} \\
A \in \cup_i [T_R(M_i)(DB)] & \\
A \in [\sqcup_i T_R(M_i)](DB) & \text{by Definition 3.12.}
\end{array}$$

We have thus shown that $[T_R(\sqcup_i M_i)](DB) \subseteq [\sqcup_i T_R(M_i)](DB)$ for every database DB. Hence $T_R(\sqcup_i M_i) \leq \sqcup_i T_R(M_i)$, by Definition 3.12. □

Having shown that T_R is continuous, we introduce some standard definitions below. They define the structure $T_R^*(M)$, which is the result of repeatedly applying the operator T_R to the structure M. Starting from the identity mapping $\mathcal{I}$, this process constructs a sequence of larger and larger structures $T_R^0(\mathcal{I})$, $T_R^1(\mathcal{I})$, $T_R^2(\mathcal{I})$ .... Since T_R is continuous, this process converges on the least fixpoint of T_R. This follows immediately by the Tarski Fixpoint theorem [47], whose proof is reproduced in Theorem 3.18 below. We first provide some preliminary definitions and a lemma.

Definition 3.14 *If R is a hypothetical rulebase, and M a hypothetical structure, then*

- $T_R^0(M) \;=\; M$
- $T_R^{i+1}(M) \;=\; T_R[T_R^i(M)]$
- $T_R^*(M) \;=\; \sqcup_{i\geq 0}\, T_R^i(M)$

Lemma 3.17 $T_R^i(\mathcal{I}) \leq T_R^{i+1}(\mathcal{I})$

Proof: (by induction on i). To prove the base case, note that $\mathcal{I} \leq T_R(\mathcal{I})$, since $\mathcal{I}$ is the smallest element in the lattice of hypothetical structures. Thus $T_R^0(\mathcal{I}) \leq T_R^1(\mathcal{I})$. To prove the inductive step, suppose that $T_R^i(\mathcal{I}) \leq T_R^(i+1)(\mathcal{I})$. Then $T_R^{i+1}(\mathcal{I}) = T_R[T_R^i(\mathcal{I})] \leq T_R[T_R^{i+1}(\mathcal{I})] = T_R^{i+2}(\mathcal{I})$, where the inequality follows by induction hypothesis and by the monotonicity of T_R. $\Box$

Theorem 3.18 *$lfp(T_R) = T_R^*(\mathcal{I})$.*

Proof: By Lemma 3.17, $T_R^0(\mathcal{I}) \leq T_R^1(\mathcal{I}) \leq T_R^2(\mathcal{I}) \cdots$ is an increasing sequence of hypothetical structures. Thus, Lemma 3.16 is applicable:

$$\begin{array}{rcll}
T_R[T_R^*(\mathcal{I})] & = & T_R[\sqcup_{i\geq 0}\, T_R^i(\mathcal{I})] & \text{by Definition 3.14.} \\
& = & \sqcup_{i\geq 0}\, T_R[T_R^i(\mathcal{I})] & \text{by Lemma 3.16.} \\
& = & \sqcup_{i\geq 0}\, T_R^{i+1}(\mathcal{I}) & \\
& = & \sqcup_{i\geq 1}\, T_R^i(\mathcal{I}) & \\
& = & \sqcup_{i\geq 0}\, T_R^i(\mathcal{I}) & \text{since } T_R^0(\mathcal{I}) \leq T_R^1(\mathcal{I}). \\
& = & T_R^*(\mathcal{I}) &
\end{array}$$

Thus $T_R^*(\mathcal{I})$ is a fixpoint of T_R. Hence $lfp(T_R) \leq T_R^*(\mathcal{I})$ since $lfp(T_R)$ is minimal.

We thus need to show the converse, that $T_R^*(\mathcal{I}) \leq lfp(T_R)$. It is sufficient to show that $T_R^i(\mathcal{I}) \leq lfp(T_R)$ for all $i \geq 0$. This has a simple proof by induction. To prove the base case, note that $T_R^0(\mathcal{I}) = \mathcal{I}$, the smallest structure. Thus $T_R^0(\mathcal{I}) \leq lfp(T_R)$. To prove the inductive step, assume that $T_R^i(\mathcal{I}) \leq lfp(T_R)$. Thus, $T_R^{i+1}(\mathcal{I}) = T_R[T_R^i(\mathcal{I})] \leq T_R[lfp(T_R)] \leq lfp(T_R)$, where the first inequality follows by induction hypothesis and the monotonicity of T_R, and the second follows because $lfp(T_R)$ is a fixpoint. $\Box$

Example 3.3 Suppose R contains exactly two rules, $A \leftarrow B[add : C]$ and $B \leftarrow C, D$. The following table shows that it takes two applications of the operator T_R to reach a fixpoint and compute $T_R^*(\mathcal{I})$. In this table, each column represents a structure $T_R^k(\mathcal{I})$ for some k. Note that as k increases from left to right, atoms are added to each column but are never subtracted, which is a reflection of Lemma 3.17. For clarity, we show only those databases DB made from the atoms C and D.

DB	$[T_R(\mathcal{I})](DB)$	$[T_R^2(\mathcal{I})](DB)$	$[T_R^3(\mathcal{I})](DB)$
–	–	–	–
C	C	C	C
D	D	D+A	A,D
C,D	C,D+B	B,C,D+A	A,B,C,D

Atoms added by T_R are preceded by a plus sign. For instance, the $+B$ in the second column/fourth row is due to the rule $B \leftarrow C, D$. This in turn causes the addition of A to the third column/third row by the rule $A \leftarrow B[add : C]$. Formally, the first application of T_R establishes that $T_R(\mathcal{I}), \{C, D\} \models B$, and thus that $T_R(\mathcal{I}), \{D\} \models B[add : C]$. The second application then establishes that $T_R^2(\mathcal{I}), \{D\} \models A$. Notice that to compute $T_R^2(\mathcal{I})(\{D\})$, we must compute $T_R(\mathcal{I})$ on other databases first.

The atom A is eventually inferred for all databases containing D. That is, if $D \in DB$ then $T_R^*(\mathcal{I}), DB \models A$. It should therefore be the case that if $D \in DB$ then $R, DB \vdash A$. Using the inference rules of Definition 3.4, it is not hard to see that this is true. $\Box$

Chapter 4

Data Complexity

This chapter[1] establishes the data complexity of the logics developed in Chapters 2 and 3. The logic of Chapter 3, which we call the *full* logic, deals with both hypothetical additions and deletions. The logic of Chapter 2, which we call the *restricted* logic, deals only with hypothetical additions. This chapter shows that the full logic is data complete for EXPTIME,[2] and the restricted logic is data complete for PSPACE.

The proof of the EXPTIME upper-bound is a straightforward consequence of the fixpoint semantics developed in Chapter 3.4: We show that the minimal model of a rulebase can be constructed in a bottom-up fashion in exponential time. To establish the PSPACE upper-bound, we develop a mixed top-down/bottom-up algorithm. It is a recursive algorithm in which Horn-style least-fixpoint computations occur at each level of recursion. There are polynomially many levels of recursion and each level takes polynomial time; so the entire computation runs in polynomial space.

To establish the lower complexity bounds, we use hypothetical rules to encode the computations of alternating turing machines. Using just hypothetical addition, we encode APTIME machines, and using both hypothetical addition and deletion, we encode APSPACE machines. In both cases, the database encodes the machine tape, and hypothetical rules encode the machine's transition table.

Data complexity is the complexity of evaluating a database query when the query is fixed and the database is regarded as input. That is, if ψ is a database query, then its data-complexity is the complexity of the language $\{\langle \overline{x}, DB\rangle \mid \overline{x} \in \psi(DB)\}$ where

---

[1] Much of this chapter originally appeared in [8] and [10].

[2] In this chapter, and throughout this dissertation, the terms EXPTIME and "exponential time" refer to DTIME$[2^{n^{O(1)}}]$.

$\overline{x}$ is a tuple, DB is a database, and $\psi(DB)$ is the value of the query applied to DB [16, 49]. This language is called the *graph* of the query ψ. The data complexity of a set of queries is the complexity of their graphs. In particular,

Definition 4.1 *A set of queries is data complete in a complexity class $\mathcal{C}$ if (1) the graph of each query is in $\mathcal{C}$, and (2) there is some query in the set whose graph is a complete language for $\mathcal{C}$.*

A deductive database can be viewed as defining a database query if one of its predicates is reserved as an output relation. For example, let R be a Horn rulebase, and let $OUT(\overline{x})$ be the output predicate. Then R and OUT define the query whose graph is the language $\{\langle \overline{x}, DB\rangle \mid R + DB \vdash_c OUT(\overline{x})\}$ where $\vdash_c$ denotes classical inference. When the predicate OUT is implicit, we call this language the graph of R.

To extend this idea to hypothetical rulebases, we replace classical inference by hypothetical inference. In doing so, however, we must carefully define what inference system $[R, dom, pred]$ we are using. In particular, we must take into account that the set of constant symbols dom and the set of predicate symbols $pred$ are not fixed, but will increase as the database increases.

The following observation simplifies this problem somewhat: Since R is fixed, the predicates involved in inference are also fixed. Although a database DB may contain other predicates, they will not be involved in hypothetical inference, and will not contribute to the data complexity. Thus, without loss of generality, we can assume that the predicate symbols in $pred$, and their associated arities, are fixed.

In contrast, the set of constant symbols dom cannot be fixed, since it must include all the constant symbols appearing in DB. As atomic formulas are added to DB, constants may be added to dom. For the purpose of proving complexity, we make dom a function of DB. It is convenient to use $dom = dom(R) + dom(DB)$, where $dom(DB)$ is the data domain of DB, and $dom(R)$ is the set of constant symbols appearing in R. With this convention, a rulebase R and a predicate OUT uniquely define a database query, the query whose graph is $\{\langle \overline{x}, DB\rangle \mid R, DB \vdash OUT(\overline{x})\}$. When the predicate OUT is implicit, we call this language the graph of the rulebase R.

This chapter establishes the complexity of these graphs when R is a hypothetical rulebase. Section 4.1 first shows that each such graph is in EXPTIME. Section 4.3 then constructs a particular rulebase whose data complexity is complete for EXPTIME. Together, these results establish Theorem 4.1, below. In this theorem, as elsewhere, the phrase "complexity of a set of rulebases" means the complexity of their graphs.

Theorem 4.1 *Let $\mathcal{R}$ be the set of rulebases of hypothetical additions and deletions. $\mathcal{R}$ is data complete for EXPTIME.*

his result applies to the unrestricted logic of hypothetical additionas and deletions. Sections 4.2 and 4.4 establish complexity results for the restricted logic of hypothetical additions:

Theorem 4.2 *Let $\mathcal{R}^+$ be the set of rulebases of hypothetical additions. $\mathcal{R}^+$ is data complete for PSPACE.*

The upper complexity bounds for both theorems are established first, and then the lower bounds.

4.1 EXPTIME Upper-Bound

This section shows that query processing in the full logic of hypothetical additions and deletions can be done in exponential time, thus establishing the upper complexity bound of theorem 4.1. The proof is based on the fixpoint semantics developed in chapter 3.4. In particular, we show that the least fixpoint of a rulebase can be constructed in exponential time. The upper complexity bound then follows immediately, since by corollary 3.12, any query can be directly evaluated from the least fixpoint.

Chapter 3.4 showed that the least fixpoint $T_R^*(\mathcal{I})$ of a rulebase R can be computed by repeatedly applying the the operator T_R to the minimal structure $\mathcal{I}$ in a bottom-up fashion. We show that this procedure runs in exponential time (exponential in the size of the data domain). The proof has three main ideas: (i) any hypothetical structure M can be represented in exponential space, (ii) each application of the operator T_R takes exponential time, and (iii) at most exponentially many applications of T_R are needed to compute the least fixpoint.

To be more precise, let $\mathcal{R} = [R, dom, pred]$ be a hypothetical rulebased system. dom and $pred$ are finite sets of constant and predicate symbols, respectively. Fixing $pred$, we show that $T_R^*(\mathcal{I})$ can be computed in time which is exponential in $|dom|$. This translates into exponential data-complexity. i.e. Since $dom = dom(R) + dom(DB)$, where $dom(R)$ is fixed, "exponential in $|dom|$" implies "exponential in $|dom(DB)|$".

Recall that $\mathcal{DB}$ is the set of all ground atomic formulas constructible from dom and $pred$. Two facts about $\mathcal{DB}$ are important: (i) every database constructed from the symbols in dom and $pred$ is a subset of $\mathcal{DB}$, and (ii) the size of $\mathcal{DB}$ is polynomial in the size of dom. (Fact (ii) follows from the fixed arity of the predicate symbols in $pred$.)

We represent a hypothetical structure M by the set $\{\langle DB', M(DB')\rangle \mid DB' \subseteq \mathcal{DB}\}$. In fact, we shall identify M with this set. To compute $T_R^*(\mathcal{I})$, M is first initialized to $\mathcal{I}$ by the assignment $M \leftarrow \{\langle DB', DB'\rangle \mid DB' \subseteq \mathcal{DB}\}$. The assignment $M \leftarrow T_R(M)$ is then executed repeatedly until M no longer changes, at which point $M = T_R^*(\mathcal{I})$.

The complexity of this procedure is easily determined, first as a function of $|\mathcal{DB}|$, and then as a function of $|dom|$. First, there are $2^{|\mathcal{DB}|}$ subsets of $\mathcal{DB}$. The representation of M therefore uses $O(2^{|\mathcal{DB}|})$ space, and each application of the operator T_R takes $2^{O(|\mathcal{DB}|)}$ time. Second, as shown in the proof of theorem 3.18, $T_R^i(\mathcal{I}) \leq T_R^{i+1}(\mathcal{I})$. Thus, each operator application makes M larger. Therefore, during each application, except the last, there is some element $\langle DB', S\rangle$ of M in which S acquires a new atomic formula. For each DB', however, this can happen at most $|\mathcal{DB}|$ times before S saturates and equals $\mathcal{DB}$. Thus, for each of the $2^{|\mathcal{DB}|}$ elements $\langle DB', S\rangle$ in M, the set S changes at most $|\mathcal{DB}|$ times. Therefore, after $|\mathcal{DB}| \times 2^{|\mathcal{DB}|}$ applications of the operator T_R, no more changes are possible, and a fixpoint is reached. Since each of these applications takes $2^{O(|\mathcal{DB}|)}$ time, $T_R^*(\mathcal{I})$ is computed in $2^{O(|\mathcal{DB}|)} \times |\mathcal{DB}| \times 2^{|\mathcal{DB}|}$ time, that is, in $2^{O(|\mathcal{DB}|)}$ time. The size of $\mathcal{DB}$, however, is polynomial in the size of dom. Thus, $T_R^*(\mathcal{I})$ is computed in $2^{|dom|^{O(1)}}$ time, that is, in time which is exponential in the size of the domain dom.

Finally, given $T_R^*(\mathcal{I})$ and a formula $OUT(\overline{x})$ in $\mathcal{DB}$, one can use definition 3.7 to determine in exponential time whether $T_R^*(\mathcal{I}), DB \models OUT(\overline{x})$. Thus, the data-complexity of the full logic of hypothetical additions and deletions is in EXPTIME.

4.2 PSPACE Upper-Bound

This section establishes the upper complexity bound for the restricted logic of hypothetical additions. In particular, this section shows that query evaluation for the restricted logic can be done in polynomial space (polynomial in the size of the database).

Instead of computing the entire mapping $T_R^*(\mathcal{I})$, in this section, we compute only the value of the mapping at the database DB, that is, we compute $[T_R^*(\mathcal{I})](DB)$. This value is a set of ground atomic formulas and is computed by an iterative computation much like the bottom-up, least-fixpoint computation of Horn logic. Unlike Horn logic, however, whenever a hypothetical rule is applied, we need an answer to a query on a hypothetical database DB'. To get this answer, we compute $[T_R^*(\mathcal{I})](DB')$ recursively. Thus, the computations of $T_R^n(\mathcal{I})$ on different databases are not synchronized, as in the previous section. The computation is implemented by a mixed iterative/recursive procedure LFP(R, DB). This procedure computes $[T_R^*(\mathcal{I})](DB)$ iteratively, and calls itself recursively to compute $[T_R^*(\mathcal{I})](DB')$ whenever needed.

Each recursive call to LFP invokes a new iterative phase in the computation. Starting with $S = DB$, atoms are added to S by repeatedly applying the rules in R until no more inferences are possible. For each rule, the conclusion is added to S iff the premise is satisfied. This is comparable to the bottom-up, least fixpoint computations of Horn logic. Hypothetical rules need special treatment however, and for each such rule, LFP calls itself recursively, invoking another iterative phase. For instance, to test the premise of the rule $A \leftarrow B[add : C]$, a new, hypothetical database $DB' = DB + \{C\}$ is created, and the rules in R are applied to DB' iteratively to saturation. If B is inferred during this phase, then $R, DB + \{C\} \vdash B$, so $R, DB \vdash B[add : C]$, so the premise of the hypothetical rule is satisfied.

LFP(R, DB) can be computed in polynomial space. First, at each level of recursion, the iterative computation takes polynomial time (as with function-free Horn logic). Second, because every hypothetical rule is an addition, the hypothetical database grows larger with each level of recursion. i.e., DB' is always larger than DB. Since there are only polynomially many distinct tuples that can be added to a database, recursion will go only to polynomial depth. LFP(R, DB) thus has polynomially many levels of recursion, each taking polynomial time, so the entire computation can be done in polynomial space.

This description of LFP is made precise by the three procedures below. For clarity, they are restricted to the propositional case, the predicate case being a straightforward generalization. The procedures implement the following inference system, which is equivalent to the inference system of chapter 2.3 in the propositional case.

1. $R, DB \vdash A$ *if* $A \in DB$

2. $R, DB \vdash A$ *if there is a rule* $A \leftarrow \phi_1 ... \phi_k$ *in* R *such that* $R, DB \vdash \phi_i$ *for each* i.

3. (a) $R, DB \vdash A[add: B_1 ... B_k]$
   if $\{B_1, ..., B_k\} \subseteq DB$ *and* $R, DB \vdash A$

   (b) $R, DB \vdash A[add: B_1 ... B_k]$
   if $\{B_1, ..., B_k\} \not\subseteq DB$ *and* $R, DB + \{B_1, ..., B_k\} \vdash A$

Inference rule 2 is implemented by the procedures LFP(R, DB) and T(R, DB, S). T applies each rule in R to DB exactly once, following Definition 3.13, and LFP calls T iteratively until saturation. Together, LFP and T execute a single iterative phase. Inference rules 1 and 3 are implemented by the procedure MATCH(R, DB, S, ϕ), which determines whether $R, DB \vdash \phi$. MATCH tests the premise of a rule, invoking a new level of recursion, and a new iterative phase, if necessary.

In these procedures, the S variables represent the inferences so far. More formally, they represent the value of $M(DB)$ for some hypothetical structure M. For instance, on the first call to the procedure T, the input variable S_1 represents $\mathcal{I}(DB)$, and the output variable S_2 represents $[T_R(\mathcal{I})](DB)$. More generally, on the $i+1^{st}$ call to T, S_1 represents $[T_R^i(\mathcal{I})](DB)$, and S_2 represents $[T_R^{i+1}(\mathcal{I})](DB)$.

```
Procedure: LFP(R, DB)
    S1 ← DB;
    S2 ← T(R, DB, S1);
    do until S1 = S2:
        S1 ← S2;
        S2 ← T(R, DB, S1);
    end;
    return(S2);

Procedure: T(R, DB, S1)
    S2 ← DB;
    for each rule A ← φ1...φm  in R do:
        if  MATCH(R, DB, S1, φj)  is true for each 1 ≤ j ≤ m
            then  S2 ← S2 ∪ {A};
    end;
    return(S2);

Procedure: MATCH(R, DB, S, φ)
    if φ is atomic  and  φ ∈ S  then  return(true);
    if φ = A[add : B1...Bk]  and  {B1, ..., Bk} ⊆ DB  and  A ∈ S
       then  return(true);
    if φ = A[add : B1...Bk]  and  {B1, ..., Bk} ⊈ DB
                              and  A ∈ LFP(R, DB ∪ {B1, ..., Bk})
       then  return(true);
       else  return(false);
```

It is not difficult to see that these procedures are correct, i.e., that $\mathrm{LFP}(R, DB) = [T_R^*(\mathcal{I})](DB)$ for any database DB. This is trivially true for the largest database $\mathcal{DB}$, consisting of all ground atomic formulas. In this case $\mathrm{LFP}(R, \mathcal{DB}) = \mathcal{DB} = [T_R^*(\mathcal{I})](\mathcal{DB})$. For any other database DB, if $\mathrm{LFP}(R, DB') = [T_R^*(\mathcal{I})](DB')$ for all DB'

containing DB, then $\text{LFP}(R, DB) = [T_R^*(\mathcal{I})](DB)$. Correctness thus follows by induction over DB.

To check the complexity, suppose that MATCH were an oracle taking constant time. LFP would then return after polynomial time, as in function-free Horn logic (Datalog). In fact, however, MATCH calls LFP recursively. But this recursion can only go to depth $|\mathcal{DB}|$ before DB saturates and equals $\mathcal{DB}$. Thus, since $\mathcal{DB}$ has polynomial size, there are only polynomially many levels of recursion, each taking polynomial time; so the entire computation can be done in polynomial space.

4.3 EXPTIME Lower-Bound

This section establishes the lower complexity bound for the full logic of hypothetical additions and deletions. In particular, it constructs a rulebase whose graph is EXPTIME-hard. The idea is to use the logic to encode the computations of an alternating PSPACE machine. Since APSPACE=EXPTIME, the result follows [19]. In addition, these machine encodings are central to the expressiveness result of section 6.4.

Suppose M is a one-tape, alternating Turing-machine which runs in polynomial space (an APSPACE machine) and $\overline{s}$ is an input string. We shall encode $\overline{s}$ as a database $DB(\overline{s})$, and M as a rulebase $R(M)$ so that

$$R(M), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s}. \tag{4.1}$$

where $ACCEPT$ is a 0-ary predicate symbol.

The important point is that the rulebase $R(M)$ is independent of the input $\overline{s}$. This allows us prove that the data-complexity of query processing is EXPTIME-hard. In particular, letting M be an APSPACE machine which recognizes an EXPTIME-complete language,[3] the result follows immediately. Combining this with the EXPTIME upper-bound of section 4.1 completes the proof of theorem 4.1.

The rest of this section describes the construction of the database $DB(\overline{s})$ and the rulebase $R(M)$.

[3] see [20] for examples of such machines.

4.3.1 Building the Database $DB(\overline{s})$

Since M is an APSPACE machine, there is an integer k such that M uses n^k tape cells, where n is the length of the input $\overline{s}$. A counter is therefore needed to represent these n^k positions, and it will be convenient to have a counter from 0 to $n^k - 1$. This is easily implemented by placing the following entries into $DB(\overline{s})$:

$$FIRST(0),\ NEXT(0,1),\ NEXT(1,2),\ NEXT(2,3)\ ...$$
$$...\ NEXT(n^k-2, n^k-1),\ LAST(n^k-1).$$

Given this counter, we can represent any configuration of the machine M. (Such a configuration is often is called an *id*.) To represent these configurations, we introduce the predicates below, one predicate for each symbol c in the tape alphabet, and one for each state q in the finite control:

- $CELL^c(j)$ means that the tape cell at position j contains the symbol c.
- $CONTROL^q(j)$ means that the finite control is in state q and is scanning the cell at position j.

It might appear that we are using more predicate symbols than necessary. For instance, we would need only two predicates $CELL(c,j)$ and $CONTROL(q,j)$ if we used constant symbols to represent tape symbols and control states. This representation is quite adequate for proving a lower complexity bound. However, the constructions of this section are also central to the expressibilty results of chapter 6, which require a rulebase that is constant-free. By using superscripts, constant symbols are easily avoided.

Given this representation, we must specify the initial *id* of the machine M. This means stating that the finite control is in state q_0, that the tape head is scanning the first tape cell, and that the input is the string $\overline{s} = \langle s_0, s_1, ..., s_{n-1}\rangle$. The i^{th} tape cell therefore contains the symbol s_i, for $i < n$, and all other tape cells are blank. This initialization is encoded by placing the following entries in the database $DB(\overline{s})$:

$$CONTROL^{q_0}(0),$$
$$CELL^{s_0}(0),\ CELL^{s_1}(1)\ , CELL^{s_2}(2)\ \ ...\ \ CELL^{s_{n-1}}(n-1),$$
$$CELL^b(n),\ CELL^b(n+1),\ CELL^b(n+2)\ \ ...\ \ CELL^b(n^k-1).$$

where b denotes a blank.

4.3.2 Building the Rulebase $R(M)$

During inference, the rulebase $R(M)$ uses hypothetical operations to simulate the action of the tape head and the finite control. For instance, tape erasure is simulated with hypothetical deletion, and tape writing, with hypothetical addition.[4] More generally, $R(M)$ simulates the transition of M from one *id* to another by shifting inference from one hypothetical database to another. Indeed, at every stage of inference, the database represents a machine *id*.

Besides defining the various computations that M performs, the rulebase also specifies which computations accept the input, and which do not. For alternating machines, acceptance is conveniently defined in terms of *accepting id*'s. Whether an *id* is accepting depends on its successor configurations and on the state of the control head. An alternating machine has three kinds of states: *accepting*, *universal* and *existential*. With these states, acceptance is defined recursively as follows:

- An *id* is accepting if it has an accepting state.
- An *id* is accepting if it has a universal state and all its successor *id*'s are accepting.
- An *id* is accepting if it has an existential state and at least one of its successor *id*'s is accepting.

An alternating machine accepts its input iff its initial *id* is accepting.

This criterion for acceptance, along with the machine's transition relation, is encoded into the rulebase $R(M)$. In particular, $R(M)$ is constructed so that if the database DB represents an *id*, then

$$R(M), DB \vdash ACCEPT \quad \textit{iff} \quad DB \textit{ represents an accepting id.} \tag{4.2}$$

But $DB(\overline{s})$ represents the initial *id*. Therefore,

$$R(M), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s}$$

[4]This is comparable to the use by Abiteboul and Vianu of database update languages to simulate non-deterministic Turing machines [1].

which proves theorem 4.1.

4.3.3 Defining the Predicate $ACCEPT$

This section defines the predicate $ACCEPT$ so that it satisfies condition (4.2). The idea is to encode the transition relation of machine M as a set of hypothetical rules. Three types of rules are distinguished, one for each type of machine state: universal, existential and accepting.

The simplest rules are for accepting states. For each accepting state, we add the following rule to $R(M)$:

$$ACCEPT \leftarrow CONTROL^{q_a}(j).$$

That is, an *id* is accepting if it contains the accepting state q_a. Since the variable j does not appear in the head of this rule, the position of the tape head is unimportant.

The rules for universal and existential states are more complex and depend on the transition relation of M. For each machine transition, there are three pieces of information that may change: q, the state of the finite control; j, the position of the tape head; and c, the tape symbol at position j. This information is encoded in the database by the formulas $CONTROL^q(j)$ and $CELL^c(j)$. For each machine transition, we delete these two formulas from the database, and add two new ones. For each c and q, the deletions are taken care of the following rule, where $TRANS^{cq}$ is a new predicate symbol:

$$\begin{aligned} ACCEPT \quad \leftarrow \quad & CONTROL^q(j),\ CELL^c(j), \\ & TRANS^{cq}(j)[del: CONTROL^q(q,j),\ CELL^c(c,j)]. \end{aligned}$$

where j is a variable.

The rules defining the predicate $TRANS^{cq}$ complete the machine transition by adding new formulas to the database. These rules encode the details of M's transition relation. For instance, given symbol c and state q, suppose that the machine has k possible transitions. Suppose that the i^{th} transition is given by the triple $\langle c_i,\ \Delta_i,\ q_i\rangle$, which means *write the symbol* c_i*, move* Δ_i *squares to the right, and go into state* q_i, where Δ_i may be 1, 0 or -1.

If q is a universal state, then a machine id containing q is accepting iff each of its k successor id's is accepting. This is expressed by the following rule, which we add to the rulebase $R(M)$:

$$\begin{aligned} TRANS^{cq}(j) \quad \leftarrow \quad & j_1 = j + \Delta_1,\ ACCEPT\,[add: CONTROL^{q_1}(j_1),\ CELL^{c_1}(j)], \\ & j_2 = j + \Delta_2,\ ACCEPT\,[add: CONTROL^{q_2}(j_2),\ CELL^{c_2}(j)], \\ & \quad \ldots \\ & j_k = j + \Delta_k,\ ACCEPT\,[add: CONTROL^{q_k}(j_k),\ CELL^{c_k}(j)]. \end{aligned}$$

where the j's are variables.

On the other hand, if q is an existential state, then a machine id containing q is accepting iff at least one of its successor id's is accepting. This is expressed by the following k rules, which we add to $R(M)$:

$$\begin{aligned} TRANS^{cq}(j) \quad &\leftarrow \quad j_1 = j + \Delta_1,\ ACCEPT\,[add: CONTROL^{q_1}(j_1),\ CELL^{c_1}(j)]. \\ TRANS^{cq}(j) \quad &\leftarrow \quad j_2 = j + \Delta_2,\ ACCEPT\,[add: CONTROL^{q_2}(j_2),\ CELL^{c_2}(j)]. \\ &\ldots \\ TRANS^{cq}(j) \quad &\leftarrow \quad j_k = j + \Delta_k,\ ACCEPT\,[add: CONTROL^{q_k}(j_k),\ CELL^{c_k}(j)]. \end{aligned}$$

where the j's are variables. The expressions $j_i = j + \Delta_i$ are abbreviations for $NEXT(j, j_i)$ or $NEXT(j_i, j)$, depending on whether Δ_i is 1 or -1, resp.

These rules encode the transition relation of the machine and the criterion for acceptance, so that the resulting rulebase, $R(M)$, satisfies condition (4.2).

4.4 PSPACE Lower-Bound

This section establishes the lower complexity bound for the restricted logic of hypothetical additions. In particular, it constructs a rulebase whose graph is PSPACE-hard. The proof is similar to that of the previous section except that instead of encoding alternating PSPACE machines, we encode alternating PTIME machines. Since APTIME=PSPACE, the result follows [19]. The machine encodings are also central to the expressiveness result of section 6.3.

The main difference between the encodings in this section and the last is that time is now represented explicitly. Without hypothetical deletion, tape erasure can no longer be simulated directly. It can, however, be simulated indirectly, by introducing a time parameter t into some of the predicates. In particular, the machine can be represented with the following predicates:

- $CELL^c(j,t)$ means that at time t, the tape cell at position j contains symbol c.
- $CONTROL^q(j,t)$ means that at time t, the finite control is in state q and is scanning the cell at position j.

For each value of t, these predicates define a machine *id*.

As in section 4.3.1, a successor predicate $NEXT(i,j)$ is stored in the database $DB(\overline{s})$ so that we can count from 0 to $n^k - 1$, where n is the length of $\overline{s}$. This time, however, the counter is used to keep track of computation steps (time) as well as tape positions (space). We can therefore simulate machines which halt in at most n^k steps, that is, machines which run in polynomial *time*.

Suppose, then, that M is a one-tape, alternating Turing-machine which runs in polynomial time (an APTIME machine), and that $\overline{s}$ is an input string. We encode $\overline{s}$ as a database $DB(\overline{s})$, and M as a rulebase $R(M)$ so that

$$R(M), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s}. \tag{4.3}$$

Again, the important point is that the rulebase $R(M)$ is independent of the input $\overline{s}$. This allows us to prove that the data-complexity of query processing is PSPACE-hard. In particular, letting M be an APTIME machine which recognizes a PSPACE-complete language,[5] the result follows immediately. Combining this with the PSPACE upper-bound of section 4.2 completes the proof of theorem 4.2.

The database $DB(\overline{x})$ and the rulebase $R(M)$ are constructed much as they were in section 4.3 for the full logic. There are, however, some important differences due to the introduction of the time variable t. The rest of this section describes these differences.

[5] See [26] for examples of such languages.

4.4.1 The Database $DB(\overline{s})$ and the Rulebase $R(M)$

Because time is now represented explicitly, the database can encode more than a single machine *id*. In fact, it can encode a sequence of *id*'s, that is, a *computation path.* An alternating Turing machine M can have many such paths, which together form an computation tree. During inference, the rulebase $R(M)$ explores this tree one path at a time. This process can be viewed as a recursive, depth-first search: each path is inserted into the database hypothetically, tested, and then retracted. The process of inserting a path is incremental. Starting with the initial *id*, the path is extended by hypothetically inserting one new *id* at a time.

The database $DB(\overline{s})$ encodes the initial *id*. That is, it states that the finite control is in state q_0, that the tape head is scanning the first tape cell, and that the input to the machine is the string $\overline{s} = \langle s_0, s_1, ..., s_{n-1}\rangle$. The i^{th} tape cell therefore contains the symbol s_i, for $i < n$, and all other tape cells are blank. This initialization is encoded by placing the following entries into the database $DB(\overline{s})$:

$$
\begin{aligned}
&CONTROL^{q_0}(0,0),\\
&CELL^{s_0}(0,0),\ CELL^{s_1}(1,0),\ CELL^{s_2}(2,0)\ \ ...\ \ CELL^{s_{n-1}}(n-1,0),\\
&CELL^{b}(n,0),\ CELL^{b}(n+1,0),\ CELL^{b}(n+2,0)\ \ ...\ \ CELL^{b}(n^k-1,0).
\end{aligned}
$$

This is identical to the encoding given in section 4.3.1 except for the presence of a third argument in each entry, representing time and whose value is set to 0. Furthermore, we view this not just as a single *id*, but as a sequence of *id*'s of length one.

This encoding assumes the existence of a counter from 0 to $n^k - 1$. As in section 4.3.1, this counter is encoded in $DB(\overline{s})$ by using a successor predicate $NEXT(i,j)$.

The rulebase $R(M)$ encodes the transition relation of machine M and the criterion for acceptance. In effect, it generates and tests all the computation paths of M. It does this by defining a predicate $ACCEPT(t)$. This predicate determines whether the *id* at the end of a computation path is accepting. In particular, if DB encodes a computation path ending at time t, then

$$R(M), DB \vdash ACCEPT(t) \quad \textit{iff} \quad \textit{the id at time } t \textit{ is accepting} \tag{4.4}$$

But $DB(\overline{s})$ represents the initial *id*, that is, a computation path of length 1 ending at time 0. Therefore,

$$\begin{aligned} R(M), DB(\overline{s}) \vdash ACCEPT(0) \quad & \textit{iff} \quad \textit{the initial id is accepting} \\ & \textit{iff} \quad M \textit{ accepts } \overline{s} \end{aligned}$$

We therefore add the following rule to $R(M)$:

$$ACCEPT \leftarrow ACCEPT(0)$$

Thus, by providing rules which define the predicate $ACCEPT(t)$, formula (4.3) can be established, thereby proving theorem 4.2.

4.4.2 Defining the Predicate $ACCEPT(t)$

To define the predicate $ACCEPT(t)$, we encode the transition relation of machine M as a set of hypothetical rules. These are a variation of the rules in section 4.3.3. Again, three types of rules are defined, one for each type of machine state: universal, existential and accepting.

For each accepting state, we add the following rule to $R(M)$:

$$ACCEPT(t) \leftarrow CONTROL^{q_a}(j,t).$$

That is, an *id* is accepting if it contains the accepting state q_a. The variable t records the time at which the accepting state is reached, and since the variable j does not appear in the head of this rule, the position of the tape head is unimportant.

The rules for universal and existential states are more complex and depend on the transition relation of the machine M. For each such transition, there are four pieces of information that may change: q, the state of the finite control; j, the position of the tape head; c, the tape symbol at position j; and t, the time associated with the current *id*. This information is encoded in the database by the formulas $CONTROL^q(j,t)$ and $CELL^c(j,t)$. For each c and q, the following rule extracts this information from the database and increments t by one, where $TRANS^{cq}$ is a new predicate symbol:

$$ACCEPT(t) \quad \leftarrow \quad CONTROL^q(j,t),\ CELL^c(j,t),\ NEXT(t,t'),\ TRANS^{cq}(j,t').$$

where t, j and t' are variables. t' represents the next moment in time, i.e., the time of the successsor *id*. Notice that this rule is not hypothetical. In particular, it does not perform any deletions, in contrast to the corresponding rule of section 4.3.3.

The rules defining the predicate $TRANS^{cq}$ perform the machine transitions, adding control information about the successor *id*'s to the database. These rules encode the details of the transition relation. For instance, given symbol c and state q, suppose that the machine has k possible transitions. Suppose that the i^{th} transition is given by the triple $\langle c_i,\ \Delta_i,\ q_i \rangle$, which means *write the symbol* c_i*, move* Δ_i *squares to the right, and go into state* q_i, where Δ_i may be 1, 0 or -1. If q is a universal state, then the machine *id* is accepting iff each of the k successor *id*'s is accepting. This is expressed by the following rule, which we add to $R(M)$:

$$\begin{aligned}
TRANS^{cq}(j,t) \ \leftarrow\ \ & j_1 = j + \Delta_1,\ ACCEPT(t)\,[add: CONTROL^{q_1}(j_1,t), CELL^{c_1}(j,t)],\\
& j_2 = j + \Delta_2,\ ACCEPT(t)\,[add: CONTROL^{q_2}(j_2,t), CELL^{c_2}(j,t)],\\
& \quad \ldots\\
& j_k = j + \Delta_k,\ ACCEPT(t)\,[add: CONTROL^{q_k}(j_k,t), CELL^{c_k}(j,t)].
\end{aligned}$$

where the t and the j's are variables.

On the other hand, if q is an existential state, then a machine *id* containing q is accepting iff at least one of its successor *id*'s is accepting. This is expressed by the following k rules, which we add to $R(M)$:

$$\begin{aligned}
TRANS^{cq}(j,t) \ &\leftarrow\ \ j_1 = j + \Delta_1,\ ACCEPT(t)\,[add: CONTROL^{q_1}(j_1,t), CELL^{c_1}(j,t)].\\
TRANS^{cq}(j,t) \ &\leftarrow\ \ j_2 = j + \Delta_2,\ ACCEPT(t)\,[add: CONTROL^{q_2}(j_2,t), CELL^{c_2}(j,t)].\\
& \qquad \ldots\\
TRANS^{cq}(j,t) \ &\leftarrow\ \ j_k = j + \Delta_k,\ ACCEPT(t)\,[add: CONTROL^{q_k}(j_k,t), CELL^{c_k}(j,t)].
\end{aligned}$$

where the t and the j's are variables.

With such rules, we encode both the transition relation of machine M and the criterion for acceptance.

4.4.3 Encoding the Frame Axiom

The above rules determine the *changes* caused to a *id* by a machine transition. The greater part of an *id*, however, remains unchanged by such transitions. Indeed, except for the tape cell under the tape head, the contents of all tape cells are unchanged. This is an instance of the *frame axiom* [32], and we must write rules to encode it. Such rules are necessary only because we are representing time explicitly; i.e., the database represents a sequence of *id*'s, and rules are needed to copy the unchanged portion of an *id* from one instant of time to the next.

In the following rule, the premise identifies those tape cells which contain the symbol c and which are not under the tape head at time t_1. The conclusion then places a c into these cells at time $t_2 = t_1 + 1$.

$$CELL^c(j, t_2) \quad \leftarrow \quad NEXT(t_1, t_2),\ CELL^c(j, t_1),\ CONTROL^q(j', t_1),\ NEQ(j, j').$$

This rule assumes that the finite control is in state q. We therefore write such a rule for each control state q and each tape symbol c.

These rules invoke the predicate $NEQ(x, y)$ which is true iff x and y represent distinct tape positions. NEQ is implemented by the following rules, which exploit the fact that $x \neq y$ iff $x < y$ or $y < x$:

$$\begin{aligned}
&NEQ(x, y) \leftarrow BEFORE(x, y). \\
&NEQ(x, y) \leftarrow BEFORE(y, x). \\
&BEFORE(x, y) \leftarrow NEXT(x, y). \\
&BEFORE(x, z) \leftarrow NEXT(x, y),\ BEFORE(y, z).
\end{aligned}$$

Combined with those of the previous two sections, these rules complete the definition of the rulebase $R(M)$, thereby establishing formula (4.4).

Chapter 5

Negation as Failure

Although the logic developed in Chapter 3 is EXPTIME complete, there are some simple, low-complexity queries that it cannot express. This is because the logic is *monotonic*: As the database expands, the answer to a query also expands. This behavior is typical of many logical inference systems, such as Horn logic, full classical logic, and modal logics. Such systems cannot express non-monotonic queries, such as relational algebra queries involving complementation. Retrieving those students who are *not* eligible to graduate is an example of a non-monotonic query. To capture such queries in the framework of hypothetical reasoning, this chapter[1] augments our hypothetical logic with a well-known non-monotonic operator: *negation-as-failure*. This is a natural extension to the logic since any practical logic-programming language must have this operator. Furthermore, as we shall see in Chapter 6, this operator allows the logic to express *all* the queries in EXPTIME, both monotonic and non-monotonic.

In the extended logic, hypothetical rules may have negated premises. Thus, rules of the form $A \leftarrow \sim B[add : C]$ are allowed. The expression $\sim B[add : C]$ is interpreted as the failure to prove $B[add : C]$. Thus, A is inferred if $B[add : C]$ cannot be inferred. This semantics is similar to the semantics of negation as commonly defined in Horn logic programming. Indeed, much of this chapter extends the theory of negation-as-failure from Horn rules to hypothetical rules, generalizing several important techniques.

A major difficulty with negation-as-failure is that it is not always well-defined. This is true both for Horn rulebases and for hypothetical rulebases. For instance, given the two rules $A \leftarrow \sim B$ and $B \leftarrow \sim A$, it is unclear whether A is to be inferred, or B, or both, or neither. Because there is no consensus on how such rulebases should be

[1] Much of this chapter originally appeared in [10].

treated, classes of Horn rulebases have been identified in which such problems do not arise. Perhaps the best known of these classes is the *stratified rulebases*. These rulebases are layered, and within each layer, a negated premise refers only to rules found in the layers below [4]. In this way, recursion never occurs through a negated premise and the semantics of negation is always well-defined.

This chapter extends the theory of stratified negation from Horn rules to hypothetical rules. First, the notion of a stratified hypothetical rulebase is defined. As in Horn logic, a rulebase can be stratified in many different ways. The semantics of a stratified rulebase is then developed, and some care is taken to show that this semantics is independent of the way in which the rulebase is stratified, as in the Horn case.

The complexity results of Chapter 4 are then extended to stratified hypothetical rulebases. In particular, it is shown that stratified negation does not increase the data complexity of hypothetical rules: The full logic of hypothetical additions and deletions remains EXPTIME complete, and the restricted logic of hypothetical additions remains PSPACE complete.

Example 5.1 The rules below are part of a stratified hypothetical rulebase that defines a student's eligibility for financial aid.[2] Intuitively, a student s is eligible for a stipend if he is a near-graduate but not a graduate. On the other hand, if he is neither a graduate nor a near-graduate, then he is eligible for a fellowship.

$$\begin{aligned} &\mathit{stipend}(s) \leftarrow \mathit{admitted}(s),\ \mathit{near\_grad}(s),\ \sim \mathit{grad}(s). \\ &\mathit{fellowship}(s) \leftarrow \mathit{admitted}(s),\ \sim \mathit{near\_grad}(s),\ \sim \mathit{grad}(s). \\ &\mathit{near\_grad}(s) \leftarrow \mathit{grad}(s)[\mathit{add} : \mathit{take}(s,c)] \end{aligned}$$

In applying the rule for $\mathit{near\_grad}(s)$, we ask if the student s is within one course of graduation. That is, is there some course c such that, if $\mathit{take}(s,c)$ were assumed to be true, then $\mathit{grad}(s)$ would also be true? The student is eligible for a fellowship only if this hypothetical test fails. Conversely, he is eligible for a stipend only if it succeeds.

[2]This example, taken from [12], is an improvement by McCarty on an example originally due to Bonner [8, 10].

Note that this hypothetical test is vacuously true if $grad(s)$ is true (as long as there exists a course somewhere in the database!). But we do not want to give a stipend to a student who has already graduated, so we include the test $\sim grad(s)$ is the rule for $stipend(s)$. This means that a stipend is available only to those students who need *exactly* one course to graduate. □

5.1 Stratification

This section gives a precise definition of stratified hypothetical rulebases. To simplify the definition, we first introduce another concept, that of *goal predicates.*

Suppose that $A \leftarrow \phi_1 ... \phi_k$ is a hypothetical rule possibly with negated premises. Each premise ϕ_i thus has six possible forms: B, $B[add : C_1...C_k]$, $B[del : C_1...C_k]$, or the negation of any of these, where B and each C_i are atomic. In each case, we call the predicate B a *goal predicate.* Thus, in the rule $A \leftarrow B_1, \sim B_2[add : C]$, both B_1 and B_2 are goal predicates. In a top-down inference procedure, goal predicates would become subgoals and would be resolved against the heads of rules. For convenience, the predicate symbol appearing in the head of a rule shall be called the *head predicate.*

Definition 5.1 *Let R be a set of hypothetical rules with (possibly) negated premises. R is stratified iff it is the disjoint union of rulebases $R_0, ..., R_k$, where for each i,*

- *If a goal predicate P appears in a negated premise in R_i, then each rule with head predicate P is contained in $\cup_{j<i} R_j$*
- *If a goal predicate P appears in a non-negated premise in R_i, then each rule with head predicate P is contained in $\cup_{j \leq i} R_j$*

In this case $R_0, ..., R_k$ is a stratification of R, and R_i is called the i_{th} stratum. We also say that $R_0, ..., R_k$ stratify R.

This definition guarantees that if recursion occurs, then it occurs only within a single stratum and that it never occurs through negation.

Notice that only goal predicates and head predicates play a role in the definition of stratification. Thus, in the rule $A \leftarrow B[add : C]$, rules defining C may appear in any stratum. In particular, they can appear above the strata in which A and B are defined. Intuitively, the atom B is "executed" in order to prove A, while the atom C is "assumed" to be true in order to prove A. This is why C is not considered for the stratification; i.e., because C is not "executed," it cannot lead to recursion through negation.

Example 5.2 The following rulebase is stratified by R_0, R_1, R_2:

$$
\begin{array}{ll}
R_2 & \left\{ \begin{array}{l} A_2(x) \leftarrow B_2(x,y),\ A_2(y)[add: C_2(y)]. \\ A_2(x) \leftarrow D_2(x),\ \sim A_1(x)[add: E(x)]. \end{array} \right. \\
R_1 & \left\{ \begin{array}{l} A_1(x) \leftarrow B_1(x,y),\ A_1(y)[add: C_1(y)]. \\ A_1(x) \leftarrow D_1(x),\ \sim A_0(x). \end{array} \right. \\
R_0 & \left\{ \begin{array}{l} A_0(x) \leftarrow B_0(x,y),\ A_0(y)[add: C_0(y)]. \\ A_0(x) \leftarrow D_0(x). \end{array} \right.
\end{array}
$$

□

5.2 Operational Semantics

This section defines the meaning of the expression $R, DB \vdash \phi$ when R is a stratified, hypothetical rulebase.[3] The definition is similar to that for stratified Horn rulebases and involves two processes: (i) A series of bottom-up fixpoint computations, one for each stratum, and (ii) the inference of negative facts, by invoking the closed world assumption between fixpoint computations. This section examines both processes in turn, and then combines them into a single inference system.

5.2.1 Negation-Free Rulebases

The fixpoint semantics developed in Section 3.4 has to be modified before it can be extended to stratified rulebases. The main difficulty is that the operator T_R does not allow the least fixpoint of a rulebase to be computed one stratum at a time. In particular, if a rulebase is divided into strata $R_0, R_1 ... R_k$, it is not possible to decompose

[3] For the restricted logic of hypothetical additions, the meaning of stratified rulebases can also be defined in terms of intuitionistic logic. See [12] for details.

T_R into separate operators $T_{R_0}, T_{R_1}...T_{R_k}$. This section solves this problem for negation-free rulebases. The main idea is to introduce a new operator V_R that is an inflationary version of T_R. V_R has exactly the same fixpoints as T_R, but it can be decomposed in a way that T_R cannot. It is therefore more convenient for developing the semantics of stratified rulebases. This section defines V_R, develops its basic properties, and shows how it can be decomposed. The next section then uses V_R to define the semantics of arbitrary stratified rulebases.

Any negation-free rulebase R is stratified, since it can be viewed as having a single stratum containing all the rules in R. Often, however, a negation-free rulebase can be stratified in other ways as well. For instance, if the negation signs in Example 5.2 are erased, then the result is a negation-free rulebase R that is stratified by R_0, R_1, R_2. Although the minimal model of such a rulebase can be computed by a single bottom-up computation, $T_R^*(\mathcal{I})$, we would like to compute the minimal model in three separate stages: (i) compute the minimal model of R_0 using the rules in R_0, (ii) extend this to the minimal model of $R_0 \cup R_1$ using the rules in R_1, and (iii) extend this to the minimal model of $R_0 \cup R_1 \cup R_2$ using the rules in R_2. That is, we would like to compute the minimal model of R one stratum at a time, as in stratified Horn logic.

To focus on this problem, suppose that $R_0, R_1...R_k$ is a stratification, and that M_i is the minimal model of $R_0 \cup ... \cup R_i$. We would like to generate M_i by applying the rules in R_i to M_{i-1}. One possibility that comes to mind is to apply T_{R_i} to M_{i-1} over and over again until saturation. We would thus have $M_i = T_{R_i}^*(M_{i-1})$. Unfortunately, this equality does not hold in general. The problem is that T_{R_i} is not inflationary; that is, in general $T_{R_i}(M) \not\supseteq M$. Intuitively, after one application of T_{R_i}, the facts represented by M are no longer available for further inference. The result is that the sequence M_{i-1}, $T_{R_i}(M_{i-1})$, $T_{R_i}^2(M_{i-1})$... is not increasing, and thus does not represent an accumulation of inferences. In fact, the sequence may not even converge and may oscillate between several values instead. Thus, the very idea of using T_R to perform a least-fixpoint computation "starting from M" is not well-defined.

For this reason, the semantics of stratified rulebases cannot be based on the operator T_R. The solution, however, is quite simple: Define a new operator V_R that is an inflationary version of T_R. This operator is identical to T_R except that it preserves the facts represented by its input. The rest of this section defines V_R, develops its basic properties, and shows that it can be used to construct the minimal model of a rulebase one stratum at a time.

Definition 5.2 *Let R be a rulebase. Then $V_R(M) = M \sqcup T_R(M)$ for all hypothetical structures M.*

Thus $V_R(M) \geq M$. That is, V_R is inflationary and does not "forget" the facts represented by M. Furthermore, V_R captures the semantics of R just as T_R does, as the following lemmas show.

Lemma 5.1 *V_R and T_R have the same fixpoints.*

Proof: Suppose that M is a fixpoint of T_R. Then $T_R(M) \leq M$. Hence $M \sqcup T_R(M) \leq M$, and so $V_R(M) \leq M$, so M is a fixpoint of V_R. Conversely, suppose that M is a fixpoint of V_M. Then, $V_R(M) \leq M$. Hence $T_R(M) \leq M \sqcup T_R(M) = V_R(M) \leq M$, so M is a fixpoint of T_R. □

Thus, the fixpoints of V_R are exactly the models of R, by Lemma 3.8.

Lemma 5.2 *V_R is monotonic and continuous.*

Proof: A straightforward consequence of the monotonicity and continuity of T_R (Lemmas 3.9 and 3.16). □

V_R thus has a least fixpoint, *lfp*(V_R), which is identical to the least fixpoint of T_R. Furthermore, by the Tarski fixpoint theorem [47], this least fixpoint can be computed in a bottom-up way. That is, *lfp*$(V_R) = V_R^*(\mathcal{I})$. The proof is parallel to that of Theorem 3.18. Thus, the two bottom-up computations $V_R^*(\mathcal{I})$ and $T_R^*(\mathcal{I})$ produce identical results. The intermediate results produced along the way are identical as well, as the following lemma shows.

Lemma 5.3 $V_R^i(\mathcal{I}) = T_R^i(\mathcal{I})$ *for any i.*

Proof: (by induction on i). The base step is trivial: $V_R^0(\mathcal{I}) = \mathcal{I} = T_R^0(\mathcal{I})$. To prove the inductive step, suppose that $V_R^i(\mathcal{I}) = T_R^i(\mathcal{I})$. Then,

$$\begin{aligned}
V_R^{i+1}(\mathcal{I}) &= V_R[V_R^i(\mathcal{I})] \\
&= V_R^i(\mathcal{I}) \sqcup T_R[V_R^i(\mathcal{I})] && \text{by Definition 5.2,} \\
&= T_R^i(\mathcal{I}) \sqcup T_R[T_R^i(\mathcal{I})] && \text{by induction hypothesis,} \\
&= T_R^i(\mathcal{I}) \sqcup T_R^{i+1}(\mathcal{I}) \\
&= T_R^{i+1}(\mathcal{I}) && \text{since } T_R^i(\mathcal{I}) \leq T_R^{i+1}(\mathcal{I}) \text{, by Lemma 3.17.}
\end{aligned}$$

□

Thus, the operators V_R and T_R are similar in many ways. The main difference is that V_R is inflationary whereas T_R is not. Intuitively, T_R "forgets" the facts represented by its input, whereas V_R does not. This difference does not matter when fixpoint computations begin at the minimal structure $\mathcal{I}$, as Lemma 5.3 shows. But it matters a great deal when they begin at some other structure M. In fact, the very idea of performing a bottom-up fixpoint computation "from M" is not well-defined unless the operator is inflationary. In particular, if the sequence M, $T_R(M)$, $T_R^2(M)$... is not increasing, then it may not converge at all and may oscillate between several values instead. In contrast, the sequence M, $V_R(M)$, $V_R^2(M)$ ... is increasing for any structure M.

We can think of the sequence M, $V_R(M)$, $V_R^2(M)$ ... as taking place not in the lattice of all hypothetical structures, but in the sublattice consisting of those structures N such that $N \geq M$. M is the minimal element of this sublattice. Furthermore, since V_R is inflationary, it is an operator on the sublattice, mapping one element of the sublattice to another. Finally, since V_R is monotonic and continuous, we can invoke Tarski's fixpoint theorem to conclude that (i) V_R has a least fixpoint in the sublattice, and (ii) the sequence M, $V_R(M)$, $V_R^2(M)$ ... converges to this least fixpoint. (See Theorems 3.11 and 3.18.) This fixpoint is the least member of the set $\{N \geq M \mid V_R(N) \leq N\}$. But by Lemmas 5.1 and 3.8, the fixpoints of V_R are exactly the models of R. We have therefore proved the following result:

Lemma 5.4 *$V_R^*(M)$ is a model of R for any hypothetical structure M. Furthermore, it is the least model N of R such that $N \geq M$.*

These properties of V operators allows us to compute the minimal model of a rulebase one stratum at a time. The idea is to perform a bottom-up least-fixpoint computation for each stratum, using the result of one stratum's computation as the starting point for the next. In particular, given a stratification $R_0, ..., R_k$ of a rulebase R, we define the following procedure, where the superscript $+$ is a reminder that we are presently dealing with negation-free rulebases.

Procedure: FIXPOINT$^+(R_0, ..., R_k)$

$M_0 \leftarrow V^*_{R_0}(\mathcal{I})$

$M_1 \leftarrow V^*_{R_1}(M_0)$

$M_2 \leftarrow V^*_{R_2}(M_1)$

$\ldots$

$M_k \leftarrow V^*_{R_k}(M_{k-1})$

return(M_k)

This procedure computes a hypothetical structure M_k. The rest of this section shows that M_k is exactly the minimal model of the rulebase R. M_k is thus independent of the stratification used to compute it, so we can use any stratification that is convenient.

Example 5.3 Let R be a rulebase stratified as follows:

$$\begin{array}{ll} R_2 & \left\{ F \leftarrow E[add : D] \right. \\ R_1 & \left\{ E \leftarrow A. \right. \\ R_0 & \left\{ \begin{array}{l} A \leftarrow B[add : C] \\ B \leftarrow C, D. \end{array} \right. \end{array}$$

Note that the first stratum, R_0, is identical to the rulebase of Example 3.3. The table below illustrates the computation of FIXPOINT$^+(R_0, R_1, R_2)$. The first column shows the four possible databases DB constructible from the two predicates C and D. The remaining columns show the values of $M_i(DB)$ for $i = 0, 1, 2$. The second column thus shows the values of $[V^*_{R_0}(\mathcal{I})](DB)$. The values in this column are equal to $[T^*_{R_0}(\mathcal{I})](DB)$, the computation of which is shown in detail in Example 3.3.

DB	$M_0(DB)$	$M_1(DB)$	$M_2(DB)$
–	–	–	+F
C	C	C	C+F
D	D+A	A,D+E	A,D,E+F
C,D	C,D+A+B	A,B,C,D+E	A,B,C,D,E+F

Atoms added by the rules in R are preceded by a plus sign. For instance, the $+E$ in the third column/third row is due to the rule $E \leftarrow A$. Likewise, all the $+F$'s in the 4^{th} column are due to the rule $F \leftarrow E[add : D]$. □

Computing the Least Model

We now show that the value computed by the procedure FIXPOINT$^+(R_0, ..., R_k)$ is exactly the least model of the rulebase $R = R_1 \cup ... \cup R_k$. Thus, FIXPOINT$^+$ can be used to compute the least model of R one stratum at a time, using any stratification that is convenient. As in Section 3.4, the development does not exploit the finiteness of the data domain. The results of this section therefore extend to an infinite domain in a straightforward way.

To show that FIXPOINT$^+$ is the minimal model of R, we introduce the following idea.

Definition 5.3 *A rule r_2 "refers" to another rule r_1 iff the head predicate of r_1 is a goal predicate of r_2. A set of rules R_2 refers to another set of rules R_1 iff some rule in R_2 refers to some rule in R_1.*

Observe that if $R_0, ..., R_k$ is a stratification of a rulebase R, then R_i does *not* refer to any stratum higher than itself. That is, R_i does not refer to R_j if $j > i$. On the other hand, R_i may refer to itself and to lower strata.

Intuitively, if R_1 does not refer to R_2, then none of the inferences made by R_2 can be used by R_1. Thus, to compute the minimal model of $R_2 \cup R_1$, we can first apply the rules in R_1 to saturation and then apply the rules in R_2. Formally, $V^*_{R_2 \cup R_1} = V^*_{R_2} V^*_{R_1}$. Proving this result is the main task of this section. The result is a consequence of two basic properties of V operators, which we establish in two lemmas. The first property says simply that larger rulebases make more inferences. The following lemma establishes this property for V operators, and a corollary extends it to their logical closures V^*.

Lemma 5.5 *If R and S are rulebases and $R \subseteq S$, then $V_R(M) \leq V_S(M)$.*

Proof: $[T_R(M)](DB) \subseteq [T_S(M)](DB)$ for any database DB, by Definition 3.13. Hence $T_R(M) \leq T_S(M)$. Thus $M \sqcup T_R(M) \leq M \sqcup T_S(M)$, and so $V_R(M) \leq V_S(M)$ by Definition 5.2. □

Corollary 5.6 *If R and S are rulebases and $R \subseteq S$, then $V_R^*(M) \leq V_S^*(M)$.*

Proof: We first show that $V_R^i(M) \leq V_S^i(M)$, by induction on i. The base step is trivial: $V_R^0(M) = M = V_S^0(M)$. To prove the inductive step, assume that $V_R^i(M) \leq V_S^i(M)$ for some i. Then,

$$\begin{aligned}
V_R^{i+1}(M) &= V_R[V_R^i(M)] \\
&\leq V_R[V_S^i(M)] \quad \text{by induction hypothesis and the monotonicity of } V_R, \\
&\leq V_S[V_S^i(M)] \quad \text{by Lemma 5.5,} \\
&= V_S^{i+1}(M)
\end{aligned}$$

Thus $V_R^*(M) = \sqcup_{i \geq 0} V_R^i(M) \leq \sqcup_{i \geq 0} V_S^i(M) = V_S^*(M)$. □

The next lemma establishes another basic property of V operators, relating them to the idea of "referral." The lemma establishes the property for V operators, and a corollary extends it to their logical closures V^*.

Lemma 5.7 *Suppose that M is a model of R_1 and that R_1 does not refer to R_2. Then, $V_{R_2}(M)$ is also a model of R_1.*

Proof: $V_{R_2}(M) = M \sqcup T_{R_2}(M)$ by definition. Thus, V_{R_2} has the effect of adding atomic values to $M(DB)$ for each database DB. But these new atomic values come only from the heads of rules in R_2. Since R_1 does not refer to R_2, the rule premises in R_1 are unaffected by the new values. That is, if $A \leftarrow \phi_1 ... \phi_k$ is an instance of a rule in R_1, then for each i,

$$M, DB \models_h \phi_i \quad \textit{iff} \quad M \sqcup T_{R_2}(M), DB \models_h \phi_i \tag{5.1}$$

for any database DB.

To show that $V_{R_2}(M)$ is a model of R_1, suppose that $V_{R_2}(M), DB \models_h \phi_1 ... \phi_k$. We must show that $V_{R_2}(M), DB \models_h A$. To see this, first note that $M, DB \models_h \phi_1 ... \phi_k$, by statement (5.1). Hence $M, DB \models_h A$, since $A \leftarrow \phi_1 ... \phi_k$ is an instance of a rule in R_1, and M is a model of R_1. Therefore $V_{R_2}(M), DB \models_h A$, by Lemma 3.7, since $V_{R_2}(M) \geq M$. $\Box$

Corollary 5.8 *Suppose that M is a model of R_1 and that R_1 does not refer to R_2. Then $V^*_{R_2}(M)$ is also a model of R_1.*

Proof: $V_{R_2}(M)$ is a model of R_1, by Lemma 5.7. Thus $V^i_{R_2}(M)$ is also a model of R_1, by induction. Hence $V^i_{R_2}(M)$ is a fixpoint of V_{R_1}. That is,

$$V_{R_1} V^i_{R_2}(M) \;\leq\; V^i_{R_2}(M) \tag{5.2}$$

In addition, $V^0_{R_2}(M)$, $V^1_{R_2}(M)$, $V^2_{R_2}(M)$, ... is an increasing sequence of hypothetical structures, since V_{R_2} is inflationary. Therefore,

$$V_{R_1}[\sqcup_{i \geq 0} V^i_{R_2}(M)] \;=\; \sqcup_{i \geq 0} V_{R_1} V^i_{R_2}(M) \tag{5.3}$$

since V_{R_1} is continuous, by Lemma 5.2. Hence,

$$\begin{array}{rcll} V_{R_1} V^*_{R_2}(M) & = & V_{R_1}[\sqcup_{i \geq 0} V^i_{R_2}(M)] & \text{by definition,} \\ & = & \sqcup_{i \geq 0} V_{R_1} V^i_{R_2}(M) & \text{by statement (5.3),} \\ & \leq & \sqcup_{i \geq 0} V^i_{R_2}(M) & \text{by statement (5.2),} \\ & = & V^*_{R_2}(M) & \text{by definition.} \end{array}$$

Thus $V^*_{R_2}(M)$ is a fixpoint of V_{R_1} and hence a model of R_1. $\Box$

The next theorem is the main result of this section. Informally, it says that if rulebase R_1 does not refer to rulebase R_2, then applying the rules in $R_2 \cup R_1$ to saturation is equivalent to first applying the rules in R_1 to saturation and then applying the rules in R_2 to saturation. A corollary then shows that the least fixpoint of a rulebase can be computed one stratum at a time.

Theorem 5.9 *If R_1 does not refer to R_2, then $V^*_{R_2 \cup R_1} = V^*_{R_2} V^*_{R_1}$.*

Proof: We show that $V^*_{R_2 \cup R_1}(M) = V^*_{R_2} V^*_{R_1}(M)$ for all hypothetical structures M. We first show that $V^*_{R_2 \cup R_1}(M) \leq V^*_{R_2} V^*_{R_1}(M)$. To see this, let $N = V^*_{R_2} V^*_{R_1}(M)$. Then N has the following two properties: (i) $N \geq M$ and (ii) N is a model of $R_2 \cup R_1$. The first property follows because V_{R_2} and V_{R_1} are inflationary. The second property is a consequence of the following observations:

	$V^*_{R_2} V^*_{R_1}(M)$ is a model of R_2, by Lemma 5.4.
Also	$V^*_{R_1}(M)$ is a model of R_1, by Lemma 5.4,
so	$V^*_{R_2} V^*_{R_1}(M)$ is a model of R_1, by Corollary 5.8.

Finally, Lemma 5.4 tells us that $V^*_{R_2 \cup R_1}(M)$ is the smallest structure satisfying properties (i) and (ii). Thus $V^*_{R_2 \cup R_1}(M) \leq V^*_{R_2} V^*_{R_1}(M)$.

We now show the converse, that $V^*_{R_2 \cup R_1}(M) \geq V^*_{R_2} V^*_{R_1}(M)$:

$$
\begin{array}{lll}
V^*_{R_2 \cup R_1}(M) & & \\
\quad = & V_{R_2 \cup R_1} V^*_{R_2 \cup R_1}(M) & \text{since } V^*_{R_2 \cup R_1}(M) \text{ is a fixpoint of } V_{R_2 \cup R_1}, \\
\quad = & V^i_{R_2 \cup R_1} V^*_{R_2 \cup R_1}(M) & \text{by induction on } i, \\
\quad = & \sqcup_{i \geq 0} V^i_{R_2 \cup R_1}[V^*_{R_2 \cup R_1}(M)] & \\
\quad = & V^*_{R_2 \cup R_1} V^*_{R_2 \cup R_1}(M) & \text{by definition}, \\
\quad \geq & V^*_{R_2} V^*_{R_2 \cup R_1}(M) & \text{by Corollary 5.6}, \\
\quad \geq & V^*_{R_2} V^*_{R_1}(M) & \text{by Corollary 5.6 and the monotonicity of } V^*_{R_2}.
\end{array}
$$

□

Corollary 5.10 *If $R_0, ..., R_k$ is a stratification of a rulebase R, then*

$$V^*_R = V^*_{R_k} V^*_{R_{k-1}} \cdots V^*_{R_0}$$

Proof: Since $R_0, ..., R_k$ is a stratification, R_i does not refer to $R_k \cup ... \cup R_{i+1}$, for all $i < k$. Thus, $V^*_{R_k \cup ... \cup R_i} = V^*_{R_k \cup ... \cup R_{i+1}} V^*_{R_i}$, by Theorem 5.9. Hence,

$$
\begin{aligned}
V^*_R &= V^*_{R_k \cup \ldots \cup R_0} \\
&= V^*_{R_k \cup \ldots \cup R_1} \; V^*_{R_0} \\
&= V^*_{R_k \cup \ldots \cup R_2} \; V^*_{R_1} V^*_{R_0} \\
&\quad \cdots \\
&= V^*_{R_k} \cdots V^*_{R_2} \; V^*_{R_1} V^*_{R_0}
\end{aligned}
$$

□

Given a stratification $R_0, ..., R_k$ of a rulebase R, it follows from Corollary 5.10 that the value returned by the procedure FIXPOINT$^+$ is the least model of R. In particular, from the definition of FIXPOINT$^+$, we see that FIXPOINT$^+(R_0, ..., R_k) = V^*_{R_k} \cdots V^*_{R_1} V^*_{R_0}(\mathcal{I})$. Therefore FIXPOINT$^+(R_0, ..., R_k) = V^*_R(\mathcal{I})$, which is just the least model of R.

This establishes the main point of this section: that using V operators, the procedure FIXPOINT$^+$ computes the least model of a negation-free rulebase one stratum at a time.

5.2.2 Rulebases with Negation

This section uses V operators to develop the semantics of arbitrary stratified rulebases. As described earlier, this semantics is similar to that for stratified Horn rulebases and involves two processes: (i) A series of bottom-up fixpoint computations, one for each stratum, and (ii) the inference of negative facts, by invoking the closed world assumption between fixpoint computations. Section 5.2.1 described the first process. The present section describes the second process and integrates it with the first, to form a single inference system.

To represent negative literals, it is convenient to introduce new predicate symbols. In particular, for each predicate symbol P we introduce a new predicate symbol $\overline{P}$, which we call the *complement* of P. The atomic formula $\overline{P}(\overline{x})$ then denotes the negative literal $\sim P(\overline{x})$. With this in mind, we replace the stratified rulebase R by a negation-free rulebase $\overline{R}$ as follows:

Definition 5.4 *Let R be a set of stratified hypothetical rules. Then $\overline{R}$ is a negation-free rulebase constructed from R as follows: In every rule of R, replace each negated premise $\sim P(\overline{x})$, $\sim P(\overline{x})[add : Q_1 ... Q_k]$ or $\sim P(\overline{x})[del : Q_1 ... Q_k]$ by the non-negated premise $\overline{P}(\overline{x})$, $\overline{P}(\overline{x})[add : Q_1 ... Q_k]$ or $\overline{P}(\overline{x})[del : Q_1 ... Q_k]$, respectively.*

Given a stratification $R_0, ..., R_k$ of a rulebase R, we shall construct the rulebases $\overline{R}_0, ... \overline{R}_k$. Since each $\overline{R}_i$ is negation-free, we can apply to it the fixpoint machinery developed in sections 3.4 and 5.2.1. In particular, for each stratum, we can define the operator $V_{\overline{R}_i}$, and we can apply this operator to an arbitrary hypothetical structure M.

Recall that when we speak of inference from a rulebase R, we always have in mind a particular rulebased system $\mathcal{R} = [R, dom, pred]$, where *dom* is a set of constant symbols and *pred* is a set of predicate symbols. We extend this convention to stratified rulebases. Also recall that $\mathcal{DB}$ is the set of ground atomic formulas constructible from *dom* and *pred*; i.e., $\mathcal{DB}$ is the set of all positive ground literals. Extending this notion to signed literals, we define $\overline{\mathcal{DB}}$ to be the set of all ground atomic formulas constructible from the constant symbols in *dom* and the predicate symbols in *pred* and their complements. Thus $\overline{\mathcal{DB}}$ contains formulas of the form $\overline{P}(\overline{a})$ as well as $P(\overline{a})$.

With this in mind, we extend the definition of an interpretation. In section 3.4, it was defined as an inflationary mapping $M : 2^{\mathcal{DB}} \rightarrow 2^{\mathcal{DB}}$. To take negative literals into account, we now define it to be an inflationary mapping $M : 2^{\mathcal{DB}} \rightarrow 2^{\overline{\mathcal{DB}}}$. To capture the intended meaning of the new predicate symbols $\overline{P}$, we define the following operation on interpretations. Given an interpretation M, this operation applies the so-called *closed world assumption*: If something is not true, then assume it is false. This is the basis of negation-as-failure.

Definition 5.5 *Let M be a hypothetical structure. Then $\overline{M}$ is also a hypothetical structure, where for each database $DB \subseteq \mathcal{DB}$ and each ground atomic formula $P(\overline{x}) \in \mathcal{DB}$,*

- $P(\overline{x}) \in \overline{M}(DB)$ *iff* $P(\overline{x}) \in M(DB)$
- $\overline{P}(\overline{x}) \in \overline{M}(DB)$ *iff* $P(\overline{x}) \notin M(DB)$

Let R be a stratified hypothetical rulebase with stratification $R_0, ..., R_k$. The following procedure defines the operational semantics of this stratification. Starting with the identity mapping $\mathcal{I}$, this procedure applies each of the rule sets $\overline{R}_i$, one after the other, in a bottom-up fashion to saturation. The operation $\overline{M}$ is used to enforce the closed world assumption between adjacent strata.

Procedure: FIXPOINT$(R_0, ..., R_k)$

$M_0 \leftarrow V^*_{\overline{R}_0}(\overline{\mathcal{I}})$

$M_1 \leftarrow V^*_{\overline{R}_1}(\overline{M}_0)$

$M_2 \leftarrow V^*_{\overline{R}_2}(\overline{M}_1)$

$\cdots$

$M_k \leftarrow V^*_{\overline{R}_k}(\overline{M}_{k-1})$

return$(\overline{M}_k)$

This procedure is a natural extension of the procedure FIXPOINT$^+$ defined in Section 5.2.1 for negation-free rulebases. This procedure also has a natural counterpart in the operational semantics of stratified Horn rulebases [4]. In both cases, the procedure constructs a preferred fixpoint. Indeed, it is not hard to see that if M = FIXPOINT(R), then M is a fixpoint of $V_{\overline{R}}$.

In general, there may be many ways to stratify a rulebase R. This is why the procedure FIXPOINT has been written as a function not just of the rulebase R, but of its stratification $R_0, ..., R_k$. However, the next section shows that for a given rulebase, the value of FIXPOINT is independent of the way it is stratified (like the procedure FIXPOINT$^+$). Thus, despite appearances, FIXPOINT is a function only of R, and we may write FIXPOINT(R) instead of FIXPOINT$(R_0, ..., R_k)$. This justifies the following definition.

Definition 5.6 *Let R be a stratified rulebase, and let ϕ be a hypothetical query. Then*

$$R, DB \vdash \phi \quad \textit{iff} \quad M, DB \models_h \phi$$

where M = FIXPOINT(R).

Example 5.4 Let R be the following stratified rulebase:

$$R_1 \left\{ \; E \leftarrow \sim C, \; \sim A, \; A[add : D] \right.$$
$$R_0 \left\{ \begin{array}{l} A \leftarrow B[add : C] \\ B \leftarrow C, D. \end{array} \right.$$

(Note that the bottom stratum, R_0, is identical to the rulebase of Example 3.3.) The corresponding negation-free rulebase $\overline{R}$ consists of the following rules, where $\overline{C}$ and $\overline{E}$ are new predicate symbols:

$$\overline{R}_1 \left\{ \; E \leftarrow \overline{C}, \overline{A}, A[add : D] \right.$$
$$\overline{R}_0 \left\{ \begin{array}{l} A \leftarrow B[add : C] \\ B \leftarrow C, D. \end{array} \right.$$

The table below illustrates the computation of FIXPOINT(R_0, R_1). The first column shows the four possible databases DB constructible from the two predicates C and D. The second column is the result of applying the closed world assumption to the first column. Thus, in each row, $\overline{C}$ is present in column 2 iff C is not present in column 1. Likewise for the other atoms A, B, D and E. The remaining columns show the values of $M_i(DB)$ and $\overline{M}_i(DB)$ for $i = 0, 1$.

DB	$\overline{\mathcal{I}}(DB)$	$M_0(DB)$	$\overline{M}_0(DB)$	$M_1(DB)$	$\overline{M}_1(DB)$
–	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}\,\overline{E}$	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}\,\overline{E}$	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}\,\overline{E}$	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}\,\overline{E} + E$	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}\,E$
C	$\overline{A}\,\overline{B}\,C\,\overline{D}\,\overline{E}$	$\overline{A}\,\overline{B}\,C\,\overline{D}\,\overline{E}$	$\overline{A}\,\overline{B}\,C\,\overline{D}\,\overline{E}$	$\overline{A}\,\overline{B}\,C\,\overline{D}\,\overline{E}$	$\overline{A}\,\overline{B}\,C\,\overline{D}\,\overline{E}$
D	$\overline{A}\,\overline{B}\,\overline{C}\,D\,\overline{E}$	$\overline{A}\,\overline{B}\,\overline{C}\,D\,\overline{E} + A$	$A\,\overline{B}\,\overline{C}\,D\,\overline{E}$	$A\,\overline{B}\,\overline{C}\,D\,\overline{E}$	$A\,\overline{B}\,\overline{C}\,D\,\overline{E}$
CD	$\overline{A}\,\overline{B}\,C\,D\,\overline{E}$	$\overline{A}\,\overline{B}\,C\,D\,\overline{E} + AB$	$A\,B\,C\,D\,\overline{E}$	$A\,B\,C\,D\,\overline{E}$	$A\,B\,C\,D\,\overline{E}$

Atoms added by the rules in $\overline{R}$ are preceded by a plus sign. For instance, the $+E$ in the fifth column/first row is due to the rule in R_1 which infers E. This rule emphasizes the hypothetical nature of the inference system. The premise of the rule states that both $\sim A$ and A must be true, but not on the same database. In particular, to infer E on a database DB, one must be able to infer $\sim A$ on DB, and to infer A on $DB + D$. This is precisely the case in the first row of the table, where $DB = \{\}$.

In the columns headed $\overline{M}_i(DB)$, the closed world assumption is simulated. Thus, in the first row, although $\overline{E}$ in present in columns 2–5, it is not present in column 6, after E has been inferred. Likewise, in the third row, although $\overline{A}$ is present in columns 2 and 3, it is not present in column 4. This removal of barred predicates reflects the non-monotonic nature of negation-as-failure. □

Negation-as-failure computes the complement of a relation. For instance, given the rule $A(x) \leftarrow B(x), \sim C(x)$, the inferred relation A is the complement of C relative to B (i.e., $A = B \sim C$). Because the complement is relative to another predicate, the rule is said to be *guarded.* In an unguarded rule such as $A(x) \leftarrow \sim B(x)$, complementation is defined relative to the domain of constant symbols: in this case, $A = dom \sim B$. It is primarily because of unguarded rules that an explicit domain *dom* is needed in the definition of a rulebase system (see Section 3.1).

It is always possible to convert unguarded rules into guarded ones by introducing a domain predicate D, where $D(x)$ means that x is in the domain *dom*. An unguarded rule such as $A(x) \leftarrow \sim B(x)$ is then equivalent to the guarded rule $A(x) \leftarrow D(x), \sim B(x)$. In this dissertation, however, we do not need to introduce a domain predicate since we have assumed that the domain is built into the inference system.

5.3 Stratification Independence

This section shows that the meaning of a stratified rulebase is independent of the way in which it is stratified. In particular, we show that the value of the procedure FIXPOINT depends only on the rulebase R and not on the particular stratification used. The development presented here is an adaptation of that given in [4] for the semantics of stratified Horn rulebases. First, we divide R into disjoint "clusters", that is, maximal sets of rules which are mutually recursive. Second, we show that any stratification of a rulebase is equivalent to a cluster stratification, in which each stratum consists of a single cluster. Finally, we show that any two cluster stratifications of a rulebase are equivalent, that is, they both result in the same value of the procedure FIXPOINT.

To define clusters and mutual recursion precisely, we first define a binary relation $\prec$ on rules. For convenience, we repeat part of the definition of "refers," as originally stated in Definition 5.3.

Definition 5.7 *Let R be a stratified rulebase, and let r_1 and r_2 be two rules in R. (i) r_2 refers to r_1 iff the head predicate of r_1 is a goal predicate of r_2. (ii) "$\prec$" denotes the reflexive, transitive closure of the relation "refers". (iii) $r_1 \equiv r_2$ iff $r_2 \prec r_1$ and $r_1 \prec r_2$.*

If $r_2 \prec r_1$, then we say that r_2 *depends on* r_1; and if $r_1 \equiv r_2$, then we say that r_1 and r_2 are *mutually recursive*. Note that mutual recursiveness is an equivalence relation. Thus, any maximal set of mutually recursive rules forms an equivalence class.

Definition 5.8 *Let R be a stratified rulebase. A cluster of R is an equivalence class of mutually recursive rules.*

It follows immediately that the clusters of R are disjoint and exhaustive. Furthermore, if $R_0, ..., R_k$ is a stratification of R, then each cluster of R is contained in some stratum R_i. Each stratum therefore consists of a disjoint union of clusters. Similar results were established for Horn rules in [4].

Clusters are a convenient device for hiding recursion from our theoretical development. There is no recursion between clusters; and because each cluster lies within a single stratum, there is no need to look within a cluster. Clusters thus serve as the "atomic units" of our analysis. Notice that if there is no recursion in a rulebase, then each cluster consists of exactly one rule. In this case, there is no recursion to hide and no need to group rules into larger units.

To describe the interactions between clusters, we extend the relation $\prec$ from single rules to rule sets.

Definition 5.9 *Let S_1 and S_2 be two subsets of a stratified rulebase R. Then $S_2 \prec S_1$ iff $r_2 \prec r_1$ for some rule $r_1 \in S_1$ and some rule $r_2 \in S_2$.*

If $S_2 \prec S_1$ then we say that S_2 *depends on* S_1. Notice that if $R_0, ..., R_k$ is a stratification of R and if R_j depends on R_i, then $j \geq i$, i.e., R_j is a higher stratum than R_i. The following basic result about dependency is immediate.

Lemma 5.11 *Let $S_0, S_1, ..., S_n$ be subsets of a stratified rulebase R. Then,*

$$S_0 \prec S_1 \cup \cdots \cup S_n \quad \textit{iff} \quad S_0 \prec S_i \ \textit{for some}\ 1 \leq i \leq n.$$

Definition 5.10 *The dependency graph of a set of clusters is the directed graph whose nodes are clusters and in which there is an edge from Q_2 to Q_1 iff Q_2 depends on Q_1 and $Q_2 \neq Q_1$.*

The following basic result is straightforward and reflects the lack of recursion between clusters.

Lemma 5.12 *The dependency graph of a set of clusters is acyclic.*

The development that follows focuses on the relation $\not\prec$, the complement of $\prec$. Informally, if S_1 does not depend on S_2, then in a bottom-up inference procedure, the rules in S_1 may be applied to saturation before the rules of S_2 are applied. The next lemma makes this idea precise. It is based on an operator U that is closely related to the V-operator developed in Section 5.2.1.

Definition 5.11 *Let S be a subset of some stratum of a stratified rulebase R. Then $U_S(M) = \overline{V^*_{\overline{S}}(\overline{M})}$ for all interpretations M.*

The operator U_S has a simple intuitive interpretation. First, the closed world assumption is applied to M. This step computes negative literals so that negative rule premises in $\overline{S}$ can be satisfied. Second, the rules in $\overline{S}$ are applied to M in a bottom-up fashion to saturation. This step computes all the positive literals derivable from M by the rules in S (for any database). Finally, the closed world assumption is applied to the result, to compute negative literals. Thus, $U_S(M)$ represents all the literals, positive and negative, derivable from M by the rules in S.

U_S can be applied to an arbitrary interpretation M, and the rule set S need not be an entire stratum. This makes U_S a convenient theoretical tool for describing the behavior of rule sets within a stratum. U_S is also closely related to the procedure FIXPOINT. In particular, if R is stratified by $R_0, R_1..., R_k$, then

$$\text{FIXPOINT}(R_0, R_1, ..., R_k) \ = \ U_{R_k} \cdots U_{R_1} U_{R_0}(\mathcal{I}) \tag{5.4}$$

The following is a basic result about U operators:

Lemma 5.13 *Let S_1 and S_2 be subsets of the same stratum of some stratified rulebase R. If $S_1 \not\prec S_2$ then $U_{S_1 \cup S_2} = U_{S_2} U_{S_1}$.*

Proof: First note that since S_1 and S_2 are subsets of the same stratum, so is $S_1 \cup S_2$. Thus the expression $U_{S_1 \cup S_2}$ is well-defined. In computing $U_{S_1 \cup S_2}(M)$, the main task is to apply the rules in $\overline{S}_1 \cup \overline{S}_2$ to saturation. Observe that since S_1 does not depend on S_2, neither does $\overline{S}_1$ depend on $\overline{S}_2$. In particular, $\overline{S}_1$ does not refer to $\overline{S}_2$. Thus, by Theorem 5.9, the rules in $\overline{S}_1$ can be applied first. That is, the rules in $\overline{S}_1$ can be run to saturation, and then the rules in $\overline{S}_2$ can be run to saturation. Formally, $V^*_{\overline{S}_1 \cup \overline{S}_2} = V^*_{\overline{S}_2} V^*_{\overline{S}_1}$. Thus, for any hypothetical structure M,

$$U_{S_1 \cup S_2}(M) \;=\; \overline{V^*_{\overline{S}_2} V^*_{\overline{S}_1}(\overline{M})} \tag{5.5}$$

Next observe that since S_1 and S_2 belong to the same stratum, negative rule premises in S_2 are not affected by the inferences of S_1. Thus, after applying the rules in $\overline{S}_1$ to saturation, we can apply the closed world assumption (to infer negative literals) before applying the rules in $\overline{S}_2$. Doing so does not affect the inferences made by $\overline{S}_2$ in any way. Thus, for any unbarred atom A,

$$A \;\in\; V^*_{\overline{S}_2}[V^*_{\overline{S}_1}(\overline{M})] \qquad \textit{iff} \qquad A \;\in\; V^*_{\overline{S}_2}[\overline{V^*_{\overline{S}_1}(\overline{M})}]$$

Hence,

$$\overline{V^*_{\overline{S}_2} V^*_{\overline{S}_1}(\overline{M})} \;=\; \overline{V^*_{\overline{S}_2} \overline{V^*_{\overline{S}_1}(\overline{M})}} \tag{5.6}$$

Finally, observe that the closed world assumption is idempotent: One application derives all the negative literals that there are to derive. Hence, $\overline{\overline{M}} = \overline{M}$, for any hypothetical structure M. In particular, $U_S(M) = \overline{U_S(M)}$, by definition 5.11. Keeping this in mind, we get the following equalities:

$$\begin{array}{rcll}
U_{S_1 \cup S_2}(M) & = & \overline{V^*_{\overline{S}_2} V^*_{\overline{S}_1}(\overline{M})} & \text{by statement (5.5),} \\
 & = & \overline{V^*_{\overline{S}_2} \overline{V^*_{\overline{S}_1}(\overline{M})}} & \text{by statement (5.6),} \\
 & = & \overline{V^*_{\overline{S}_2} U_{S_1}(M)} & \text{by Definition 5.11,} \\
 & = & \overline{V^*_{\overline{S}_2} \overline{U_{S_1}(M)}} & \text{since } U_{S_1}(M) = \overline{U_{S_1}(M)}, \\
 & = & U_{S_2}[U_{S_1}(M)] & \text{by Definition 5.11.}
\end{array}$$

Hence $U_{S_1 \cup S_2} = U_{S_2} U_{S_1}$. $\Box$

Corollary 5.14 *Let R be a rulebase with stratification $R_0, ..., R_k$. If $R_{i+1} \not\prec R_i$ then $U_{R_{i+1}} U_{R_i} = U_{R_i} U_{R_{i+1}}$.*

Proof: Since $R_0, ..., R_k$ is a stratification, $R_i \not\prec R_{i+1}$. Thus neither R_i nor R_{i+1} depends on the other. Thus, since they are adjacent strata, they can be combined into a single stratum. Thus, R_i and R_{i+1} can be viewed as subsets of a single stratum, so that Lemma 5.13 applies. Applying the lemma twice, $U_{R_{i+1}} U_{R_i} = U_{R_1 \cup R_2} = U_{R_i} U_{R_{i+1}}$ $\Box$

Lemma 5.13 allows us to decompose a single stratum into smaller strata. Our strategy is to apply this lemma over and over again, breaking a stratum down into smaller and smaller pieces, until each stratum consists of a single cluster. Before we can do this, however, we must linearize the set of clusters.

Definition 5.12 *Let $S_0, ..., S_n$ be subsets of a stratified rulebase R. Then the sequence $S_0, ..., S_n$ is linear iff $S_i \not\prec S_j$ for each $i < j$.*

In a linear sequence, a rule set S_i does not depend on any of the rule sets which follow it. Thus, a bottom-up inference procedure can first apply the rules in S_0 to saturation, then the rules in S_1, etc. The following is a basic result about linear sequences of clusters.

Lemma 5.15 *Let $Q_0, ..., Q_n$ be the clusters of a stratified rulebase R. Then $Q_0, ..., Q_n$ is linear iff it is a stratification of R.*

The following lemma is the main result about clusters and linearity.

Lemma 5.16 *Any set of clusters can be arranged in a linear sequence.*

Proof: By lemma 5.12, the dependency graph of the clusters is acyclic. We can therefore assign a depth to each cluster Q, where $depth(Q)$ is the length of the *longest* directed path from Q to a leaf (i.e., a sink). Now print the clusters out in ascending order of depth, where clusters of equal depth are printed in any order. The result is a sequence of clusters $Q_0, ..., Q_n$ where Q_0 is printed first. This sequence is linear. To see this, note that for any two distinct clusters Q_i and Q_j, if Q_i depends on Q_j, then the longest path from Q_i to a leaf is at least as long as any path from Q_i through Q_j to a leaf. Hence $depth(Q_i) > depth(Q_j)$. Thus, Q_j is printed first; that is, $j < i$. We have therefore shown that if $Q_i \prec Q_j$, then $j < i$. Conversely, if $i < j$ then $Q_i \not\prec Q_j$. Hence, the sequence $Q_0, ..., Q_n$ is linear. □

By the discussion following Definition 5.8, each stratum of a rulebase consists of a disjoint union of clusters. Lemma 5.16 tells us that these clusters can be linearized. By Lemma 5.15, each cluster in this sequence can be viewed as a small stratum in its own right. In this way, each stratum of a rulebase can be split into a number of smaller strata, each consisting of a single cluster. The following theorem and its corollary develop this idea further.

Theorem 5.17 *Let R be a stratified rulebase. If R_i is a stratum of R, and $Q_0, Q_1, ..., Q_n$ are the clusters of R_i arranged in a linear order, then*

$$U_{R_i} \;=\; U_{Q_n} \cdots U_{Q_1} U_{Q_0}$$

Proof: $Q_i \not\prec Q_j$ for $i < j \leq n$, since $Q_0, ..., Q_n$ is linear. Thus $Q_i \not\prec Q_{i+1} \cup ... \cup Q_n$, by Lemma 5.11. Thus, $U_{Q_n \cup ... \cup Q_i} = U_{Q_n \cup ... \cup Q_{i+1}} U_{Q_i}$, by Lemma 5.13. Hence,

$$\begin{aligned} U_{R_i} &= U_{Q_n \cup ... \cup Q_0} \\ &= U_{Q_n \cup ... \cup Q_1} U_{Q_0} \\ &= U_{Q_n \cup ... \cup Q_2} U_{Q_1} U_{Q_0} \\ &\quad \cdots \\ &= U_{Q_n} \cdots U_{Q_2} U_{Q_1} U_{Q_0} \end{aligned}$$

□

Corollary 5.18 *Let R be a rulebase with stratification $R_0 ... R_k$. For each i, let $Q_i^0 ... Q_i^{n_i}$ be the clusters of stratum R_i arranged in a linear order. Then $Q_0^0 ... Q_0^{n_0} ... Q_k^0 ... Q_k^{n_k}$ is also a stratification of R. Furthermore, $U_{R_k} \cdots U_{R_0} \;=\; U_{Q_k^{n_k}} \cdots U_{Q_0^0}$.*

Thus every stratification of a rulebase R is equivalent to a cluster stratification, in which each stratum consists of a single cluster. In particular, by statement (5.4), these two stratifications will result in the same value of the procedure FIXPOINT. We now show that any two cluster stratifications of R are equivalent.

Theorem 5.19 *Let $\mathcal{L} = Q_0, ..., Q_n$ and $\mathcal{L}' = Q'_0, ..., Q'_n$ be two linear orderings of the clusters of a rulebase R. Then $\mathcal{L}$ and $\mathcal{L}'$ are equivalent; that is,*

$$U_{Q_n} \cdots U_{Q_0} \;=\; U_{Q'_n} \cdots U_{Q'_0}$$

Proof: We transform $\mathcal{L}$ into $\mathcal{L}'$ using a bubble sort. In particular, we use $\mathcal{L}'$ to specify a fixed linear order of the clusters, and we swap adjacent clusters in $\mathcal{L}$ whenever their relative order is different from that in $\mathcal{L}'$. The important point is that each swap operation preserves linearity and equivalence.

To see this, suppose that Q_i and Q_{i+1} are two adjacent clusters in $\mathcal{L}$ and that we are about to swap them. Then Q_{i+1} comes before Q_i in $\mathcal{L}'$. Hence $Q_{i+1} \not> Q_i$ since $\mathcal{L}'$ is linear. Thus, after swapping Q_i and Q_{i+1} in $\mathcal{L}$, the resulting sequence of clusters will still be linear. In addition, $U_{Q_{i+1}} U_{Q_i} = U_{Q_i} U_{Q_{i+1}}$, by corollary 5.14 (since $Q_0, ..., Q_n$ is a stratification of R, by Lemma 5.15). Thus, after swapping Q_i and Q_{i+1} in $\mathcal{L}$, the resulting sequence is equivalent to the original; that is,

$$U_n \cdots U_{i+1} U_i \cdots U_0 \;=\; U_n \cdots U_i U_{i+1} \cdots U_0$$

Let us continue to swap adjacent elements in $\mathcal{L}$ whenever their relative order is different from that in $\mathcal{L}'$. In this way, we slowly transform $\mathcal{L}$ into $\mathcal{L}'$. Since each swap preserves equivalence and linearity, $\mathcal{L}$ and $\mathcal{L}'$ must themselves be equivalent. $\Box$

The following corollary is the main result of this section. It is an immediate consequence of Corollary 5.18 and Theorem 5.19.

Corollary 5.20 *Let R be a stratified rulebase, and let $R_0, ..., R_k$ and $R'_0, ..., R'_{k'}$ be any two stratifications of R. Then,*

$$U_{R_k} \cdots U_{R_0} \;=\; U_{R'_{k'}} \cdots U_{R'_0}$$

Thus, by statement (5.4),

$$\text{FIXPOINT}(R_0, ..., R_k) \;=\; \text{FIXPOINT}(R'_0, ..., R'_{k'})$$

This establishes the main point of this section: The procedure FIXPOINT is a function only of R, and not of its stratification. We can therefore write FIXPOINT(R) instead of FIXPOINT($R_0, ...R_k$). Thus the semantics developed in Section 5.2 provides a meaning for a stratified rulebase that is independent of any particular stratification.

5.4 Data Complexity

Stratified negation does not increase the data complexity of hypothetical rules. That is, the stratified logic of hypothetical additions is data-complete for PSPACE, and the stratified logic of hypothetical additions and deletions is data-complete for EXPTIME. In keeping with our earlier terminology, we refer to the former as the *restricted* stratified logic, and to the latter as the *full* stratified logic.

The lower complexity bounds are immediate since both of the two stratified logics contains its negation-free version as a special case. The upper bounds are not difficult either. First note that given a stratified rulebase R, computing a stratification $R_0, ..., R_k$ is independent of the database, and thus adds only a constant term to the data complexity. Likewise for computing the negation-free rule sets $\overline{R}_0, ..., \overline{R}_k$.

Next, recall from section 4.1 that an interpretation M can be represented in exponential space in a straightforward way (exponential in the size of the database DB). Thus, in the procedure FIXPOINT(R), the computation of $\overline{M}_i$ from M_i can be performed in exponential time. Furthermore, by the argument given in section 4.1, $V^*_{\overline{R}_i}(\overline{M}_i)$ can be computed from $\overline{M}_i$ in exponential time (exponential in the size of the database). In evaluating FIXPOINT(R), there are k computations of this form. Since k is a constant independent of the database, the entire computation can be done in exponential time. Thus the data-complexity of the full stratified logic of hypothetical additions and deletions is in EXPTIME.

For the restricted stratified logic, a PSPACE upper bound is obtained by an adaptation of the iterative/recursive procedures of section 4.2. These procedures are defined below for the propositional case. The predicate case is a straightforward generalization.

In comparing these procedures to those of Section 4.2, note that each procedure has a new argument, i, that refers to a stratum in the rulebase. In addition, the procedure V replaces the earlier procedure T. The only difference between the two is that V joins the input facts to the inferred facts before returning, reflecting Definition 5.2. Finally, a fourth procedure, COMP, has been defined, which when given $M(DB)$, returns $\overline{M}(DB)$.

Procedure: LFP(i, R, DB)

  if $i = -1$ then return(DB);

  $S_1 \leftarrow$ COMP[LFP($i-1, R, DB$)];

  $S_2 \leftarrow$ V(i, R, DB, S_1);

  do until $S_1 = S_2$:

    $S_1 \leftarrow S_2$;

    $S_2 \leftarrow$ V(i, R, DB, S_1);

  end;

  return(S_2);

Procedure: V(i, R, DB, S_1)

  $S_2 \leftarrow DB$;

  for each rule $A \leftarrow \phi_1 ... \phi_m$ in $\overline{R}_i$ do:

    if MATCH(i, R, DB, S_1, ϕ_j) is true for each $1 \leq j \leq m$

      then $S_2 \leftarrow S_2 \cup \{A\}$;

  end;

  return($S_1 \cup S_2$);

Procedure: MATCH(R, DB, S, ϕ)

if ϕ is atomic and $\phi \in S$ then return(*true*);

if $\phi = A[add : B_1 ... B_k]$ and $\{B_1, ..., B_k\} \subseteq DB$ and $A \in S$

then return(*true*);

if $\phi = A[add : B_1 ... B_k]$ and $\{B_1, ..., B_k\} \not\subseteq DB$

and $A \in \text{LFP}(R, DB \cup \{B_1, ..., B_k\})$

then return(*true*);

else return(*false*);

Procedure: COMP(S_1)

$S_2 \leftarrow \{\}$;

for each atom P do:

if $P \in S_1$ then $S_2 \leftarrow S_2 \cup \{P\}$;

else $S_2 \leftarrow S_2 \cup \{\overline{P}\}$;

end;

return(S_2);

For each value of i, the above procedures implement essentially the same computation as the three procedures of section 4.2. There are two main differences. First, instead of using all the rules in R, only those in the i^{th} stratum are used. This shows up in the procedure V, which uses the negation-free rules in $\overline{R}_i$. Second, the initialization of the procedure LFP is different. This is because the fixpoint computation for the i^{th} stratum begins at $\overline{M}_{i-1}$, not at $\mathcal{I}$. Thus, the variable S_1 in LFP is initialized not to DB, but to $\overline{M}_{i-1}(DB)$. The recursion on i terminates when $i = -1$ (Intuitively, we are "below" the bottom stratum). For each value of i, $\text{LFP}(i, R, DB) = M_i(DB)$. Hence, if the top stratum is R_k, then by definition 5.6,

$$R, DB \vdash A \quad \textit{iff} \quad A \in \text{LFP}(k, R, DB)$$

It is not difficult to see that these procedures are correct. The argument is an elaboration of that presented at the end of Section 4.2. First, define M_i' to be the hypothetical structure such that $M'_i(DB) = \text{LFP}(i, R, DB)$ for every database DB. We argue by induction on i that $M'_i = M_i$ for $-1 \leq i \leq k$ (where $M_{-1} = \mathcal{I}$). The basis step is trivial, since $M'_{-1} = \mathcal{I} = M_{-1}$. To see the inductive step, suppose that $M'_{i-1} = M_{i-1}$. For the largest database $\mathcal{DB}$, it is trivially true that $M'_i(\mathcal{DB}) = \mathcal{DB} = M_i(\mathcal{DB})$. For any other database DB, the value of $M_i(DB)$ is determined by the values $M_i(DB')$ for databases DB' containing DB. Likewise for $M'_i(DB)$. In fact, if $M'_i(DB') = M_i(DB')$ for all $DB' \supset DB$, then $M'_i(DB) = M_i(DB)$. Thus $M'_i(DB) = M_i(DB)$ for all DB, by induction on DB. Hence $M'_i = M_i$.

Data complexity is not difficult to establish either. Again, the argument is an elaboration of that presented at the end of Section 4.2. As in that section, LFP is a recursive function in which each level of recursion takes polynomial time. Now, however, $\text{LFP}(i, R, DB)$ invokes itself recursively in two different ways: (i) through the function MATCH, and (ii) through its own procedure body, by calling $\text{LFP}(i-1, R, DB)$. With each recursive call, the database DB and the integer i change monotonically: either i is decremented by 1, or a new tuple is added to DB. However, i can change at most $k+1$ times before it reaches -1, and DB can change at most $|\mathcal{DB}|$ times before it saturates and equals $\mathcal{DB}$, the set of all ground atomic formulas. Thus, there are at most $k+1+|\mathcal{DB}|$ levels of recursion. But k is a constant independent of the database, and $|\mathcal{DB}|$ is polynomial in the database size. Thus, there are polynomially many levels of recursion, each taking polynomial time. So the entire computation can be done in polynomial space.

This establishes a PSPACE upper bound on the data-complexity of the restricted stratified logic of hypothetical additions. Combining this result with our comments from the beginning of the section gives the following two theorems.

Theorem 5.21 *The data-complexity of the restricted stratified logic of hypothetical additions is PSPACE-complete.*

Theorem 5.22 *The data-complexity of the full stratified logic of hypothetical additions and deletions is EXPTIME-complete.*

Chapter 6

Expressibility

Negation-as-failure increases the expressibility of the logic developed in Chapter 3. Without negation, the logic is *monotonic*: As entries are added to the database, the set of inferences can only increase. There are therefore many queries that the negation-free logic cannot express, including most relational algebra queries involving complementation. Augmenting the logic with negation-as-failure, as in Chapter 5, makes the logic non-monotonic and more expressive.

In this chapter[1], we characterize the queries that the augmented logic can express. We show in particular that the full stratified logic is expressively complete for EXPTIME. That is, any typed, generic query that can be computed in exponential time can be expressed as a stratified rulebase of hypothetical additions and deletions. We also show that the restricted stratified logic is expressively complete for PSPACE. That is, any typed, generic query that can be computed in polynomial space can be expressed as a stratified rulebase of hypothetical additions.

Informally, a query is said to be generic if it treats all constant symbols in the database "equally". For instance, when a rulebase contains no constant symbols, then no constant is given special treatment, and the rulebase expresses a generic query. For this reason, constant-free rulebases play a special role in our development. We show in particular that constant-free hypothetical rulebases express *exactly* the generic queries in PSPACE and EXPTIME, no more and no less.

It is worthwhile pointing out that these results are achieved without assuming a linear order on the data domain. This is assumption is sometimes used in the literature to achieve expressive completeness [29, 49, 17]. For hypothetical logics, however, if there

[1]This chapter elaborates on material that originally appeared in [8] and [10].

is no *a-priori* order on the domain, then an order can be generated by the rulebase and inserted into the database hypothetically. This technique is central to the expressibility results of this chapter.

The expressibility result for PSPACE is particularly interesting, because it establishes a strong connection between an important complexity class, polynomial space, and a well-known logic, intuitionistic logic. Other database languages have been developed that are expressively complete for PSPACE [1, 2], but the language of hypothetical insertion is unique in being a declarative language based on a well-established logic. It is also unique in having a practical, top-down proof procedure based on unification, in the logic-programming tradition [38]. Furthermore, the results on PSPACE can be extended, as shown in Chapter 8, to include the polynomial time hierarchy and its various levels, thereby linking intuitionistic logic to other important complexity classes.

6.1 Examples

Before developing the formal results of this chapter, we provide several examples that illustrate the expressibility of hypothetical reasoning with negation. These examples also supply an intuitive understanding of why a linearly ordered domain is not needed.

Chapter 4 showed that the logic of hypothetical additions and deletions is complete for exponential time. Despite this computational power, there are many simple queries that the negation-free logic cannot express. Consider for instance the query $EVEN$, which determines the parity of a database relation D: $EVEN$ returns *true* if D has an even number of entries, and *false* otherwise. $EVEN$ is non-monotonic since as entries are added to D, the value of $EVEN$ "flips back and forth" between *true* and *false*; i.e., once $EVEN$ becomes true, it does not remain true, as a monotonic query would.

Without negation, our hypothetical logic is monotonic and cannot express the query $EVEN$. To be precise: There is no hypothetical rulebase R such that for all databases DB,

$$R, DB \vdash EVEN \quad \textit{iff} \quad DB \text{ has an even number of entries of the form } D(x).$$

where $EVEN$ is a 0-ary predicate symbol. This is true not just of hypothetical logics, but of all monotonic logics, including Datalog (without negation), Prolog (without negation), and full classical logic. In this sense, the expressibility of negation-free rulebases is limited. Regardless of their computational power, there are many simple queries that they cannot express, simply because the queries are non-monotonic.

Augmenting a logic with negation-as-failure gives it the power to express such queries. Example 6.1, for instance, shows that $EVEN$ can be expressed as a stratified rulebase of hypothetical additions and deletions. (It is worthwhile noting that $EVEN$ cannot be expressed in Datalog even when stratified negation is allowed [16, 17].) The operation of this rulebase is best viewed as top-down. From this perspective, the first two rules select and remove elements one-by-one from the database relation D. As elements are removed, the rules "flip back and forth" between the two queries $EVEN$ and ODD until the relation D is empty. The third and fourth rules terminate the recursion by assigning an $EVEN$ parity to the empty relation. Notice that it does not matter in which order the elements of D are removed, since every order gives the same answer: Either every order results in a proof of $EVEN$ or every order results in a proof of ODD. This idea of order independence is central to the proof of expressive completeness given later in this chapter.

Example 6.1 Suppose R is the following collection of rules:

$$EVEN \leftarrow D(x),\ ODD[del : D(x)].$$

$$ODD \leftarrow D(x),\ EVEN[del : D(x)].$$

$$EVEN \leftarrow \sim SOMELEFT.$$

$$SOMELEFT \leftarrow D(x).$$

Then $R, DB \vdash EVEN$ *iff* DB has an even number of entries of the form $D(x)$. □

$EVEN$ can also be expressed in the restricted logic of stratified hypothetical additions. The idea is to mark elements instead of deleting them. In particular, after an element x has been selected, we hypothetically add the formula $MARK(x)$ to the database. In this way, we record which elements have been selected. We then select unmarked elements from the relation D, instead of undeleted elements. The following rules do exactly this.

Example 6.2 Suppose R is the following collection of rules:

$$EVEN \leftarrow SELECT(x),\ ODD[add : MARK(x)].$$
$$ODD \leftarrow SELECT(x),\ EVEN[add : MARK(x)].$$
$$EVEN \leftarrow \sim SOMELEFT.$$
$$SOMELEFT \leftarrow SELECT(x).$$
$$SELECT(x) \leftarrow D(x),\ \sim MARK(x).$$

Then $R, DB \vdash EVEN$ *iff* DB contains an even number of entries of the form $D(x)$. □

With negation-as-failure, it is possible to write simple hypothetical rulebases which solve complex problems. In the next example, the rulebase R solves the problem of finding a Hamiltonian path in a directed graph. Since this is an NP-complete problem, the example provides a simple demonstration of why the data complexity of hypothetical reasoning is not in P. As in the two examples above, the use of deletion in this example can be simulated by a combination of hypothetical addition and negation-as-failure.

Example 6.3 Suppose that DB is a database representing a directed graph. That is, $NODE(a) \in DB$ iff a is a node in the graph, and $EDGE(a, b) \in DB$ iff there is an edge in the graph from a to b. Suppose also that R is the following collection of rules:

$$YES \leftarrow NODE(x),\ PATH(x)[del : NODE(x)].$$
$$PATH(x) \leftarrow EDGE(x, y), NODE(y),\ PATH(y)[del : NODE(y)].$$
$$PATH(x) \leftarrow \sim SOMELEFT.$$
$$SOMELEFT \leftarrow NODE(x).$$

Then $R, DB \vdash YES$ iff the graph represented by DB has a directed Hamiltonian path. □

The rulebase in this example is best understood from the perspective of top-down inference. During inference, the rulebase tries to construct a Hamiltonian path one node at a time. The first rule selects a node x at which the path is to begin. The second rule is then applied repeatedly, selecting a node y connected by an edge to the last node in the path. Each time a node is selected, it is removed from the graph. In this way, no node is selected twice. The last two rules say that a Hamiltonian path has been found when there are no more nodes left in the graph, that is, when every node has been visited exactly once. Note that each node selection is non-deterministic, so in effect, the rulebase searches the graph for all possible Hamiltonian paths.

The next example illustrates perhaps most clearly why hypothetical rules with negation do not need a linear order. It is a variation on the example developed in Section 2.3.2. In that example, the database DB encoded a circuit whose input lines were linearly ordered. The linear order was used by the rulebase to select the input wires one at a time. We present the same example here but without the assumption of a linear order. Instead, we assume that the input wires are simply marked as such. That is, if a is a constant symbol denoting an input wire, then the database should contain the atomic formula $INPUT(a)$.

The main problem is to write rules which select the input wires one at a time. We can do this by using the techniques of the three examples above: Each time a wire is selected, we either delete it from the database or we mark it as selected so that we do not select it again. Negation-as-failure is then used to determine when there are no more unmarked wires left to select. The rules below do exactly this, exploiting the fact that as each wire is selected, it is "marked" with a 0 or a 1. We add these six rules to the four Horn rules of Section 2.3.2 that define the predicate $EVAL$.

$$VALID \leftarrow SELECT(x),\ VALID[add: ONE(x)],\ VALID[add: ZERO(x)]$$
$$VALID \leftarrow \sim SOMELEFT,\ EVAL.$$
$$SOMELEFT \leftarrow SELECT(x).$$
$$SELECT(x) \leftarrow INPUT(x),\ \sim MARKED(x).$$
$$MARKED(x) \leftarrow ONE(x).$$
$$MARKED(x) \leftarrow ZERO(x).$$

These rules repeatedly select an input wire and assign it a value of 0 or 1. The process is similar that of Section 2.3.2 except that now the order in which wires are selected is non-deterministic. This must be the case, since now there is no linear order on the wires and no way for the rulebase to prefer one ordering over another. After input values have been assigned in this way, the second rule invokes the predicate $EVAL$. $EVAL$ is true iff the output of the circuit evaluates to 1, as described in Section 2.3.2. If $EVAL$ is inferred for every possible set of input values, then the predicate $VALID$ is inferred, and only then. These rules therefore form a rulebase R such that $R, DB \vdash VALID$ iff DB represents a circuit whose output is 1 for every set of input values.

Notice that this result is true regardless of the order in which the input wires are selected. This is because circuit evaluation is a generic query; i.e., the constant symbols are all uninterpreted and have no special meaning. Under such conditions, it does not matter in which order they are selected. The next section describes the notion of genericity in more detail.

6.2 Generic Queries

The rest of this chapter shows that our two stratified hypothetical logics are expressively complete for EXPTIME and PSPACE, resp. That is, any generic database query in EXPTIME can be expressed as a rulebase of hypothetical additions and deletions, and any generic database query in PSPACE can be expressed as a rulebase of hypothetical additions.

Informally, a query is generic if it gives no special treatment to any of the constant symbols in the database. For instance, when a rulebase contains no constant symbols, then it treats all constants equally, and it expresses a generic query. For this reason, constant-free rulebases play a special role in our development. In particular, we show that constant-free hypothetical rulebases express exactly the generic queries in PSPACE and EXPTIME.

Before establishing these results, we need a more precise definition of what it means for a query to be generic and for a rulebase R to express such a query. This section provides these definitions. It is then shown that hypothetical rulebases without constant symbols are guaranteed to express only generic queries.

The first step is to give a precise definition of "relational database". The following definition is due to Chandra and Harel [15, 16].

Definition 6.1 *Let U be a countable set, called the universal data domain. A relational database DB of type $\overline{\alpha} = \langle \alpha_1, ..., \alpha_m \rangle$ is a tuple $\langle D, R_1, ..., R_m \rangle$ where D is a finite subset of U and R_i is an α_i-ary relation over D, i.e., $R_i \subseteq D^{\alpha_i}$. D is called the domain of DB, written $dom(DB)$.*

In logical systems such as ours, a relational database is represented as a set of ground atomic formulas. U is a universal set of constant symbols, and for each relation R_i there is a predicate symbol P_i whose ground atomic formulas represent R_i.

A *database query* is a mapping which takes a relational database as input and returns a relation as output. A database query is *generic* iff renaming the constants in the database causes the constants in the output to be renamed in the same way. This is known as the *consistency criterion*, and it formalizes the idea that the constant symbols in a database are uninterpreted.

To be more precise, we define a renaming to be a one-to-one mapping f of the universal domain U onto itself. A renaming can be extended to tuples, relations and databases in the obvious way. In particular, if DB is a database, then fDB denotes the renamed database. With this in mind, we give the following definition, which is due to Chandra and Harel [15, 16].

Definition 6.2 *A generic database query of type $\overline{\alpha} \rightarrow \alpha_0$ is a partial function ψ which takes a relational database DB of type $\overline{\alpha}$ and returns a relation $\psi(DB)$ over $dom(DB)$ of arity α_0. In addition, ψ must satisfy the following consistency criterion: For any database DB of type $\overline{\alpha}$, and any renaming f of the constant symbols,*

$$\psi(fDB) = f\psi(DB)$$

Recall from Section 3.1 that hypothetical inference is defined with respect to a particular inference system $[R, dom, pred]$, where *dom* and *pred* are finite sets of constant and predicate symbols, resp. As discussed in Chapter 4, *dom* is not fixed, but depends on the database DB that is being queried. In particular, *dom* consists of the data domain of DB plus the set of constant symbols appearing in R. (Thus, in the special case in which R is constant-free, *dom* is equal to the data domain of DB.) With this in mind, we give the following precise definition of expressibility.

Definition 6.3 *Suppose that ψ is a database query of type $\overline{\alpha} \rightarrow \alpha_0$. A hypothetical rulebase R expresses ψ iff for all relational databases DB of type $\overline{\alpha}$,*

$$R, DB \vdash Q(\overline{x}) \quad \textit{iff} \quad \overline{x} \in \psi(DB)$$

where Q is a predicate symbol of arity α_0.

Note that a hypothetical rulebase may express non-generic queries. For instance, consider the rulebase consisting of the single rule $Q(a,b) \leftarrow P(a)$, where a and b are constants. This rule expresses a query ψ for which $\psi(\{P(a)\}) = (a,b)$. ψ is not generic since if we rename a to be c, then $\psi(\{P(c)\}) = \{\}$, and in particular, $\psi(\{P(c)\}) \neq \{(c,b)\}$. Thus, if DB is renamed, then $\psi(DB)$ is not necessarily renamed in the same way. ψ is non-generic for another reason as well. Since $\psi(\{P(a)\}) = (a,b)$, the set $\psi(DB)$ may contain constant symbols which are not in the domain of DB, such as the constant b. This also contradicts the definition of genericity.

In this example, the rulebase expressing ψ contains constant symbols. This is the source of its non-genericity. In particular, if a rulebase contains no constant symbols, then it is guaranteed to be generic. This is established by the next lemma and its corollary.

Lemma 6.1 *Let f be a renaming. Then, for any hypothetical rulebase R, any database DB, and any hypothetical query ϕ,*

$$R, DB \vdash \phi \quad \textit{iff} \quad fR, fDB \vdash f\phi$$

Proof: Follows by a straightforward induction over the length of derivations. In particular, each of the inference rules in Definition 3.4 is invariant under a renaming of the constant symbols. $\Box$

Corollary 6.2 *Any query expressed by a constant-free rulebase is generic.*

Proof: Let R be a constant free rulebase, and let ψ be a query expressed by R. Thus, $R, DB \vdash Q(\overline{x})$ iff $\overline{x} \in \psi(DB)$, for some predicate symbol Q. Since R is constant free, the domain of the inference system is just the domain of the database $dom(DB)$. Hence, $\psi(DB)$ is a relation over $dom(DB)$. ψ thus satisfies the first condition of genericity.

We must also show that ψ satisfies the consistency criterion. To see this, first note that since R is constant-free, $R = fR$ for any renaming f. Thus,

$$
\begin{array}{lll}
 & \overline{x} \in \psi(DB) & \\
\textit{iff} & R, DB \vdash Q(\overline{x}) & \text{by Definition 6.3.} \\
\textit{iff} & fR, fDB \vdash Q(f\overline{x}) & \text{by Lemma 6.1.} \\
\textit{iff} & R, fDB \vdash Q(f\overline{x}) & \text{since } R = fR. \\
\textit{iff} & f\overline{x} \in \psi(fDB) & \text{by definition.}
\end{array}
$$

This is true for any database DB and any renaming f. In particular, we can use fDB instead of DB, and f^{-1} instead of f.[2] Furthermore, for any set S of tuples, $\overline{x} \in S$ iff $f\overline{x} \in fS$. Keeping these points in mind, we get the following:

$$
\begin{array}{ll}
 & \overline{x} \in \psi(fDB) \\
\textit{iff} & f^{-1}\overline{x} \in \psi(f^{-1}fDB) \\
\textit{iff} & f^{-1}\overline{x} \in \psi(DB) \\
\textit{iff} & ff^{-1}\overline{x} \in f\psi(DB) \\
\textit{iff} & \overline{x} \in f\psi(DB)
\end{array}
$$

Thus $\psi(fDB) = f\psi(DB)$ for any renaming f. Hence ψ is generic. $\Box$

From Corollary 6.2 and from the upper complexity bounds of Section 5.4, it follows that any hypothetical rulebase which in constant-free expresses a generic query in EXPTIME. In the rest of this chapter, we prove the converse, that any generic query in EXPTIME can be expressed as a constant-free, hypothetical rulebase. We prove a similar result for PSPACE and the restricted logic of hypothetical additions. Since the PSPACE result is slightly simpler to prove, we present it first.

[2] Since a renaming f is one-to-one and onto, its inverse f^{-1} always exists and is itself a renaming.

6.3 PSPACE Expressibility

This section shows that any typed generic query that can be computed in polynomial space can be expressed as a stratified rulebase of hypothetical additions. In particular, this section proves the following theorem.

Theorem 6.3 *Suppose that φ is a generic database query of type $\overline{\alpha} \rightarrow \alpha_0$ which is computable in polynomial space. Then there is a stratified rulebase $R(\varphi)$ of hypothetical additions which expresses φ. Furthermore, $R(\varphi)$ can be constructed so that it is constant-free.*

The last sentence in this theorem leads to an interesting corollary. We have already established that a constant-free stratified rulebase of hypothetical additions expresses a generic query in PSPACE. Theorem 6.3 gives us the converse. We therefore have the following characterization, which is a main result of this chapter.

Corollary 6.4 *Constant-free stratified rulebases of hypothetical additions express exactly the generic database queries in PSPACE.*

The rest of this section proves Theorem 6.3. The proof relies on the Turing machine simulations developed in Section 4.4. In this respect, it is similar to other expressibility proofs in the literature (e.g., [29, 49]). One difference, however, is that the results presented here do not assume that the data domain is linearly ordered. Linear orders are typically used to simulate counters, which in turn, are used to simulate the movement of Turing-machine tape heads. Stratified hypothetical rules do not need an *a priori* domain ordering since they can generate a linear order for themselves: First they "guess" a linear order non-deterministically, and then they add it to the database hypothetically. One difficulty with this approach is that our machine simulations are sensitive to which linear order is used; so that different "guesses" can lead to different query answers. We show, however, that generic queries are robust in that they return the same answer no matter which linear order is used.

It should also be pointed out that the proof of Theorem 6.3 does involve another assumption which is common in the literature, namely, that the data domain has at least two elements in it. This assumption is needed in order to build a counter. Given a domain of size n, we use the n domain elements to simulate a counter in base n. Each domain element is treated as a separate digit, and at least two digits are needed to build a powerful enough counter.

With this in mind, we prove Theorem 6.3 by first reducing it to the problem of expressing yes/no queries, that is, queries which return either *true* or *false*. Most of this section is thus devoted to proving the following lemma.

Lemma 6.5 *Let ψ be a generic query of type $\overline{\alpha} \rightarrow 0$ which is computable in polynomial space. Then there is a stratified rulebase $R(\psi)$ of hypothetical additions which expresses ψ. That is, for all databases DB of type $\overline{\alpha}$,*

$$R(\psi), DB \vdash YES \quad \textit{iff} \quad \psi(DB) = true$$

where YES is a predicate symbol of arity zero. Furthermore, $R(\psi)$ can be constructed so that it is constant free.

Given this lemma, Theorem 6.3 can be proved in a straightforward way as follows:

Proof: (of Theorem 6.3) Let φ be a generic database query of type $\langle \alpha_1, ..., \alpha_m \rangle \rightarrow \alpha_0$ which is computable in polynomial space. Now define ψ to be the query of type $\langle \alpha_0, \alpha_1, ..., \alpha_m \rangle \rightarrow 0$ such that for all databases DB of type $\langle \alpha_1, ..., \alpha_m \rangle$,

$$\psi[DB + P_0(\overline{x})] = true \quad \textit{iff} \quad \overline{x} \in \varphi(DB)$$

where P_0 is a predicate symbol of arity α_0 not appearing in DB. Note that $DB + P_0(\overline{x})$ is a database of type $\langle \alpha_0, \alpha_1, ..., \alpha_m \rangle$. For all other databases of this type, define the value of ψ to be *false*. It is not difficult to see that because φ is generic and in PSPACE, so is ψ. Thus a rulebase $R(\psi)$ exists as described in Lemma 6.5. The rulebase $R(\varphi)$ of Theorem 6.3 is then constructed by adding to $R(\psi)$ the following rule:

$$Q(\overline{x}) \leftarrow YES[add : P_0(\overline{x})].$$

In reasoning with this rule, the inference system effectively generates all possible α_0-tuples $\overline{x}$ over the data domain $dom(DB)$. For each such tuple, $P_0(\overline{x})$ is hypothetically added to the database, and then $R(\psi)$ is used to determine whether $\overline{x} \in \varphi(DB)$. That is, for all $\overline{x} \in [dom(DB)]^{\alpha_0}$,

$$
\begin{array}{lll}
 & R(\varphi), DB \vdash Q(\overline{x}) & \\
\textit{iff} & R(\psi), DB + P_0(\overline{x}) \vdash \mathit{YES} & \text{by construction.} \\
\textit{iff} & \psi[DB + P_0(\overline{x})] = \mathit{true} & \text{since } R(\psi) \text{ expresses } \psi. \\
\textit{iff} & \overline{x} \in \varphi(DB) & \text{by definition.}
\end{array}
$$

□

Thus, Theorem 6.3 is true if Lemma 6.5 is true. The rest of this section is devoted to the proof of Lemma 6.5.

6.3.1 The Rulebase $R'(M)$

If ψ is a yes/no query computable in PSPACE, then there is an APTIME machine M which accepts the language $\{DB \mid \psi(DB) = \mathit{true}\}$. Section 4.4 showed how to construct a rulebase $R(M)$ which encodes the computations of an arbitrary APTIME machine. This construction, however, is not exactly the rulebase needed here. The construction of Section 4.4 assumed that the initial *id* of machine M was encoded as a database $DB(\overline{s})$, where $\overline{s}$ is the machine's input string. $DB(\overline{s})$ was also assumed to contain a counter. $DB(\overline{s})$ is therefore not the same as the database DB which we are given here. We must therefore provide rules which when given DB, will construct $DB(\overline{s})$. Adding these rules to the rulebase $R(M)$ gives a new rulebase $R'(M)$. This section describes the construction of $R'(M)$.

The construction is given in three stages. We first provide rules which generate a linear order for the data domain. We then provide rules which use the linear order to build a counter. We finally provide rules which encode the database DB as a string suitable for input to a Turing machine.

All the rules described in this section are constant free. Since the rulebase $R(M)$ constructed in Section 4.4 is also constant free, the rulebase $R'(M)$ is therefore constant free as well. This fact will be important in proving Lemma 6.5.

Generating a Linear Order

If the data domain is linearly ordered, then a counter can be implemented with Horn rules. In fact, by using predicates of arity k, a counter from 0 to n^k-1 can be constructed in a straightforward way. For this reason, the assumption of a linearly ordered domain is sometimes used in the literature [29, 49], especially when expressibility results are established in terms of complexity classes, as in Lemma 6.5. With hypothetical rules, however, this assumption is unnecessary. This section provides a set of hypothetical rules which "guess" a linear order and hypothetically add it to the database. As we shall see, because the machine M computes a generic yes/no query, it is insensitive to the particular order that is used; i.e., M computes the same value (*true* or *false*) for every linear order. A similar idea was used in implementing the query $EVEN$ in Examples 6.1 and 6.2.

The rules below use the techniques of Example 6.2 to guess and hypothetically assert a linear order on the data domain. From the perspective of top-down inference, the rules non-deterministically select elements from the domain, one after another. As each domain element y is selected, it is used to hypothetically extend a linear order. It is also "marked" so that it will not be selected again. Marking is done by hypothetically adding the formula $MARK(y)$ to the database. Selection of domain elements continues until there are no unmarked elements left.

$$YES \leftarrow SELECT(x),\ ORDER(x)[add: FIRST_1(x),\ MARK(x)].$$
$$ORDER(x) \leftarrow SELECT(y),\ ORDER(y)[add: NEXT_1(x,y),\ MARK(y)].$$
$$ORDER(x) \leftarrow \sim SOMELEFT,\ ACCEPT[add: LAST_1(x)].$$
$$SOMELEFT \leftarrow SELECT(y).$$
$$SELECT(y) \leftarrow \sim MARK(y).$$

For each different ordering of the domain, a different hypothetical database is generated. When the domain elements are selected in the order $a_1, a_2 ... a_n$, the following entries are hypothetically added to the database:

$$\begin{gathered} FIRST_1(a_1),\ NEXT_1(a_1,a_2),\ NEXT_1(a_2,a_3)\ ... \\ ...\ NEXT_1(a_{n-1},a_n),\ LAST_1(a_n). \end{gathered} \tag{6.1}$$

This is in addition to the original database predicates $P_0, ..., P_m$.

After a linear order has been inserted, the third rule tries to infer the atom $ACCEPT$. This invokes a simulation of machine M, as described in Section 4.4, so that $ACCEPT$ is inferred iff the machine accepts its input.

Note that the above rules select a linear order non-deterministically. Furthermore, they infer the atom YES iff $ACCEPT$ is inferred for at least one of these orders. The behavior of the rules is thus described formally as follows:

$$R'(M), DB \vdash YES \quad \textit{iff} \quad \vee_L \; [R'(M), DB + L \vdash ACCEPT] \tag{6.2}$$

Here L is a set of atomic formulas representing a linear order, such as formulas (6.1) above; and the disjunction $\vee_L$ is over all linear orders L of the data domain.

As we shall see, if M computes a generic query, then it is insensitive to which linear order is used. That is, either $ACCEPT$ is inferred for all linear orders, or for none. In this case, YES is inferred iff M accepts its input.

Counting

Each linear ordering of the data domain $dom(DB)$ provides a counter from 0 to $n-1$. In addition, it induces an ordering on $[dom(DB)]^k$, the space of k-tuples of domain elements. This provides for the possibility of constructing a counter from 0 to $n^k - 1$. Such a counter can be used to represent n^k points in time and n^k points on machine tape, which is adequate for representing the computations of an APTIME machine. This section describes the construction of a counter assuming that the data domain is linearly ordered.

The rules below define three predicates: $FIRST(\overline{x})$, $NEXT(\overline{x}, \overline{y})$, and $LAST(\overline{y})$, where $\overline{x}$ and $\overline{y}$ are k-tuples. $FIRST(\overline{x})$ and $LAST(\overline{y})$ are true iff $\overline{x}$ and $\overline{y}$ represent the integers 0 and $n^k - 1$, respectively. $NEXT(\overline{x}, \overline{y})$ is true iff $\overline{y}$ represents the integer $\overline{x} + 1$. To define these predicates, the rules implement a counter in base n. Each of the n domain elements is treated as a distinct digit, where the linear order on the domain orders the digits. For clarity, the rules below use the symbols 0 and 9 to represent the first and last elements of the domain, resp. These symbols are variables, not constants.

That is, they can be replaced by the symbols y and z without changing the meaning of the rules.

$$FIRST(0, ..., 0) \leftarrow FIRST_1(0).$$
$$LAST(9, ..., 9) \leftarrow LAST_1(9).$$
$$NEXT(x_k, ...x_4, x_3, x_2, v,\ x_k, ...x_4, x_3, x_2, u) \leftarrow NEXT_1(u, v).$$
$$NEXT(x_k, ...x_4, x_3, v, 0,\ x_k, ...x_4, x_3, u, 9) \leftarrow NEXT_1(u, v), FIRST_1(0), LAST_1(9).$$
$$NEXT(x_k, ...x_4, v, 0, 0,\ x_k, ...x_4, u, 9, 9) \leftarrow NEXT_1(u, v), FIRST_1(0), LAST_1(9).$$
$$\cdots$$
$$NEXT(x_k, v, 0, 0, ...0,\ x_k, u, 9, 9, ...9) \leftarrow NEXT_1(u, v), FIRST_1(0), LAST_1(9).$$
$$NEXT(v, 0, 0, 0, ...0,\ u, 9, 9, 9, ...9) \leftarrow NEXT_1(u, v), FIRST_1(0), LAST_1(9).$$

Each of the rules defining $NEXT$ encodes the process of "carrying a 1", that is, the process of adding 1 to the number $(x_k, ...x_i, u, 9, ..., 9)$ to get a number of the form $(x_k, ...x_i, u+1, 0, ...0)$. Since our numbers have k digits, there are k places where a 1 can be carried, and so there are k rules.

Initializing the Machine

With the above counter, we can write rules which compute the initial *id* of the machine M. Following the construction of Section 4.4, we use the following predicates to represent the machine, where $\overline{p}$ and $\overline{t}$ are k-tuples:

- $CELL^c(\overline{p}, \overline{t})$ means that at time $\overline{t}$, the tape cell at position $\overline{p}$ contains symbol c.
- $CONTROL^q(\overline{p}, \overline{t})$ means that at time $\overline{t}$, the finite control is in state q and is scanning the cell at position $\overline{p}$.

To specify the initial *id*, the rules must specify the initial tape contents as well as the initial state and position of the finite control. Of these two tasks, initializing the finite control is by far the easiest. It can be accomplished by a single rule which says that at time 0, the control is in state q_0 and is scanning the tape cell at position 0:

$$CONTROL^{q_0}(\overline{p}, \overline{t}) \leftarrow FIRST(\overline{p}),\ FIRST(\overline{t}).$$

The rest of this section describes the initialization of the machine tape. This amounts to encoding the database DB as a string suitable for input to a Turing machine. The database itself consists of a set of relations $P_0, P_1, ..., P_m$ of arities $\alpha_0, \alpha_1, ..., \alpha_m$, resp. These relations must be encoded "onto the input tape" of machine M; that is, they must be encoded as a set of atomic formulas of the form $CELL^c(\overline{p}, \overline{t})$. Furthermore, hypothetical rules must be provided to carry out this encoding.

Any reasonable machine-encoding of the relations will do, and it is convenient to use bit-maps. That is, an initial segment of the machine tape is divided into $m+1$ blocks, and a bit-map representation of P_i is stored in the i^{th} block. A bit-map is a contiguous segment of tape in which each tape cell corresponds to a possible database tuple. If the tuple is in the database, then the cell contains a 1; otherwise, a 0. Because the data domain is linearly ordered, a tuple $(x_1, ..., x_{\alpha_i})$ can be treated as a tape address. Thus, in the bit-map representation of P_i, the tape cell with address $(x_1, ..., x_{\alpha_i})$ contains a 1 iff $P_i(x_1, ..., x_{\alpha_i})$ is in the database. The use of negation-as-failure is crucial to setting the tape cell to 0 when $P_i(x_1, ..., x_{\alpha_i})$ is *not* in the database.

Below, is a set of stratified Horn rules which set up the blocks of tape and store the bit-maps. They define three predicates $INITIAL^0$, $INITIAL^1$, and $INITIAL^b$, each of arity $k+1$. Informally, $INITIAL^c(\overline{p})$ means that the tape cell at position $\overline{p}$ is to be initialized to symbol c, where c is 0, 1 or b. (Recall that b denotes a blank.) For clarity, the definition of these predicates is done in several steps, utilizing several rule sets.

The first rule-set computes the bit-map representation for each database relation P_i. It defines new predicates $BITMAP_i^0$ and $BITMAP_i^1$ of arity $\alpha_i + 1$. Informally, $BITMAP_i^c(\overline{x})$ means that position $\overline{x}$ in the bit map for P_i contains the symbol c, where c is either 0 or 1.

$$BITMAP_i^1(\overline{x}) \leftarrow P_i(\overline{x}).$$

$$BITMAP_i^0(\overline{x}) \leftarrow \sim P_i(\overline{x}).$$

Each bit-map is stored in a block of tape containing n^γ cells, where $\gamma = 1 + max\{\alpha_0, ..., \alpha_m\}$. This assures that each block contains enough space to store any of the predicates. Each block also contains unused cells, which will be filled with blanks.

In this way, the bit-maps are separated by blank tape and can thereby be distinguished from each other by the program in the finite control.

Each bit-map is stored at the low end of a block. For this reason, the bit corresponding to $BITMAP_i^c(x_1, ..., x_{\alpha_i})$ is stored in block i at address $(0, 0, ..., 0, x_1, ..., x_{\alpha_i})$, where the symbol 0 stands for the first domain element. The following rules define the predicates $BLOCK_i^0$ and $BLOCK_i^1$, of arity $\gamma + 1$, which represent the i^{th} block of the tape. Informally, $BLOCK_i^c(\overline{y})$ means that position $\overline{y}$ in block i contains the symbol c, where c is either 0 or 1.

$$BLOCK_i^0(u, ..., u, \overline{x}) \leftarrow FIRST_1(u),\ BITMAP_i^0(\overline{x}).$$
$$BLOCK_i^1(u, ..., u, \overline{x}) \leftarrow FIRST_1(u),\ BITMAP_i^1(\overline{x}).$$

The blocks are placed in consecutive order at the beginning of the machine tape. Thus, the element $BLOCK_i^c(x_1, ..., x_\gamma)$ is stored on tape at address $(0, ..., 0, i, x_1, ..., x_\gamma)$, where the symbol i in the address stands for the i^{th} domain element and is used to identify the i^{th} block.[3] The following rules encode the blocks onto the machine tape. There are two rules for each block, and they all contribute to defining the predicates $INPUT^0$ and $INPUT^1$, of arity $k+1$, which represent the tape. Informally, $INPUT^c(\overline{p})$ means that the tape cell at position $\overline{p}$ initially contains symbol c, where c is 0 or 1.

$$INPUT^0(u, ..., u, y, \overline{x}) \leftarrow FIRST_1(u),\ DOM_i(y),\ BLOCK_i^0(\overline{x}).$$
$$INPUT^1(u, ..., u, y, \overline{x}) \leftarrow FIRST_1(u),\ DOM_i(y),\ BLOCK_i^1(\overline{x}).$$

The literal $DOM_i(y)$ means that y is the i^{th} element of the data domain. The predicates DOM_0, DOM_1, ... ,DOM_m are defined by $m + 1$ rules as follows:

$$DOM_0(y) \leftarrow FIRST_1(y).$$
$$DOM_i(y) \leftarrow DOM_{i-1}(x),\ NEXT_1(x, y).$$

[3] This encoding assumes that the data domain contains at least $m + 1$ elements, one for each block. However,if the number i is encoded in binary, then only two domain elements are needed. In this case, more tuple-positions would be needed to designate the i^{th} block. In fact, $\lceil log_2(m+1) \rceil$ positions would be required, one for each digit in the binary representation of i. For clarity's sake, we do not elaborate on this scheme.

The predicates $INPUT^c$ encode the 0's and 1's of the bit-maps, but not the intervening blanks. The following rules put a blank in all tape cells which have not yet been initialized. They define the predicates $INITIAL^0$, $INITIAL^1$ and $INITIAL^b$, of arity $k+1$, which represent the complete initialization of the machine tape. Informally, $INITIAL^c(\overline{p})$ means that the tape cell at position $\overline{p}$ initially contains the symbol c, where c is 0, 1, or b.

$$
\begin{aligned}
&INITIAL^0(\overline{p}) \leftarrow INPUT^0(\overline{p}).\\
&INITIAL^1(\overline{p}) \leftarrow INPUT^1(\overline{p}).\\
&INITIAL^b(\overline{p}) \leftarrow \sim ZERO\_OR\_ONE(\overline{p}).\\
&ZERO\_OR\_ONE(\overline{p}) \leftarrow INPUT^0(\overline{p}).\\
&ZERO\_OR\_ONE(\overline{p}) \leftarrow INPUT^1(\overline{p}).
\end{aligned}
$$

Finally, we must state that the initial tape contents are just the tape contents at time 0. We thus write a rule of the following form for $c \in \{0, 1, b\}$:

$$CELL^c(\overline{p}, \overline{t}) \leftarrow INITIAL^c(\overline{p}),\ FIRST(\overline{t}).$$

Given an alternating PTIME machine M, we add the rules of this section and those of Sections 6.3.1 and 6.3.1 to the rulebase $R(M)$ described in Section 4.4. The collection of all these rules is called $R'(M)$. It is a stratified rulebase of hypothetical additions.

6.3.2 Basic Properties of $R'(M)$

The rules in $R'(M)$ can be divided into three broad categories: (i) those that encode the database onto the machine tape, (ii) those that simulate the machine M, and (iii) those that generate a linear order. This section reviews the rules in each of these categories and describes some of their important formal properties. These properties are central to the proof of expressive completeness given in the next section.

The rules in category (i) assume that the domain of the database is linearly ordered. They also assume that the linear order is represented as a set of atomic formulas L. Given a linear order L and a database DB, these rules generate a set of atomic formulas of the form $CELL^c(\overline{p}, \overline{t})$. These formulas represent an encoding of the database on the tape of a Turing machine; that is, they represent a string of 1's, 0's and blanks. The rules therefore define a mapping s which takes a set $DB + L$ as input and returns a string $s(DB + L)$ as output. Although the machine tape is infinite, the machine only uses a finite portion, i.e., the first n^k tape cells, where n is the size of the domain of DB. We assume, therefore, that $s(DB + L)$ is a finite string of this length. We shall prove an important property of this string shortly, but we first use it to describe the behavior of the rules in $R'(M)$.

The rules in category (ii) try to infer the atom $ACCEPT$. In doing so, they simulate the computations of the Turing machine M, as described in Section 4.4. The input to the machine is the string $s(DB+L)$ described above. The main property of these rules is that given the database DB and the linear order L, they infer the atom $ACCEPT$ iff the machine accepts its input. This is described formally as follows:

$$R'(M), DB + L \vdash ACCEPT \quad \textit{iff} \quad M[s(DB + L)] = \textit{true} \tag{6.3}$$

Here we are treating the machine M as a mapping from strings to boolean values.

The rules in category (iii) try to infer the atom YES. In doing so, they guess a linear order non-deterministically and insert it into the database hypothetically. After a linear order has been generated in this way, the rules try to infer the atom $ACCEPT$, which invokes the Turing machine simulation described above. The main property of these rules is that given a database DB, they infer the atom YES iff $ACCEPT$ is inferred for some linear order L. This is expressed formally by statement (6.2) in Section 6.3.1, which we reproduce here:

$$R'(M), DB \vdash YES \quad \textit{iff} \quad \vee_L \ [R'(M), DB + L \vdash ACCEPT]$$

Here the disjunction $\vee_L$ is over all linear orders L of the data domain.

By combining this with statement 6.3 above, we get the following statement, which summarizes the behavior of the rulebase $R'(M)$:

$$R'(M), DB \vdash YES \quad \textit{iff} \quad \vee_L \ M[s(DB + L)] \tag{6.4}$$

This statement is true for any alternating PTIME machine M.

As described above, the rules in category (i) define a mapping s which encodes the database DB as a string. This mapping is a function of both the database DB and the linear order L; so that in general, different linear orders will result in different encodings of the same database. The rest of this section shows that this encoding is generic function of $DB + L$. That is, if the constant symbols in the linear order and the database are renamed in the same way, then the encoding does not change. This property, which is essential to the proof of expressive completeness, is described formally by the following lemma.

Lemma 6.6 $s(DB + L) = s[f(DB + L)]$ *for any renaming* f.

Proof: We focus on individual characters in the string $s(DB + L)$. In particular, let S_i^c be a zero-ary predicate which is true iff the i^{th} character of the string is c, for $0 \leq i \leq n^k - 1$. S_i^c thus represents a yes/no database query. We claim that this query is generic. To see this, note that for any fixed i, the predicate S_i^c can be defined by a set of constant-free rules. In particular, it can be defined by the following set of $i + 2$ rules:[4]

$$
\begin{aligned}
&\mathit{ENCODE}_0(\overline{x}) \leftarrow \mathit{FIRST}(\overline{x}).\\
&\mathit{ENCODE}_1(\overline{y}) \leftarrow \mathit{ENCODE}_0(\overline{x}),\ \mathit{NEXT}(\overline{x}, \overline{y}).\\
&\mathit{ENCODE}_2(\overline{y}) \leftarrow \mathit{ENCODE}_1(\overline{x}),\ \mathit{NEXT}(\overline{x}, \overline{y}).\\
&\qquad \ldots\\
&\mathit{ENCODE}_i(\overline{y}) \leftarrow \mathit{ENCODE}_{i-1}(\overline{x}),\ \mathit{NEXT}(\overline{x}, \overline{y}).\\
&S_i^c \leftarrow \mathit{ENCODE}_i(\overline{y}),\ \mathit{INITIAL}^c(\overline{y}).
\end{aligned}
$$

In this rule set, the first $i + 1$ rules compute the encoding of the integer i as a tuple $\overline{y}$. These rules use the predicates *FIRST* and *NEXT* which are defined in the rulebase $R'(M)$. The last rule examines position i on the machine tape to see if it contains the symbol c. This rule uses the predicate $\mathit{INITIAL}^c$, also defined in $R'(M)$.

Recall that the predicates $\mathit{INITIAL}^c$ and *NEXT* are defined only for the first n^k positions on the machine tape. The predicate S_i^c is thus properly defined for $0 \leq i \leq$

[4] These rules are presented only for the purpose of proving Lemma 6.6. They are not part of the rulebase $R'(M)$ constructed in Section 6.3.1.

$n^k - 1$, as required. (For larger values of i, these rules provide a value for S_i^c, but that value is unimportant.)

Let R_i^c be the rulebase consisting of the $i+2$ rules given above together with the necessary rules from $R'(M)$. R_i^c can be constructed for any fixed i, and it provides a complete definition of the predicate S_i^c. Furthermore, R_i^c is constant free, so by Corollary 6.2, it defines a generic query. Thus, for any renaming f,

$$R, DB + L \vdash S_i^c \quad \textit{iff} \quad R, f(DB + L) \vdash S_i^c$$

This is true for *any* value of i, and in particular, it is true for $0 \le i \le n^k - 1$. Thus, the i^{th} character of the string $s(DB + L)$ is invariant under a renaming of $DB + L$. Since this is true for any character in the string, the entire string is invariant. That is, $s(DB + L) = s[f(DB + L)]$. □

Lemma (6.6) says that renaming both the database and the linear order in the same way has no effect on the string which is provided as input to the Turing machine M. To illustrate this point graphically, consider the three diagrams below. In each diagram, the sequence of 1's, 0's and blanks represents an encoding of a database as a string, and the entries beneath show the interpretation given to each string position.[5] Diagram 1 shows the encoding of the database $DB = \{P(a,b),\ P(b,b),\ Q(b)\}$ under the linear order $a < b$. Diagram 2 shows the encoding when the linear order is changed, and diagram 3 shows the encoding when both the linear order and the database are changed.

1. Encoding the database $\{P(b,a),\ P(b,b),\ Q(b)\}$ under the linear order $a < b$:

0	0	1	1	0	1	b	b
$P(a,a)$	$P(a,b)$	$P(b,a)$	$P(b,b)$	$Q(a)$	$Q(b)$	–	–

[5]According to Section 6.3.1, each block should contain eight tape cells, the final four being blank. For brevity, we have shown only four tape cells per block.

2. Encoding the database $\{P(b,a),\ P(b,b),\ Q(b)\}$ under the linear order $b < a$:

1	1	0	0	1	0	b	b
$P(b,b)$	$P(b,a)$	$P(a,b)$	$P(a,a)$	$Q(b)$	$Q(a)$	–	–

3. Encoding the database $\{P(a,b),\ P(a,a),\ Q(a)\}$ under the linear order $b < a$:

0	0	1	1	0	1	b	b
$P(b,b)$	$P(b,a)$	$P(a,b)$	$P(a,a)$	$Q(b)$	$Q(a)$	–	–

Diagrams 1 and 2 show clearly that changing just the linear order can change the encoding of the database. More importantly, diagrams 1 and 3 show that renaming both the database and the linear order in the same way does not change the encoding.

6.3.3 Expressive Completeness

Section 6.3.2 described some of the basic properties of the rulebase $R'(M)$. This section uses them to complete the proof of Lemma 6.5, and thus establish expressive completeness. In particular, we show that for an appropriate choice of M, a rulebase of the form $R'(M)$ can express any generic yes/no query. $R'(M)$ can thus be used as the rulebase $R(\psi)$ in Lemma 6.5.

The behavior of $R'(M)$ is given by statement (6.4) in Section 6.3.2, which we reproduce here:

$$R'(M), DB \vdash YES \quad \textit{iff} \quad \vee_L \;\; M[s(DB+L)] \tag{6.5}$$

Here, the disjunction $\vee_L$ is over all linear orders L of the data domain, and $s(DB+L)$ is the encoding of the database DB as a string. This statement is true for any alternating PTIME machine M. Note that in this statement, we are treating the machine M as a mapping from strings to boolean values.

The encoding $s(DB + L)$ is a function of both the database DB and the linear order L. Each linear order therefore results in a different encoding. Since each of these encodings is equally good, we shall pick a particular order L_0 and consider only the encoding defined by it. It is not difficult to see that any linear order of the data domain can be derived from L_0 by a renaming. In fact,

$$\{L \mid L \textit{ is a linear order}\} \;=\; \{fL_0 \mid f \textit{ is a renaming}\}$$

where we are restricting ourself here to renamings f which map the data domain $dom(DB)$ onto itself.

We can use L_0 to select Turing machines for computing database queries. That is, given a yes/no query ψ, there is some machine M which evaluates ψ when the database is encoded onto the tape using L_0. Formally $\psi(DB) = M[s(DB + L_0)]$. If, in addition, ψ is computable in polynomial space, then we can assume that M is an alternating PTIME, and we can encode it as a rulebase $R'(M)$ satisfying statement (6.5). In this case, we get the following result, which exploits the genericity of the encoding s:

$$\begin{array}{lll}
 & R'(M), DB \vdash \mathit{YES} & \\
\textit{iff} & \vee_L \; M[s(DB + L)] & \text{where } L \text{ ranges over all linear orders.} \\
\textit{iff} & \vee_f \; M[s(DB + fL_0)] & \text{where } f \text{ ranges over all renamings.} \\
\textit{iff} & \vee_f \; M(s[f^{-1}(DB + fL_0)]) & \text{since } s \text{ is generic, by Lemma 6.6.} \\
\textit{iff} & \vee_f \; M[s(f^{-1}DB + L_0)] & \\
\textit{iff} & \vee_f \; \psi(f^{-1}DB) & \text{by the definition of } M.
\end{array}$$

In short,

$$R'(M), DB \vdash \mathit{YES} \quad \textit{iff} \quad \vee_f \; \psi(f^{-1}DB)$$

This statement says that the rulebase $R'(M)$ does not necessarily express the query ψ, even though the machine M computes ψ. Instead, it expresses the query $\vee_f \psi(f^{-1}DB)$. However, if ψ is generic, then we can simplify this expression. In particular, $\psi(fDB) = \psi(DB) = \psi(f^{-1}DB)$ for any renaming f, and so $\vee_f \psi(f^{-1}DB) = \vee_f \psi(DB) = \psi(DB)$. Hence,

$$R'(M), DB \vdash \mathit{YES} \quad \textit{iff} \quad \psi(DB)$$

Thus, if ψ is a generic yes/no query in PSPACE, then it is expressed by the stratified hypothetical rulebase $R'(M)$. This establishes Lemma 6.5, and completes the proof of expressive completeness.

6.4 EXPTIME Expressibility

This section shows that any database query which is computable in exponential time can be expressed as a stratified rulebase of hypothetical additions and deletions. Furthermore, those rulebases which are constant free express exactly the generic queries in EXPTIME. The proof is almost identical to that of the previous section. There are two main differences: (i) an APSPACE machine is encoded, not an APTIME machine, and (ii) the process of initializing the machine is more subtle. Section 4.3 showed how to encode APSPACE machines. This section shows how to initialize them.

With hypothetical deletions, initializing the machine is less straightforward than with hypothetical additions. If we were to use the technique of Section 6.3, we would simply write Horn rules such as the following:

$$CONTROL^{q_0}(\overline{p}) \leftarrow FIRST(\overline{p}).$$
$$CELL^{c}(\overline{p}) \leftarrow INITIAL^{c}(\overline{p}).$$

These rules invoke the predicates $INITIAL^{c}(\overline{p})$, which mean that the cell at position $\overline{p}$ is to be initialized to the value c. The problem with such rules is that the formulas $CONTROL^{q_0}(\overline{p})$ and $CELL^{c}(\overline{p})$ are not part of the database, but are inferred. This is important because hypothetical deletion does not delete inferred facts; it only deletes facts from the database. Since hypothetical deletion plays a crucial role in our simulation of APSPACE machines, the machine id must be inserted into the database, and cannot be inferred.

The rules below insert the initial id into the database hypothetically. They are invoked after the rules of Section 6.3.1 have generated a linear order. They first initialize the finite control. Then they iterate over all tape positions $\overline{x}$, placing the appropriate symbol in each tape cell. Finally, they try to infer the atom $SIMULATE$, which is meant to invoke a machine simulation, as described in Chapter 4.3. (i.e., $SIMULATE$ is true iff the machine accepts its input.)

$$
\begin{aligned}
ACCEPT &\leftarrow FIRST(\overline{x}),\ INITIAL^c(\overline{x}), \\
&\quad\ TAPE(\overline{x})[add: CELL^c(\overline{x}),\ CONTROL^{q_0}(\overline{x})]. \\
TAPE(\overline{x}) &\leftarrow NEXT(\overline{x}, \overline{y}),\ INITIAL^c(\overline{y}), \\
&\quad\ TAPE(\overline{y})[add: CELL^c(\overline{y})]. \\
TAPE(\overline{x}) &\leftarrow LAST(\overline{x}),\ SIMULATE.
\end{aligned}
$$

We write rules of this form for $c \in \{0, 1, b\}$. These rules combined with the constructions of Section 6.3 give the following results.

Theorem 6.7 *Suppose that φ is a database query of type $\overline{\alpha} \rightarrow \alpha_0$ which is computable in exponential time. Then there is a stratified rulebase $R(\varphi)$ of hypothetical additions and deletions which expresses φ. Furthermore, $R(\varphi)$ can be constructed so that it is constant-free.*

As in Section 6.3, the last sentence in this theorem leads to an interesting corollary. It has already been established that a stratified rulebase of hypothetical additions and deletions which is constant-free expresses a generic query in EXPTIME. Theorem 6.7 gives us the converse. We therefore have the following characterization, which is a main result of this chapter.

Corollary 6.8 *Constant-free stratified rulebases of hypothetical additions and deletion express exactly the generic database queries in EXPTIME.*

Chapter 7

Linear Recursion

The restricted logic of hypothetical additions has important properties not enjoyed by the full logic of hypothetical additions and deletions. This chapter and the next develop these properties in detail. This chapter[1] develops syntactic restrictions that reduce the complexity of the restricted logic. In particular, we require that rules be *linear*. Informally, a rule is linear iff recursion occurs through only one premise. In Horn-clause logic, "linear rules play an important role because, (i) there is a belief that most 'real life' recursive rules are linear, and (ii) algorithms have been developed to handle them efficiently" [6]. The main result of this chapter is that linearity reduces the complexity of the logic of hypothetical additions from PSPACE to NP in the negation-free case. (The next chapter studies the interaction between linearity and negation.)

In defining linear recursion for hypothetical rules, we generalize the definition given in [6] for Horn rules. Central to this definition is the concept of mutually recursive predicates.[2]

Definition 7.1 *Let R be a rulebase. (i) A predicate P refers to a predicate Q iff Q is a goal predicate of some rule in R with head predicate P. (ii) The reflexive, transitive closure of "refers" is denoted by the symbol "$\prec$". (iii) $P \equiv Q$ iff $P \prec Q$ and $Q \prec P$.*

If $P \prec Q$, then we say that P *depends* on Q; and if $P \equiv Q$, then we say that P and Q are *mutually recursive*. Note that mutual recursiveness is an equivalence relation. In Example 6.2, the predicates $EVEN$ and ODD are mutually recursive. In Section 2.3.2, the predicates $ZERO$ and ONE are mutually recursive.

---

[1]This chapter elaborates on material that originally appeared in [9].

[2]It is worth comparing the definition of mutually recursive *predicates* given here to the definition of mutually recursive *rules* given in Section 5.3.

Definition 7.2 *(Linearity) Let r be a rule in a rulebase R. (i) r is recursive iff it has at least one goal predicate Q that is mutually recursive with its head predicate P. (ii) r is linear iff it has exactly one such goal predicate. (iii) The rulebase R is linear iff every recursive rule in R is linear.*

Example 7.1

A Linear Rulebase	A Non-Linear Rulebase
$A \leftarrow B[add : D_1],\ G[add : D_2].$	$A \leftarrow B[add : D_1],\ C[add : D_2]$
$B \leftarrow C[add : E_1],\ H.$	$B \leftarrow C[add : E_1],\ A[add : E_2]$
$C \leftarrow A[add : F_1]$	$C \leftarrow A[add : F_1],\ B[add : F_2]$

$\square$

Examples from earlier chapters provide richer and less abstract examples of linear and non-linear rulebases. In Section 6.1, the rulebase in each of the first three examples is linear, and the rulebase in the fourth example is non-linear. In Section 3.2, the rulebase is linear. In Section 2.3.2, the rulebase has two sources of non-linearity, one involving the predicate SET, and another involving the predicates $ZERO$ and ONE.

The following theorem is the main result of this chapter.

Theorem 7.1 *In the negation free case, the data complexity of linear rulebases of hypothetical additions is complete for NP.*

The lower complexity bound can be established in a straightforward way by using the machine encodings of Section 4.4. First note that an NP machine is a special kind of alternating PTIME machine, one in which there are no universal control states. For such a machine, the encodings of Section 4.4 give a linear rulebase. Linear rulebases can thus encode arbitrary NP machines. This establishes the lower complexity bound of Theorem 7.1.

The rest of this chapter establishes the upper bound. Our strategy is to show that linear rulebases give rise to proof trees which are "narrow" and which therefore have a "small" number of nodes. More precisely, we show that if R is linear, and if the expression $R, DB \vdash \psi$ is true, then the expression has a proof tree of polynomial size. Most of this section is devoted to proving this result. Given this result, it is easy to show that hypothetical inference for linear rulebases is in NP: Given an expression $R, DB \vdash \psi$, we non-deterministically "guess" a possible proof tree of polynomial size; we then check, in polynomial time, whether it is a legal proof. This establishes the upper bound of Theorem 7.1.

7.1 Proof Trees

This section gives a precise definition of "proof tree" for hypothetical inference and establishes some basic results about them. The results apply to any rulebase of hypothetical additions, linear or not. The next section establishes stronger results that apply only to linear rulebases. For the purpose of this chapter, expressions of the form $R, DB \vdash \psi$ shall be called *inference expressions* and shall sometimes be denoted by the symbol ξ.

Definition 7.3 *(Proof Trees) A proof tree for a rulebase R is a finite, rooted tree in which each node is labeled by an inference expression of the form $R, DB \vdash \psi$. Furthermore, each node in the tree must satisfy the following three conditions:*

- *If the node is labeled by the expression $R, DB \vdash A$ where $A \in DB$, then the node has no children.*
- *If the node is labeled by the expression $R, DB \vdash A$ where $A \notin DB$, then for some ground instance $A \leftarrow \psi_1, ..., \psi_m$ of a rule in R, the node has exactly m children and the i^{th} child is labeled by the expression $R, DB \vdash \psi_i$.*
- *If the node is labeled by the expression $R, DB \vdash A[add : B]$, then the node has exactly one child, which is labeled by the expression $R, DB + B \vdash A$.*

Definition 7.3 is intended to mimic the hypothetical inference system defined in Section 3.1. In fact, it is not hard to see that the expression $R, DB \vdash \psi$ is derivable in this system iff the expression labels the root of some proof tree. In this case, we say that the tree is a proof of the expression $R, DB \vdash \psi$.

In general, a proof tree may be of arbitrary size, since an inference expression ξ may appear any number of times on a single branch, due to cycles in the proof. Such repetitions turn out to be redundant however. To show the existence of small proof trees, we focus on trees in which no expression occurs more than once on any given branch. This motivates the following definition.

Definition 7.4 *(Minimal Proof Trees) A proof tree is minimal iff no inference expression appears more than once on each branch of the tree.*

The next lemma is a basic result about minimal proof trees. It shows that any proof tree can be reduced step-by-step to a minimal proof tree.

Lemma 7.2 *The inference expression $R, DB \vdash \psi$ is true iff it has a minimal proof tree.*

Proof: $R, DB \vdash \psi$ is true iff it has a proof tree. We show that any proof tree can be transformed into a minimal proof tree. If a proof tree is not minimal, then it has a branch in which some inference expression ξ appears more than once. On this branch, let the occurrence of ξ which is closest to the root be called number 1, and let the next closest occurrence be called number 2. Now construct a new tree by replacing the subtree rooted at occurrence 1 by the subtree rooted at occurrence 2. We have thus replaced one subtree by a smaller subtree. Furthermore, the new tree is a proof tree. Because a proof tree is finite, we can apply this transformation repeatedly until a minimal proof tree is obtained. $\Box$

For each rulebase R, there are exponentially many expressions of the form $R, DB \vdash \psi$ (exponential in the size of the data domain). This is because there are exponentially many databases DB. The length of any branch of a minimal proof tree is thus bounded above by an exponential. For rulebases of hypothetical additions *and* deletions, this bound is tight, as demonstrated by the example in Section 3.2. For a rulebase of hypothetical additions, however, a tighter bound is possible. In this case, as the next lemma shows, the depth of a minimal proof tree is polynomial. The crucial fact in the proof is that as one proceeds down a branch of a proof tree, the database never shrinks and can only get larger, because entries may be added to the database hypothetically, but never deleted.

Definition 7.5 *(Paths) A path in a tree is a sequence of nodes* $q_0, q_1, q_2...q_m$ *such that* q_{i+1} *is a child of* q_i *for* $0 \leq i < m$*.*

Lemma 7.3 *In a minimal proof tree for a rulebase of hypothetical additions, every path has length* $O(n^{2k_0})$*, where* n *is the size of the data domain and* k_0 *is the maximum number of distinct variables in any rule in the rulebase.*

Proof: Let $q_0, q_1, ..., q_m$ be a path in a minimal proof tree, and suppose that node q_i is labeled by the inference expression $R, DB_i \vdash \psi_i$. Because the proof tree is minimal, these expressions are all distinct. And because the rulebase R contains no hypothetical deletions, the databases increase as we traverse the path; that is, $DB_i \subseteq DB_{i+1}$. We can thus divide the path into segments such that within each segment, the database is the same, and between segments, it increases.

Within each segment, node labels have the form $R, DB \vdash \psi_i$ where the queries ψ_i are all distinct and ground. Thus, the length of a segment is no greater than the number of distinct ground queries that can appear in a proof tree. To estimate this number, observe that except for the root node, each query in a proof tree comes from the premise of a rule. Furthermore, each rule has $O(n^{k_0})$ ground instances, each giving rise to a fixed number of ground queries. For example, the ground rule $A \leftarrow B[add : C], D[add : E]$ gives rise to four queries: $B[add : C]$ and $D[add : E]$ as well as B and D. We thus have $O(n^{k_0})$ ground rules, each giving rise to $O(1)$ ground queries. A proof tree therefore contains $O(n^{k_0})$ distinct ground queries. Thus each path segment has length $O(n^{k_0})$.

Lastly, in going from one path segment to the next, the database increases, that is, a ground atomic formula is added to it. This can only happen $O(n^{k_0})$ times, however, before the database saturates, i.e., before the rulebase can no longer add new formulas to the database. Thus, the path contains $O(n^{k_0})$ segments each of length $O(n^{k_0})$. Its total length is therefore $O(n^{2k_0})$. □

Corollary 7.4 *A minimal proof tree for a rulebase of hypothetical additions has polynomial depth (polynomial in the size of the data domain).*

7.2 Proof Trees for Linear Rulebases

Corollary 7.4 assures us that for a rulebase of hypothetical additions, every minimal proof tree has polynomial depth. Such a tree may still have exponential size however, as demonstrated by Example 2.9 and by the rulebase presented in Section 2.3.2. However, both these examples involve non-linear rulebases. Can such examples be constructed with linear rulebase? This section shows that they cannot. In particular, we show that every minimal proof tree of a linear rulebase has polynomial size.

Linear recursion is defined in terms of mutual recursion, which is an equivalence relation between predicate symbols. Any maximal set of mutually-recursive predicates thus forms an equivalence class. These classes play a central role in our analysis, and for the purpose of this section, we refer to them as *clusters*.[3] It follows immediately that the clusters of a rulebase are disjoint and exhaustive.

Definition 7.6 *(Clusters) A cluster of a rulebase R is a maximal set of mutually recursive predicates.*

Clusters separate the recursive and non-recursive behaviors of a rulebase. Each cluster itself is an island of recursion in which each predicate depends on every other predicate; whereas between clusters, there is no recursion. To study the non-recursive behavior of a rulebase, we look at the dependency of one cluster upon another.

Definition 7.7 *(Dependency Graphs) The dependency graph of a rulebase is a directed graph whose nodes are the clusters of the rulebase. Furthermore, for any two distinct clusters C_1 and C_2, there is an edge from C_1 to C_2 iff some predicate in C_1 depends on a predicate in C_2.*

[3] Unlike the clusters defined in Section 5.3, which are sets of mutually recursive *rules*, the clusters defined here are sets of mutually recursive *predicates*.

Because there is no recursion between clusters, the dependency graph is acyclic. This property allows us to assign a rank to each node in the graph based on the maximum distance from the node to a leaf (sink) in the graph. We can then extend this idea from dependency graphs to proof trees in a straightforward way, allowing us to rank the nodes of a proof tree. The main result of this section is then proved by an induction over node rank.

Definition 7.8 *(Rank) Let R be a rulebase.*

- *The rank of a cluster of R is the maximum distance in the dependency graph from the cluster to a leaf (where leaves have rank 1).*
- *The rank of a hypothetical formula ψ is the rank of the cluster to which its goal predicate belongs.*
- *If a node in a proof tree is labeled by the expression $R, DB \vdash \psi$, then the rank of the node is the rank of ψ.*

Intuitively, rank in a proof tree can be understood by dividing each branch of the tree into "recursive phases," which deal with predicates from a particular cluster. That is, within a recursive phase, each node is labeled by an expression of the form $R, DB \vdash A$ or $R, DB \vdash A[add : B]$ where A belongs to a particular cluster. The rank of a node is then the maximum number of recursive phases from the node to a leaf.

The following lemma states two basic facts about rank. The second statement is true because in a linear rulebase, at most one premise of a rule belongs to the same cluster as the head of the rule.

Lemma 7.5 (i) *In any proof tree, the rank of a node is no greater than the rank of its parent.* (ii) *In a proof tree for a linear rulebase, each node has at most one child of the same rank as itself.*

We are now in a position to place an upper bound on the size of a proof tree for a linear rulebase. We do this by estimating the number of descendants of each node. The proof is by induction on node rank. i.e., We first estimate the number of descendants for nodes of rank 1, then of rank 2, etc. The following lemma does precisely this and is the main result of this section.

Lemma 7.6 *In a minimal proof tree for a linear rulebase, a node of rank r has $O(n^{2k_0 r})$ descendants, where n is the size of the data domain and k_0 is the maximum number of distinct variables in any rule in the rulebase.*

Proof: (By induction on rank)

Basis: Let q_0 be a node of rank 1. Because the rulebase is linear, q_0 can have at most one child whose rank is 1, by Lemma 7.5. Every other child must have a rank which is strictly less than 1. But 1 is the smallest rank. Thus, q_0 has at most one child q_1, and its rank is 1. Similarly, q_1 has at most one child q_2, and its rank is also 1. Thus we have a path $q_0, q_1, q_2 ... q_m$ in the proof tree, consisting of the descendants of q_0. By Lemma 7.3, this path has length $O(n^{2k_0})$.

Induction: First note that each node in the proof tree has at most m_0 children, where m_0 is the maximum number of premises of any rule in the rulebase. Since a rulebase is independent of n, so is m_0.

Now suppose that the lemma is true for nodes of rank $r-1$, and let q_0 be a node of rank r. By Lemma 7.5, q_0 has at most one child of rank r. Call it q_1. Similarly, q_1 has at most one child of rank r. Call it q_2. In this way, we define a path $q_0, q_1, q_2 ... q_m$. By Lemma 7.3, this path has length $O(n^{2k_0})$. Furthermore, since q_{i+1} is the one child of q_i having rank r, every other child of q_i has rank at most $r-1$. Each of these children therefore has $O[n^{2k_0(r-1)}]$ descendants, by induction hypothesis.

Every node in the proof tree has at most m_0 children, of whatever rank. Thus, there are $O(n^{2k_0})$ nodes on the path $q_0 ... q_m$, and each of these nodes has at most m_0 other children, and each of these children has $O[n^{2k_0(r-1)}]$ descendants. Thus, the total number of descendants of q_0 is

$$O(n^{2k_0}) \ + \ O(n^{2k_0}) \times m_0 \times O[n^{2k_0(r-1)}] \quad = \quad O(n^{2k_0 r})$$

□

The rank of a node in a proof tree is just the rank of some cluster in the rulebase R. Let r_0 be the maximum rank of any cluster in R. Then r_0 is an upper bound on the rank of every node in a proof tree for R. In particular, r_0 bounds the rank of the root node. Thus, by Lemma 7.6, the root of a proof tree has $O(n^{2k_0 r_0})$ descendants. Since k_0 and r_0 are independent of n, we have the following result.

Corollary 7.7 *A minimal proof tree for a linear rulebase R has polynomial size (polynomial in the size of the data domain).*

By combining Corollary 7.7 and Lemma 7.2, we get the following result.

Theorem 7.8 *Let R be a linear rulebase of hypothetical additions. Then the inference expression* $R, DB \vdash \psi$ *is true iff it has a proof tree of polynomial size (polynomial in the size of the data domain).*

7.3 A Proof Procedure

It follows immediately from Theorem 7.8 that inference for linear rulebases is in NP. To see this, first non-deterministically "guess" a possible proof tree of polynomial size. Then check in polynomial time whether it represents a legal proof. This section presents a proof procedure based on this idea. An adaptation of this procedure is used in Chapter 8 to obtain upper complexity bounds for stratified linear rulebases.

The procedure PROVE below attempts to generate a proof tree for the expression $R, DB \vdash \psi$, and it attempts to do so in a top-down, non-deterministic manner. The procedure effectively combines the guessing phase with the checking phase. Note that each of the conditions 1–3 in this procedure corresponds to a condition in Definition 7.3.

Procedure: PROVE(DB, ψ)

1. if $\psi \in DB$ then return($true$);

2. elseif ψ is atomic

 then *choose* a ground instance $A \leftarrow \psi_1 ... \psi_m$ of a rule in R;

 if $A = \psi$ then return[$\wedge_i$ PROVE(DB, ψ_i)]; else return(*false*);

3. elseif $\psi = A[add : B]$ then return[PROVE$(DB + B, A)$];

The choice point in condition 2 of this procedure is non-deterministic. It "guesses" a ground instance of a rule. The subsequent test, "if $A = \psi$," then determines whether the guess can be used to extend the proof tree in a top-down fashion. If the guess fails the test, then the procedure returns *false*; if it passes the test, then the procedure calls itself recursively, thereby extending the proof tree.

Suppose that for a particular sequence of guesses, PROVE(DB, ψ) returns *true*. Then during execution, the procedure effectively generated a proof tree for $R, DB \vdash \psi$ in pre-order (i.e., depth first and left to right). Conversely, any proof tree for $R, DB \vdash \psi$ can be generated in this way by some sequence of guesses. These comments hold for any rulebase R of hypothetical additions, whether linear or non-linear. We thus have the following result.

Lemma 7.9 *The expression $R, DB \vdash \psi$ has a proof tree iff for some sequences of guesses,* PROVE(DB, ψ) *returns true (in which case, the procedure generates a proof tree in pre-order during execution.)*

For any sequence of guesses, the execution time of PROVE(DB, ψ) is a polynomial function of the size of the tree that it generates. In the special case in which R is linear, Theorem 7.8 assures us that if $R, DB \vdash \psi$ is true, then it has a proof tree of polynomial size. We can force the procedure PROVE to generate only polynomial-size proof trees by augmenting it with a counter so that it halts (and fails) after polynomial time. Calling the augmented procedure PROVE$^{\mathrm{NP}}$, we have the following result.

Lemma 7.10 *If R is linear, then the inference expression $R, DB \vdash \psi$ is true iff* PROVE$^{\mathrm{NP}}(DB, \psi)$ *returns true for some sequence of guesses.*

PROVE^{NP} runs in non-deterministic polynomial time. Therefore inference for linear rulebases is in NP. This establishes the upper bound of Theorem 7.1.

Although this completes the main task of this section, we can use the development to produce an alternative proof that inference for non-linear rulebases is in PSPACE.[4] The proof is similar to the one just given for linear rulebases, but it exploits the polynomial depth of minimal proof trees, instead of polynomial size. We first combine Lemma 7.2 and Corollary 7.4 to give the following result.

Lemma 7.11 *Suppose R is a rulebase of hypothetical additions (not necessarily linear). Then the expression $R, DB \vdash \psi$ is true iff it has a proof tree of polynomial depth.*

We augment the procedure PROVE with a counter so that it halts (and fails) after polynomially-many levels of recursion. Call the augmented procedure PROVE′. This procedure generates only proof trees of polynomial depth. Conversely, any proof tree of polynomial depth can be generated by PROVE′ for some sequence of guesses. Thus, by Lemma 7.11, an inference expression $R, DB \vdash \phi$ is true iff $\text{PROVE}'(DB, \phi)$ returns true for some sequence of guesses.

The complexity of PROVE′ is easily established. For any particular sequence of guesses, PROVE′ makes polynomially-many recursive calls, each taking polynomial time. This computation can be done in polynomial space. Thus, the entire computation, including guesses, runs in non-deterministic polynomial space.

Inference for the logic of hypothetical additions is therefore in NPSPACE. Inference is therefore in PSPACE, since PSPACE = NPSPACE, by Savitch's Theorem [26].

[4]This was originally proved in Section 4.2.

Chapter 8

Negation and Linear Recursion

Chapter 7 studied linear rulebases of hypothetical additions in the negation-free case. This chapter[1] studies the case with negation, i.e., stratified linear rulebases. Chapter 7 showed that in the absence of negation, linear recursion reduces the data complexity of hypothetical additions from PSPACE to NP. This chapter shows that with negation, the data complexity is between NP and PSPACE, and that the precise complexity depends on the number and type of strata in the rulebase. We then use these results to characterize the expressibility of stratified linear rulebases.

We first show that the data complexity increases with the number of strata. Furthermore, only those strata containing non-Horn rules have a significant effect on the complexity. We say that such strata are *hypothetical*, and we show that for each hypothetical stratum added to a rulebase, the data complexity climbs one level in the polynomial time hierarchy. Using these results, we obtain an exact characterization of the database queries at each level in the hierarchy. Finally, using a well-known connection between the polynomial time hierarchy and second order logic [30], we show that stratified linear rulebases without constant symbols express exactly the second-order definable queries.

The polynomial-time hierarchy is a sequence of complexity classes between P and PSPACE. It is based on the idea of an oracle Turing-machine [26] and can be defined recursively as follows:

- $\Sigma_0^{\mathrm{P}} = \Delta_0^{\mathrm{P}} = P$
- $\Sigma_{k+1}^{\mathrm{P}} = \mathrm{NP}^{\Sigma_k^{\mathrm{P}}}$ = those languages accepted in non-deterministic polynomial time

[1]This chapter elaborates on material that originally appeared in [9].

by an oracle machine whose oracle is a language in Σ_k^{P}.

- $\Delta_{k+1}^{\mathrm{P}} = P^{\Sigma_k^{\mathrm{P}}}$ = those languages accepted in deterministic polynomial time by an oracle machine whose oracle is a language in Σ_k^{P}.
- PHIER $= \cup_k \Sigma_k^{\mathrm{P}}$

It is well known that $P \subseteq \Sigma_k^{\mathrm{P}} \subseteq \Delta_{k+1}^{\mathrm{P}} \subseteq \Sigma_{k+1}^{\mathrm{P}} \subseteq$ PHIER $\subseteq$ PSPACE.[2]

This chapter establishes several complexity results. The simplest result to state is the following: The data complexity of linear rulebases with k strata is complete for Σ_k^{P}. It turns out, however, that additional strata can be added to a rulebase with only a small cost in complexity as long as the new strata contain only Horn rules. In particular, for linear rulebases having k *hypothetical* strata, the data complexity is between Σ_k^{P} and $\Delta_{k+1}^{\mathrm{P}}$. Thus, no matter how many Horn strata are added, the data complexity will not exceed $\Delta_{k+1}^{\mathrm{P}}$. Finally, we show that if Horn strata are added *between* two hypothetical strata, and not to the top of a rulebase, then complexity is not affected at all. That is, for rulebases which have k hypothetical strata, and for which the top stratum is hypothetical, the data complexity is complete for Σ_k^{P}.

As discussed in Chapter 6, it does not necessarily follow from these complexity results that stratified linear rulebases are expressively complete for PHIER. That is, the data complexity of a logic may be complete for Σ_k^{P}, yet there may be many database queries in Σ_k^{P} that it cannot express. By employing the techniques of Chapter 6, however, we show that linear rulebases are expressively complete for PHIER. That is, any generic database query whose data complexity is in PHIER can be represented as a stratified linear rulebase of hypothetical additions. Rulebases in which the top stratum is hypothetical play a key role in this development. In particular, rulebases having k hypothetical strata and a hypothetical top stratum are are expressively complete for Σ_k^{P}.

Although linear rulebases express every generic query in PHIER, they also express some non-generic queries. Non-genericity is caused by constant symbols in the rulebase,

---

[2] Although considered likely, it is an open question as to whether any of these containments are strict.

which cause the rulebase to give special treatment to some elements of the data domain. By barring constant symbols from the rulebase, non-genericity is eliminated. Thus, stratified linear rulebases that are constant free express exactly the typed generic queries in PHIER. This gives an exact characterization of the database queries in PHIER. In a similar fashion, we characterize the queries in Σ_k^{P}.

It should be noted that these results apply only to the restricted logic of hypothetical additions. With both addition and deletion, we can use the techniques of Section 4.3 to construct a linear rulebase that encodes the computations of any deterministic PSPACE machine. In the full logic, therefore, linear rulebases having just one stratum are PSPACE-hard, which is harder than any level in the polynomial time hierarchy. Furthermore, since there is no "polynomial space hierarchy", there appears to be no natural way of extending the results of this chapter to the full logic of hypothetical additions and deletions.

Strictly speaking, every rule in our logic is hypothetical, but for the purpose of this chapter, it is convenient to distinguish between hypothetical rules and Horn rules. We say that a rule is *hypothetical* iff it has at least one hypothetical modality in its premise; otherwise, we say it is *Horn*. This applies to rules with or without negation in their premises. Likewise, we say that a stratum is hypothetical iff it contains at least one hypothetical rule; otherwise, we say it is Horn. With this in mind, the following three theorems are a precise statement of the main complexity results of this chapter. (Section 8.3 states the main expressibility results.) Each theorem is formulated in terms of a set R_k^i of linear rulebases. The first theorem is about rulebases having k strata.

Definition 8.1 *$\mathcal{R}_k^1$ is a set of stratified linear rulebases of hypothetical additions. In particular, $R \in \mathcal{R}_k^1$ iff there is a stratification of R with at most k strata.*

Theorem 8.1 *For $k \geq 1$, the data complexity of $\mathcal{R}_k^1$ is complete for Σ_k^{P}.*

The second theorem is about rulebases having k hypothetical strata.

Definition 8.2 *$\mathcal{R}_k^2$ is a set of stratified linear rulebases of hypothetical additions. In particular, $R \in \mathcal{R}_k^2$ iff there is a stratification of R with at most k hypothetical strata.*

Theorem 8.2 *For $k \geq 1$, the data complexity of $\mathcal{R}_k^2$ is bounded below by Σ_k^{P} and bounded above by $\Delta_{k+1}^{\mathrm{P}}$.*

Theorem 8.2 does not establish tight complexity bounds. The next theorem establishes tight bounds by requiring that the top stratum of a rulebase be hypothetical.

Definition 8.3 *$\mathcal{R}_k^3$ is a set of stratified linear rulebases of hypothetical additions. In particular, $R \in \mathcal{R}_k^3$ iff there is a stratification of R with at most k hypothetical strata and for which the top stratum is hypothetical.*

Theorem 8.3 *For $k \geq 1$, the data complexity of $\mathcal{R}_k^3$ is complete for Σ_k^{P}.*

These three theorems remain true if in the definition of $\mathcal{R}_k^i$, the phrase "at most k" is replaced by "exactly k". Furthermore, the rulebases in $\mathcal{R}_k^i$ do not have to be entirely linear: The hypothetical strata must be linear, but the Horn strata may be non-linear. For succinctness sake, however, we refer simply to "linear rulebases". With this in mind, note that Theorem 8.2 can be extended to the case $k = 0$. In this case, $\mathcal{R}_k^2$ is just the set of stratified Horn rulebases, whose data complexity is well-known to be complete for P, i.e., for Σ_0^P.

8.1 Upper Complexity Bounds

This section shows that the data complexity of a linear rulebase with at most k hypothetical strata is in $\Delta_{k+1}^{\mathrm{P}}$. Furthermore, if its top stratum is hypothetical, then its complexity is in Σ_k^{P}. This establishes the upper bounds of Theorems 8.1, 8.2 and 8.3. To do this, we present a mixed top-down/bottom-up proof procedure in which each stratum is processed separately. Each Horn stratum is processed by a bottom-up procedure that runs in polynomial time, and each hypothetical stratum is processed by a top-down procedure that runs in non-deterministic polynomial time.

We define a series of procedures $\mathrm{PROVE}_1, ..., \mathrm{PROVE}_m$, one for each stratum. PROVE_i takes two arguments, a database DB and a hypothetical query ψ, where the goal predicate of ψ is defined at or below the i^{th} stratum. The value of $\mathrm{PROVE}_i(DB, \psi)$

is true iff $R, DB \vdash \psi$. PROVE_i invokes PROVE_{i-1} as a subroutine, i.e., as an oracle. Thus, the entire proof procedure can be viewed as a cascade of oracle machines.

Each procedure PROVE_i is one of two types depending on the form of the i^{th} stratum of the rulebase. If the i^{th} stratum is Horn, then PROVE_i is a bottom-up, iterative procedure based on the bottom-up procedure for computing the least fixpoint of a Horn rulebase. This procedure can be viewed as a P machine that uses uses PROVE_{i-1} as an oracle. On the other hand, if the i^{th} stratum is hypothetical, then PROVE_i is a top-down, recursive procedure based on the procedure PROVE developed in Chapter 7 for negation-free linear rulebases. The procedure PROVE_i can be viewed as an NP machine that uses PROVE_{i-1} as an oracle. The overall proof procedure can thus be viewed as a cascade of oracle machines each of which is either a P machine or an NP machine. The important point is that at most k of these machines are NP machines. Thus, the overall procedure is in $\Delta^{\mathrm{P}}_{k+1}$. In addition, if the top stratum of the rulebase is hypothetical, then the top oracle-machine is an NP machine, so the overall procedure is in Σ^{P}_k.

Suppose, for example, that the rulebase has three strata, all of which are hypothetical. Then the data complexity of the rulebase is $\mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}} = \Sigma^{\mathrm{P}}_3$. On the other hand, suppose that the middle stratum is Horn. Then the data complexity of the rulebase is in $\mathrm{NP}^{\mathrm{P}^{\mathrm{NP}}} = \mathrm{NP}^{\mathrm{NP}} = \Sigma^{\mathrm{P}}_2$. Here we have used the identity $\mathrm{NP}^{\mathrm{P}} = \mathrm{NP}$. Finally, suppose that the top two strata are Horn and the bottom stratum is hypothetical. Then the data complexity of the rulebase is in $\mathrm{P}^{\mathrm{P}^{\mathrm{NP}}} = \mathrm{P}^{\mathrm{NP}} = \Delta^{\mathrm{P}}_2$. Here we have used the identity $\mathrm{P}^{\mathrm{P}} = \mathrm{P}$.

For convenience, we write $\mathrm{PROVE}^{\mathrm{P}}_i$ to indicate that PROVE_i is a P machine, and $\mathrm{PROVE}^{\mathrm{NP}}_i$ to indicate that it is an NP machine.

8.1.1 The Procedure $\mathrm{PROVE}^{\mathrm{P}}_i$

If the i^{th} stratum of the rulebase R is Horn, then it is processed by the procedure $\mathrm{PROVE}^{\mathrm{P}}_i$. $\mathrm{PROVE}^{\mathrm{P}}_i$ is a bottom-up, iterative procedure very similar to the bottom-up procedure for computing the least fixpoint of a Horn rulebase. Starting with $S = DB$, atoms are added to S by repeatedly applying the rules in the i^{th} stratum of R. For each rule, the head of a rule is added to S iff S satisfies the premises of the rule. This process is repeated until no new atoms are inferred.

Since the i^{th} stratum is Horn, none of its rules are hypothetical. Therefore, unlike the procedures of Sections 5.4 and 4.2, the procedures below are not recursive. Furthermore, except for a brief initialization phase, the database DB is fixed during the entire execution. In particular, the procedures do not invoke themselves recursively on a larger database DB'. These procedures therefore run in polynomial time instead of polynomial space.

Although the i^{th} stratum is Horn, it may contain rules whose premises are defined in lower strata. In this case, the procedure PROVE_{i-1} is invoked as a subroutine to determine whether these premises are true. If PROVE_{i-1} is treated as an oracle, which returns in constant time, then $\text{PROVE}_i^{\text{P}}$ runs in polynomial time, just like the least-fixpoint computation for a set of function-free Horn rules.

The procedure $\text{PROVE}_i^{\text{P}}(DB, \psi)$ is given below. Before computing a "least fixpoint", it makes some elementary decisions. First, it determines whether ψ is an atomic query. If not, it converts it to one by adjusting the database. Then, it determines whether ψ is defined in the i^{th} stratum. If not, the problem is passed on to PROVE_{i-1}. Only if both these tests are passed is a least fixpoint computation actually carried out. In this case, ψ is atomic and $\text{PROVE}_i^{\text{P}}(DB, \psi)$ returns *true* iff ψ is in the "least fixpoint."

Procedure: $\text{PROVE}_i^{\text{P}}(DB, \psi)$

  if $\psi = A[add : B]$ then return[$\text{PROVE}_i^{\text{P}}(DB \cup \{B\}, A)$];

  elseif ψ is defined below the i^{th} stratum, then return[$\text{PROVE}_{i-1}(DB, A)$];

  elseif $\psi \in \text{LFP}_i(DB)$ then return(*true*);

  else return(*false*);

The "least fixpoint" computation itself is implemented by the four procedures below. The procedures LFP_i and V_i apply the rules in the the i^{th} stratum of R repeatedly until saturation, as in Horn clause logic. The procedure $\text{MATCH}_i(DB, S, \psi)$ determines whether the premise ψ is trivially true given the database DB and the inferred atoms

S. If ψ is defined in the i^{th} stratum, then ψ is a positive literal, and MATCH$_i$ simply checks to see whether $\psi \in S$. If ψ is defined below the i^{th} stratum, then MATCH$_i$ invokes PROVE$_{i-1}$ as an oracle. In this case, ψ may be a positive or a negative literal. For convenience, these procedures are defined for the propositional case, the predicate case being a straightforward generalization.

Procedure: LFP$_i(DB)$
    $S_1 \leftarrow DB$;
    $S_2 \leftarrow \mathrm{V}_i(DB, S_1)$;
    do until $S_1 = S_2$
        $S_1 \leftarrow S_2$;
        $S_2 \leftarrow \mathrm{V}_i(DB, S_1)$;
    end do;
    return(S_2);

Procedure: V$_i(DB, S_1)$
    $S_2 \leftarrow S_1$;
    do for each rule $A \leftarrow \psi_1 ... \psi_m$ in the i^{th} stratum of R
        if MATCH$_i(DB, S_1, \psi_j)$ is true for each $1 \leq j \leq m$
            then $S_2 \leftarrow S_2 \cup \{A\}$;
        end do;
    return(S_2);

Procedure: MATCH$_i(DB, S, \psi)$
    if ψ is defined below the i^{th} stratum then return[COMP$_{i-1}(DB, \psi)$];
    elseif $\psi \in S$ then return(*true*);
                        else return(*false*);

Procedure: COMP$_{i-1}(DB, \psi)$
    if $\psi = \sim \psi_0$ then return[*not*(PROVE$_{i-1}(DB, \psi_0)$)];
                        else return[PROVE$_{i-1}(DB, \psi)$];

8.1.2 The Procedure $\mathrm{PROVE}_i^{\mathrm{NP}}$

If the i^{th} stratum of the rulebase R is hypothetical, then it is processed by the procedure $\mathrm{PROVE}_i^{\mathrm{NP}}$. $\mathrm{PROVE}_i^{\mathrm{NP}}$ is a top-down, recursive procedure very similar to the procedure $\mathrm{PROVE}^{\mathrm{NP}}$ developed in Section 7.3 for negation-free linear rulebases. To determine whether $R, DB \vdash \psi$, the procedure non-deterministically "guesses" a rule that concludes ψ and then tries to prove each of the rule premises. This process of expanding subgoals non-deterministically continues recursively until either (i) a subgoal is trivially satisfied by the database, or (ii) no appropriate rules can be found, or (iii) the appropriate rules belong to a lower stratum, in which case the procedure PROVE_{i-1} is invoked as a subroutine. If PROVE_{i-1} is treated as an oracle, which returns in constant time, then $\mathrm{PROVE}_i^{\mathrm{NP}}$ runs in non-deterministic polynomial time, just like the procedure $\mathrm{PROVE}^{\mathrm{NP}}$ of Section 7.3.

Procedure: $\mathrm{PROVE}_i^{\mathrm{NP}}(DB, \psi)$

  if ψ is defined below the i^{th} stratum then return[$\mathrm{COMP}_{i-1}(DB, \psi)$];

  elseif $\psi \in DB$ then return(*true*);

  elseif ψ is atomic

    then *choose* a ground instance $A \leftarrow \psi_1, ..., \psi_m$ of a rule

      in the i^{th} stratum of R;

      if $A = \psi$ then return[$\wedge_i$ $\mathrm{PROVE}_i^{\mathrm{NP}}(DB, \psi_i)$]; else return(*false*);

  elseif $\psi = A[add : B]$ then return[$\mathrm{PROVE}_i^{\mathrm{NP}}(DB + B, A)$];

Procedure: $\mathrm{COMP}_{i-1}(DB, \psi)$

  if $\psi = \sim \psi_0$ then return[$not(\mathrm{PROVE}_{i-1}(DB, \psi_0))$];

    else return[$\mathrm{PROVE}_{i-1}(DB, \psi)$];

To show that $\mathrm{PROVE}_i^{\mathrm{NP}}$ runs in NP time, we invoke the analysis of proof trees given in Chapter 7. Actually, this analysis must be modified slightly to take oracle invocations into account. The first step is to extend the notion of proof trees from negation-free rulebases to stratified rulebases. Since the procedure $PROVE_i^{\mathrm{NP}}$ applies only to the i^{th} stratum, it is sufficient to define proof trees for a stratum (instead of for the entire rulebase).

In Chapter 7, the nodes of a proof tree are labeled by expressions of the form $R, DB \vdash \psi$. Furthermore, at leaf nodes, ψ is atomic and $\psi \in DB$. To define proof trees for the i^{th} stratum, two extensions are necessary. First, only rules in the i^{th} stratum may be used to construct the proof tree. Second, at leaf nodes, the formula ψ may be a hypothetical formula defined *below* the i^{th} stratum. In this case, the leaf is labeled by the expression $R, DB \vdash \psi$ iff $\mathrm{PROVE}_{i-1}(DB, \psi)$ is true. This extension takes oracle invocations into account, representing them as leaves of a proof tree. In this way, an oracle invocation is treated like a database access, adding just a single node to the proof tree.

Assuming that PROVE_{i-1} is correct, all the results of Chapter 7 are true with respect to this modified definition of proof trees. In particular, if the i^{th} stratum of R is linear, and if ψ is defined in the i^{th} stratum, then the expression $R, DB \vdash \psi$ is true iff it has a proof tree of polynomial size. As in Chapter 7, we assume that the procedure $\mathrm{PROVE}_i^{\mathrm{NP}}$ has a counter attached to it so that it halts after polynomial time. In this way, it non-deterministically generates and tests all possible proof trees of polynomial size, and therefore runs in NP time.

This establishes the upper bounds of Theorems 8.1, 8.2 and 8.3.

8.2 Lower Complexity Bounds

This section establishes the lower bounds of Theorems 8.1, 8.2 and 8.3. In particular, this section constructs a stratified linear rulebase whose data complexity is Σ_k^{P} hard. This rulebase has exactly k strata, each of which is hypothetical.

Each stratum of the rulebase encodes an arbitrary NP oracle machine. Furthermore, the machine encoded in one stratum uses the machine encoded in the stratum below as its oracle. By using hypothetical addition, the higher machine provides input to the lower one. That is, M_i hypothetically writes into the database, and M_{i-1} reads this data. Likewise, M_{i-1} provides input to M_{i-2}. In this way, a rulebase having k strata can encode a cascade of k distinct NP oracle-machines, i.e., a Σ_k^{P} machine. Besides providing a lower complexity bound, these encodings are crucial to the expressibility results of Section 8.3.

Suppose that L is a language in Σ_k^{P} and $\overline{s}$ is a string. We encode $\overline{s}$ as a database $DB(\overline{s})$ and construct a rulebase $R(L)$ with k strata so that

$$R(L), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad \overline{s} \in L \tag{8.1}$$

where $ACCEPT$ is a 0-ary predicate. The important point is that the rulebase $R(L)$ is independent of the string $\overline{s}$. We can therefore conclude that the data-complexity of $R(L)$ is Σ_k^{P} hard. In particular, let L be a language which is Σ_k^{P} complete.[3] The hardness result then follows immediately.

The rest of this section describes the construction of $DB(\overline{s})$ and $R(L)$. The notion of a Σ_k^{P} machine is central to these constructions.

A Σ_k^{P} machine consists of a cascade of NP oracle machines $M_k, ..., M_1$. An oracle machine has two tapes, a work tape and an oracle tape. Because $M_k, ..., M_1$ are cascaded, the oracle tape of M_i is the work tape of M_{i-1}. In this way, M_i can ask questions of M_{i-1}. To do this, M_i first writes a string on the work tape of M_{i-1} and then asks M_{i-1} whether it accepts the string. M_i suspends its own computations while M_{i-1} is computing. In effect, M_k invokes M_{k-1} as a subroutine. Likewise, M_{k-1} invokes M_{k-2} as a subroutine. Etc. The language accepted by this compound machine is the language accepted by M_k when it is the root of this cascade of machines.

Because M_k runs in (non-deterministic) polynomial time, it can only write a string of polynomial length. M_{k-1} therefore runs in (non-deterministic) polynomial time both in terms of its own input and in terms of the input to M_k, that is, in terms of the string $\overline{s}$. By a simple induction, it follows that each invocation of machine M_i runs in time which is polynomial in the length of $\overline{s}$. More specifically, suppose that M_i runs in non-deterministic time $O(m^{l_i})$, where m is the length of the input to M_i. Then, as part of the compound machine, each invocation of M_i runs in non-deterministic time $O[n^{(l_i+\cdots+l_k)}]$, where n is the length of $\overline{s}$. Thus, any invocation of any of the M_i runs

[3] See [16] for examples of such languages.

in non-deterministic time $O(n^l)$, where $l = l_1 + \cdots + \; l_k$.

Keeping this in mind, we describe the database $DB(\overline{s})$ and the rulebase $R(L)$.

8.2.1 Building the Database $DB(\overline{s})$

The language L in statement 8.1 is a language in Σ_k^{P}. There is therefore a Σ_k^{P} machine which accepts it, that is, a sequence of cascaded NP oracle machines $M_k, ..., M_1$. As discussed above, there is an integer l such that any invocation of any of these machines runs in (non-deterministic) time n^l, where n is the length of the input $\overline{s}$. A counter is therefore needed to represent n^l points in time and n^l positions on tape. We represent this counter by placing the following atomic formulas in the database $DB(\overline{s})$:

$$FIRST(0),\ NEXT(0,1),\ NEXT(1,2),\ NEXT(n^l-2, n^l-1),\ LAST(n^l-1).$$

Given this counter, we can represent the configuration of the Σ_k^{P} machine at each point in its computation. To represent the oracle machine M_i, we introduce the predicates below, one predicate for each symbol c in the tape alphabet, and one for each state q in the finite control. This representation assumes that M_i has exactly two tape heads, one for its work tape and one for its oracle tape.

- $CELL_i^c(j,t)$ means that at time t, the work tape of machine M_i has the symbol c in the cell at position j.
- $CONTROL_i^q(j_1, j_2, t)$ means that at time t, the finite control of machine M_i is in state q, its work head is over cell j_1 of the work tape, and its oracle head is over cell j_2 of the oracle tape.

For each value of t, these atomic formulas define k distinct id's, one for each oracle machine.

Note that we need not have introduced separate predicates for each tape symbol and control state. Instead, we might have introduced new constant symbols c and q and defined just two predicates for each machine, $CELL_i(c, j, t)$ and $CONTROL_i(q, j_1, j_2, t)$. By using separate predicates, however, we can construct the rulebase $R(L)$ so that it contains no constant symbols. This property will be important in Section 8.3 when we use the constructions of this chapter to characterize the generic queries in Σ_k^{P}.

The database $DB(\overline{s})$ describes the initial tape contents of the machines $M_k, ..., M_1$. Since $M_{k-1}, ..., M_1$ act as oracles, their work tapes are initially blank. This is represented by placing the following atomic formulas in $DB(\overline{s})$ for each $i \leq k-1$:

$$CELL_i^b(0,0),\ CELL_i^b(1,0)\ ...\ CELL_i^b(n^l-1,0).$$

where b denotes a blank. These formulas states that at time 0, the tape cell at position j contains a blank, for $0 \leq j \leq n^l - 1$.

For machine M_k, the work tape initially contains the input string $\overline{s} = \langle s_0, s_1 ... s_{n-1} \rangle$ followed by blank tape cells. This is represented by placing the following atomic formulas in $DB(\overline{s})$:

$$\begin{array}{l} CELL_k^{s_0}(0,0),\ CELL_k^{s_1}(1,0)\ \cdots\ CELL_k^{s_{n-1}}(n-1,0), \\ CELL_k^b(n,0),\ CELL_k^b(n+1,0)\ \cdots\ CELL_k^b(n^l-1,0). \end{array}$$

This states that at time 0, the symbol s_j appears in cell j for $0 \leq j \leq n-1$, and a blank appears in cell j for $n \leq j \leq n^l - 1$.

This completes the construction of the database $DB(\overline{s})$. It defines a counter from 0 to $n^l - 1$ and specifies the initial contents of each machine tape. Note that this construction can be performed in polynomial time and space (polynomial in the length of $\overline{s}$).

8.2.2 Building the Rulebase $R(L)$

This section constructs the rulebase $R(L)$ so that it encodes the oracle machines $M_k, ..., M_1$. This rulebase has exactly k strata, where the i^{th} stratum encodes machine M_i. M_i uses M_{i-1} as its oracle, and by using hypothetical addition, the rules in the i^{th} stratum encode the process whereby M_i provides input to M_{i-1}. That is, M_i hypothetically writes into the database, and M_{i-1} then reads the data. In this way, information is transferred from a higher stratum to a lower one.

Central to the encodings of M_i is the unary predicate $ACCEPT_i$. This predicate determines whether M_i accepts its input, and the i^{th} stratum of $R(L)$ consists mainly of linear rules for defining it. From the perspective of top-down inference, the rules defining $ACCEPT_i$ generate a computation path for M_i, that is, a series of machine *id*'s. As each *id* is generated, it is inserted into the database hypothetically. In this way, a computation path for M_i is "grown" one *id* at a time. Since M_i is a non-deterministic machine, it may have many computation paths. The rulebase generates each path non-deterministically during top-down inference. If any path reaches an accepting state, then $ACCEPT_i(t)$ is true, where t is the time at which the computation began.

More formally, consider a particular computation path of machine M_i. Let $DB_i(t)$ be a database encoding the *id* at time t on this path. $R(L)$ is constructed so that $ACCEPT_i$ has the following property:

$$
\begin{array}{ll}
 & R(L), DB_i(t) \ \vdash \ ACCEPT_i(t) \\
\textit{if} & R(L), DB_i(t) + DB_i(t+1) \ \vdash \ ACCEPT_i(t+1) \\
\textit{if} & R(L), DB_i(t) + DB_i(t+1) + DB_i(t+2) \ \vdash \ ACCEPT_i(t+2) \\
 & \qquad \ldots \\
\textit{if} & R(L), DB_i(t) + DB_i(t+1) + \cdots + DB_i(t+k) \ \vdash \ ACCEPT_i(t+k) \\
\textit{if} & \textit{the id represented by } DB_i(t+k) \ \textit{ has an accepting control state.}
\end{array}
$$

This represents a top-down line of inference. In trying to infer $ACCEPT_i(t)$, the rules in the i^{th} stratum of $R(L)$ first examine the database $DB_i(t)$ to determine whether the finite control is in an accepting state. If not, then $DB_i(t+1)$ is generated and added to the database hypothetically. The process then repeats: In trying to infer $ACCEPT_i(t+1)$, the rules in $R(L)$ examine the database $DB_i(t+1)$ to determine whether the finite control is in an accepting state. If not, then $DB_i(t+2)$ is generated and added to the database hypothetically. The process is repeated until an *id* with an accepting control state is generated. In this way, as inference proceeds, the database is hypothetically expanded and represents a growing computation path.

The above line of inference represents a particular computation path of M_i. Since M_i is non-deterministic, it may have many such paths. Each computation path gives rise to a different line of inference.

The database $DB_i(t)$ represents an *id* of machine M_i. Since M_i is non-deterministic, it may have many computation paths which begin at this *id*. For each such path, $R(L)$ generates a top-down line of inference like the one just described. $ACCEPT_i(t)$ is true iff at least one of these paths reaches an accepting state. $ACCEPT_i$ thus has the following important property:

$$R(L), DB_i(t) \vdash ACCEPT_i(t) \quad \textit{iff} \quad DB_i(t) \textit{ represents an accepting id.}$$

Accepting *id*'s were defined in Section 4.3.2 for alternating Turing machines, but the definition applies equally well to non-deterministic machines. Essentially, an *id* is accepting iff some computation path leads from this *id* to an *id* with an accepting control state. Note that M_i accepts its input iff its initial *id* is accepting. Thus, once we can detect accepting *id*'s, we can determine whether computations terminate successfully.

The rulebase $R(L)$ encodes not just a single machine M_i, but a composite machine made up of $M_k, ..., M_1$. At any point during its computations, several of the M_i may have been started and may be in the middle of a computation. Top-down inference simulates these computations; and so at any point during inference, the database may contain many computation paths, one for each M_i. When the composite machine has invoked oracles to a depth of j, machines $M_k, ..., M_{k-j}$ have been started, and the database contains $j+1$ computation paths. At this point, only machine M_{k-j} is actually running and only its computation path is growing, the others being in a state of suspended computation, waiting for their oracles to finish.

If at some point, M_{k-j} invokes its oracle, then M_{k-j} will be suspended, M_{k-j-1} will begin computing, and a new computation path will begin in the database. This new path is completely hypothetical: After M_{k-j-1} returns its answer, the database is restored to the state it was in just before M_{k-j-1} was invoked. In this way, no trace is left in the database of the computations performed by M_{k-j-1}. Machine M_{k-j} then continues its own computations as if it were given the correct answer by an oracle which performed no computations at all. Intuitively, oracle computations are performed in a "hypothetical world" and do not consume any "real time".

This outlines the way in which the predicates $ACCEPT_i$ are computed. Given rules defining these predicates, it is a straightforward matter to define the predicate $ACCEPT$

of statement (8.1). This predicate is defined by a single rule which initiates the computations of the entire composite machine. Recall that the database $DB(\overline{s})$ encodes the initial tape contents for all the machines $M_k, ..., M_1$. To simulate the computations of the composite machine, we must add the initial control information for machine M_k. (The control information for the other machines is added when they invoked as oracles.) The following rule inserts this information into the database hypothetically and initiates the simulation of machine M_k:

$$ACCEPT \;\; \leftarrow \;\; ACCEPT_k(0)\,[add: CONTROL_k^{q_0}(0,0,0)].$$

The hypothetical addition says that at time 0, machine M_k is the initial state q_0 and both tape heads are at position 0. The attempt to infer $ACCEPT_k(0)$ then succeeds iff M_k accepts the input $\overline{s}$. This rule is adequate in that it results in a rulebase $R(L)$ that satisfies statement (8.1). We are trying, however, to construct a rulebase that is free of constant symbols. For this reason, we use the following equivalent rule instead:

$$ACCEPT \;\; \leftarrow \;\; FIRST(x),\; ACCEPT_k(x)\,[add: CONTROL_k^{q_0}(x,x,x)].$$

Given a rulebase that defines the predicates $ACCEPT_k, ..., ACCEPT_1$, we add this rule to it to form the rulebase $R(L)$. The rest of this chapter provides rules defining these predicates.

8.2.3 Implementing the Predicate $ACCEPT_i$

The predicate $ACCEPT_i$ is defined by three types of rules: (i) those which detect accepting states, (ii) those which encode the transition relation of M_i, and (iii) those which encode the mechanism for invoking the oracle M_{i-1}. Negation-as-failure is used only in rules of type (iii), to detect those cases in which the oracle returns *no*. We treat each of the three types of rule in turn.

(i) Suppose that q_a is an accepting state of machine M_i. Then any *id* containing q_a is an accepting *id*. This is easily encoded with the following rule:

$$ACCEPT_i(t) \;\leftarrow\; CONTROL_i^{q_a}(j_1, j_2, t).$$

The variable t records the time at which acceptance occurs, and the variables j_1 and j_2 signify that the tape-head positions are unimportant.

(ii) For each element of the machine's transition relation, we write a single hypothetical rule. For example, suppose that M_i has the following transition:

> If the finite control is in state q and the work head is scanning symbol b, then (1) write symbol c onto the work tape and move the work head one cell to the right, (2) write symbol d onto the oracle tape and move the oracle head one cell to the right, and (3) put the finite control into state q'.

This is encoded with the following rule:

$$\begin{aligned}&ACCEPT_i(t) \;\leftarrow\; CONTROL_i^q(j_1, j_2, t),\; CELL_i^b(j_1, t),\\ &\quad NEXT(t, t'),\; NEXT(j_1, j_1'),\; NEXT(j_2, j_2'),\\ &\quad ACCEPT_i(t')\,[add : CONTROL_i^{q'}(j_1', j_2', t'),\; CELL_i^c(j_1', t'),\; CELL_{i-1}^d(j_2', t')].\end{aligned}$$

The body of this rule does several things. The first line determines whether the finite control of machine M_i is in state q and whether its work head is scanning the symbol b. If so, then the second line computes the next point in time t', the next position of the work head j_1', and the next position of the oracle head j_2'. The third line then inserts the new control information into the database hypothetically. Notice that the symbol d is inserted into the work tape of machine M_{i-1}, which is also the oracle tape of M_i. These hypothetical insertions specify the next *id* in the computation path of machine M_i. The third line then continues the simulation of M_i's computations by asking whether the new *id* is accepting.

Since M_i is a non-deterministic machine, it may have many successor *id*'s. Each possibility is represented by a distinct rule. From the perspective of top-down inference, it is the non-deterministic choice of which rule to invoke which determines which computation path is followed.

(iii) Finally, we encode the mechanism for invoking oracles. When machine M_i invokes its oracle, the oracle replies either *yes* or *no*, depending on whether it accepts or rejects the string which M_i has written onto its oracle tape. Because M_i is an oracle machine, its finite control has three special states: $q_?$, q_y and q_n. When M_i enters state $q_?$, the oracle is invoked and M_i is suspended until the oracle returns an answer. If the oracle returns *yes*, then M_i enters state q_y; and if it returns *no*, then M_i enters state q_n.

We encode this mechanism with the following two rules. These rules use the predicate $ORACLE_{i-1}(t)$, which is true iff the oracle returns *yes*. $ORACLE_{i-1}$ is easily defined in terms of $ACCEPT_{i-1}$, and its definition will be given shortly.

$$\begin{aligned} ACCEPT_i(t) \quad &\leftarrow \quad CONTROL_i^{q_?}(j_1, j_2, t),\ ORACLE_{i-1}(t),\ NEXT(t, t'), \\ &\qquad ACCEPT_i(t')\,[add: CONTROL_i^{q_y}(j_1, j_2, t')]. \\ ACCEPT_i(t) \quad &\leftarrow \quad CONTROL_i^{q_?}(j_1, j_2, t),\ \sim ORACLE_{i-1}(t),\ NEXT(t, t'), \\ &\qquad ACCEPT_i(t')\,[add: CONTROL_i^{q_n}(j_1, j_2, t')]. \end{aligned}$$

The body of each rule has four premises. Consider the first rule. The first premise determines whether the finite control of M_i is in state $q_?$. If so, the second premise invokes machine M_{i-1} as an oracle. If the oracle returns *yes*, then the third premise computes the next point in time t'. The fourth premise then inserts new control information into the database hypothetically. This information specifies a new *id* in which the finite control is in state q_y. Note that the position of the work head j_1 and the position of the oracle head j_2 remain unchanged. In addition, the fourth premise continues the simulation of M_i's computations by asking whether the new *id* is accepting.

The second rule is similar to the first except that it determines whether the oracle returns *no*. If so, it puts the finite control of M_i into state q_n and continues the computations. Notice that an oracle answer of *no* is represented by the failure to prove the subgoal $ORACLE_{i-1}(t)$. This is the only place in the rulebase where negation-as-failure is used, but its use here is essential. Without it, data complexity could not climb from one level in the polynomial time hierarchy to the next.

The following rule defines the predicate $ORACLE_{i-1}$. It puts M_{i-1} into its initial configuration and begins the simulation of its computations:

$$ORACLE_{i-1}(t) \quad \leftarrow \quad FIRST(j),\ ACCEPT_{i-1}(t)\,[add: CONTROL_{i-1}^{q_0}(j, j, t)].$$

This rule places the tape heads of M_{i-1} at the beginning of their respective tapes and puts the finite control into its initial state state q_0. The attempt to infer $ACCEPT_{i-1}(t)$ then succeeds iff M_{i-1} accepts the string which is on its work tape. Until this rule is invoked, there are no formulas in the database of the form $CONTROL^q_{i-1}(j_1, j_2, t)$. This rule thus defines a unique starting point for the computations of M_{i-1}.

8.2.4 The Frame Axiom

The above rules determine the *changes* caused to an *id* by a machine transition. The greater part of an *id*, however, remains unchanged by such transitions. Indeed, except for those cells under the tape heads, the contents of the machine tapes remain unchanged. This is an instance of the *frame axiom* [32], and we must write rules to encode it. Such rules are necessary only because we are representing time explicitly; i.e., the database represents a sequence of *id*'s, and rules are needed to copy the unchanged portion of an *id* from one instant of time to the next.

In Horn logic programming, negation-as-failure normally plays a central role in any encoding of the frame axiom [32]. We do not have this luxury however. The rulebase $R(L)$ which we are constructing must have no more than k strata, and we have already created k of them, one for each oracle machine. Any attempt to use negation-as-failure will add new strata to the rulebase. We therefore go to some pains to keep our encoding of the frame axiom negation-free.

The frame axiom applies to all tape cells which are *not* beneath a tape head, but it may also apply to some cells which *are*. This is because an oracle machine does not always write on its tapes. If the machine has not yet been started, for instance, then it does not write. Even if it has been started, it may not write because it has invoked its oracle and is in a state of suspended computation. We say that a machine is *active* at time t iff it has been started and is not suspended. Note that exactly one machine is active at any given time.

We encode these ideas with the rules below, which define two predicates for each machine, $ACTIVE_i$ and $INACTIVE_i$. $ACTIVE_i(t)$ means that machine M_i is active at time t, and $INACTIVE_i(t)$ means that it is not active at time t. We could use negation-as-failure to define $INACTIVE_i$ in terms of $ACTIVE_i$, but this would introduce a new stratum into the rulebase. Instead, the rules use the predicates $CONTROL_j^q$ to determine whether M_i is active. In these rules, t, j_1 and j_2 are variables.

$$\begin{aligned} &ACTIVE_i(t) \leftarrow CONTROL_i^q(j_1, j_2, t) && \text{for all } q \neq q_? \\ &INACTIVE_i(t) \leftarrow CONTROL_i^{q_?}(j_1, j_2, t) && \\ &INACTIVE_i(t) \leftarrow ACTIVE_j(t) && \text{for all } j > i \end{aligned}$$

The first of these three rules says that M_i is active if it has been started and is not suspended, i.e., if it is in some state q other than $q_?$. The second rule says that M_i is inactive if it has been started and is suspended, i.e., if it is in state $q_?$. The third rule says that M_i is inactive if it has not been started, i.e., if some machine above it is active (Recall that M_i has been started iff every machine above it is suspended).

There is another complicating factor: Machines M_i and M_{i+1} share a tape, the work tape of M_i being the oracle tape of M_{i+1}. Thus, a tape may have two different tape heads writing on it (although at most one of them will be writing at any given time). To deal with this, we first write the rules below, which identify the positions of the work head and the oracle head of machine M_i at time t. These rules define the predicates $HEAD_i^w$ and $HEAD_i^o$. $HEAD_i^w(j,t)$ means that at time t, the work head of machine M_i is positioned over cell j. Likewise for $HEAD_i^o(t,j)$ and the oracle head.

$$\begin{aligned} HEAD_i^w(j,t) &\leftarrow CONTROL_i^q(j,j',t) \\ HEAD_i^o(j,t) &\leftarrow CONTROL_i^q(j',j,t) \end{aligned}$$

where t, j and j' are variables. We write a pair of rules like this for every control state q.

We can assume that if a machine is active, then both its tape heads are writing. We thus need to identify those tape cells which are *not* beneath a tape head of an active machine. These are exactly the cells to which the frame axiom applies. To do this we introduce two predicates. The first predicate is $NWRITING_i^w(j,t)$, which means that at time t, the work head of machine M_i is not writing anything at position j of its work tape. This predicate is defined by the following rules:

$$NWRITING_i^w(j,t) \leftarrow INACTIVE_i(t)$$
$$NWRITING_i^w(j,t) \leftarrow ACTIVE_i(t),\ HEAD_i^w(j',t),\ NEQ(j,j').$$

The first rule handles the case when M_i is inactive. In this case it is not writing on any cell of its work tape. The second rule handles the case when M_i is active. In this case, the machine is only writing at cell j', the position of the work head. Consequently, it is not writing on any tape cell j which is not equal to j'. The predicate NEQ determines whether two tape cells are not equal. This predicate can be defined without negation by noting that $j_1 \neq j_2$ iff $j_1 < j_2$ or $j_2 < j_1$. This is encoded by the following rules:

$$NEQ(j_1,j_2) \leftarrow BEFORE(j_1,j_2)$$
$$NEQ(j_1,j_2) \leftarrow BEFORE(j_2,j_1)$$
$$BEFORE(j_1,j_2) \leftarrow NEXT(j_1,j_2)$$
$$BEFORE(j_1,j_3) \leftarrow NEXT(j_1,j_2),\ BEFORE(j_2,j_3).$$

In a similar fashion, we define a predicate $NWRITING_i^o(j,t)$, which means that at time t, the oracle head of machine M_i is not writing anything at position j of its oracle tape. This predicate is defined by the following rules:

$$NWRITING_i^o(j,t) \leftarrow INACTIVE_i(t)$$
$$NWRITING_i^o(j,t) \leftarrow ACTIVE_i(t),\ HEAD_i^o(j',t),\ NEQ(j,j').$$

Given the two predicates $NWRITING_i^w(j,t)$ and $NWRITING_i^o(j,t)$, we can encode the frame axiom in a straightforward way. The following rule applies the axiom to all tape cells which do not have a tape head writing on them. In this rule, j is a position on the work tape of machine M_i. Only two tape heads can write on this cell, the work head of M_i and the oracle head of M_{i+1}. If neither of these heads is in fact writing on the cell, then its contents are propagated forward from time t to time $t+1$. We create a rule like this for each tape symbol c.

$$CELL_i^c(j,t') \leftarrow NEXT(t,t'),\ CELL_i^c(j,t),\ NWRITING_i^w(j,t),\ NWRITING_{i+1}^o(j,t).$$

This completes the encoding of the frame axiom. Adding these rules to those of the previous sections defines a rulebase $R(L)$ satisfying statement (8.1). This statement, in turn, establishes the lower complexity bounds of Theorems 8.1, 8.2, and 8.3.

8.3 Expressibility

We have shown that for each level of the polynomial time hierarchy Σ_k^{P}, there is a linear rulebase whose data complexity is complete for Σ_k^{P}. However, as discussed in Chapter 6, it does not necessarily follow that linear rulebases are expressively complete for PHIER, since the data complexity of a logic may be complete for Σ_k^{P} and yet there may be many database queries in Σ_k^{P} that it cannot express. The example of Datalog (function-free Horn logic) is instructive. The complexity of Datalog is complete for P, yet without negation it can only express monotonic queries. Even with negation, it cannot express the query *EVEN* described in Section 6.1. In contrast to Datalog, rulebases of hypothetical additions have no such gaps in their expressibility. By employing the techniques of Chapter 6, this section shows that linear rulebases are expressively complete for PHIER. That is, any generic database query in PHIER can be represented as a stratified linear rulebase of hypothetical additions. Rulebases in which the top stratum is hypothetical play a key role in this development. In particular, rulebases that have k hypothetical strata and a hypothetical top-stratum are are expressively complete for Σ_k^{P}.

Although linear rulebases express every generic query in PHIER, they also express some non-generic queries. Non-genericity is caused by constant symbols in the rulebase, which cause the rulebase to give special treatment to some elements of the data domain. By barring constant symbols from the rulebase, non-genericity is eliminated. Thus, stratified linear rulebases that are constant free express exactly the generic queries in PHIER. This gives an exact characterization of the database queries in PHIER. In addition, Immerman has shown that the generic queries in PHIER are exactly the queries definable in second-order logic [30]. Therefore, stratified linear rulebases without constant symbols express exactly the second-order definable queries.

Besides characterizing the queries in PHIER, we also characterize the queries in Σ_k^{P}. This characterization is based on the set $\mathcal{R}_k^3$, defined at the beginning of this chapter. $\mathcal{R}_k^3$ is the set of linear rulebases that have k hypothetical strata and a hypothetical top stratum. By Theorem 8.3, its data complexity is complete for Σ_k^{P}. This section uses the techniques of Section 6.3 to show that $\mathcal{R}_k^3$ is also *expressively* complete for Σ_k^{P}. By looking only at the constant-free rulebases in $\mathcal{R}_k^3$, we get exactly the generic queries in Σ_k^{P}.

The sets $\mathcal{R}_k^1$ and $\mathcal{R}_k^2$ were also defined at the beginning of this chapter, and similar complexity results were derived for them. However, it is only for $\mathcal{R}_k^3$ that we can establish expressive completeness. In the case of $\mathcal{R}_k^2$, the complexity bounds given by Theorem 8.2 are not tight enough. And in the case of $\mathcal{R}_k^1$, the techniques of Chapter 6 do not work, because they require a rulebase with more than k strata. Neither of these problems exists for $\mathcal{R}_k^3$, whence it forms the basis of our expressibility results. In particular, we prove the following lemma, which establishes the expressive completeness of $\mathcal{R}_k^3$.

Lemma 8.4 *Let ϕ be a generic database query of type $\overline{\alpha} \to \beta$ whose graph is in Σ_k^{P}. Then, there is a rulebase $R(\phi)$ in $\mathcal{R}_k^3$ that expresses ϕ. That is, for any database DB of type $\overline{\alpha}$,*

$$R(\phi), DB \vdash OUT(\overline{x}) \quad \textit{iff} \quad \overline{x} \in \phi(DB) \tag{8.2}$$

where OUT is a predicate symbol of arity β.

We provide only a sketch of the proof for Lemma 8.4, since the details are almost identical to those of Theorem 6.3.

Since the graph of ϕ is in Σ_k^{P}, there is a cascade of NP oracle machines $M_k, ..., M_1$ that recognizes it, i.e., that recognizes the language $\{\langle DB, \overline{x}\rangle \mid \overline{x} \in \phi(DB)\}$. Section 8.2 showed how to construct a rulebase $R(M_k...M_1)$ that encodes the computations of a compound machine such as this. However, this construction is not exactly the rulebase needed here. The construction of Section 8.2 assumes that the initial tape contents of $M_k, ..., M_1$ are encoded as a database $DB(\overline{s})$, where $\overline{s}$ is the input to machine M_k. $DB(\overline{s})$ is also assumed to contain a counter. $DB(\overline{s})$ is therefore different from the database DB given here. We must therefore provide rules which when given DB and $\overline{x}$, will construct $DB(\overline{s})$.

Section 6.3 developed rules for doing precisely this. These rules perform two main tasks: (i) They hypothetically generate a linear order of the data domain, and (ii) they encode the database DB and the tuple $\overline{x}$ in terms of the predicates $CELL_k^c(j,t)$. Step (i) provides the rulebase with a counter, and step (ii) effectively encodes the database and the tuple onto the input tape of machine M_k. Let us call these rules ΔR. If ϕ is a generic query, then adding ΔR to $R(M_k...M_1)$ gives a rulebase $R(\phi)$ that satisfies statement (8.2), as shown in Section 6.3. To prove Lemma 8.4, we need only show that $R(\phi)$ is in $\mathcal{R}_k^3$.

The first point to observe is that there is no mutual recursion between the predicates of ΔR and those of $R(M_k...M_1)$. A review of Section 6.3 shows that the top stratum of ΔR invokes the predicate $ACCEPT$, which is defined in $R(M_k...M_1)$. In its turn, $R(M_k...M_1)$ invokes the machine predicates $CELL_i^c$ and $CONTROL_i^q$, which are defined in the lower strata of ΔR. The important point is that this is the only way in which the predicates of ΔR and $R(M_k...M_1)$ depend on each other. In combining these two rulebases, we can therefore sandwich $R(M_k...M_1)$ in between the top two strata of ΔR. Since $R(\phi)$ and $R(M_k...M_1)$ are both linear, it follows immediately that each stratum of $R(\phi)$ is linear, and therefore that $R(\phi)$ itself is linear. $R(\phi)$ thus satisfies the first requirement for membership in $\mathcal{R}_k^3$.

We must also show that $R(\phi)$ has exactly k hypothetical strata and that its top stratum is hypothetical. Recall the following properties of the constructions of Sections 8.2 and 6.3: (i) $R(M_k...M_1)$ has exactly k strata, each of them hypothetical, and (ii) the top stratum of ΔR is hypothetical (consisting mostly of rules for generating a linear order) and all its lower strata are Horn (consisting mostly of rules for encoding the database onto the machine tape). Since $R(M_k...M_1)$ is sandwiched between the top two strata of ΔR, the top $k+1$ strata of the resulting rulebase are hypothetical, and all lower strata are Horn. The important point, however, is that the top two strata of the new rulebase can be merged into a single stratum. This is possible because the top stratum does not use negation-as-failure when invoking predicates defined in the next-to-top stratum (In particular, the top stratum invokes the predicate $ACCEPT$ only in a positive sense, as seen in the rules of Section 6.3.1). $R(\phi)$ thus has a stratification in which the top k strata are hypothetical and all lower strata are Horn. Hence $R(\phi)$ is in $\mathcal{R}_k^3$, thus completing the proof of Lemma 8.4.

Although $\mathcal{R}_k^3$ expresses every typed generic query in Σ_k^{P}, it also expresses some non-generic queries. As discussed above, non-genericity is caused by constant symbols in the rulebase, which cause the rulebase to give special treatment to some elements of the data domain. By barring constant symbols from the rulebase, non-genericity is eliminated. Hence, rulebases in $\mathcal{R}_k^3$ that are constant free express exactly the typed generic queries in Σ_k^{P}. We therefore make the following definition.

Definition 8.4 *$\mathcal{R}_k^4$ is the set of rulebases in $\mathcal{R}_k^3$ that are constant free.*

In constructing the rulebases ΔR and $R(M_k \ldots M_1)$, care was taken to use only rules that are constant free. The rulebase $R(\phi)$ of Lemma 8.4 is thus in $\mathcal{R}_k^4$, and it follows that $\mathcal{R}_k^4$ expresses every generic query in Σ_k^{P}. In addition, since $\mathcal{R}_k^4$ is constant free, it expresses only generic queries, by Corollary 6.2. We therefore have the following theorem, which is the main result of this section.

Theorem 8.5 *$\mathcal{R}_k^4$ expresses exactly the typed generic queries in Σ_k^{P}.*

This theorem immediately gives rise to a characterization of the typed generic queries in PHIER. Any such query will be in Σ_k^{P} for some k. It is thus expressible as a stratified linear rulebase of hypothetical additions that is constant free. Conversely, any such rulebase expresses a typed generic query in Σ_k^{P}, where k is the number of strata in the rulebase. Since PHIER $= \cup_k \Sigma_k^{\mathrm{P}}$, the following result is immediate.

Corollary 8.6 *Linear rulebases of hypothetical additions that are constant free express exactly the typed generic queries in PHIER.*

Immerman has shown that the typed generic queries in PHIER are precisely the queries definable in second-order logic [30]. This gives us another characterization of the queries expressed by stratified linear rulebases.

Corollary 8.7 *Stratified linear rulebases of hypothetical additions that are constant free express exactly the second-order definable queries.*

Chapter 9

Predicate Substitution

This chapter[1] develops a method for reusing and modifying deductive databases. The need for such techniques occurs when new rulebased applications differ only slightly from existing ones or when an application is to be incrementally updated. Such techniques are particularly important when reprogramming is expensive or unreliable. Unfortunately, these issues have not yet received significant attention in the deductive database literature. This chapter attempts to address them, focusing on a particular class of modifications that we call *predicate substitution*. The idea is to reuse and modify a set of rules by substituting one predicate symbol for another. Augmented with this capability, deductive databases can better exploit the polymorphism that exists in many database operations.

Transitive closure is a simple example of such an operation. It is polymorphic in that every binary relation has a transitive closure. Nevertheless, in many deductive database systems, one cannot write a rulebase that computes the transitive closure of an arbitrary binary relation. Instead, one can only write rules that compute the transitive closure of a particular relation. The following rules, for example, compute the transitive closure of the relation R:

$$TR(x,y) \leftarrow R(x,y).$$
$$TR(x,z) \leftarrow R(x,y),\, TR(y,z).$$

To compute the transitive closure of another relation, say S, one must write a new set of rules, defining a new predicate TS in terms of S. In this way, a user is forced to rewrite rulebases if he wants to apply them to different sets of data.

[1]This chapter elaborates on material that originally appeared in [11].

Predicate substitution carries out this rewriting automatically. For instance, to compute the transitive closure of S, the user specifies that S be substituted for R in the two rules above. Two new rules, equivalent to the following, are then generated:

$$TS(x, y) \leftarrow S(x, y).$$
$$TS(x, z) \leftarrow S(x, y), TS(y, z).$$

If necessary, new predicates such as TS are also generated. In this way, predicate substitution allows a user to specify new rulebases by reusing and modifying old ones.

To make these ideas precise, we develop a logic programming language that combines logical inference with predicate substitution. Syntactically, the main feature of this language is a modal-like operator that specifies predicate substitutions. This operator appears in rule bodies, just as hypothetical operators do, and we refer to such rules as *rules with substitution.* These rules allow a user to select other rules for reuse and modification and to combine the modified rules into larger rulebases. Rulebases defined in this way can themselves be reused and modified, so that the process of constructing rulebases is closed under predicate substitution.

There is a close connection between rules with substitution and the hypothetical rules developed in earlier chapters. One similarity is syntactic, since both hypothetical rules and rules with substitution employ modal-like operators in their premises. The similarity goes beyond syntax however, and in Section 9.2, we outline how to treat predicate substitution as a hypothetical operator, one that operates on the database, not on the rulebase. In fact, without too much difficulty, one can develop a semantics for predicate substitution that is similar to the semantics developed in Chapter 3 for hypothetical inference.

Such a semantics, however, obscures the role of predicate substitution in the reuse and modification of rulebases. For this reason, we develop an alternate semantics, one not based on hypothetical databases. In this semantics, rulebases *without* substitution are the models of rulebases *with* substitution. This semantics corresponds to the intuition that rules with substitution are written by a user who has rules without substitution "in mind." This intuition is similar to the view taken of ordinary Datalog rules, in which the user has an "intended model" in mind (the least model of the rules), which he could have written if he had enough space and enough time. Hence, it can be argued that in going from Datalog to Datalog-with-substitution, we achieve an upgrade similar to that in going from relational algebra to Datalog.

To support this view, we show that predicate substitution is a concise notation for describing large rulebases. This conciseness is achieved by reusing and modifying rule sets, possibly many times. In fact, by combining recursion and substitution, it is possible to reuse and modify rule sets infinitely many times. Using this idea, we show that a *finite* set of rules with substitution can specify an *infinite* set of rules without substitution. In such cases, substitution provides an extremely concise notation for representing sets of Datalog rules. Fortunately, it is not necessary to materialize all these rules, since our inference system works directly on the rule specifications, not on the rules themselves. This is similar to inference in Datalog in that a query can be evaluated without materializing the least fixpoint of the rulebase.

Since Datalog with substitution can specify an infinite rulebase, predicate substitution is more than just a syntactic convenience: It actually increases the expressive power of Datalog. In fact, we show that Datalog with substitution can solve an EXPTIME-complete problem—a task that is impossible for Datalog without substitution.[2]

We believe that substitution is a principle that can be applied to logic-based languages generally in order to facilitate reuse and modification. In this chapter, however, and in the rest of this dissertation, we focus on Horn logics and on Datalog in particular. In this chapter, several examples are given to illustrate predicate substitution and its utility. Predicate substitution is then defined precisely and an inference system is developed for Horn rules with substitution.

9.1 Examples

This section presents several examples of predicate substitution and discusses several other possible applications. Each of the examples uses Horn logic with substitution to

[2]In this chapter, as throughout this dissertation, the terms EXPTIME and "exponential time" refer to DTIME[$2^{n^{O(1)}}$].

exploit the polymorphic nature of certain database operations in a way that Horn logic alone cannot. In these examples, predicate substitution is represented by modal-like expressions of the form $A[P/Q]$ where P is a base predicate. Intuitively, this expression means, "substitute Q for P in the rules defining A." After performing the substitution, we treat this expression as an ordinary predicate of classical logic, referring to as a *predicate with substitution*. Such expressions can be used in queries or in rule bodies, just as other predicates can.

Example 9.1 Let R be the following rulebase, which defines the transitive closure A of a base relation A_0:

$$A(x, y) \leftarrow A_0(x, y).$$
$$A(x, z) \leftarrow A_0(x, y), A(y, z).$$

Thus $R, DB \vdash A(x, y)$ iff (x, y) is in the transitive closure of A_0, for any database DB. To compute the transitive closure of some other relation B_0, we can rewrite these rules replacing A_0 by B_0. The predicate with substitution $A(x, y)[A_0/B_0]$ does this automatically. That is $R, DB \vdash A(x, y)[A_0/B_0]$ iff (x, y) is in the transitive closure of B_0. This predicate with substitution can also be used in rule bodies to define other predicates. In this way, the following rule defines B to be the transitive closure of B_0:

$$B(x, y) \leftarrow A(x, y)[A_0/B_0]$$

This rule is called a rule with substitution. Intuitively, it means that the Horn rules defining B are the same as those defining A with A_0 replaced by B_0. If we add this rule to the rulebase R, then $R, DB \vdash B(x, y)$ iff (x, y) is in the transitive closure of B_0. In this way, predicate substitution exploits the polymorphic nature of transitive closure. $\Box$

Example 9.2 Given a rulebase R that sorts people by age, we reuse it, with slight modifications, to sort people by weight. We suppose that the database DB defines the predicates $Age(x, y)$ and $Weight(x, w)$, which mean that person x has age y and weight

w, respectively. We suppose also that the rulebase defines a predicate $\mathit{SortAge}(x,n)$ that sorts people by age. That is,

$$R, DB \vdash \mathit{SortAge}(x,n)$$

iff person x is the n^{th} oldest person in the database. We do not know how the predicate *SortAge* is implemented, nor what algorithm it uses. We know only that the rules defining *SortAge* are written in terms of the base predicate *Age*.

To sort people by weight instead of by age, we can rewrite these rules substituting *Weight* for *Age*. The predicate with substitution $\mathit{SortAge}(x,n)[\mathit{Age}/\mathit{Weight}]$ does this automatically. That is,

$$R, DB \vdash \mathit{SortAge}(x,n)[\mathit{Age}/\mathit{Weight}]$$

iff x is the n^{th} heaviest person in the database. This predicate with substitution can be used in rule bodies to define other predicates. In this way, the following rule defines a predicate *SortWeight* that sorts people by weight:

$$\mathit{SortWeight}(x,n) \ \leftarrow\ \mathit{SortAge}(x,n)[\mathit{Age}/\mathit{Weight}]$$

This rule is called a rule with substitution. Intuitively, it means that the Horn rules defining *SortWeight* are the same as those defining *SortAge* except that *Age* is replaced by *Weight*. In this way, predicate substitution exploits the fact that sorting is polymorphic on the sorted attribute. □

In a predicate with substitution such as $A[P/Q]$ the predicate P must be a base predicate, but the predicate Q need not be. The next example illustrates this.

Example 9.3 Continuing with the previous example, suppose that the predicate $\mathit{SortAge}(x,n)$, which sorts people by age, is written in terms of the base predicate $\mathit{Person}(x)$, which means that x is a person, and in terms of the base predicate $\mathit{Age}(x,y)$. We suppose that the database also defines predicates $\mathit{Student}(s)$ and $\mathit{Take}(s,c)$, which mean respectively that s is a full-time student and that s has taken course c. We want to use the predicate *SortAge* to sort students instead of people, but only the full-time students who have taken *cs250*. The following rules do exactly this:

$$\mathit{Student250}(s) \leftarrow \mathit{Student}(s), \mathit{Take}(s, cs250).$$
$$\mathit{Sort250}(s, n) \leftarrow \mathit{SortAge}(s, n)\,[\mathit{Person}/\mathit{Student250}]$$

The first rule defines a predicate *Student250* identifying the full-time students who have taken *cs250*. The second rule defines a predicate *Sort250* that sorts these students by age. Conceptually, *Sort250* is defined by Horn rules that are identical to the Horn rules defining *SortAge* except that the *base* predicate *Person* is replaced by the *derived* predicate *Student250*. If we wanted to, we could carry this process one step further and apply a substitution to the predicate *Sort250* so that the students are sorted by grade instead of by age. □

The next example describes a module of rules for solving a graph problem. This general-purpose module is reused and modified in order to solve certain problems in airline flight reservations.

Example 9.4 Let G be a weighted graph in which each edge has an associated cost. Given a path between two nodes, the cost of the path is the sum of the weights on each edge. Let R be a set of rules that for any pair of nodes x and y, computes the minimum cost of all the paths from x to y. The rules in R have one base predicate, $\mathit{Weight}(x, y, w)$, which means that the graph has an edge from x to y of weight w. The output predicate of R is $\mathit{MinCost}(x, y, w)$ where w is the minimum cost of a path from node x to node y.

Suppose that we wish to use the set of rules R in an airline reservation system. The graph G now represents flight connections in the United States. We would like to reuse the rules R for many different notions of cost. In particular, we would like to know the following three parameters for the connections between Tucson, Arizona and Bismarck, North Dakota: (i) the smallest number of stopovers, (ii) the minimum flying time, and (iii) the minimum price. For this purpose, we can reuse the module R, substituting the following predicates for the predicate $Weight$:

- $W_1(x, y, w)$ where w is 1 if x and y are directly connected, and 0 otherwise. In this case, we want to calculate the minimum number of stopovers.

- $W_2(x, y, w)$ where w is the price of a direct flight from x to y. In this case, we want to calculate the minimum price of the flight (assuming that the total cost is the sum of the costs of the individual direct-flights).

- $W_3(x, y, w)$ where w is the flying time for a direct flight from x to y. In this case, we want to calculate the minimum flying time.

Using these three different predicates, we can reuse the minimum-cost module by simply writing the following rules:

$$MinStopovers(x, y, w) \leftarrow MinCost(x, y, w)[Weight/W_1]$$
$$MinPrice(x, y, w) \leftarrow MinCost(x, y, w)[Weight/W_2]$$
$$MinFlyingTime(x, y, w) \leftarrow MinCost(x, y, w)[Weight/W_3]$$

In each case, the rules in the module R are copied and modified so that *Weight* is replaced by W_i. In this way, a possibly complex and large module R does not have to be rewritten every time it is reused. □

The above examples are typical of a class of applications in which a module is to be used repeatedly with different inputs. In such cases, we simply "plug in" different definitions for the base predicates of the module.

In addition, rules with substitution seem well suited to the reuse and modification of expert database systems. Such systems have a variety of applications, such as computer-based medical and legal consultation systems. Kowalski and Sergot, for instance, have encoded the British Nationality Act in Prolog, and McCarty and Sridharan have developed expert systems for reasoning about contract law and corporate tax law [39]. As laws are amended, such systems have to be updated. For instance, sections of the income tax act could be amended to treat residents as citizens. In this case, every occurrence of "citizen" could be replaced by a new predicate meaning "citizen or resident". Similarly, other bodies of law could be amended to replace the predicate "male" by "person". Such updates could easily be expressed and implemented with predicate substitutions.

The need to often reuse and modify existing rule bases occurs in rulebased specifications of the operations of large enterprises, such as big corporations, computer operating systems or various legal systems, such as tax-law, citizenship law etc. When changes are small, it simply does not make sense to rewrite possibly-huge rulebased specifications. It is definitely more attractive to reuse as much as possible of the old definition.

From the above examples it may appear that adding the principle of substitution may increase the convenience of a language but not its expressive power. We show, however, that the expressive power increases as well, and in particular, that predicate substitution increases the expressive power of Datalog (function-free Horn logic). To illustrate this increased power, Section 9.5.2 provides a set of rules that use recursion through substitution to solve an EXPTIME-complete problem—a task impossible for Datalog.

9.2 The Hypothetical Approach

There is a close connection between predicate substitution and hypothetical inference. The most evident similarity is syntactic, since both hypothetical rules and rules with substitution employ modal-like operators in their premises. The similarity goes beyond syntax however, and in this section, we outline how predicate substitution can be treated as a hypothetical operator, one that operates on the database, not on the rulebase.

To see why this is possible, recall that predicate substitution allows rulebases to be applied to different sets of data. The problem is that the data is stored in one set of relations, whereas the rulebase operates on another set. This mismatch can be solved in two distinct ways: (i) by modifying the rulebase, or (ii) by modifying the database. These two approaches lead to two different theories of predicate substitution. This section discusses the latter approach (whereas the rest of this dissertation develops the former approach).

The latter approach, modifying the database, can be formulated in terms of hypothetical transactions. For example, suppose we are given the following two rules for computing the transitive closure of the relation R:

$$TR(x, y) \leftarrow R(x, y).$$
$$TR(x, z) \leftarrow R(x, y),\, TR(y, z).$$

By employing transactions, we can use these rules to compute the transitive closure of some other relation S. We do this by first erasing the data in R, and then copying the data in S to R, so that the extensions of R and S are equal. Using the two rules above, we then compute TR, the transitive closure of R, which is now equal to the transitive closure of S. In effect, we use R as a temporary relation to hold data during inference.

It is important that these transactions be hypothetical. If they are not, then several problems arise. First, the data in R is lost, since it is overwritten by the data in S. Second, concurrency is not possible, since R can be overwritten by only one relation at a time. Third, the solution is not declarative, since transactions are procedural. These problems disappear, however, when the transactions are hypothetical. This is because each hypothetical transaction operates on its own hypothetical database. Since the real database is not affected, data is not lost and concurrency is not hindered. Furthermore, hypothetical transactions admit a declarative semantics, as shown in Sections 3.3 and 5.2. The next example illustrates how to express these hypothetical transactions in our logic of hypothetical insertion and deletion.

Example 9.5 As in Examples 9.2 and 9.3, suppose we are given a rulebase that sorts people by age. We wish to use this rulebase to sort students. The database contains predicates $Person(x)$, $Student(x)$ and $Age(x, y)$, and we are told that the rulebase defines a predicate $SortAge(x, n)$, which means that x is the n^{th} oldest person in the database. We are also told that the rules defining *SortAge* are written in terms of the predicates *Person* and *Age*. We wish to use these rules to sort the data in the predicate *Student*. The following hypothetical rules do exactly this. They define a predicate $SortStudent(x, n)$, which means that x is the n^{th} oldest student in the database.

$$
\begin{aligned}
SortStudent(x,n) &\leftarrow Erase(x,n) \\[4pt]
Erase(x,n) &\leftarrow Person(y),\ Erase(x,n)\,[del : Person(y)]. \\
Erase(x,n) &\leftarrow Empty,\ Copy(x,n). \\
Empty &\leftarrow \sim NonEmpty \\
NonEmpty &\leftarrow Person(y) \\[4pt]
Copy(x,n) &\leftarrow Select(y),\ Copy(x,n)\,[add : Person(y)]. \\
Copy(x,n) &\leftarrow Done,\ SortAge(x,n). \\
Done &\leftarrow \sim NotDone \\
NotDone &\leftarrow Select(y). \\
Select(y) &\leftarrow Student(y),\ \sim Person(y).
\end{aligned}
$$

From a top-down perspective, these rules operate in three phases: (i) they hypothetically erase all data from the database relation *Person*, (ii) they hypothetically copy all data from *Student* into *Person*, and (iii) they invoke the predicate *SortAge* to sort the data in *Person* by age. In effect, *Person* is used as a temporary relation to hold the student data while it is being sorted.

The rules operate in much the same way as those of Examples 6.1 and 6.2, which define the predicate $EVEN$. The first rule, defining the predicate *SortStudent*, begins phase (i) by invoking the predicate *Erase*. The second rule then recursively selects and deletes entries from *Person*. When *Person* is empty, the third rule begins phase (ii) by invoking the predicate *Copy*. The sixth rule then recursively selects entries from *Student* and adds them to *Person*. Finally, when all the entries in *Student* have been copied to *Person*, the seventh rule begins phase (iii) by invoking the predicate *SortAge*. Notice that the predicate *SortAge* is treated as a "black box". We do not know how it is implemented nor what algorithm it uses. We know only that it operates on the predicates *Person* and *Age*. □

The technique illustrated in Example 9.5 achieves the goal of applying a rulebase to different sets of data. It is somewhat clumsy, however, since it does not express in a simple and direct way the idea that one predicate is being substituted for another. Instead of *specifying* the substitution, the rules implement it in a logic program that recursively inserts and deletes individual tuples. This clumsiness manifests itself in several ways:

- It introduces recursion into what might otherwise be a non-recursive rulebase. This could preclude the use of certain optimization strategies and could also cause us to over estimate the computational complexity of a rulebase.

- It requires the use of negation-as-failure, thereby complicating the semantics of predicate substitution and detracting from its purely declarative nature.

- It gives little idea of the complexity of predicate substitution. It tells us that EXPTIME is an upper bound, since this is the complexity of hypothetical inference, but it says nothing about the lower bound. Since predicate substitution is a useful operation by itself, we would like to know its complexity in the absence of other hypothetical operations. Establishing this complexity will be awkward, if not impossible, unless we can formulate predicate substitution more crisply.

- Although predicate substitution is a deterministic process, the technique of Example 9.5 treats it as a non-deterministic one. That is, the rules choose tuples from *Person* and *Student* in some non-deterministic order. If *Person* has n tuples and *Student* has m, then there are $n!m!$ different orders to choose from. This exacerbates the problem of optimization, since in practice, some way must be found to ensure that the inference mechanism does not repeat the entire substitution process for each possible order. Problems like this are typical in logic-programming systems, and are typically solved by introducing extra-logical mechanisms, such as Prolog's cut symbol. However, this approach is unattractive for two reasons: it detracts from the declarative nature of predicate substitution, and it shifts the burden of optimization from the system to the user.

These objections to the technique of Example 9.5 are rooted in the attempt to express predicate substitution in terms of hypothetical insertion and deletion. The objections can be overcome by treating predicate substitution as a primitive hypothetical operation, one which is carried out "all at once." Formally, we would introduce a new hypothetical operator $[P/Q]$ that erases P and copies the data in Q to P. The expression $B[P/Q]$ then means that B is true after performing the transaction $[P/Q]$. Under this interpretation, expressions of the form $A \leftarrow B[P/Q]$ are hypothetical rules. It is not difficult to develop a formal semantics for these rules by adapting the semantics for hypothetical insertion and deletion developed in Chapter 3. Under such a semantics, the rulebase in Example 9.5 would be equivalent to the following single rule:

$$SortStudent(x,n) \;\leftarrow\; SortAge(x,n)[Person/Student] \tag{9.1}$$

Besides being more succinct, this rule is non-recursive and negation-free and clearly specifies the users intent to perform a predicate substitution.

Thus, from both the hypothetical point of view and the rulebase-modification point of view, we are lead to the same notation for predicate substitution. How then should we develop the proof theory and model theory of this language? We could treat predicate substitution as an atomic hypothetical operator and adapt the semantics of hypothetical inference. This solution, however, is an exercise that promises no interesting new results or insights beyond what we have already developed in earlier chapters. For this reason, we develop predicate substitution in a completely new fashion, as an operation that reuses and modifies rulebases. From this perspective, it is no longer an operation on databases, but a concise notation for specifying rulebases. In later sections, this perspective leads us to new results, new inference procedures, and a novel interpretation of the complexity of predicate substitution.

9.3 The Rulebase-Modification Approach

This section develops predicate substitution in terms of the reuse and modification of rulebases. The syntax of predicates and rules with substitution is defined precisely, and an inference system is developed in terms of a copy-and-substitute mechanism. The development is initially informal, leaning on a number of illustrative examples. The mechanism is then defined precisely and an axiomatization is given.

Definition 9.1 *Every atomic formula is a predicate with substitution. Furthermore, if α is a predicate with substitution, then so is $\alpha[P/Q]$, where P and Q are predicate* *symbols of equal arity.*

Expressions of the form $[P/Q]$ shall be called *predicate substitutions*, or simply *substitutions*. They should be distinguished from the more familiar form of substitution in which terms are substituted for variables. For instance, in the expression $\{u/f(x), v/y\}$, the terms $f(x)$ and y are to be substituted for the variables u and v, resp. We call these expressions *term substitutions* to distinguish them from predicate substitutions. Throughout this chapter, and the rest of this dissertation, the symbol $[\theta]$ denotes a predicate substitution and σ denotes a term substitution.

Definition 9.2 *(Rules) A rule with substitution is an expression of the form $\beta \leftarrow \beta_1 ... \beta_k$, where β and each β_i are predicates with substitution. If β is atomic, then the rule is called a Horn rule with substitution.*

From the database perspective, a rulebased system consists of two parts: a set of rules, called the *rulebase*; and a set of ground atomic formulas, called the *database*. This dichotomy leads naturally to two types of relations: *base relations* and *derived relations*. A base relation is defined by the database, and an derived relation is defined by the rulebase. The division of relations in this way is common in deductive databases. Furthermore, by separating the rulebase from the database in this way, we are able to treat the rulebase as a mapping from base relations to derived relations. Substitution is then a composition of one map with another.

To represent these two types of relations, we use two types of predicate symbols: base and derived. Base predicate symbols are used to construct the atomic formulas in the database but may not appear in the heads of rules. Conversely, derived predicate symbols may appear in the heads of rules but may not appear in the database.

9.3.1 The Copy-and-Substitute Mechanism

We treat predicates with substitution as derived predicates. For instance, the expression $A(x)[P/Q]$ is treated as just another derived predicate. This predicate has a special meaning however: The rules defining $A(x)[P/Q]$ must be identical to the rules defining $A(x)$ except that P is replaced everywhere by Q.

To be more precise, we define the intension of $A(x)[P/Q]$ in terms of a copy-and-substitute mechanism. First, we copy all the rules in the rulebase, and in this copy, each predicate $B(x)$ is given a new name $B(x)[P/Q]$.[3] In this way, the original rulebase and its copy can co-exist as a single rulebase without interfering with each other. Second, each occurrence of $P(\overline{x})[P/Q]$ in the premises of these rules is replaced by $Q(\overline{x})$. In this way, substitution is carried out. Finally, for all base predicates $B(\overline{x})$, the new predicate $B(\overline{x})[P/Q]$ is replaced by $B(\overline{x})$. In this way, the new rulebase is able to access the database. The result is a rulebase which is identical to the original except that P has been replaced everywhere by Q; and instead of defining $A(x)$, the new rulebase defines $A(x)[P/Q]$, for every derived predicate A.

Example 9.6 Let $\mathcal{S}$ be the following rulebase, where B and C are base predicates:

$$A(x) \leftarrow B(x), P(x).$$
$$A(x) \leftarrow C(x,y), P(x), P(y).$$

Then the expression $A(x)[P/Q]$ is treated as a new predicate symbol defined by the following rules.

$$A(x)[P/Q] \leftarrow B(x), Q(x).$$
$$A(x)[P/Q] \leftarrow C(x,y), Q(x), Q(y).$$

Notice that the definition of $A(x)[P/Q]$ is identical to the definition of $A(x)$ except that P has been replaced by Q in the premise of every rule. Conceptually, these rules are generated automatically; and in particular, they do not have to be supplied by the user. □

The above example is the simplest possible in that there is only one derived predicate and no recursion. The next two examples illustrate the substitute-and-copy mechanism for slightly more complex rulebases.

[3] We could equally well use the notation $B'(x)$ instead of $B(x)[P/Q]$. The latter notation simply provides a systematic way of providing new and distinct predicate names for each substitution.

Example 9.7 Let $\mathcal{S}$ be the following rulebase, where C and D are base predicates:

$$A(x) \leftarrow P(x), C(x,y), B(y).$$
$$B(x) \leftarrow D(x,y), P(y).$$

Then the predicates $A(x)[P/Q]$ and $B(x)[P/Q]$ are defined by the following rules:

$$A(x)[P/Q] \leftarrow Q(x), C(x,y), B(y)[P/Q].$$
$$B(x)[P/Q] \leftarrow D(x,y), Q(y).$$

Since these rules are to be treated classically, we can make the following inference:

$$A(x)[P/Q] \leftarrow Q(x), C(x,y), D(y,z), Q(z).$$

□

Example 9.8 Let $\mathcal{S}$ be the following rulebase, where C is a base predicate:

$$A(x) \leftarrow P(x), C(x,y), A(y)$$
$$A(x) \leftarrow P(x)$$

The following rules then define the predicate $A(x)[P/Q]$:

$$A(x)[P/Q] \leftarrow Q(x), C(x,y), A(y)[P/Q]$$
$$A(x)[P/Q] \leftarrow Q(x)$$

Since these rules are to be treated classically, we can make the following inferences (among others):

$$A(x)[P/Q] \leftarrow Q(x), C(x,y), Q(y)$$
$$A(x)[P/Q] \leftarrow Q(x), C(x,y), Q(y), C(y,z), Q(z)$$

□

The idea of copying rules, renaming predicates and making substitutions applies to Horn rulebases with substitution as well as to those without. Furthermore, the idea can be applied recursively; that is, once predicates such as $A(x)[P/Q]$ have been defined, they can be used to define more complex predicates such as $A(x)[P/Q][R/S]$. These two ideas are illustrated in the next example.

Example 9.9 Let $\mathcal{S}$ be the following rulebase, where Q and R are base predicates:

$$A(x) \leftarrow B(x)[P/Q]$$
$$B(x) \leftarrow P(x), R(x).$$

We then get the following rules:

$$A(x)[R/S] \leftarrow B(x)[P/Q][R/S]$$
$$B(x)[P/Q] \leftarrow Q(x), R(x).$$
$$B(x)[P/Q][R/S] \leftarrow Q(x), S(x).$$

Since these rules are to be treated classically, we can make the following inferences:

$$A(x) \leftarrow Q(x), R(x).$$
$$A(x)[R/S] \leftarrow Q(x), S(x).$$

□

The copy-and-substitute mechanism outlined above describes informally the idea of substitution for Horn rules with substitution. The following definition makes the idea precise.

Definition 9.3 *Let $\mathcal{S}$ be a Horn rulebase with substitution. Then $\mathcal{S}^*$ is the set of rules with substitution constructed from $\mathcal{S}$ as follows:*[4]

1. $\mathcal{S} \subseteq \mathcal{S}^*$

2. *If* $\beta \leftarrow \beta_1 ... \beta_n$ *is in* $\mathcal{S}^*$, *then*
   $\beta[\theta] \leftarrow \beta_1[\theta] ... \beta_n[\theta]$ *is in* $\mathcal{S}^*$ *for every predicate substitution* $[\theta]$.

3. *The rule* $P(\overline{x})[P/Q] \leftarrow Q(\overline{x})$ *is in* $\mathcal{S}^*$ *for every substitution* $[P/Q]$.

4. *The rule* $B(\overline{x})[P/Q] \leftarrow B(\overline{x})$ *is in* $\mathcal{S}^*$ *for every substitution* $[P/Q]$ *and every base predicate* $B \neq P$.

[4]To be precise, $\mathcal{S}^*$ is the smallest set of rules satisfying these four conditions.

The second item in the definition of $\mathcal{S}^*$ ensures that for every rule defining β there is a copy of the rule in $\mathcal{S}^*$ defining $\beta[\theta]$. The third item carries out the substitution $[P/Q]$, so that every occurrence of $P(\overline{x})[P/Q]$ is replaced by $Q(\overline{x})$. The fourth item ignores the substitution when it is applied to base predicates not equal to P, so that every occurrence of $B(\overline{x})[P/Q]$ is replaced by $B(\overline{x})$. The rulebase $\mathcal{S}^*$ thus represents the result of all possible copy-and-substitute operations. As such, it provides a complete definition of all possible predicates with substitution.

9.3.2 Inference from Databases

The copy-and-substitute mechanism developed above takes a rulebase with substitution $\mathcal{S}$ and generates a larger set of rules $\mathcal{S}^*$. This is not a complete inference mechanism however. To be useful, a rulebase must be augmented with a database and the rules used to infer new atomic formulas. Having generated $\mathcal{S}^*$, we now treat it as a set of Horn rules and apply it to a database using classical inference.

Definition 9.4 *Suppose β is a predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. Then,*[5]

$$\mathcal{S}, DB \vdash \beta \quad \textit{iff} \quad \mathcal{S}^* \cup DB \vdash_c \beta$$

Example 9.10 Suppose DB consists of the atoms $Q(b)$, $R(b)$, $Q(a)$, $S(a)$, $P(b)$. Applying rulebase $\mathcal{S}$ from Example 9.9 to this database gives the following inferences (among others):

$$\begin{array}{ll} \mathcal{S}, DB \vdash B(b)[P/Q] & \qquad \mathcal{S}, DB \vdash A(b) \\ \mathcal{S}, DB \vdash A(a)[P/Q] & \qquad \mathcal{S}, DB \vdash B(b) \end{array}$$

$\square$

The set of rules $\mathcal{S}^*$ is infinite and contains an infinite number of distinct predicate symbols. As the next example shows, this is because substitution is not idempotent;

[5] In this definition, as throughout this dissertation, the symbol $\vdash_c$ denotes classical inference, where all free variables are universally quantified.

that is, in general, $A(x)[B/C][B/C]$ is not the same as $A(x)[B/C]$. Thus, from a finite set of predicate symbols, it is possible to create new and distinct predicates whose names have arbitrarily length, such as $A(x)[B/C][B/C][B/C]$. Because of this, $\mathcal{S}^*$ contains an infinite number of distinct predicate symbols and is itself infinite.

Example 9.11 Suppose $\mathcal{S}$ consists of the following two rules, where B and P are base predicates:

$$A(x) \leftarrow P(x)$$
$$Q(x) \leftarrow B(x,y), P(y)$$

The predicates $A(x)[P/Q]$, $A(x)[P/Q][P/Q]$ and $A(x)[P/Q][P/Q][P/Q]$ are then defined as follows:

$$A(x)[P/Q] \leftarrow Q(x)$$
$$A(x)[P/Q][P/Q] \leftarrow B(x,y), Q(y)$$
$$A(x)[P/Q][P/Q][P/Q] \leftarrow B(x,y), B(y,z), Q(z)$$

In general, the definition of $A(x)[P/Q]^i$ is different from the definition of $A(x)[P/Q]^{i+1}$, and so they denote distinct predicates. In other words, substitution is not idempotent. $\Box$

9.3.3 An Axiomatization

Because $\mathcal{S}^*$ is infinite, its value is largely conceptual; that is, it provides a concise definition of inference with substitution. To actually perform inference, however, it is impractical to first generate $\mathcal{S}^*$ and then invoke classical inference procedures. This section offers a more practical alternative by providing a finite axiomatization which is sound and complete. The inference system below can be operated in either a top-down or a bottom-up manner. When operated top-down, it attempts only those substitutions which are relevant to proving its current goal.

Lemma 9.1 *Let β be a predicate with substitution, $\mathcal{S}$ a Horn rulebase with substitution, and DB a database. Then $\mathcal{S}, DB \vdash \beta$ iff β can be inferred from the following inference system:*

Axioms:

1. *For every substitution* $[P/Q]$,

   $P(\overline{x})[P/Q] \leftarrow Q(\overline{x})$

2. *For every substitution* $[P/Q]$ *and every base predicate* $B \neq P$,

   $B(\overline{x})[P/Q] \leftarrow B(\overline{x})$

3. *Every rule in* $\mathcal{S}$.

4. *Every atomic formula in* DB.

Inference Rules:

1. *For any predicate substitution* $[\theta]$,

   $$\frac{\beta \leftarrow \beta_1 ... \beta_n}{\beta[\theta] \leftarrow \beta_1[\theta] ... \beta_n[\theta]}$$

2. *For any term substitution* σ,

   $$\frac{\beta \leftarrow \beta_1 ... \beta_n}{(\beta \leftarrow \beta_1 ... \beta_n)\sigma}$$

3. *Modus Ponens:*

   $$\frac{\begin{array}{l}\beta_1 \; ... \; \beta_n \\ \beta \leftarrow \beta_1 ... \beta_n\end{array}}{\beta}$$

Given a predicate β, the first inference rule effectively makes copies of the rules defining β in order to derive rules defining $\beta[\theta]$. The first two axioms come directly from the definition of $\mathcal{S}^*$. The other axioms and inference rules are familiar from classical logic: the second inference rule makes all instances of a rule available for inference, and the third rule applies rules to atomic facts to infer more atomic facts.

9.4 Basic Properties

This section develops some of the basic formal properties of the rule set $\mathcal{S}^*$ defined in Section 9.3.1. The properties all deal with the derivation of Horn rules from $\mathcal{S}^*$ and are central to the proofs of soundness and completeness given in Section 10. We first prove a compactness result for entailment with substitution. The remaining development is then divided into two subsections, each leading to a central lemma.

To prove the first result, recall that definition 9.3 provides four items that contribute rules to $\mathcal{S}^*$. Using this definition, each rule in $\mathcal{S}^*$ can be generated as follows: First pick a rule from items 1, 3 or 4; then apply item 2 to it some number of times (possibly zero times). It follows that every rule in $\mathcal{S}^*$ originates either as a rule in $\mathcal{S}$ or as a rule described by item 3 or 4. From this observation, we get the following results.

Lemma 9.2 *If $\rho^1 ... \rho^n$ are rules in $\mathcal{S}^*$, then there is a finite subset $\mathcal{S}_o$ of $\mathcal{S}$ of cardinality at most n such that $\rho^1 ... \rho^n$ are all in $\mathcal{S}_o^*$.*

Proof: Each rule ρ^i originates as a single rule ρ_o^i from item 1, 3 or 4 of Definition 9.3. Let $\mathcal{S}_o = \{\rho_o^1, ..., \rho_o^n\} \cap \mathcal{S}$. Then $\mathcal{S}_o$ is a finite subset of $\mathcal{S}$ of cardinality at most n, as required. Furthermore, we claim that each ρ^i is in $\mathcal{S}_o^*$. To see this, consider two cases. If ρ^i originates from item 3 or 4, then $\rho \in \mathcal{R}^*$ for every rulebase $\mathcal{R}$. In particular $\rho^i \in \mathcal{S}_o^*$. On the other hand, if ρ^i originates from item 1, then $\rho_0^i \in \mathcal{S}$ and so $\rho_0^i \in \mathcal{S}_0$. Furthermore $\rho^i \in \mathcal{R}^*$ for any rulebase $\mathcal{R}$ containing ρ_o^i. In particular $\rho^i \in \mathcal{S}_o^*$. $\Box$

Corollary 9.3 *(Compactness) Suppose that $\mathcal{S}^* \vdash_c \rho$ where ρ is a Horn rule with substitution, and $\mathcal{S}$ is a (possibly infinite) rulebase with substitution. Then $\mathcal{S}_o^* \vdash_c \rho$ for some finite subset $\mathcal{S}_o$ of $\mathcal{S}$.*

Proof: Suppose that $\mathcal{S}^* \vdash_c \rho$. Then, by the compactness of classical entailment, there is a finite subset $\mathcal{R}$ of $\mathcal{S}^*$ such that $\mathcal{R} \vdash_c \rho$. By Lemma 9.2, there is a finite subset $\mathcal{S}_o$ of $\mathcal{S}$ such that $\mathcal{R} \subseteq \mathcal{S}_o^*$. Therefore $\mathcal{S}_o^* \vdash_c \mathcal{R}$ and $\mathcal{R} \vdash_c \rho$, and so $\mathcal{S}_o^* \vdash_c \rho$. $\Box$

9.4.1 Properties I.

This section shows that if $\mathcal{S}_1 \vdash_c \mathcal{S}_2$ then $\mathcal{S}_1^* \vdash_c \mathcal{S}_2^*$, where predicates with substitution are treated as ordinary classical predicates. This result is used throughout this chapter and the next and is crucial to the proof of completeness in Section 10.2.

To prove this result, we view a predicate substitution $[\theta]$ as a mapping $f_{[\theta]}$ from predicates with substitution to predicates with substitution. In particular, if β is a predicate with substitution, then $f_{[\theta]}(\beta) = \beta[\theta]$. For instance, if P is an atom, then

$$
\begin{aligned}
&f_{[D/C]}(P) = P[D/C] \\
&f_{[A/B]}(P[D/C]) = P[D/C][A/B] \\
&f_{[A/B]}(P[D/C][A/B]) = P[D/C][A/B][A/B]
\end{aligned}
$$

Note that $f_{[\theta]}$ is a one-to-one mapping. For convenience, we shall often write $\beta[\theta]$ instead of $f_{[\theta]}(\beta)$. This mapping can be extended to rules and rule sets in the natural way:

$$
\begin{aligned}
(\beta \leftarrow \beta_1 ... \beta_n)[\theta] \;&=\; \beta[\theta] \leftarrow \beta_1[\theta] ... \beta_n[\theta] \\
\mathcal{S}[\theta] \;&=\; \{\rho[\theta] \mid \rho \in \mathcal{S}\}
\end{aligned}
$$

where $\mathcal{S}$ is a set of rules with substitution.

When predicates with substitution are treated as ordinary predicates of classical logic, the mapping $f_{[\theta]}$ becomes a renaming of the predicate symbols. Because this renaming is one-to-one, we have the following result:

Lemma 9.4 *$\mathcal{S} \vdash_c \rho$ iff $\mathcal{S}[\theta] \vdash_c \rho[\theta]$ for any rule with substitution ρ and any set $\mathcal{S}$ of rules with substitution.*

Corollary 9.5 *If $\mathcal{S}^* \vdash_c \rho$ then $\mathcal{S}^* \vdash_c \rho[\theta]$, for any rule with substitution ρ and any set $\mathcal{S}$ of Horn rules with substitution.*

Proof: By Definition 9.3, the set $\mathcal{S}^*$ is closed under the renaming $[\theta]$. That is, $\mathcal{S}^*[\theta] \subseteq \mathcal{S}^*$. Hence $\mathcal{S}^* \vdash_c \mathcal{S}^*[\theta]$. Furthermore, since $\mathcal{S}^*$ is a set of rules with substitution, we can apply Lemma 9.4 to it. Keeping these points in mind, suppose that $\mathcal{S}^* \vdash_c \rho$. Then $\mathcal{S}^*[\theta] \vdash_c \rho[\theta]$, by Lemma 9.4, and so $\mathcal{S}^* \vdash_c \rho[\theta]$, since $\mathcal{S}^* \vdash_c \mathcal{S}^*[\theta]$. $\Box$

The main result of this section is an immediate corollary of the following lemma.

Lemma 9.6 *If $\mathcal{S}_1^* \vdash_c \mathcal{S}_2$ then $\mathcal{S}_1^* \vdash_c \mathcal{S}_2^*$ where $\mathcal{S}_1$ and $\mathcal{S}_2$ are sets of Horn rules with substitution.*

Proof: Treating $\mathcal{S}_1^*$ as a classical rulebase, let $cl(\mathcal{S}_1^*)$ be the set of Horn rules classically entailed by $\mathcal{S}_1^*$ (the "closure" of $\mathcal{S}_1^*$). This closure has the following properties:

1. $\mathcal{S}_2 \subseteq cl(\mathcal{S}_1^*)$.

2. If $\beta \leftarrow \beta_1 ... \beta_n$ is in $cl(\mathcal{S}_1^*)$, then $\beta[\theta] \leftarrow \beta_1[\theta] ... \beta_n[\theta]$ is in $cl(\mathcal{S}_1^*)$ for every predicate substitution $[\theta]$.

3. The rule $P(\overline{x})[P/Q] \leftarrow Q(\overline{x})$ is in $cl(\mathcal{S}_1^*)$ for every substitution $[P/Q]$.

4. The rule $B(\overline{x})[P/Q] \leftarrow B(\overline{x})$ is in $cl(\mathcal{S}_1^*)$ for every substitution $[P/Q]$ and every base predicate $B \neq P$.

The first item is by hypothesis, the second by Corollary 9.5, and the last two are from the definition definition of $\mathcal{S}_1^*$. The set $cl(\mathcal{S}_1^*)$ thus satisfies each of the four conditions in the definition of $\mathcal{S}_2^*$. By definition, however, $\mathcal{S}_2^*$ is the smallest set satisfying these conditions. Thus $\mathcal{S}_2^* \subseteq cl(\mathcal{S}_1^*)$. Therefore, each rule in $\mathcal{S}_2^*$ is classically entailed by $\mathcal{S}_1^*$, and so $\mathcal{S}_1^* \vdash_c \mathcal{S}_2^*$. $\Box$

$\mathcal{S}_1 \subseteq \mathcal{S}_1^*$ for any rulebase with substitution $\mathcal{S}_1$. Thus, if $\mathcal{S}_1 \vdash_c \mathcal{S}_2$ then $\mathcal{S}_1^* \vdash_c \mathcal{S}_2$, which gives the following corollary of Lemma 9.6.

Corollary 9.7 *If* $\mathcal{S}_1 \vdash_c \mathcal{S}_2$ *then* $\mathcal{S}_1^* \vdash_c \mathcal{S}_2^*$ *where* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are Horn rulebases with substitution*

It follows immediately that if $\mathcal{S}_1$ and $\mathcal{S}_2$ are classically equivalent, then so are $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$. Hence, rulebases with substitution that are classically equivalent express exactly the same database queries.

Corollary 9.8 *If* $\mathcal{S}_1$ *is classically equivalent to* $\mathcal{S}_2$*, then for any database DB and any predicate with substitution* β*,*

$$\mathcal{S}_1, DB \vdash \beta \quad \textit{iff} \quad \mathcal{S}_2, DB \vdash \beta$$

9.4.2 Properties II.

This section elucidates the relationship between a predicate with substitution $\beta[\theta]$ and the simpler predicate with substitution β. In particular, it is shown that reasoning about $\beta[\theta]$ can be broken into two phases: (i) ignoring $[\theta]$ and reasoning just about β, and (ii) ignoring β and reasoning just about $[\theta]$. The precise statement of this result (Lemma 9.10) is crucial to the proof of soundness given in Section 10.3. For clarity, the development is restricted to the propositional case. The proof of the predicate case requires the introduction of unification and is a straightforward, though tedious generalization.

The main idea behind inference with substitution is to treat the rule set $\mathcal{S}^*$ as a Horn rulebase and to perform classical inference with it. Resolution is a well-known method of classical inference, and it has a particularly simple formulation in the context of Horn rules. In particular, if $A \leftarrow B$ and $B \leftarrow C$ are two Horn rules, then they can be resolved together to derive a new Horn rule $A \leftarrow C$. Horn rules are thus closed under resolution. Furthermore, if R is a Horn rulebase, then except for tautologies such as $A \leftarrow A$, all the Horn rules entailed by R can be derived by resolution, i.e., by resolving the rules in R with one another. This idea is made precise by the following characterization of classical inference, which is the starting point for the rest of this section.

Lemma 9.9 *Suppose that R is a set of Horn rules. Then the Horn rules classically entailed by R are given by the following inference system:*

Axioms: *Every Horn rule $A \leftarrow A_1 ... A_n$ which is a tautology, i.e., for which $A = A_i$ for some i.*

Inference Rules:

For each rule $A \leftarrow A_1 ... A_n$ in R, and for any string of atoms $B_1 ... B_m$,

$$\frac{A_i \leftarrow B_1 ... B_m \quad \text{for } \; 1 \leq i \leq n}{A \leftarrow B_1 ... B_m}$$

Before applying this inference system to the rule set $\mathcal{S}^*$, we need to make a simple but important observation about the structure of $\mathcal{S}^*$. In Definition 9.3, there are four items that contribute rules to $\mathcal{S}^*$. The first item contributes rules whose heads are atomic (since each rule in $\mathcal{S}$ has an atomic head). The third and fourth items contribute rules whose heads contain exactly one substitution. Only the second item contributes rules whose heads have more than one substitution, and these rules have the form $\beta[\theta] \leftarrow \beta_1[\theta]...\beta_n[\theta]$, where $\beta \leftarrow \beta_1...\beta_n$ is also a rule in $\mathcal{S}^*$. This observation is exploited in the proof of the following lemma, which is the main result of this section.

Lemma 9.10 *Let $\mathcal{S}$ be a set of Horn rules with substitution. If $B_1...B_m$ are atoms, then $\mathcal{S}^* \vdash_c \beta[\theta] \leftarrow B_1...B_m$ iff the following two conditions are satisfied for some set of atoms $A_1...A_n$,*

$$\mathcal{S}^* \quad \vdash_c \quad \beta \leftarrow A_1...A_n \tag{9.2}$$

$$\textit{and} \quad \mathcal{S}^* \quad \vdash_c \quad A_j[\theta] \leftarrow B_1...B_m \quad \textit{for each } j \tag{9.3}$$

Proof: The *if* direction follows immediately from Corollary 9.5 and resolution. To prove the *only if* direction, suppose that the rule $\beta[\theta] \leftarrow B_1...B_m$ is classically entailed by $\mathcal{S}^*$. It is therefore derivable by the inference system of Lemma 9.9. The proof is by induction on the length of this derivation.

Basis: If $\beta[\theta] \leftarrow A_1...A_n$ is derivable in 0 steps, then it must be an axiom. Thus $\beta[\theta] = A_j$ for some j. But this is impossible, since each A_j is atomic, whereas $\beta[\theta]$ is not. Hence the base case never occurs and is therefore trivially true.

Induction: Suppose that $\beta[\theta] \leftarrow B_1...B_m$ is derivable from $\mathcal{S}^*$ in $k+1$ steps. We consider two cases, depending on whether or not β is atomic:

Case (i): If β is atomic, then we are done, since

$$\mathcal{S}^* \quad \vdash_c \quad \beta \leftarrow \beta$$

$$\textit{and} \quad \mathcal{S}^* \quad \vdash_c \quad \beta[\theta] \leftarrow B_1...B_m$$

The first line has form (9.2), and the second line has form (9.3).

Case (ii): If β is non-atomic, then $\beta[\theta]$ contains more than one substitution. Therefore, as noted above, every rule in $\mathcal{S}^*$ that infers $\beta[\theta]$ is due to item 2 of Definition

9.3. These rules have the form $\beta[\theta] \leftarrow \beta_1[\theta]...\beta_n[\theta]$ where $\beta \leftarrow \beta_1...\beta_n$ is also a rule in $\mathcal{S}^*$. These are the only rules in $\mathcal{S}^*$ that infer $\beta[\theta]$.

Thus, by Lemma 9.9, since $\beta[\theta] \leftarrow B_1...B_m$ is derivable in $k+1$ steps, there must be a rule $\beta[\theta] \leftarrow \beta_1[\theta]...\beta_n[\theta]$ in $\mathcal{S}^*$ such that the expression $\beta_i[\theta] \leftarrow B_1...B_m$ is derivable in k steps or less, for each i. Thus, by induction hypothesis, there is a set of atoms $A^i_1...A^i_{n_i}$ such that

$$
\begin{array}{lll}
 & \mathcal{S}^* \;\; \vdash_c & \beta_i \leftarrow A^i_1...A^i_{n_i} \\
\textit{and} & \mathcal{S}^* \;\; \vdash_c & A^i_j[\theta] \leftarrow B_1...B_m \quad \textit{for each } j.
\end{array}
$$

The second line has form (9.3). In addition, since $\beta \leftarrow \beta_1...\beta_n$ is in $\mathcal{S}^*$, we can combine it with the first line to get the following, which has form (9.2):

$$\mathcal{S}^* \;\; \vdash_c \;\; \beta \leftarrow A^1_1...A^1_{n_1}\; A^2_1...A^2_{n_2} \;...\; A^m_1...A^m_{n_m}$$

□

9.5 The Power of Substitution

It might seem that predicate substitution is a mere syntactic device, one that increases the convenience of using a deductive database system, but not its power. In this section, we show that this is not true, by proving that predicate substitution increases the power of Datalog (function-free Horn logic). Without substitution, the data-complexity of Datalog is complete for polynomial time. It therefore cannot solve any problem that is EXPTIME-complete. In this section, we show that Datalog with substitution *can* solve EXPTIME-complete problems. This boost in expressive power comes about from recursion through substitution, which allows us to reuse and modify a rulebase infinitely many times.

Our strategy is to use Datalog with substitution to encode the computations of an alternating Turing machine. Alternating machines are non-deterministic in that they can have many possible transitions at any point in a computation. Unlike ordinary non-deterministic machines, however, alternating machines may require that *all* transitions be successful, not just one. To compare alternating machines to other machines, it is useful to view computation as a search through a space of possible machine configurations. Intuitively, an alternating machine searches an *and/or* tree, whereas a non-deterministic machine searches an *or* tree, and a deterministic machine searches a tree having only one branch. Evidence suggests that alternation increases the power of a machine. For instance, alternation gives PSPACE machines the power of EXPTIME machines. That is, APSPACE = EXPTIME [19]. By encoding the computations of APSPACE machines, we can therefore solve EXPTIME-complete problems.

We used alternating machines in Chapter 4 to establish lower bounds on the complexity of hypothetical insertion and deletion. Those techniques cannot be applied directly to predicate substitution however, because substitution is not as "fine grained" as insertion and deletion. In particular, predicate substitution cannot simulate the detailed movements of a control head by inserting and deleting individual tuples. Instead, substitution performs the grosser operation of replacing one entire relation by another. To accommodate this, we adopt a different view of computation. Instead of viewing computation as the reading and writing of one character at a time, we view it as a string transformation in which all characters may change simultaneously. By representing a string as a predicate, string transformation becomes predicate substitution.

For purposes of presentation, we develop this idea in two stages. First, in Section 9.5.1, we introduce the idea of computation as a series of string transformations. We illustrate the idea by encoding the computations of a deterministic PSPACE machine in a Datalog rulebase with substitution. Second, in Section 9.5.2, we consider *alternating* string transformations. We illustrate the idea by encoding the computations of an alternating PSPACE machine in a Datalog rulebase with substitution. For another illustration of the power of predicate substitution, we refer the interested reader to [11], where we use Datalog with substitution to solve the problem of propositional satisfiability.

9.5.1 Solving a PSPACE-complete Problem

In this section, we show that Datalog with substitution can solve a PSPACE-complete problem. This preliminary result already suggests the power of predicate substitution, since without substitution, Datalog cannot solve such problems, unless P = PSPACE. Our strategy is to construct a Datalog rulebase with substitution that encodes the computations of an arbitrary PSPACE machine. This demonstrates that the data complexity of Datalog with substitution is PSPACE-hard.

To state the main result precisely, suppose that M is a PSPACE machine. Without loss of generality, we can assume that M has a single tape. We encode this machine as a database $DB(M)$. Given an input string $\overline{s}$ for the machine, we encode it as another database $DB(\overline{s})$. Finally, we construct a Datalog rulebase with substitution R that simulates the computations of M on input $\overline{s}$. In particular,

$$R,\, DB(M) + DB(\overline{s}) \;\vdash\; ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s} \tag{9.4}$$

where $ACCEPT$ is a 0-ary predicate. The important points are that the rulebase R is independent of the input $\overline{s}$ and that the database $DB(\overline{s})$ can be constructed in polynomial time (polynomial in the size of $\overline{s}$). From these facts, we can conclude that the data complexity of Datalog with substitution is PSPACE-hard. The result follows immediately by letting M be a machine that recognizes a PSPACE-complete language.[6]

The rest of this section describes the construction of $DB(M)$, $DB(\overline{s})$ and R.

Strings and Transformations

To encode the computations of a PSPACE machine, we view computation as a sequence of string transformations, where each string represents a machine configuration. A configuration records the contents of the machine tape and the position and state of the control head. We represent a configuration as a string in a standard way, and we represent the transition function of the machine as a string transformation. We choose this representation because it is easy to implement with Horn rules and predicate substitution.

[6] See [26] for examples of such languages.

There are many ways to encode a machine configuration as a string. We describe a standard encoding that is convenient for the purpose of this section. First, we encode the machine tape as a string in the obvious way, so that each tape cell is represented by one string character. Thus, a tape holding the characters $s_1, s_2, ..., s_n$ is represented by the string $s_1 s_2 ... s_n$. Second, for technical reasons, we add a special character \$ to the front and end of the string, to produce the string $\$ s_1 s_2 ... s_n \$$. Third, for each tape character s_i and each control state q_j we treat the ordered pair $\langle s_i, q_j \rangle$ as a character and add it to the string alphabet. We use these compound characters to encode the state and position of the control head. For example, if the control head is in state q_j and is scanning the tape cell at position i, then in the string $\$ s_1 ... s_n \$$, we replace the character s_i by $\langle s_i, q_j \rangle$. The configuration of the machine is therefore encoded by the following string:

$$\$ s_1 ... s_{i-1} \langle s_i, q_j \rangle s_{i+1} ... s_n \$$$

For a given PSPACE machine and a given input, we can assume that all configurations are strings of the same length. This is because the work space used by the machine is bounded by a polynomial. Thus, there is an integer k such that for inputs of length n, the machine uses no more than n^k tape cells. Adding the two special characters, each configuration has $n^k + 2$ characters. The important points are that the length of a configuration does not change during a computation, and that the initial configuration can be constructed in polynomial time.

Given this encoding of configurations, we represent the transition function of a machine as a string transformation. This transformation maps each character in a string onto another character, where the target character depends not on the entire input string, but only on a few of its characters (reflecting the *local* nature of computation). Formally, the transformation is specified by a function f that takes three characters x, y and z as input, and returns a single character $f(x, y, z)$ as output. x y and z denote any three consecutive characters in a string. For instance, if a string contains the sequence abc, and if $f(a, b, c) = d$, then the string is transformed by replacing this occurrence of

b by d. Formally, the string $a_1...a_n$ is mapped onto the string $b_1...b_n$ where[7]

$$b_1 = a_1 \tag{9.5}$$

$$b_n = a_n \tag{9.6}$$

$$b_j = f(a_{j-1}, a_j, a_{j+1}) \quad \textit{for} \;\; 1 < j < n \tag{9.7}$$

Thus, the end points of the string do not change (leaving the two special characters invariant), and the interior points are transformed by the function f. We use the symbol t to denote the entire string transformation. Thus $t(a_1...a_n) = b_1...b_n$. In general, t takes a string $\overline{a}$ as input, and returns a string $t(\overline{a})$ of the same length as output.

The ternary function f can encode the transition function of any one-tape Turing machine. For example, suppose the transition function includes the following instruction: *If the control head is in state q_1 and is reading the character a, then write the character b, move one square to the left, and go into state q_2.* In this case, any string of the form "$...w\,x\,\langle a, q_1\rangle\,y\,z...$" is mapped onto the string "$...w\,\langle x, q_2\rangle\,b\,y\,z...$". The ternary function f thus satisfies the following equations:

$$\begin{aligned} f(w, x, \langle a, q_1\rangle) &= \langle x, q_2\rangle \\ f(x, \langle a, q_1\rangle, y) &= b \\ f(\langle a, q_1\rangle, y, z) &= y \end{aligned}$$

for any characters w, x, y and z.

Of course, during any step of a computation, most characters in a configuration do not change. Thus $f(x, y, z) = y$ as long as x, y and z represent tape characters, not control states. This is an instance of the frame axiom [36]. In Chapters 4 and 8, we carefully encoded the frame axiom in the rulebase. That is unnecessary here since the frame axiom is encoded in the function f instead.

Finally, note that the function f is finite and depends only on the machine M, not on the machine input $\overline{s}$. We can therefore spend as much time as we like constructing it.

[7]In the rest of this section, the symbol n denotes the length of a machine configuration, not the length of the machine's input.

When f encodes the transition function of a PSPACE machine, we can use the transformation t to simulate the computations of the machine. This transformation maps one configuration of the machine onto the next. That is, if $\overline{a}$ is a string representing the configuration of the machine at some point in a computation, then $t(\overline{a})$ represents the next configuration in the computation, $t^2(\overline{a})$ the next configuration after that, etc. If $\overline{a}$ is an initial configuration for the machine, then we can simulate the entire computation by generating one configuration after another. In this case, the machine accepts its input iff some configuration $t^i(\overline{a})$ is generated in which the finite control is in an accepting state.

Data and Rules

In this section, we construct a Datalog rulebase with substitution that implements the string transformations described above. The rulebase has two parts. First, given a string $\overline{a}$ encoded in the database, a set of Horn rules derives the transformed string $t(\overline{a})$. Second, a recursive rule with substitution applies this transformation over-and-over again to generate the strings $t^i(\overline{a})$. The atom $ACCEPT$ is inferred iff one of these strings encodes an accepting machine state. In this way, the rulebase satisfies condition (9.4), as required. It is worth pointing out that the entire rulebase consists of seven rules: six non-recursive Horn rules and one recursive rule with substitution. This simple structure is enough to simulate any PSPACE machine and achieve PSPACE-hardness.

The PSPACE machine itself is encoded in the database. To encode the finite control, we store the ternary function f and the set of accepting states. We thus introduce a 4-ary predicate F and add the formula $F(a, b, c, d)$ to the database iff $f(a, b, c) = d$. To store the accepting states, we introduce a unary predicate $ACCEPTING$. For each compound character $\langle c, q\rangle$, we add the formula $ACCEPTING(\langle c, q\rangle)$ to the database iff q is an accepting state. (The appearance of any of these compound characters in a machine configuration tells us that the machine has accepted its input.) These formulas make up the database $DB(M)$ in statement (9.4).

To encode the machine's initial configuration, we need a counter to represent string positions. Thus, as in Section 4.3, we introduce three base predicates, $FIRST$, $NEXT$

and $LAST$. For strings of length n, we add the following formulas to the database:

$$FIRST(1),\quad NEXT(1,2),\quad NEXT(2,3)\ \ldots\ NEXT(n-1,n),\quad LAST(n).$$

Given this counter, we introduce a binary predicate S to represent the characters in a string. The formula $S(c,j)$ means that character c appears at position j in the string. If the initial configuration of the machine is given by the string $\overline{a} = a_1 \ldots a_n$, then we add the following formulas to the database:

$$S(a_1,1),\quad S(a_2,2),\quad S(a_3,3)\ \ldots\ S(a_n,n).$$

These formulas make up the database $DB(\overline{s})$ in statement (9.4). (The machine input $\overline{s}$ is encoded in the initial configuration $\overline{a}$.)

The next task is to construct the rulebase R. This rulebase has two parts: (i) a set of Horn rules that implement the transformation t, and (ii) additional rules that determine whether the the machine M accepts its input.

Part (i): To implement the string transformation t, we add the Horn rules below to the rulebase R. These three rules correspond to conditions (9.5)–(9.7), respectively. Given a string $\overline{a}$ stored in the predicate S, these rules derive the transformed string $t(\overline{a})$ and store it in the predicate S_0.

$$\begin{aligned}
S_0(x,j) &\leftarrow FIRST(j),\ S(x,j). \\
S_0(x,j) &\leftarrow LAST(j),\ S(x,j). \\
S_0(x_4,j) &\leftarrow F(x_1,x_2,x_3,x_4),\ NEXT(i,j),\ NEXT(j,k), \\
&\qquad S(x_1,i),\ S(x_2,j),\ S(x_3,k).
\end{aligned}$$

where i, j, k and all the x's are variables. The first two rules ensure that the end points of the string do not change. The third rule implements the function f that transforms the interior points of the string.

The Horn rules defining S_0 represent one application of the transformation t. To apply the transformation more than once, we apply the predicate substitution $[S/S_0]$ to the rulebase. This substitution reuses and modifies the rules to produce another set of rules representing another application of t. Specifically, the substitution defines a new predicate $S_0(x,j)\,[S/S_0]$ defined by the following three rules:

$$
\begin{aligned}
S_0(x,j)\,[S/S_0] &\leftarrow FIRST(j),\ S_0(x,j). \\
S_0(x,j)\,[S/S_0] &\leftarrow LAST(j),\ S_0(x,j). \\
S_0(x_4,j)\,[S/S_0] &\leftarrow F(x_1,x_2,x_3,x_4),\ NEXT(i,j),\ NEXT(j,k), \\
& \qquad S_0(x_1,i),\ S_0(x_2,j),\ S_0(x_3,k).
\end{aligned}
$$

The definition of the predicate $S_0(x,j)\,[S/S_0]$ is the same as the definition of $S_0(x,j)$ except that S has been replaced by S_0 in the body of every rule. These rules are not supplied by the user, but are generated automatically by the copy-and-substitute mechanism described in Section 9.3. Like the rules defining S_0, these new rules represent the transformation t, but they operate on different predicates. Given a string $\overline{b}$ stored in the predicate $S_0(x,j)$, these rules derive the transformed string $t(\overline{b})$ and store it in the predicate $S_0(x,j)\,[S/S_0]$.

By applying the substitution $[S/S_0]$ many times, we can generate many more sets of rules, each representing the transformation t. In general, by applying the substitution $i+1$ times, we generate the following three rules:

$$
\begin{aligned}
S_0(x,j)\,[S/S_0]^{i+1} &\leftarrow FIRST(j),\ S_0(x,j)\,[S/S_0]^{i}. \\
S_0(x,j)\,[S/S_0]^{i+1} &\leftarrow LAST(j),\ S_0(x,j)\,[S/S_0]^{i}. \\
S_0(x_4,j)\,[S/S_0]^{i+1} &\leftarrow F(x_1,x_2,x_3,x_4),\ NEXT(i,j),\ NEXT(j,k), \\
& \qquad S_0(x_1,i)\,[S/S_0]^{i},\ S_0(x_2,j)\,[S/S_0]^{i},\ S_0(x_3,k)\,[S/S_0]^{i}.
\end{aligned}
$$

Given a string stored in the predicate $S_0(x,j)\,[S/S_0]^{i}$, these rules derive the transformed string and store it in the predicate $S_0(x,j)\,[S/S_0]^{i+1}$.

Each trio of rules generated in this way represents an application of the transformation t. Equally important, the output predicate of one trio is the input predicate of the next. The rules can therefore be cascaded to produce multiple applications of t. Thus, if the predicate $S(x,j)$ represents the string $\overline{a}$, then the following predicates with substitution represent the strings $t(\overline{a})$, $t^2(\overline{a})$, $t^3(\overline{a})$, and $t^4(\overline{a})$, respectively:

$$
\begin{aligned}
& S_0(x,j) \\
& S_0(x,j)\,[S/S_0] \\
& S_0(x,j)\,[S/S_0]\,[S/S_0] \\
& S_0(x,j)\,[S/S_0]\,[S/S_0]\,[S/S_0]
\end{aligned}
$$

In general, the predicate $S_0(x,j)\,[S/S_0]^i$ represents the string $t^{i+1}(\overline{a})$.

This technique generates infinitely many predicates, each with a unique name. For a given database, however, these predicates will not all have unique extensions. This is because there are only finitely many strings of length n. In fact, there are k^n different strings for a string alphabet of size k. Predicate substitution thus allows us to represent exponentially many different machine configurations (exponential in the size of the database), which is enough to encode the computations of any PSPACE machine.

Part (ii): Besides generating configurations, the rulebase must determine whether the machine accepts its input. To this end, we introduce the idea of an accepting configuration. A configuration $\overline{a}$ is accepting iff it contains an accepting control-state or the next configuration $t(\overline{a})$ is accepting. A machine therefore accepts its input iff the initial configuration is accepting. To test for acceptance, we add the following Horn rule to the rulebase R:

$$ACCEPT_0 \;\leftarrow\; S(x,j),\; ACCEPTING(x).$$

where x and j are variables. This rule tests for a character x that represents an accepting control state at some location j in the configuration.

Because this rule is written in terms of the predicate S, it applies only to the initial configuration $\overline{a}$ stored in the database. However, we can apply this rule to other configurations by using predicate substitution. For example, the following rules determine whether the configurations $\overline{a}$, $t(\overline{a})$, $t^2(\overline{a})$ and $t^3(\overline{a})$ are accepting, respectively:

$$\begin{aligned}
ACCEPT &\leftarrow ACCEPT_0 \\
ACCEPT &\leftarrow ACCEPT_0\,[S/S_0] \\
ACCEPT &\leftarrow ACCEPT_0\,[S/S_0]\,[S/S_0] \\
ACCEPT &\leftarrow ACCEPT_0\,[S/S_0]\,[S/S_0]\,[S/S_0]
\end{aligned} \tag{9.8}$$

In general, the following rule determines whether the configuration $t^i(\overline{a})$ is accepting:

$$ACCEPT \;\leftarrow\; ACCEPT_0\,[S/S_0]^i$$

By using recursion through substitution, we can generate all these rules automatically. The following rules, which we add to the rulebase R, do precisely this:

$$\begin{aligned} ACCEPT &\leftarrow ACCEPT\,[S/S_0] \\ ACCEPT &\leftarrow ACCEPT_0 \end{aligned}$$

These rules infer the atom $ACCEPT$ iff some configuration $t^i(\overline{a})$ is accepting, that is, iff the machine accepts its input. Intuitively, this recursion can be unfolded to produce an infinite number of non-recursive rules. For example, unfolding the first four levels of recursion produces the four rules (9.8) above. After unfolding, each of these rules is treated as a Horn rule in classical logic, thus effectively producing an infinite Datalog rulebase.

This completes the construction of the rulebase R. Together with the databases $DB(M)$ and $DB(\overline{s})$, this rulebase satisfies condition (9.4). Observe that since this condition represents a PSPACE-complete problem, no *finite* Datalog rulebase without substitution can satisfy this condition, unless P = PSPACE.

9.5.2 Solving an EXPTIME-complete Problem

The rulebase constructed in Section 9.5.1 is relatively simple: It has only one recursive rule, which itself has only one premise. Its power to specify a PSPACE-complete problem comes from its use of recursion through substitution. In this section, we use recursion through substitution in more complex ways to specify more complex problems. In particular, we construct a rulebase in which predicate substitution appears in more than one premise and in more than one rule. During inference, this rulebase simulates the computations of an arbitrary *alternating* PSPACE machine. Since APSPACE = EXPTIME, the data complexity of Datalog with substitution is EXPTIME-hard.

To state the main result of this section precisely, suppose that M is an alternating PSPACE machine. Without loss of generality, we can assume that M has a single tape. We encode this machine as a database $DB(M)$. Given an input string $\overline{s}$ for the machine, we encode it as another database $DB(\overline{s})$. Finally, we construct a Datalog rulebase with substitution R that simulates the computations of M on input $\overline{s}$. That is,

$$R, DB(M) + DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s} \tag{9.9}$$

where $ACCEPT$ is a 0-ary predicate. The important points are that the rulebase R is independent of the input $\overline{s}$ and that the database $DB(\overline{s})$ can be constructed in polynomial time (polynomial in the size of $\overline{s}$). From these facts, we can conclude that the data complexity of Datalog with substitution is EXPTIME-hard. This result follows immediately by letting M be a machine that recognizes a EXPTIME-complete language.[8]

The rest of this section describes the construction of $DB(M)$, $DB(\overline{s})$ and R.

Strings and Transformations

To encode the computations of alternating PSPACE machines, we need to enrich the view of computation developed in Section 9.5.1. In that section, we viewed computation as a *sequence* of machine configurations, so that each configuration had exactly one successor configuration. In this section, we view computation as a *tree* of configurations, so that each configuration can have many successors. Instead of a transition *function*, the finite control of a machine thus has a transition *relation*. As in Section 9.5.1, we encode each machine configuration as a string. Because the machines still use only polynomial space, we can still assume that for a given machine and a given input, all configurations are strings of the same length.

In Section 9.5.1, we represented the state transitions of a machine as a ternary function f. This approach is not possible here because machine transitions are now specified by a relation, not a function. However, without loss of generality, we can assume that for each tape character, a state of the finite control has exactly two successor states (if any). We can therefore represent the transition relation of a machine as two functions f_1 and f_2.

As before, these functions specify string transformations. Formally, a string $a_1 ... a_n$

[8] See [19] for examples of such languages.

has two successors $b_1^1...b_n^1$ and $b_1^2...b_n^2$ where

$$b_1^i = a_1 \tag{9.10}$$

$$b_n^i = a_n \tag{9.11}$$

$$b_j^i = f_i(a_{j-1}, a_j, a_{j+1}) \quad \textit{for} \;\; 1 < j < n \tag{9.12}$$

Thus, as before, the end points of the string do not change, and the interior points are transformed by the functions f_1 and f_2. We use the symbols t_1 and t_2 to denote these two string transformations. Thus $t_i(a_1...a_n) = b_1^i...b_n^i$. In general, t_i takes a string $\overline{a}$ as input, and returns a string $t_i(\overline{a})$ of the same length as output. If $\overline{a}$ encodes the initial configuration of an alternating PSPACE machine, then we can apply t_1 and t_2 recursively to generate a binary tree of configurations rooted at $\overline{a}$ representing the computation tree of the machine.

Data and Rules

We implement the strings and transformations described above as a rulebase with substitution. As before, we store the finite control and the initial configuration of the machine in the database, and we implement the transformations t_1 and t_2 as a set of Horn rules. Then, starting from the initial configuration, we use predicate substitution to apply the transformations over-and-over again. It is worth noting that the entire rulebase consists of twelve rules: nine non-recursive Horn rules and three mutually-recursive rules with substitution. This structure is enough to simulate any APSPACE machine and achieve EXPTIME-hardness.

Besides specifying the computation tree of the machine, the rulebase also determines whether the tree represents an accepting computation. For alternating machines, acceptance is conveniently defined in terms of accepting configurations. Whether a configuration is accepting depends on its successor configurations and on the state of the control head. An alternating machine has three kinds of states: *accepting*, *universal* and *existential*. Each configuration has exactly one control state q. The configuration is accepting iff

- q is an accepting state, or

- q is a universal state and all successor configurations are accepting, or
- q is an existential state and at least one successor configurations is accepting.

An alternating machine accepts its input iff its initial configuration is accepting.

The machine itself is encoded in the database. Storing the finite control is slightly more complicated than in Section 9.5.1 since we now have two transition functions and three kinds of control states. To store the control states, we introduce three unary predicates, $ACCEPTING$, $UNIVERSAL$ and $EXISTENTIAL$. For each compound character $\langle c, q\rangle$, we add the formula $EXISTENTIAL(\langle c, q\rangle)$ to the database iff q is an existential state. Likewise for $ACCEPTING$ and $UNIVERSAL$. These formulas make up the database $DB(M)$ in statement (9.9).

We store the initial configuration of the machine exactly as in Section 9.5.1. Thus, to store the configuration $\overline{a} = a_1 ... a_n$, we add the following formulas to the database:

$$FIRST(1),\quad NEXT(1,2),\quad NEXT(2,3)\ ...\ NEXT(n-1,n),\quad LAST(n),$$
$$S(a_1,1),\quad S(a_2,2),\quad S(a_3,3)\ ...\ S(a_n,n).$$

These formulas make up the database $DB(\overline{s})$ in statement (9.9). (The machine input $\overline{s}$ is encoded in the initial configuration $\overline{a}$.)

The next task is to construct the rulebase R. As in Section 9.5.1, this rulebase has two parts: (i) a set of Horn rules that implement the string transformations t_1 and t_2, and (ii) additional rules that determine whether the the machine accepts its input.

Part (i): To implement the string transformation t_i, we add the Horn rules below to the rulebase R. These three rules correspond to the three conditions (9.10)–(9.12), respectively. Given a string $\overline{a}$ stored in the predicate S, these rules derive the transformed string $t_i(\overline{a})$ and store it in the predicate S_i.

$$\begin{aligned}
S_i(x,j) &\leftarrow FIRST(j),\ S(x,j).\\
S_i(x,j) &\leftarrow LAST(j),\ S(x,j).\\
S_i(x_4,j) &\leftarrow F_i(x_1,x_2,x_3,x_4),\ NEXT(i,j),\ NEXT(j,k),\\
&\qquad S(x_1,i),\ S(x_2,j),\ S(x_3,k).
\end{aligned}$$

where i, j, k and all the x's are variables. The first two rules ensure that the end points of the string do not change. The third rule implements the function f_i that transforms the interior points of the string.

The three Horn rules defining S_i represent a single application of the transformation t_i to the string S encoded in the database. As in Section 9.5.1, we use predicate substitution to apply the transformations over-and-over again. Now, however, we have two substitutions to apply, $[S/S_1]$ and $[S/S_2]$, and the order in which we apply them determines how the transformations t_1 and t_2 are composed. For example, if the predicate $S(x, j)$ represents the string $\overline{a}$, then the following predicates with substitution represent the strings $t_1(\overline{a})$, $t_2t_1(\overline{a})$, $t_1t_2t_1(\overline{a})$, $t_1t_1t_2t_1(\overline{a})$ and $t_2t_1t_1t_2t_1(\overline{a})$, respectively:

$$
\begin{aligned}
& S_1(x, j) \\
& S_1(x, j)\,[S/S_2] \\
& S_1(x, j)\,[S/S_2]\,[S/S_1] \\
& S_1(x, j)\,[S/S_2]\,[S/S_1]\,[S/S_1] \\
& S_1(x, j)\,[S/S_2]\,[S/S_1]\,[S/S_1]\,[S/S_2]
\end{aligned}
$$

Part (ii): Besides generating configurations, the rulebase must determine whether the machine accepts its input. We therefore construct rules that determine whether a configuration is accepting.

The first step is to determine what kind of control state a configuration has. The following Horn rules accomplish this:

$$
\begin{aligned}
& ACCEPT \leftarrow ACCEPTING(x),\ S(x, j). \\
& UNIV \leftarrow UNIVERSAL(x),\ S(x, j). \\
& EXIST \leftarrow EXISTENTIAL(x),\ S(x, j).
\end{aligned}
$$

where x and j are variables. The predicate $EXIST$ is inferred iff the configuration represented by S contains an existential control state. Likewise for the predicates $UNIV$ and $ACCEPT$.

If the configuration has an accepting control state, then we are done. If not, then we can examine its successor configurations with predicate substitution. For example, the predicates $ACCEPT[S/S_1]$ and $ACCEPT[S/S_2]$ ask whether the two successor configurations are accepting. By using recursion through substitution, we can recursively examine all successor configurations. The following rules, which we add to the rulebase R, do exactly this:

$$\begin{aligned}
ACCEPT &\leftarrow UNIV,\ ACCEPT[S/S_1],\ ACCEPT[S/S_2] \\
ACCEPT &\leftarrow EXIST,\ ACCEPT[S/S_1] \\
ACCEPT &\leftarrow EXIST,\ ACCEPT[S/S_2]
\end{aligned}$$

The first rule says that a universal configuration is accepting if both its successors is accepting. The second and third rules say that an existential configuration is accepting if at least one of its successors is accepting.

Applied to the database $DB(M) + DB(\overline{s})$ constructed above, these rules infer the atom $ACCEPT$ iff the initial configuration is accepting, that is, iff machine M accepts input $\overline{s}$. This rulebase therefore satisfies condition (9.9). Observe that since this condition represents an EXPTIME-complete problem, no finite Datalog rulebase without substitution can satisfy this condition.

Chapter 10

The Semantics of Substitution

Chapter 9 developed the proof theory for a logic that reuses and modifies rulebases. This chapter[1] develops the model theory. In this theory, rulebases *without* substitution are the models for rulebases *with* substitution. This semantics corresponds to the intuition that rules with substitution are written by a user who has rules without substitution "in mind." This intuition is similar to the view taken of ordinary Datalog rules, in which the user has an "intended model" in mind (the least model of the rules), which he could have written if he had enough space and enough time. Like a Datalog rulebase, a rulebase with substitution may have many models, yet, as we show, it has exactly one minimal model. In this way, just as Datalog provides a concise notation for describing large *databases*, Datalog with substitution provides a concise notation for describing large *rulebases*. This conciseness is achieved by reusing and modifying rule sets, possibly many times.

As in the theory of Horn logic [48, 5], we complement the model-theoretic semantics with a fixpoint semantics. We define a monotonic "T-operator" which can be applied to a rulebase in a bottom-up fashion, generating the minimal model (or least fixpoint) piece-by-piece. In Horn logic, the T-operator takes a database as input and returns another database as output. In the logic of predicate substitution, the T-operator takes a Horn rulebase as input and returns another Horn rulebase as output.

Starting with the empty rulebase, the T-operator is applied over-and-over again until the minimal model is generated. Each application of the operator expands rules with substitution into equivalent Horn rules. If there is recursion through substitution, it may take infinitely many applications to generate all the Horn rules in the minimal

[1] This chapter elaborates on material that originally appeared in [11].

model. Because the operator is continuous, however, any particular Horn rule will be generated in a finite number of applications. In this way, the Horn rules represented by a rulebase with substitution can be materialized. Section 10.4.3 provides several examples of the unfolding of rules with substitution into rules without substitution.

By combining recursion and substitution, it is possible to reuse and modify rule sets infinitely many times. Using this idea, we show that a *finite* set of rules with substitution can specify an *infinite* set of rules without substitution. Examples 10.11 and 10.12 at the end of this chapter show two ways to do this.

Sometimes, the least model is not just infinite, but "strongly infinitely" in that it is not equivalent to any finite Horn rulebase. This kind of strong infiniteness arises because predicate substitution increases the power of Datalog from P to EXPTIME. In particular, when a rulebase with substitution solves an EXPTIME-complete problem, as in Section 9.5, it is impossible to represent it as a *finite* Datalog rulebase without substitution. The semantics developed in this chapter thus provides a new explanation for the data complexity of predicate substitution: When a Datalog rulebase *with* substitution is EXPTIME-complete, it represents an infinite Datalog rulebase *without* substitution, one that is not equivalent to a finite Datalog rulebase.

10.1 Models and Entailment

This section develops the model theory for predicate substitution. In this theory, a model is a Horn rulebase without substitution. Such models support the idea that a rulebase *with* substitution is a short-hand notation for a (possibly larger) rulebase *without* substitution, which the user might have written had he had the time, the space, and the inclination. Thus,

Definition 10.1 *An interpretation is a set R of Horn rules without substitution.*

The rest of this section defines what it means for an interpretation R to be a model of a rule with substitution. That is, this section gives a precise meaning to the expression $R \models \alpha \leftarrow \beta_1 ... \beta_n$, where each β_i is a predicate with substitution.

10.1.1 A Simple Case

Before considering the general case, we focus on a simple rule: $A(x) \leftarrow B(x)[P/Q]$. We develop a precise definition for the models of this rule, giving several examples. The development is done in stages, starting with a crude definition of model, and slowly refining it. Each refinement illustrates an important aspect of the model theory.

Informally, the rule $A(x) \leftarrow B(x)[P/Q]$ means that the definition of predicate A is the same as that of predicate B with P replaced by Q. For instance, suppose that B is defined by the following two rules:

$$B(x) \leftarrow E(x), P(x).$$
$$B(x) \leftarrow F(x,y), P(x), P(y).$$

Then A is defined by the following two rules:

$$A(x) \leftarrow E(x), Q(x).$$
$$A(x) \leftarrow F(x,y), Q(x), Q(y).$$

If R is the Horn rulebase consisting of these four rules, then we say that R is a *model* of the rule $A(x) \leftarrow B(x)[P/Q]$, and we write $R \models A(x) \leftarrow B(x)[P/Q]$. The underlying principle is that in R, the Horn rules defining B give rise to Horn rules defining A. If more Horn rules defining B are added to R, then more Horn rules defining A must also be added; otherwise, R will not remain a model of $A(x) \leftarrow B(x)[P/Q]$.

In this example, the definition of A is *identical* to that of B with P replaced by Q. Strict identity is not required however. In general, the rule $A(x) \leftarrow B(x)[P/Q]$ will be part of a larger rulebase $\mathcal{S}$, and other rules in $\mathcal{S}$ will also contribute to the definition of A. When these rules are taken into account, there may be many other Horn rules in the definition of A. In the next example, the predicates B and C both contribute rules to the definition of A.

Example 10.1 Suppose that R is the following Horn rulebase, where D, E and F are base predicates.

$$B(x) \leftarrow E(x), P(x).$$
$$B(x) \leftarrow F(x,y), P(x), P(y).$$
$$C(x) \leftarrow G(x,y), P(y).$$

$$A(x) \leftarrow E(x), Q(x).$$
$$A(x) \leftarrow F(x,y), Q(x), Q(y).$$
$$A(x) \leftarrow G(x,y), Q(y).$$

Then, R is a model of $A(x) \leftarrow B(x)[P/Q]$ and of $A(x) \leftarrow C(x)[P/Q]$. □

These examples illustrate what it means for a Horn rulebase R to be a model of the rule $A(x) \leftarrow B(x)[P/Q]$: For each rule in R whose head is $B(x)$, there must be a corresponding rule in R whose head is $A(x)$. Furthermore, the bodies of these two rules must be identical except that wherever the former contains a P, the latter contains a Q.

The process of changing P's to Q's can be formalized by using the rulebase R^*, defined in Section 9.3.1. R^* defines the predicate $B(x)[P/Q]$ to be the same as $B(x)$ with P replaced by Q. Unlike the rulebases considered in Section 9.3.1, however, the rulebase R considered here is a simple Horn rulebase. R does not define predicates in terms of substitution, and it does not involve recursion through substitution. The definition of R^* is therefore straightforward. Using R^*, we can test whether R is a model of $A(x) \leftarrow B(x)[P/Q]$. For example,

if $B(x) \leftarrow E(x), P(x)$ is a rule in R,
then $B(x)[P/Q] \leftarrow E(x), Q(x)$ is in a rule in R^*,
therefore $A(x) \leftarrow E(x), Q(x)$ should be a rule in R.

More generally, for any string of atomic formulas $\alpha_1 ... \alpha_n$,

if $B(x)[P/Q] \leftarrow \alpha_1 ... \alpha_n$ is a rule in R^*,
then $A(x) \leftarrow \alpha_1 ... \alpha_n$ should be a rule in R.

The first line in this statement extracts a rule that defines $B(x)$ and converts P to Q everywhere in the rule body. The second line then uses the new body to construct a rule for $A(x)$. This idea forms the basis for a formal definition of Horn models.

The examples given above are relatively simple in that the predicates A and B are defined directly in terms of base predicates. For more complex models, the rule $B(x)[P/Q] \leftarrow \alpha_1 ... \alpha_n$ may not be in R^* per se, but may be implied by it. Similarly, the rule $A(x) \leftarrow \alpha_1 ... \alpha_n$ may be implied by R without actually being in it. We therefore pose the following definition of Horn models:

A Horn rulebase R is a model of $A(x) \leftarrow B(x)[P/Q]$ iff for any string of atomic formulas γ,

$$\textit{if} \quad R^* \models_c B(x)[P/Q] \leftarrow \gamma \quad \textit{then} \quad R \models_c A(x) \leftarrow \gamma \tag{10.1}$$

Example 10.2 Suppose that R is the following Horn rulebase, where F is a base predicate:

$$B(x) \leftarrow P(x), C(x).$$
$$C(x) \leftarrow F(x,y), P(y).$$
$$A(x) \leftarrow Q(x), F(x,y), Q(y).$$

Then R is a model of $A(x) \leftarrow B(x)[P/Q]$. To see this, note that $B(x)[P/Q]$ and $C(x)[P/Q]$ are defined in R^* as follows:

$$B(x)[P/Q] \leftarrow Q(x), C(x)[P/Q]$$
$$C(x)[P/Q] \leftarrow F(x,y), Q(y).$$

Thus $B(x)[P/Q] \leftarrow Q(x), F(x,y), Q(y)$ is the one and only expansion of these rules. □

Example 10.3 Suppose that R is the following Horn rulebase, where F is a base predicate:

$$B(x) \leftarrow P(x), C(y).$$
$$C(x) \leftarrow F(x,y), P(y).$$
$$A(x) \leftarrow Q(x), C'(x).$$
$$C'(x) \leftarrow F(x,y), Q(y).$$

Then $R \models A(x) \leftarrow B(x)[P/Q]$. To see this, note that

$$\begin{aligned} R^* &\models_c B(x)[P/Q] \leftarrow Q(x), F(x,y), Q(y). \\ \textit{and} \quad R &\models_c A(x) \leftarrow Q(x), F(x,y), Q(y). \end{aligned}$$

□

Example 10.3 is similar to Example 10.2 except that a new predicate symbol C' has been introduced into the model. The use of C' is somewhat arbitrary: Any other predicate symbol could be used as long as it is a new symbol. This introduction of new predicate symbols is the model-theoretic counterpart of the predicate with substitution $C(x)[P/Q]$ used in the proof theory. In the proof theory, predicates with substitution are treated as ordinary predicates of classical logic. The tag $[P/Q]$ attached to $C(x)$ simply ensures that $C(x)[P/Q]$ is a new predicate which is not used for any purpose other that reusing and modifying $C(x)$. Models such as the one in Example 10.3 capture the idea that $C(x)[P/Q]$ represents any new predicate symbol.

In addition, unlike the proof theory, a model need not introduce any new predicates at all. Example 10.2, for instance, recognizes that the predicate $C'(x)$ is just a place holder and can be done away with.

Statement (10.1) captures the idea that a Horn model may contain intermediate predicates. It is still an incomplete definition, however, because it ignores unification. Intuitively, the rule $A(x) \leftarrow B(x)[P/Q]$ means that the definition of A is the same as that of B with P replaced by Q. We would expect, therefore, that the following Horn rulebase is a model of this rule.

$$\begin{aligned} B(fx) &\leftarrow D(x,y), P(y). \\ A(fx) &\leftarrow D(x,y), Q(y). \end{aligned} \tag{10.2}$$

Where f is a function symbol, and fx is an abbreviation for $f(x)$. We would also expect that the following Horn rulebase is not a model of the rule:

$$B(fx) \leftarrow D(x,y), P(y). \tag{10.3}$$

Statement (10.1) does not treat these cases properly, however, since it refers only to rules whose heads are $A(x)$ and $B(x)$, not $A(fx)$ or $B(fx)$.

If we are to reuse and modify the definition of B, then we must use *all* the rules defining B, not just those that define $B(x)$ per se. Statement (10.1) must therefore be refined to include rules that define $B(fx)$, $B(gx)$, $B(a)$, and in general, $B(x)\sigma$ for any term substitution σ. The following statement does exactly this:

> *A Horn rulebase R is a model of $A(x) \leftarrow B(x)[P/Q]$ iff for any string of atomic formulas γ, and any term substitution σ,*
>
> $$\textit{if} \quad R^* \models_c B(x)[P/Q]\sigma \leftarrow \gamma \quad \textit{then} \quad R \models_c A(x)\sigma \leftarrow \gamma \qquad (10.4)$$

Rulebase (10.2) above satisfies this definition, whereas rulebase (10.3) does not.

10.1.2 The General Case

So far, the only rule with substitution that we have considered is the relatively simple rule $A(x) \leftarrow B(x)[P/Q]$. The meaning of this rule is given by statement (10.4). But what about more complex rules, such as rules with multiple premises, and recursive rules? Intuitively, statement (10.4) says that the premise of a rule with substitution should be expanded out into a string of atomic formulas γ. To handle rules with multiple premises, we simply require that each premise be expanded in this way. This leads to the following definition of satisfaction, which is a straightforward generalization of statement (10.4). As we shall see, this definition also gives a natural interpretation of recursive rules.

Definition 10.2 *(Satisfaction) Let R be a set of Horn rules, let α be an atomic formula, and let $\beta_1 ... \beta_n$ be predicates with substitution. Then, $R \models \alpha \leftarrow \beta_1 ... \beta_n$ iff for all strings of atomic formulas γ, and all term substitutions σ,*

$$R \models_c \alpha\sigma \leftarrow \gamma \quad \textit{if} \quad R^* \models_c \beta_i\sigma \leftarrow \gamma \quad \textit{for each } i.$$

If $\mathcal{S}$ is a Horn rulebase with substitution and R is a Horn rulebase without substitution, then we say that R satisfies $\mathcal{S}$ iff R satisfies each rule in $\mathcal{S}$. In this case, we also say that R is a *model* of $\mathcal{S}$.

Example 10.4 Suppose that R is the following Horn rulebase, where F is base a predicate:

$$B(x) \leftarrow P(x).$$
$$C(x) \leftarrow F(x,y), M(y).$$
$$A(x) \leftarrow Q(x), F(x,y), N(y).$$

Then R is a model of the rule $A(x) \leftarrow B(x)[P/Q], C(x)[M/N]$. To see this, note that $B(x)[P/Q]$ and $C(x)[M/N]$ are defined as follows:

$$B(x)[P/Q] \leftarrow Q(x)$$
$$C(x)[M/N] \leftarrow F(x,y), N(y).$$

□

We are now in a position to consider recursive rules with substitution. For instance, what are the models of the rule $A(x) \leftarrow A(x)[P/Q]$? This question can be answered much as it was in the non-recursive case: Given a set of rules for A defined in terms of P, the rule $A(x) \leftarrow A(x)[P/Q]$ specifies additional rules for A, defined in terms of Q. For instance, suppose that R contains the following two rules:

$$A(x) \leftarrow E(x), P(x).$$
$$A(x) \leftarrow F(x,y), P(x), P(y).$$

Then, if R is a model of $A(x) \leftarrow A(x)[P/Q]$, then R should also contain these rules:

$$A(x) \leftarrow E(x), Q(x).$$
$$A(x) \leftarrow F(x,y), Q(x), Q(y).$$

In fact, if R consists of these four rules, then R satisfies Definition 10.2. Note that R itself is non-recursive. Thus a recursive rule can have non-recursive models.

If a rule is added to R defining A in terms of P, then another rule must also be added, defining A in terms of Q; otherwise, R will not remain a model of $A(x) \leftarrow A(x)[P/Q]$.

Example 10.5 Suppose that R is the following Horn rulebase, where E, F and G are base predicates:

$$A(x) \leftarrow E(x), P(x).$$
$$A(x) \leftarrow F(x,y), P(x), P(y).$$
$$A(x) \leftarrow G(x,y), P(y).$$
$$A(x) \leftarrow E(x), Q(x).$$
$$A(x) \leftarrow F(x,y), Q(x), Q(y).$$
$$A(x) \leftarrow G(x,y), Q(y).$$

Then $R \models A(x) \leftarrow A(x)[P/Q]$. □

Before leaving this section, it is worthwhile making a few observations about the notion of satisfaction that we have just developed. The first observation is that for Horn rules, satisfaction reduces to classical entailment. That is, we have the following basic result.

Lemma 10.1 *If R is a Horn rulebase, and $\alpha \leftarrow \alpha_1 ... \alpha_n$ is a Horn rule, then*

$$R \models \alpha \leftarrow \alpha_1 ... \alpha_n \quad \textit{iff} \quad R \models_c \alpha \leftarrow \alpha_1 ... \alpha_n$$

Proof: The *only if* direction is immediate since $R \subseteq R^*$. In the other direction, suppose that $R^* \models_c \alpha \leftarrow \gamma$. Because R is Horn, every rule in R^* with an atomic conclusion has only atomic premises. Thus, the only rules in R^* that can be used to derive $\alpha \leftarrow \gamma$ are Horn rules. But the set of Horn rules in R^* is just R. Hence $R \models_c \alpha \leftarrow \gamma$. □

It is also worth noting that satisfaction is not monotonic in R. As the next example shows, if $R_1 \subseteq R_2$, then it is not necessarily the case that if $R_1 \models \alpha \leftarrow \beta$ then $R_2 \models \alpha \leftarrow \beta$. This should not be surprising, since satisfaction is not generally monotonic in logical systems. In classical logic, for instance, if s_1 and s_2 are first-order structures and $s_1 \subseteq s_2$, then it is not necessarily true that if $s_1 \models_c \psi$ then $s_2 \models_c \psi$ for an arbitrary first-order sentence ψ.

Example 10.6 Suppose that R_1 and R_2 are the following Horn rulebases, where E and F are base predicates:

$$R_1 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow E(x), Q(x). \end{array} \right\} \qquad R_2 = \left\{ \begin{array}{l} B(x) \leftarrow F(x,y), P(x), P(y). \\ B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow E(x), Q(x). \end{array} \right\}$$

Then R_1 is a model of $A(x) \leftarrow B(x)[P/Q]$, but the larger rulebase R_2 is not. $\Box$

10.1.3 Entailment

As in Section 9.3, our goal is not to derive Horn rules, but to apply rules with substitution to a database to derive atomic formulas. Section 9.3 gave a proof theoretic definition of what it means to derive an atomic formula α from a rulebase with substitution $\mathcal{S}$ and a database DB. This section gives a semantic definition.

Given a database DB, the following statement says that α is entailed by the rulebase $\mathcal{S}$ iff it is entailed by each model of $\mathcal{S}$:

$$\mathcal{S}, DB \models \alpha \quad \textit{iff} \quad R \cup DB \models_c \alpha \quad \textit{for all Horn models } R \textit{ of } \alpha.$$

This is a natural definition of entailment. To be completely general, however, we extend this definition from atomic formulas α to an arbitrary predicate with substitution β. Recalling that the meaning of β is given by R^*, we make the following definitions.

Definition 10.3 *Let R be a set of Horn rules, β a predicate with substitution, and DB a database. Then $R, DB \models \beta$ iff $R^* \cup DB \models_c \beta$.*

By Lemma 10.1, if β is atomic, then $R, DB \models \beta$ iff $R \cup DB \models_c \beta$.

Definition 10.4 *(Entailment) Suppose $\mathcal{S}$ is a set of rules with substitution, DB is a database for $\mathcal{S}$, and β is a predicate with substitution. Then $\mathcal{S}, DB \models \beta$ iff $R, DB \models \beta$ for every model R of $\mathcal{S}$.*

10.2 Completeness

This section establishes the completeness of the inference system of Section 9.3 with respect to the semantics of substitution developed in Section 10.1. That is, we prove the following theorem:

Theorem 10.2 *(Completeness) Suppose β is a predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. If $\mathcal{S}, DB \models \beta$ then $\mathcal{S}, DB \vdash \beta$.*

To prove completeness, we define a rulebase $R_{\mathcal{S}}$ called the *canonical model* of $\mathcal{S}$. This model, defined proof-theoretically, provides the necessary link between inference and semantics. Completeness follows from the basic properties of this model.

Definition 10.5 *If $\mathcal{S}$ is a set of Horn rules with substitution, then $R_{\mathcal{S}}$ is the set of Horn rules that are classically entailed by $\mathcal{S}^*$. That is,*

$$R_{\mathcal{S}} = \{\alpha \leftarrow \alpha_1 ... \alpha_n \mid \mathcal{S}^* \models_c \alpha \leftarrow \alpha_1 ... \alpha_n\}$$

where each α and each α_i are atomic formulas.

Since $\mathcal{S}^* \models_c R_{\mathcal{S}}$, the next result follows immediately from Lemma 9.6.

Lemma 10.3 $\mathcal{S}^* \models_c R^*_{\mathcal{S}}$

The next result shows that the canonical model $R_{\mathcal{S}}$ is a model of $\mathcal{S}$.

Lemma 10.4 $R_{\mathcal{S}} \models \mathcal{S}$

Proof: Choose a rule $\alpha \leftarrow \beta_1 ... \beta_n$ in $\mathcal{S}$. Since this rule is also in $\mathcal{S}^*$, it follows that $\mathcal{S}^* \models (\alpha \leftarrow \beta_1 ... \beta_n)\sigma$ for any term substitution σ. Hence, for any string of atomic formulas γ,

if	$R^*_{\mathcal{S}} \models_c \beta_i\sigma \leftarrow \gamma$	for each i
then	$\mathcal{S}^* \models_c \beta_i\sigma \leftarrow \gamma$	for each i, by Lemma 10.3
	$\mathcal{S}^* \models_c \alpha\sigma \leftarrow \gamma$	since $\mathcal{S}^* \models_c \alpha\sigma \leftarrow \beta_1\sigma ... \beta_n\sigma$
	$\alpha\sigma \leftarrow \gamma \in R_{\mathcal{S}}$	by definition
	$R_{\mathcal{S}} \models_c \alpha\sigma \leftarrow \gamma$	

Thus $R_{\mathcal{S}} \models \alpha \leftarrow \beta$ by Definition 10.2. $R_{\mathcal{S}}$ therefore satisfies each rule in $\mathcal{S}$; so it is a model of $\mathcal{S}$. □

The next result shows that inference with substitution is complete with respect to satisfaction in the canonical model.

Lemma 10.5 *If* $R_{\mathcal{S}}, DB \models \beta$ *then* $\mathcal{S}, DB \vdash \beta$, *where* DB *is a database,* β *is a predicate with substitution,* $\mathcal{S}$ *is a Horn rulebase with substitution.*

Proof:

if	$R_{\mathcal{S}}, DB \models \beta$	
then	$R^*_{\mathcal{S}} + DB \models_c \beta$	by Definition 10.3
	$\mathcal{S}^* + DB \models_c \beta$	by Lemma 10.3
	$\mathcal{S}^* + DB \vdash_c \beta$	by classical completeness
	$\mathcal{S}, DB \vdash \beta$	by Definition 9.4

□

Completeness follows immediately. By definition, if $\mathcal{S}, DB \models \beta$ then $R, DB \models \beta$ for any model R of $\mathcal{S}$. Thus $R_{\mathcal{S}}, DB \models \beta$ by Lemma 10.4. Hence $\mathcal{S}, DB \vdash \beta$ by Lemma 10.5.

10.3 Soundness

This section establishes the soundness of the inference system of Section 9.3 with respect to the semantics of substitution developed in Section 10.1. That is, we prove the following theorem:

Theorem 10.6 *(Soundness) Suppose* β *is a predicate with substitution,* $\mathcal{S}$ *is a Horn rulebase with substitution, and* DB *is a database. If* $\mathcal{S}, DB \vdash \beta$ *then* $\mathcal{S}, DB \models \beta$.

By definition $\mathcal{S}, DB \vdash \beta$ iff $\mathcal{S}^* \cup DB \vdash_c \beta$ where $\mathcal{S}^*$ is treated as a set of classical Horn rules. Therefore $\mathcal{S}, DB \vdash \beta$ iff β is derivable by the following, simple inference system:

Axioms: *Each element of DB is an axiom.*

Inference Rules: *For each rule* $\beta \leftarrow \beta_1 ... \beta_n$ *in* $\mathcal{S}^*$ *and each term substitution* σ*,*

$$\frac{\beta_1\sigma ... \beta_n\sigma}{\beta\sigma}$$

To prove Theorem 10.6, it is sufficient to show that this inference system is sound. This follows immediately from the next lemma.

Lemma 10.7 *Suppose* β *and* $\beta_1 ... \beta_n$ *are predicates with substitution,* $\mathcal{S}$ *is a rulebase with substitution, and DB is a database. Then,*

1. $\mathcal{S}, DB \models DB$
2. *Suppose* $\beta \leftarrow \beta_1 ... \beta_n$ *is a rule in* $\mathcal{S}^*$ *and* σ *is a term substitution.*
   If $\mathcal{S}, DB \models \beta_i\sigma$ *for each* i*, then* $\mathcal{S}, DB \models \beta\sigma$.

The first item is trivially true by Definitions 10.4 and 10.3, and it shows that the axioms of the inference system are sound. The second item shows that the rules of inference are sound. The rest of this section proves the second item.

Definition 10.2 gives the semantic meaning of $R \models \beta \leftarrow \beta_1 ... \beta_n$ when β is atomic. This is the most important case because inference has been defined only for rulebases containing rules of this kind. However, for the purpose of proving Lemma 10.7, it is convenient to have a more general definition of satisfaction. The next definition does exactly this. Note that this definition is purely a formal device and is not intended as a meaningful semantics for rules with substitution in the head.

Definition 10.6 *(Generalized Satisfaction) Let* R *be a set of Horn rules, and* β *and* $\beta_1 ... \beta_n$ *be predicates with substitution. Then,* $R \models \beta \leftarrow \beta_1 ... \beta_n$ *iff for all term substitutions* σ*, and all strings of atomic formulas* γ*,*

$$R^* \models_c \beta\sigma \leftarrow \gamma \quad \textit{if} \quad R^* \models_c \beta_i\sigma \leftarrow \gamma \quad \textit{for each } 1 \leq i \leq n.$$

If $\mathcal{S}$ is a rulebase with substitution and R is a Horn rulebase, then we say that R satisfies $\mathcal{S}$ iff R satisfies each rule in $\mathcal{S}$. In this case, we also say that R is a *model* of $\mathcal{S}$.

The following lemma shows that Definition 10.6 is in fact a generalization of definition 10.2, thus justifying the use of the same symbol, $\models$, for both notions

Lemma 10.8 *Let* $\models$ *denote satisfaction and let* $\models^*$ *denote generalized satisfaction. If* α *is atomic and* $\beta_1 ... \beta_n$ *are predicates with substitution, then*

$$R \models \alpha \leftarrow \beta_1 ... \beta_n \quad \textit{iff} \quad R \models^* \alpha \leftarrow \beta_1 ... \beta_n$$

for any Horn rulebase R *without substitution.*

Proof: It is sufficient to show that

$$R \models_c \alpha \leftarrow \gamma \quad \textit{iff} \quad R^* \models_c \alpha \leftarrow \gamma$$

for any atomic formula α and any string of atomic formulas γ. The *only if* direction is immediate since $R \subseteq R^*$. In the other direction, suppose that $R^* \models_c \alpha \leftarrow \gamma$. Because R is Horn, every rule in R^* with an atomic conclusion has only atomic premises. Thus, the only rules in R^* that can be used to derive $\alpha \leftarrow \gamma$ are Horn rules. But the set of Horn rules in R^* is just R. Hence $R \models_c \alpha \leftarrow \gamma$. $\Box$

The following, straightforward lemma lists some basic properties of generalized satisfaction.

Lemma 10.9 *Let* R *be a Horn rulebase without substitution, and let* ρ *be a rule with substitution. Then*

1. *If* $R \models \beta_1 \leftarrow \beta_2$ *and* $R \models \beta_2 \leftarrow \beta_3$ *then* $R \models \beta_1 \leftarrow \beta_3$.
2. *If* $R \models \rho$ *then* $R \models \rho\sigma$ *any term substitution* σ.
3. *If* $R^* \models_c \rho$ *then* $R \models \rho$.

The next lemma is an important step in the proof of soundness. It establishes semantic properties corresponding to steps in the construction of $\mathcal{S}^*$ (Definition 9.3).

Lemma 10.10 *Let* R *be a Horn rulebase without substitution. Then,*

1. *If* $R \models \beta \leftarrow \beta_1 ... \beta_n$ *then* $R \models \beta[\theta] \leftarrow \beta_1[\theta] ... \beta_n[\theta]$, *for every predicate substitution* $[\theta]$.

2. $R \models P(\overline{x})[P/Q] \leftarrow Q(\overline{x})$ *for every predicate substitution* $[P/Q]$.

3. $R \models B(\overline{x})[P/Q] \leftarrow B(\overline{x})$ *for every predicate substitution* $[P/Q]$ *and every base predicate* $B \neq P$.

Proof: Refer to the third item of Lemma 10.9. As a special case, R satisfies every rule in R^*. In particular, it satisfies the rules described in the last two items of Definition 9.3. This proves items 2 and 3. To prove item 1, refer to Definition 10.6. Suppose that for each i, $R^* \models_c \beta_i[\theta] \leftarrow B_1 ... B_m$ where $B_1 ... B_m$ are atomic. We must show that $R^* \models_c \beta[\theta] \leftarrow B_1 ... B_m$. To prove this, first note that by Lemma 9.10, for each i, there is a set of atoms $A^i_1 ... A^i_{n_i}$ such that

$$R^* \models_c \beta_i \leftarrow A^i_1 ... A^i_{n_i} \tag{10.5}$$

$$\textit{and} \quad R^* \models_c A^i_j[\theta] \leftarrow B_1 ... B_m \quad \textit{for each } j. \tag{10.6}$$

Therefore,

$R^* \models_c \beta_i \leftarrow A^i_1 ... A^i_{n_i}$	by statement (10.5).
$R^* \models_c \beta_i \leftarrow A^1_1 ... A^1_{n_1}\ A^2_1 ... A^2_{n_2}\ ...\ A^m_1 ... A^m_{n_m}$	
$R^* \models_c \beta \leftarrow A^1_1 ... A^1_{n_1}\ A^2_1 ... A^2_{n_2}\ ...\ A^m_1 ... A^m_{n_m}$	since $R \models \beta \leftarrow \beta_1 ... \beta_n$.
$R^* \models_c \beta[\theta] \leftarrow (A^1_1 ... A^1_{n_1}\ A^2_1 ... A^2_{n_2}\ ...\ A^m_1 ... A^m_{n_m})[\theta]$	by Corollary 9.5
$R^* \models_c \beta[\theta] \leftarrow B_1 ... B_n$	by statement (10.6).

□

Lemma 10.10 corresponds to items 2, 3 and 4 in the construction of $\mathcal{S}^*$ (Definition 9.3). The following corollary combines these results with item 1 to develop the main result about generalized satisfaction, and a central result in the proof of soundness. Note that the corollary would not make sense if the notion of generalized satisfaction had not been introduced, since many of the rules in $\mathcal{S}^*$ have non-atomic heads.

Corollary 10.11 *If* $R \models \mathcal{S}$ *then* $R \models \mathcal{S}^*$, *for any Horn rulebase* R, *and any Horn rulebase* $\mathcal{S}$ *with substitution.*

Proof: Let $cl(R)$ be the set of formulas entailed by R, that is, $cl(R) = \{\rho \mid R \models \rho\}$. By assumption, $\mathcal{S} \subseteq cl(R)$, so $cl(R)$ satisfies item 1 in Definition 9.3, and by Lemma 10.10, $cl(R)$ satisfies the other three items in Definition 9.3. But R^* is defined to be the smallest set satisfying all four of these items. Hence $\mathcal{S}^* \subseteq cl(R)$, that is, $R \models \mathcal{S}^*$.
□

In practice, rulebases and databases are finite. Nothing in the development exploits this fact, however, but the proof of the next lemma would be somewhat simpler if we did. The proof avoids the assumption, and keeps the result completely general, by invoking the compactness theorem for classical inference. The compactness theorem says that if a formula is derivable, then it is derivable from a finite set of axioms.

Lemma 10.12 *If* $R \models \beta \leftarrow \beta_1 ... \beta_n$ *and* $R, DB \models \beta_i$ *for each* i *from* 1 *to* n, *then* $R, DB \models \beta$. *Here,* R *is a Horn rulebase without substitution and* β *and* β_i *are predicates with substitution.*

Proof: Suppose that $R \models \beta \leftarrow \beta_1 ... \beta_n$. Then

if	$R, DB \models \beta_i$	for each i,
then	$R^* \cup DB \models_c \beta_i$	by Definition 10.3
	$R^* \cup \gamma_i \models_c \beta_i$	for some finite subset γ_i of DB,
		by the compactness of classical logic.
	$R^* \cup \gamma \models_c \beta_i$	for each i, where $\gamma = \gamma_1 \cup ... \cup \gamma_n$
	$R^* \models_c \beta_i \leftarrow \gamma$	for each i, by the deduction theorem
	$R^* \models_c \beta \leftarrow \gamma$	by Definition 10.6, since $R \models \beta \leftarrow \beta_1 ... \beta_n$
	$R^* \cup \gamma \models_c \beta$	by the deduction theorem
	$R^* \cup DB \models_c \beta$	since $\gamma \subseteq DB$
	$R, DB \models \beta$	by Definition 10.3.

□

We are now in a position to complete the proof of Lemma 10.7. As pointed out, the first item in the lemma is trivial. To prove the second item, suppose that $\beta \leftarrow \beta_1 ... \beta_n$ is a rule in $\mathcal{S}^*$ and σ is a term substitution. Thus, by Corollary 10.11, if R is a model of $\mathcal{S}$, then $R \models \beta \leftarrow \beta_1 ... \beta_n$. Hence,

$$R \models \beta\sigma \leftarrow \beta_1\sigma...\beta_n\sigma$$

by the second item of Lemma 10.9. Keeping this in mind, suppose that $\mathcal{S}, DB \models \beta_i\sigma$ for each i. Then $R, DB \models \beta_i\sigma$ for each i, and so $R, DB \models \beta\sigma$ by Lemma 10.12. This is true for any model R of $\mathcal{S}$. Hence $\mathcal{S}, DB \models \beta\sigma$.

This completes the proof of Lemma 10.7, which in turn, completes the proof of soundness.

10.4 Fixpoint Semantics

The model theory of predicate substitution is similar to the classical model theory of Horn logic in several respects. For instance, as shown in Section 10.1, a rulebase with substitution $\mathcal{S}$ has many Horn models, just as a Horn rulebase R has many classical models. Furthermore, adding rules to $\mathcal{S}$ eliminates some of its models, just as the addition of Horn rules to R eliminates some of its classical models. In this section, we show that the parallel can be taken much further. For instance, although a Horn rulebase has many models, it has only one *minimal* model. Likewise, we show that a rulebase with substitution has only one minimal model. In both cases, the minimal model can be viewed as the model that the user "had in mind" when he wrote the rulebase. The rulebase is thus a description of the user's intended model. In Horn logic, this model can be constructed, if necessary, by a bottom-up, iterative procedure. We develop a similar procedure for rulebases with substitution. As in Horn logic, this procedure is based on the repeated application of a "T operator." The main difference between the two logics is that a Horn rulebase specifies a database (i.e., a first-order structure), whereas a rulebase with substitution specifies a Horn rulebase. Substitution thus elevates Horn rules to the meta-level: Whereas they are normally the specification language, here they are the objects being specified.

Before developing the fixpoint theory of predicate substitution, we give the following example, which illustrates the difference between a model and a minimal model for rulebases with substitution.

Example 10.7 Suppose that $\mathcal{S}_1$ and $\mathcal{S}_2$ are the following rulebases with substitution, were E and F are base predicates:

$$\mathcal{S}_1 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow B(x)[P/Q] \end{array} \right\} \qquad \mathcal{S}_2 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ B(x) \leftarrow F(x,y), P(x), P(y). \\ A(x) \leftarrow B(x)[P/Q] \end{array} \right\}$$

Then each of the following Horn rulebases is a model of $\mathcal{S}_1$, but only R_3 is a model of $\mathcal{S}_2$. In addition, R_1 is the minimal model of $\mathcal{S}_1$, and R_3 is the minimal model of $\mathcal{S}_2$.

$$R_1 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow E(x), Q(x). \end{array} \right\} \qquad R_2 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow E(x), Q(x). \\ A(x) \leftarrow F(x,y), Q(x), Q(y). \end{array} \right\}$$

$$R_3 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ B(x) \leftarrow F(x,y), P(x), P(y). \\ A(x) \leftarrow E(x), Q(x). \\ A(x) \leftarrow F(x,y), Q(x), Q(y). \end{array} \right\}$$

□

In each model in Example 10.7, all the rules defining $B(x)$ are reused and modified to produce rules defining $A(x)$. In the minimal model, however, these are the *only* rules defining A. Furthermore, in the minimal model, the only rules defining B are those specified by $\mathcal{S}_i$. The minimal model thus has no superfluous rules: It contains the rules specified by $\mathcal{S}_i$ and no others. For this reason, the minimal model is central to the model theory of predicate substitution, and it is the focus of this section.

In Horn logic, a central feature of the fixpoint theory is a lattice of classical, first-order structures [48, 5]. Viewing first-order structures as a lattice simplifies the theoretical development, since the main results are a consequence of the lattice-theoretic properties of Horn rules. Once these properties are established, the details of logical syntax and inference are abstracted out of the theory.

Similarly, a central feature of the fixpoint theory of predicate substitution is a lattice of Horn rulebases. Given a rulebase with substitution $\mathcal{S}$, each element of the lattice is a potential model of $\mathcal{S}$. To move from point to point in this lattice, we define an operator $T_{\mathcal{S}}$. As in the fixpoint theory of Horn logic, this operator takes a lattice element as input, and returns another lattice element as output. An important property of this operator is that the fixpoints of $T_{\mathcal{S}}$ are exactly the models of $\mathcal{S}$. This property is an essential link between the fixpoint theory and the model theory, allowing us to study the models of a rulebase by studying the fixpoints of its T operator. Two other important properties of this operator are its *monotonicity* and *continuity*. Once these two properties are established, the main results of this section can be proved by lattice theory, and in particular, by invoking Tarski's fixpoint theorem in the usual way [47].

The first main result is that a rulebase with substitution has a unique minimal model. As in Horn logic, the minimal model is canonical in that the formulas satisfied in the minimal model are exactly the formulas entailed by the rulebase. We can therefore regard the rulebase as a specification of the minimal model. In this view, the minimal model is the model "intended" by the user, the model he "had in mind" when he wrote the rulebase. In the case of a Horn rulebase R, the minimal model is the *database* specified by R. In the case of a rulebase with substitution $\mathcal{S}$, the minimal model is the *Horn rulebase* specified by $\mathcal{S}$. Note that a minimal model need not be materialized in order to answer queries, since query processing can be carried out directly on the rulebase that describes it. The rulebase is thus an executable specification of the minimal model.

The second main result of this section is that the minimal model can be materialized, if necessary, by an iterative, bottom-up procedure. Starting with the empty interpretation, the procedure generates larger and larger interpretations until it finally generates the minimal model of the rulebase. To generate these interpretations, the procedure applies the T operator repeatedly, with each application converting one interpretation into a larger one.

Given a rulebase with substitution $\mathcal{S}$, each application of the operator $T_{\mathcal{S}}$ "unfolds" the rules in $\mathcal{S}$ into Horn rules. As the operator is applied again and again, more and more unfolding occurs. In this way, the description of a Horn rulebase is gradually unfolded into the Horn rulebase itself. If the description is non-recursive—if $\mathcal{S}$ has no recursion through substitution—then the unfolding process is similar to macro expansion, terminating after a finite number of steps. If the description is recursive, then the unfolding process may go on indefinitely. However, because the T operator is continuous, the process converges to the minimal model, with each Horn rule in the minimal model being generated after a finite number of steps.

10.4.1 The Lattice of Horn Rulebases

The central feature of any fixpoint semantics is a lattice of interpretations. In the case of predicate substitution, the lattice is the set of all Horn rulebases (finite or infinite). The elements of this lattice are ordered according to classical entailment. That is, if R_1 and R_2 are Horn rulebases, then $R_1 \geq R_2$ iff $R_1 \models_c R_2$. In this way, equivalent interpretations are equivalent lattice elements. That is, R_1 and R_2 are classically equivalent iff $R_1 \geq R_1$ and $R_2 \geq R_1$. In this case, we write $R_1 \equiv R_2$.

A lattice is *complete* if any set of lattice elements $\mathcal{R}$ has a least upper bound $\sqcup\mathcal{R}$ and a greatest lower bound $\sqcap\mathcal{R}$. Thus, if $R_o \geq R$ for all $R \in \mathcal{R}$, then $R_o \geq \sqcup\mathcal{R}$. Note that the least upper bound is unique up to equivalence: If R_1 and R_2 are least upper bounds of $\mathcal{R}$, then $R_1 \geq R_2$ and $R_2 \geq R_1$, so $R_1 \equiv R_2$. Similarly, the greatest lower bound is unique up to equivalence. As a special case, a complete lattice has a greatest element $\top$, which is the greatest upper bound of all the lattice elements. Likewise, the lattice has a least element $\perp$.

The following lemma shows that the set of all Horn rulebases is a complete lattice. The least element of this lattice is the empty rulebase, and the greatest element is the set of all Horn rules.

Lemma 10.13 *The set of all Horn rulebases without substitution is a complete lattice under the following operations:*

1. $R_1 \geq R_2$ *iff* $R_1 \models_c R_2$
2. $\sqcup\mathcal{R} = \cup\mathcal{R}$ *for any set* $\mathcal{R}$ *of Horn rulebases.*

3. $\sqcap\mathcal{R} \;=\; \cap_{R\in\mathcal{R}}\ \{\rho \mid R \models_c \rho\}$ *where each* ρ *is a Horn rule.*

Proof: We prove each item in turn.

1. Since classical entailment is transitive and reflexive, it is a partial order.

2. We must show that $\sqcup\mathcal{R}$ is the least upper bound of the elements in $\mathcal{R}$ under the partial order $\geq$. To prove this, suppose that R_0 is a Horn rulebase such that $R_0 \geq R$ for every R in $\mathcal{R}$. Then $R_0 \models_c R$ for every R in $\mathcal{R}$. Thus $R_0 \models_c \cup\mathcal{R}$, so $R_0 \geq \sqcup\mathcal{R}$. Hence $\sqcup\mathcal{R}$ is minimal.

3. We must show that $\sqcap\mathcal{R}$ is the greatest lower bound of the elements in $\mathcal{R}$.

if	$R \geq R_0$	for every R in $\mathcal{R}$
then	$R \models_c R_0$	for every R in $\mathcal{R}$
	$R_0 \subseteq \{\rho \mid R \models_c \rho\}$	for every R in $\mathcal{R}$
	$R_0 \subseteq \cap_{R\in\mathcal{R}}\ \{\rho \mid R \models_c \rho\}$	
	$R_0 \subseteq \sqcap\mathcal{R}$	
	$\sqcap\mathcal{R} \models_c R_0$	
	$\sqcap\mathcal{R} \geq R_0$	

Thus $\sqcap\mathcal{R}$ is maximal.

□

The following example shows that in general $R_1 \sqcap R_2 \neq R_1 \cap R_2$. The definition of $\sqcap$ is thus necessarily more complex than that of $\sqcup$. Since $R_1 \sqcap R_2$ is infinite, the lattice must include infinite rulebases.

Example 10.8

$$\begin{array}{lllll} \textit{if} & R_1 \;=\; \{A \leftarrow P,\;\; P \leftarrow B\} & \textit{and} & R_2 \;=\; \{A \leftarrow Q,\;\; Q \leftarrow B\} \\ \textit{then} & R_1 \sqcap R_2 \;\equiv\; \{A \leftarrow B\} & \textit{and} & R_1 \cap R_2 \;=\; \{\}. \end{array}$$

□

The following basic result shows that rulebases higher in the lattice satisfy more formulas.

Lemma 10.14 *If* $R_2 \geq R_1$ *and* $R_1, DB \models \beta$ *then* $R_2, DB \models \beta$*, for any database* DB *and any predicate with substitution* β*.*

Proof: Suppose that $R_2 \geq R_1$. Then $R_2 \models_c R_1$, so $R_2^* \models_c R_1^*$ by Corollary 9.7. Hence

if	$R_1, DB \models \beta$	
then	$R_1^* \cup DB \models_c \beta$	by definition
	$R_2^* \cup DB \models_c \beta$	since $R_2^* \models_c R_1^*$
	$R_2, DB \models \beta$	by definition

□

10.4.2 The T operator

Another central feature of any fixpoint semantics is a "T operator," which takes one lattice element as input and returns another lattice element as output. The T operator provides a way of "moving" from point to point within the lattice. There is one such operator $T_{\mathcal{S}}$ associated with every rulebase $\mathcal{S}$. An important property of this operator is that the fixpoints of $T_{\mathcal{S}}$ are exactly the models of $\mathcal{S}$. This property is an essential link between the fixpoint theory and the model theory, allowing us to study the models of a rulebase by studying the fixpoints of its T operator.

In the fixpoint theory of Horn logic, the input and output of the T operator is a database. In the simplest case, if the Horn rulebase contains the rule $A \leftarrow B$, and the input contains the atom B, then the output contains the atom A.

The operator $T_{\mathcal{S}}$ developed in this section is more general. For one thing, the input R and the output $T_{\mathcal{S}}(R)$ are Horn rulebases, not databases. To take the simplest case, if $\mathcal{S}$ contains the Horn rule $A \leftarrow B$, and if R contains the rule $B \leftarrow C$, then $T_{\mathcal{S}}(R)$ contains the rule $A \leftarrow C$. Of course, $\mathcal{S}$ may also contain non-Horn rules such as $A \leftarrow B[P/Q]$. In this case, the rulebase R^* provides the rules defining $B[P/Q]$. For instance, if $\mathcal{S}$ contains the rule $A \leftarrow B[P/Q]$, and if $R^* \models^c B[P/Q] \leftarrow C$, then $T_{\mathcal{S}}(R)$ contains the rule $A \leftarrow C$. Notice that we ask about the rules *entailed* by R^*, not just about the rules *in* R^*. This allows the rule premises in R^* to be expanded until they are atomic.

The following definition formalizes these considerations. It should be compared to the definition of Horn model given in Section 10.1 (Definition 10.2).

Definition 10.7 *(T operator) Let $\mathcal{S}$ be a Horn rulebase with substitution, and let R be a Horn rulebase (without substitution). Then, $T_\mathcal{S}$ is a mapping that takes a Horn rulebase R as input and returns another Horn rulebase $T_\mathcal{S}(R)$ as output. In particular, the value of $T_\mathcal{S}(R)$ is*

$$\cup_\sigma \ \{\alpha\sigma \leftarrow \gamma \ \mid \ R^* \models_c \beta_i\sigma \leftarrow \gamma \text{ for each premise } \beta_i \text{ in some rule } \alpha \leftarrow \beta_1 ... \beta_n \ \text{ in } \mathcal{S}\}$$

where the union is over all term substitutions σ, and each γ is a string of atomic formulas.

Intuitively, the operator $T_\mathcal{S}$ attempts to "unfold" the rules with substitution in $\mathcal{S}$ and convert them into Horn rules, just as the operator T_R of Horn logic attempts to "unfold" the Horn rules in R and convert them to atoms.

The following basic result shows that the operator $T_\mathcal{S}$ maps equivalent rulebases onto equivalent rulebases.

Lemma 10.15 *If $R_1 \equiv R_2$ then $T_\mathcal{S}(R_1) \equiv T_\mathcal{S}(R_2)$.*

Proof: In fact, we shall prove that $T_\mathcal{S}(R_1) = T_\mathcal{S}(R_2)$. If $R_1 \equiv R_2$, then R_1 and R_2 are classically equivalent, and so by Corollary 9.8, R_1^* and R_2^* are classically equivalent too. Thus, referring to Definition 10.7, R_1^* satisfies $\beta_i\sigma \leftarrow \gamma$ iff R_2^* does. Thus $T_\mathcal{S}(R_1)$ contains $\alpha\sigma \leftarrow \gamma$ iff $T_\mathcal{S}(R_2)$ does. Hence $T_\mathcal{S}(R_1) = T_\mathcal{S}(R_2)$. $\Box$

A fixpoint of $T_\mathcal{S}$ is a Horn rulebase R such that $R \geq T_\mathcal{S}(R)$. The following result says that the fixpoints of $T_\mathcal{S}$ are exactly the models of $\mathcal{S}$.

Lemma 10.16 $R \models \mathcal{S}$ *iff* $R \geq T_\mathcal{S}(R)$

Proof: In the *if* direction, suppose that $R \models_c T_\mathcal{S}(R)$. We must show that $R \models \mathcal{S}$. To prove this, suppose that $\alpha \leftarrow \beta_1 ... \beta_n$ is a rule in $\mathcal{S}$. Then, for any term substitution σ, and any string of atomic formulas γ,

if	$R^* \models_c \beta_i\sigma \leftarrow \gamma$	for each i
then	$\alpha\sigma \leftarrow \gamma \ \in \ T_{\mathcal{S}}(R)$	by Definition 10.7
	$R \models_c \alpha\sigma \leftarrow \gamma$	since $R \models_c T_{\mathcal{S}}(R)$.

Thus $R \models \alpha \leftarrow \beta_1 ... \beta_n$ by Definition 10.2. This is true for any rule in $\mathcal{S}$. Hence $R \models \mathcal{S}$.

In the *only if* direction, suppose that $R \models \mathcal{S}$. We must show that $R \models_c T_{\mathcal{S}}(R)$. Suppose that $\alpha\sigma \leftarrow \gamma$ is a rule in $T_{\mathcal{S}}(R)$. Then, for some rule $\alpha \leftarrow \beta_1 ... \beta_n$ in $\mathcal{S}$,

$$R^* \models_c \beta_i\sigma \leftarrow \gamma$$

for each i, by Definition 10.7. Furthermore, $R \models \mathcal{S}$ by hypothesis, so

$$R \models \alpha \leftarrow \beta_1 ... \beta_n$$

Combining these two statements, we get $R \models_c \alpha\sigma \leftarrow \gamma$ by Definition 10.2. Therefore R classically entails every rule in $T_{\mathcal{S}}(R)$. Hence $R \models_c T_{\mathcal{S}}(R)$. $\Box$

Monotonicity

This section shows that the operator $T_{\mathcal{S}}$ is *monotonic*: If the input to the operator increases, then so does the output. Because it is monotonic, the operator has a least fixpoint $lfp(T_{\mathcal{S}})$, which is also the unique minimal model of the rulebase $\mathcal{S}$. The minimal model is canonical in that the formulas satisfied in this model are exactly the formulas entailed by the rulebase $\mathcal{S}$. Thus, as in Horn logic, the rulebase $\mathcal{S}$ can be viewed as a *description* of the minimal model $lfp(T_{\mathcal{S}})$. In this way, a rulebase *with* substitution specifies a rulebase *without* substitution. Note that the rulebase without substitution need never be materialized, since query processing can be carried out directly on $\mathcal{S}$. The rulebase $\mathcal{S}$ is an thus executable specification of a Horn rulebase.

Lemma 10.17 *$T_{\mathcal{S}}$ is monotonic. That is, if $R_1 \geq R_2$ then $T_{\mathcal{S}}(R_1) \geq T_{\mathcal{S}}(R_2)$.*

Proof: Suppose that $R_1 \models_c R_2$. We show that $T_{\mathcal{S}}(R_2) \subseteq T_{\mathcal{S}}(R_1)$. First note that $R_1^* \models_c R_2^*$ by Corollary 9.7. Thus, referring to Definition 10.7,

$$\begin{array}{lll} \textit{if} & \alpha\sigma \leftarrow \gamma \ \in \ T_{\mathcal{S}}(R_1) & \\ \textit{then} & R_1^* \models_c \beta_i\sigma \leftarrow \gamma & \text{for some rule } \alpha \leftarrow \beta_1 ... \beta_n \text{ in } \mathcal{S} \\ & R_2^* \models_c \beta_i\sigma \leftarrow \gamma & \text{since } R_1^* \models R_2^* \\ & \alpha\sigma \leftarrow \gamma \ \in \ T_{\mathcal{S}}(R_2) & \text{since } \alpha \leftarrow \beta_1 ... \beta_n \text{ is in } \mathcal{S}. \end{array}$$

Thus every rule in $T_{\mathcal{S}}(R_1)$ is also in $T_{\mathcal{S}}(R_2)$. □

Since $T_{\mathcal{S}}$ is a monotonic operator on a complete lattice, it has a unique minimal fixpoint. This result is due to Tarski [47], whose proof we reproduce below.

Theorem 10.18 *The operator $T_{\mathcal{S}}$ has a least fixpoint $\mathit{lfp}(T_{\mathcal{S}})$. That is, $\mathit{lfp}(T_{\mathcal{S}}) \leq R$ for all fixpoints R of $T_{\mathcal{S}}$.*

Proof: Let $\mathcal{R}$ be the set of fixpoints of $T_{\mathcal{S}}$. Then,

$$\begin{array}{ll} R \geq \sqcap\mathcal{R} & \text{for every element } R \text{ in } \mathcal{R} \\ T_{\mathcal{S}}(R) \geq T_{\mathcal{S}}(\sqcap\mathcal{R}) & \text{for every element } R \text{ in } \mathcal{R}\text{, since } T_{\mathcal{S}} \text{ is monotonic} \\ R \geq T_{\mathcal{S}}(\sqcap\mathcal{R}) & \text{for every element } R \text{ in } \mathcal{R}\text{, since } R \geq T_{\mathcal{S}}(R) \\ \sqcap\mathcal{R} \geq T_{\mathcal{S}}(\sqcap\mathcal{R}) & \end{array}$$

$\sqcap\mathcal{R}$ is thus a fixpoint of $T_{\mathcal{S}}$. Furthermore, $R \geq \sqcap\mathcal{R}$ for every fixpoint R. Hence, if $\sqcap\mathcal{R} \geq R$, then $\sqcap\mathcal{R} \equiv R$. Thus, up to equivalence, $\sqcap\mathcal{R}$ is the least fixpoint of $T_{\mathcal{S}}$. □

The least fixpoint of $T_{\mathcal{S}}$ is a canonical model of $\mathcal{S}$ in that the queries satisfied in this model are exactly the queries entailed by $\mathcal{S}$. The following result makes this idea precise.

Corollary 10.19 *Let $\mathcal{S}$ be a Horn rulebase with substitution, β be a predicate with substitution, and DB be a database. Then*

$$\mathcal{S}, DB \models \beta \quad \textit{iff} \quad \mathit{lfp}(T_{\mathcal{S}}), DB \models \beta$$

Proof: In the *only if* direction, suppose that $\mathcal{S}, DB \models \beta$. Then $R, DB \models \beta$ for all models R of $\mathcal{S}$. In particular $\mathit{lfp}(T_{\mathcal{S}}), DB \models \beta$ since $\mathit{lfp}(T_{\mathcal{S}})$ is a model of $\mathcal{S}$.

In the *if* direction, suppose that $\mathit{lfp}(T_{\mathcal{S}}), DB \models \beta$. Then

$R, DB \models \beta$ for all $R \geq lfp(T_{\mathcal{S}})$, by Lemma 10.14

$R, DB \models \beta$ for all fixpoints R of $T_{\mathcal{S}}$, since $R \geq lfp(T_{\mathcal{S}})$

$R, DB \models \beta$ for all models R of $\mathcal{S}$, since a model is a fixpoint

$\mathcal{S}, DB \models \beta$ by definition.

□

Continuity

This section shows that the operator $T_{\mathcal{S}}$ is *continuous*: If a sequence of input values converges to a point R, then the sequence of output values converges to the point $T_{\mathcal{S}}(R)$. Because the operator is continuous, its least fixpoint $lfp(T_{\mathcal{S}})$ can be constructed by an iterative, bottom-up procedure. Abstractly, this procedure starts at the bottom of the lattice, and climbs higher and higher up the lattice, converging on the least fixpoint.

We prove continuity in two lemmas. The first lemma is a consequence of the compactness result of Section 9.4.

Lemma 10.20 *Suppose that ρ is a Horn rule, and that $R_1 \leq R_2 \leq R_3 ...$ is an increasing sequence of Horn rulebases. Then,*

$$\textit{if} \quad (\sqcup_j R_j)^* \models_c \rho \quad \textit{then} \quad R_k^* \models_c \rho \quad \textit{for some } k.$$

Proof: Suppose that $(\sqcup_j R_j)^* \models_c \rho$. Then by Corollary 9.3, there is a finite subset S of $\sqcup_j R_j$ such that

$$S^* \models_c \rho \tag{10.7}$$

Since $S \subseteq \cup_j R_j$, each rule in S is in R_j for some j. Since S is finite, these j's have a maximum value k. Hence $S \subseteq \cup_{j \leq k} R_j$. Therefore

$$\cup_{j \leq k} R_j \models_c S \tag{10.8}$$

But $R_k \models_c R_j$ for $j \leq k$, by hypothesis. Thus $R_k \models_c \cup_{j \leq k} R_j$, and so $R_k \models_c S$ by (10.8). Thus $R_k^* \models_c S^*$ by Corollary 9.7, and so $R_k^* \models_c \rho$ by (10.7). □

Continuity has a simple intuitive meaning. Suppose that $R_0, R_1, R_2 ...$ is a sequence of values which "converges" to the value R. Then, an operator T is continuous iff the sequence $T(R_0)$, $T(R_1)$, $T(R_2)$ ... "converges" to the value $T(R)$. For the purpose of this section, we need only a simple definition of convergence: We say that an increasing sequence $R_0 \leq R_1 \leq R_2$... converges to the value $\sqcup_i R_i$. Note that by monotonicity, $T_{\mathcal{S}}(R_0) \leq T_{\mathcal{S}}(R_1) \leq T_{\mathcal{S}}(R_2)$ is also an increasing sequence, which therefore converges to $\sqcup_j T_{\mathcal{S}}(R_j)$. With this in mind, we show that the operator $T_{\mathcal{S}}$ is continuous.

Lemma 10.21 *$T_{\mathcal{S}}$ is continuous. That is,*

$$\textit{if} \quad R_1 \leq R_2 \leq R_3 \ ... \quad \textit{then} \quad T_{\mathcal{S}}(\sqcup_{j\geq 1}\ R_j) \ \equiv \ \sqcup_{j\geq 1}\ T_{\mathcal{S}}(R_j)$$

Proof: First note that $\sqcup_j R_j \geq R_k$ for all k. Thus $T_{\mathcal{S}}(\sqcup_j R_j) \geq T_{\mathcal{S}}(R_k)$ for all k, by monotonicity. Hence $T_{\mathcal{S}}(\sqcup_j R_j) \geq \sqcup_k T_{\mathcal{S}}(R_k)$.

To prove the inequality in the other direction, suppose that $\alpha\sigma \leftarrow \gamma$ is a rule in $T_{\mathcal{S}}(\sqcup_j R_j)$. Then, by Definition 10.7, there is a rule $\alpha \leftarrow \beta_1...\beta_n$ in $\mathcal{S}$ such that $(\sqcup_j R_j)^* \models_c \beta_i\sigma \leftarrow \gamma$ for each i. Therefore, by Lemma 10.20, for each i there is an integer k_i such that

$$R^*_{k_i} \ \models_c \ \beta_i\sigma \leftarrow \gamma \tag{10.9}$$

Let k be the maximum of the k_i's. Then $R_{k_i} \leq R_k$ by hypothesis, so $R_k \models_c R_{k_i}$. Thus $R^*_k \models R^*_{k_i}$ by Lemma 9.7. Hence $R^*_k \models_c \beta_i\sigma \leftarrow \gamma$ for each i, by (10.9). But $\alpha \leftarrow \beta_1...\beta_n$ is a rule in $\mathcal{S}$. Thus $\alpha\sigma \leftarrow \gamma$ is in $T_{\mathcal{S}}(R_k)$ by Definition 10.7. Hence $\alpha\sigma \leftarrow \gamma$ is in $\cup_j T_{\mathcal{S}}(R_j)$, that is, in $\sqcup_j T_{\mathcal{S}}(R_j)$.

We have thus shown that every rule in $T_{\mathcal{S}}(\sqcup_j R_j)$ is also in $\sqcup_j T_{\mathcal{S}}(R_j)$. Thus $T_{\mathcal{S}}(\sqcup_j R_j) \subseteq \sqcup_j T_{\mathcal{S}}(R_j)$, and so $T_{\mathcal{S}}(\sqcup_j R_j) \leq \sqcup_j T_{\mathcal{S}}(R_j)$. $\Box$

Having shown that $T_{\mathcal{S}}$ is continuous, we introduce the standard definitions below. They define the structure $T^*_{\mathcal{S}}(R)$ which is the result of repeatedly applying the operator $T_{\mathcal{S}}$ to the Horn rulebase R.

Definition 10.8 *Let $\mathcal{S}$ be a Horn rulebase with substitution, and let R be a Horn rulebase without substitution. Then,*

- $T^0_{\mathcal{S}}(R) \ = \ R$
- $T^{i+1}_{\mathcal{S}}(R) \ = \ T_{\mathcal{S}}[T^i_{\mathcal{S}}(R)]$

- $T^*_{\mathcal{S}}(R) \;=\; \sqcup_{i\geq 0}\, T^i_{\mathcal{S}}(R)$

Starting from the empty rulebase $\{\}$, the operator $T_{\mathcal{S}}$ produces a sequence of larger and larger Horn rulebases $T^0_{\mathcal{S}}(\{\})$, $T^1_{\mathcal{S}}(\{\})$, $T^2_{\mathcal{S}}(\{\})$ ... The main result of this section is that this sequence converges to the least fixpoint of $T_{\mathcal{S}}$. This result follows from the monotonicity and continuity of the operator $T_{\mathcal{S}}$, as shown by Tarski [47].

Theorem 10.22 $T^*_{\mathcal{S}}(\{\}) \;\equiv\; \mathit{lfp}(T_{\mathcal{S}})$

Proof: The empty rulebase $\{\}$ is the least element in the lattice of rulebases. Therefore $\{\} \leq \mathit{lfp}(T_{\mathcal{S}})$. Thus $T_{\mathcal{S}}(\{\}) \leq T_{\mathcal{S}}[\mathit{lfp}(T_{\mathcal{S}})] \leq \mathit{lfp}(T_{\mathcal{S}})$ since $T_{\mathcal{S}}$ is monotonic and $\mathit{lfp}(T_{\mathcal{S}})$ is a fixpoint. By induction $T^j_{\mathcal{S}}(\{\}) \leq \mathit{lfp}(T_{\mathcal{S}})$ for any j. Thus $\sqcup_j T^j_{\mathcal{S}}(\{\}) \leq \mathit{lfp}(T_{\mathcal{S}})$.

To prove the inequality in the other direction, first note that $\{\} \leq T_{\mathcal{S}}(\{\})$. Thus $T_{\mathcal{S}}(\{\}) \leq T^2_{\mathcal{S}}(\{\})$ since $T_{\mathcal{S}}$ is monotonic, and by induction $T^i_{\mathcal{S}}(\{\}) \leq T^{i+1}_{\mathcal{S}}(\{\})$. Thus $T^0_{\mathcal{S}}(\{\}) \leq T^1_{\mathcal{S}}(\{\}) \leq T^2_{\mathcal{S}}(\{\})$ ... Hence, by Lemma 10.21,

$$\begin{aligned}
T_{\mathcal{S}}[\sqcup_{j\geq 0}\, T^j_{\mathcal{S}}(\{\})] &\equiv \sqcup_{j\geq 0}\, T_{\mathcal{S}}[T^j_{\mathcal{S}}(\{\})] \\
&= \sqcup_{j\geq 0}\, T^{j+1}_{\mathcal{S}}(\{\}) \\
&= \sqcup_{j\geq 1}\, T^j_{\mathcal{S}}(\{\}) \\
&\leq \sqcup_{j\geq 0}\, T^j_{\mathcal{S}}(\{\})
\end{aligned}$$

Thus $\sqcup_j T^j_{\mathcal{S}}(\{\})$ is a fixpoint of $T_{\mathcal{S}}$. Hence $\sqcup_j T^j_{\mathcal{S}}(\{\}) \geq \mathit{lfp}(T_{\mathcal{S}})$ since $\mathit{lfp}(T_{\mathcal{S}})$ is the least fixpoint. $\Box$

10.4.3 Examples

As shown in the previous section, the minimal model of a rulebase $\mathcal{S}$ can be generated by applying the operator $T_{\mathcal{S}}$ over and over again. With each application of the operator, substitution rules in $\mathcal{S}$ are expanded into Horn rules. The first application of $T_{\mathcal{S}}$ extracts the Horn rules in $\mathcal{S}$. The second application combines these Horn rules with the substitution rules in $\mathcal{S}$ to generate new Horn rules. The process then repeats: Each application of the operator generates new Horn rules, which are then combined with the rules in $\mathcal{S}$ to produce still more Horn rules. If at any point, no new Horn rules are generated, then the minimal model has been reached. The examples below illustrate this step-by-step expansion of substitution rules into Horn rules.

The last two examples, 10.11 and 10.12, show two ways in which a finite rulebase with substitution can be expanded into an infinite set of Horn rules. Example 10.11 uses function symbols to generate an infinite set of rules, each of fixed length. In contrast, Example 10.12 uses no function symbols, but generates longer and longer rules. In both cases, the least model of the rulebase with substitution is an infinite Horn rulebase.

To demonstrate the generation of Horn rules, we introduce a simple delta notation. In general, $T_{\mathcal{S}}^{i-1}(\{\}) \subseteq T_{\mathcal{S}}^{i}(\{\})$, so we define $\Delta T_{\mathcal{S}}^{i}(\{\})$ to be $T_{\mathcal{S}}^{i}(\{\}) - T_{\mathcal{S}}^{i-1}(\{\})$. $\Delta T_{\mathcal{S}}^{i}(\{\})$ is thus the set of new Horn rules created by the i^{th} application of $T_{\mathcal{S}}$.

Example 10.9 Suppose the rulebase $\mathcal{S}$ consists of the following three rules, where B, C and D are base predicates:

$$
\begin{aligned}
&A(x) \leftarrow B(x), C(x).\\
&A(x) \leftarrow D(x,y), C(x), C(y).\\
&A'(x) \leftarrow A(x)[C/C']
\end{aligned}
$$

It then takes two applications of the operator $T_{\mathcal{S}}$ to generate the minimal model of $\mathcal{S}$. That is, $lfp(T_{\mathcal{S}}) = T_{\mathcal{S}}^{2}(\{\})$. This model consists of four Horn rules, generated as follows:

$$
\left.\begin{aligned}
&A(x) \leftarrow B(x), C(x).\\
&A(x) \leftarrow D(x,y), C(x), C(y).
\end{aligned}\right\} \in T_{\mathcal{S}}^{1}(\{\})
$$
$$
\left.\begin{aligned}
&A'(x) \leftarrow B(x), C'(x).\\
&A'(x) \leftarrow D(x,y), C'(x), C'(y).
\end{aligned}\right\} \in \Delta T_{\mathcal{S}}^{2}(\{\})
$$

The definition of the predicate A' is thus the same as the definition of A except that C has been replaced everywhere by C'. $\square$

Typically, not all the substitution rules in $\mathcal{S}$ will be applicable during each application of the operator $T_{\mathcal{S}}$. At first, only a few substitution rules may be applicable. However, as more and more Horn rules are generated, more and more substitutions become available for conversion into Horn rules. This is illustrated by the following propositional example.

Example 10.10 Suppose the rulebase $\mathcal{S}$ consists of the following $n+1$ rules, where each B_i is a base predicate:

$$\begin{aligned}
A_1 &\leftarrow B_1, B_2, B_3 ... B_n. \\
A_2 &\leftarrow A_1[B_1/C_1] \\
A_3 &\leftarrow A_2[B_2/C_2] \\
&\cdots \\
A_{n+1} &\leftarrow A_n[B_n/C_n]
\end{aligned}$$

It then takes $n+1$ applications of the operator $T_{\mathcal{S}}$ to generate the minimal model of $\mathcal{S}$. That is, $lfp(T_{\mathcal{S}}) = T_{\mathcal{S}}^{n+1}(\{\})$. This model consists of $n+1$ Horn rules, generated as follows:

$$\begin{array}{rcl}
A_1 \leftarrow B_1, B_2, B_3 ... B_n. & \in & T_{\mathcal{S}}^1(\{\}) \\
A_2 \leftarrow C_1, B_2, B_3 ... B_n. & \in & \Delta T_{\mathcal{S}}^2(\{\}) \\
A_3 \leftarrow C_1, C_2, B_3 ... B_n. & \in & \Delta T_{\mathcal{S}}^3(\{\}) \\
& \cdots & \\
A_{n+1} \leftarrow C_1, C_2, C_3 ... C_n. & \in & \Delta T_{\mathcal{S}}^{n+1}(\{\})
\end{array}$$

The $i+1^{st}$ application of $T_{\mathcal{S}}$ effectively makes a copy of the rule defining A_i, substitutes C_i for B_i in the premise, and then uses this as the definition of a new predicate A_{i+1}. $\square$

In general, when there is recursion through substitution, the minimal model of $\mathcal{S}$ will be infinite, and generating this model will require an infinite number of iterations of the operator $T_{\mathcal{S}}$. Unlike the previous examples, the next example cannot be terminated after a finite number of iterations.

Example 10.11 Suppose the rulebase $\mathcal{S}$ consists of the following three rules:

$$A(x) \leftarrow B(x,a), B(x,b), B(x,c).$$
$$A(x) \leftarrow A(x)[B/B']$$
$$B'(x,y) \leftarrow B(x,fy)$$

where f is a unary function symbol. Then, for any integer $i \geq 1$,

$$\begin{array}{lll} \textit{if} & A(x) \leftarrow B(x,u), B(x,v), B(x,w). & \in \ T^i_{\mathcal{S}}(\{\}) \\ \textit{then} & A(x) \leftarrow B'(x,u), B'(x,v), B'(x,w). & \in \ T^{i+1}_{\mathcal{S}}(\{\}) \\ \textit{hence} & A(x) \leftarrow B(x,fu), B(x,fv), B(x,fw). & \in \ T^{i+1}_{\mathcal{S}}(\{\}) \end{array}$$

where u, v and w are terms. Thus,

$$\begin{array}{ll} A(x) \leftarrow B(x,a), B(x,b), B(x,c). & \in \ T^1_{\mathcal{S}}(\{\}) \\ A(x) \leftarrow B(x,fa), B(x,fb), B(x,fc). & \in \ \Delta T^2_{\mathcal{S}}(\{\}) \\ A(x) \leftarrow B(x,f^2a), B(x,f^2b), B(x,f^2c). & \in \ \Delta T^3_{\mathcal{S}}(\{\}) \\ \quad \ldots & \\ A(x) \leftarrow B(x,f^ia), B(x,f^ib), B(x,f^ic). & \in \ \Delta T^i_{\mathcal{S}}(\{\}) \\ \quad \ldots & \end{array}$$

□

Example 10.11 shows how a finite rulebase with substitution can represent an infinite rulebase without substitution. This property may not be too surprising in this particular case, since the rulebase involves function symbols. In traditional logic programming, it is easy to construct rulebases whose least model is infinite as long as function symbols are available. The next example, however, shows that the least model of a rulebase with substitution can be infinite even in the *absence* of function symbols. Intuitively, the infiniteness comes about because the *length* of the rules in the minimal model is unbounded. In fact, each application of the T-operator generates longer and longer rules. In this respect, Datalog with substitution is different from Datalog without substitution, for which the minimal model is always finite.

Example 10.12 Suppose the rulebase $\mathcal{S}$ consists of the following three rules:

$$A(x) \leftarrow A(x)[P/Q]$$
$$A(x) \leftarrow P(x)$$
$$Q(x) \leftarrow B(x,y), P(y).$$

where P is a base predicate. Then, for any sequence of variables $x_1, x_2 ... x_{j+1}$, the operator $T_{\mathcal{S}}$ can generate a rule with the formulas $B(x_1,x_2), B(x_2,x_3)...B(x_j,x_{j+1})$ in the rule premise. In particular,

$$\begin{array}{lll} \textit{if} & A(x_1) \leftarrow B(x_1,x_2)...B(x_{j-1},x_j), P(x_j). & \in \;\; T^i_{\mathcal{S}}(\{\}) \\ \textit{then} & A(x_1) \leftarrow B(x_1,x_2)...B(x_{j-1},x_j), Q(x_j). & \in \;\; T^{i+1}_{\mathcal{S}}(\{\}) \\ \textit{hence} & A(x_1) \leftarrow B(x_1,x_2)...B(x_{j-1},x_j), B(x_j,x_{j+1}), P(x_{j+1}). & \in \;\; T^{i+1}_{\mathcal{S}}(\{\}) \end{array}$$

for any integer $i \geq 1$. Thus, since the rule $A(x) \leftarrow P(x)$ is in $\mathcal{S}$,

$$\begin{array}{l} A(x_1) \leftarrow P(x_1) \;\; \in \;\; T^1_{\mathcal{S}}(\{\}) \\ A(x_1) \leftarrow B(x_1,x_2), P(x_2) \;\; \in \;\; \Delta T^2_{\mathcal{S}}(\{\}) \\ A(x_1) \leftarrow B(x_1,x_2), B(x_2,x_3), P(x_3) \;\; \in \;\; \Delta T^3_{\mathcal{S}}(\{\}) \\ \qquad \ldots \\ A(x_1) \leftarrow B(x_1,x_2), B(x_2,x_3)...B(x_{i-1},x_i), P(x_i) \;\; \in \;\; \Delta T^i_{\mathcal{S}}(\{\}) \\ \qquad \ldots \end{array}$$

Thus, the rules in $\Delta T^i_{\mathcal{S}}(\{\})$ test the database for "B chains" of length i. Each application of the operator $T_{\mathcal{S}}$ takes a rule that tests for chains of length i and transforms it into a rule that tests for chains of length $i+1$. In this way, an infinite Horn rulebase is built up, one rule at a time, by reusing and modifying the seed rule $A(x) \leftarrow P(x)$ an infinite number of times. □

Example 10.12 shows that without using function symbols, a finite rulebase with substitution can represent an infinite rulebase without substitution. In this particular case, however, we can also represent the infinite rulebase in a finite way *without* using substitution. Instead, we introduce a new predicate symbol C, as follows:

$$\begin{aligned} A(x) &\leftarrow C(x) \\ C(x) &\leftarrow B(x,y), C(y). \\ C(x) &\leftarrow P(x) \\ C(x) &\leftarrow Q(x) \\ Q(x) &\leftarrow B(x,y), P(y). \end{aligned} \tag{10.10}$$

This rulebase generates exactly the same Horn rules for the predicate A as the rulebase of Example 10.12. To see this, we first unfold the recursion in the second rule to generate an infinite set of rules for C:

$$C(x_1) \leftarrow B(x_1, x_2), B(x_2, x_3)...B(x_{i-1}, x_i), C(x_i).$$

By combining these rules with the first, third and fourth rules in (10.10), we get an infinite set of rules for A:

$$\begin{aligned} A(x_1) &\leftarrow B(x_1, x_2), B(x_2, x_3)...B(x_{i-1}, x_i), P(x_i). \\ A(x_1) &\leftarrow B(x_1, x_2), B(x_2, x_3)...B(x_{i-1}, x_i), Q(x_i). \end{aligned}$$

In fact, these are the only rules entailed by (10.10) that define the predicate A in terms of the original predicates B, P and Q. Furthermore, these rules are exactly the rules for A generated in Example 10.12. In this way, the rulebase *with* substitution in Example 10.12 and the rulebase *without* substitution in (10.10) both represent the same infinite set of Horn rules.

Example (10.10) is particularly simple in that it has only one Horn rule defining the predicate A, and this Horn rule is reused and modified by only one rule with substitution. More generally, A could be defined by a large and complicated set of Horn rules, which in turn could be reused and modified by a complicated set of rules with substitution. In such cases, it would not be easy to write an equivalent set of rules without substitution, such as (10.10). In fact, sometimes it is impossible. This impossibility arises because predicate substitution increases the power of Datalog from P to EXPTIME. In particular, when a rulebase with substitution solves an EXPTIME-complete problem, as in Section 9.5, it is impossible to represent it as a finite Datalog rulebase. In this case, the minimal model of the rulebase with substitution is an infinite set of Datalog rules, and this infinite set cannot be represented by *any* finite set of Datalog rules. In such cases, predicate substitution provides an extremely concise notation for expressing large Datalog programs.

References

[1] S. Abiteboul and V. Vianu. A Transaction Language Complete for Database Update Specification. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 260–268, 1987. Published in expanded form as Rapports de Recherche no. 715, INRIA, 78153 Le Chesnay Cedex, France.

[2] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. Technical Report 900, Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay Cedex, France, 1988.

[3] A.V. Aho and J.D. Ullman. Universality of Data Retrieval Languages. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, pages 110–120, San Antonio, TX, Jan. 1979.

[4] K.R. Apt, H.A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann, 1988.

[5] K.R. Apt and M.H. Van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[6] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., May 28–30 1986.

[7] A.J. Bonner. A Logic for Hypothetical Reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Saint Paul, Minnesota, August 21–26 1988. American Association for Artificial Intelligence (AAAI).

[8] A.J. Bonner. Hypothetical Datalog: Complexity and Expressibility. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 144–160. Springer-Verlag, 1988. Published as volume 326 of *Lecture Notes in Computer Science*.

[9] A.J. Bonner. Hypothetical Datalog: Negation and Linear Recursion. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 286–300, Philadelphia, PA, March 1989.

[10] A.J. Bonner. Hypothetical Datalog: Complexity and Expressibility. *Theoretical Computer Science (TCS)*, 76:3–51, 1990. Special issue on the 2^{nd} International Conference on Database Theory (ICDT). Also appears, in abridged form, in volume 326 of *Lecture Notes in Computer Science*, pages 144–160. Springer Verlag, 1988.

[11] A.J. Bonner and T. Imielinski. The Reuse and Modification of Rulebases by Predicate Substitution. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 437–451, Fondazione Cini, Venice, Italy, March 26–30 1990.

[12] A.J. Bonner and L.T. McCarty. Adding Negation-as-Failure to Intuitionistic Logic Programming. In *Proceedings of the North American Conference on Logic Programming (NACLP)*, pages 681–703, Austin, Texas, Oct 29–Nov 1 1990.

[13] A.J. Bonner, L.T. McCarty, and K. Vadaparty. Expressing Database Queries with Intuitionistic Logic. In *Proceedings of the North American Conference on Logic Programming (NACLP)*, pages 831–850, Cleveland, Ohio, October 16–20 1989.

[14] A.K. Chandra. Theory of Database Queries. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 1–9, Austin, Texas, March 1988.

[15] A.K. Chandra and D. Harel. Computable Queries for Relational Databases. *Journal of Computer and System Sciences (JCSS)*, 21(2):156–178, 1980.

[16] A.K. Chandra and D. Harel. Structure and Complexity of Relational Queries. In *Proceedings of the Symposium on the Foundations of Computer Science (FOCS)*, pages 333–347, 1980.

[17] A.K. Chandra and D. Harel. Horn Clauses and the Fixpoint Query Hierarchy. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 158–163, 1982.

[18] A.K. Chandra and D. Harel. Horn Clause Queries and Generlizations. *Journal of Logic Programming (JLP)*, 2(1):1–15, 1985.

[19] A.K. Chandra, D. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[20] A.K. Chandra and L.J. Stockmeyer. Alternation. In *Proceedings of the Symposium on the Foundations of Computer Science (FOCS)*, pages 98–108, 1976.

[21] B.F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, 1980.

[22] M.C. Fitting. *Intuitionistic Logic, Model Theory and Forcing.* North-Holland, 1969.

[23] The Committee for Advanced DBMS function. Third-Generation Database System Manifesto. *SIGMOD Record*, 19(3):31–44, September 1990. Also published as Memorandum No. UCB/ERL M90/28, Electronics Research Laboratory, College of Engineering, University of California, Berkeley.

[24] D.M. Gabbay. N-Prolog: an Extension of Prolog with Hypothetical Implications. II. Logical Foundations and Negation as Failure. *Journal of Logic Programming*, 2(4):251–283, 1985.

[25] D.M. Gabbay and U. Reyle. N-Prolog: an Extension of Prolog with Hypothetical Implications. I. *Journal of Logic Programming (JLP)*, 1(4):319–355, 1984.

[26] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[27] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic.* Methuen and Co. Ltd., London, 1968.

[28] Richard Hull and Jianwen Su. On the Expressive Power of Database Queries with Intermediate Types. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 39–51, Austin, Texas, March 1988.

[29] N. Immerman. Relational Queries Computable in Polynomial Time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 147–152, 1982.

[30] N. Immerman. Languages that Capture Complexity Classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[31] R.H. Sprague Jr. and H.J. Watson, editors. *Decisions Support Systems: Putting Theory into Practice.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[32] R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[33] S. Kripke. Semantical Analysis of Intuitionistic Logic. I. In J.N. Crossley and M.A.E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North Holland, Amsterdam, 1965.

[34] S. Manchanda and D.S. Warren. A Logic-based Language for Database Updates. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 10, pages 363–394. Morgan Kaufmann, 1988.

[35] Sanjay Manchanda. *A Dynamic Logic Programming Language for Relational Updates.* PhD thesis, The University of Arizona, Tuscon, Arizona 85721, January 1988.

[36] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artifical Intelligence. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. Reprinted in "Readings in AI".

[37] L.T. McCarty. Clausal Intuitionistic Logic. I. Fixed-Point Semantics. *Journal of Logic Programming (JLP)*, 5(1):1–31, 1988.

[38] L.T. McCarty. Clausal Intuitionistic Logic. II. Tableau Proof Procedures. *Journal of Logic Programming (JLP)*, 5(2):93–132, 1988.

[39] L.T. McCarty and N.S. Sridharan. The Representation of an Evolving System of Legal Concepts. II. Prototypes and Deformations. In *Proceedings of the Seventh IJCAI*, pages 246–253, 1981.

[40] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming (JLP)*, 6:79–108, 1989.

[41] G. Nadathur and D. Miller. An Overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 810–827, Cambridge, MA, 1988. MIT Press.

[42] R.L. Olson and R.H. Sprague Jr. Financial Planning in Action. In R.H. Sprague Jr. and H.J. Watson, editors, *Decisions Support Systems: Putting Theory into Practice*, pages 373–381. Prentice Hall, Englewood Cliffs, NJ, 1989.

[43] Editor: A.P. Seth. Database Research at Bellcore. *SIGMOD Record*, 19(3):45–52, September 1990.

[44] R. Statman. Intuitionistic Propositional Logic is Polynomial-Space Complete. *Theoretical Computer Science (TCS)*, 9(1):67–72, 1979.

[45] M. Stonebraker. Hypothetical databases as views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 224–229, 1981.

[46] M. Stonebraker and K. Keller. Embedding expert knowledge and hypothetical databases into a data base system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 58–66, Santa Monica, CA, 1980.

[47] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[48] M.H. Van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

[49] M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 137–146, 1982.

[50] L. Vieille, P. Bayer, V. Kuchenhoff, and A. Lefebvre. EKS-V1, A Short Overview. Presented at the AAAI-90 Workshop on Knowledge Base Management Systems, July 1990, Boston, USA.

[51] D.S. Warren. Database Updates in Pure Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 244–253, 1984.

Vita

Anthony J. Bonner

1973–1977	B.Sc. with Honours in Mathematics and Physics, University of Toronto, Toronto, Ontario, Canada.
1977–1983	Defence Scientist, Defence Research Establishment Atlantic (DREA), Dartmouth, Nova Scotia, Canada.
1978–1983	Attendance at Dalhousie University, Halifax, Nova Scotia, Canada. Part time.
1983	J.N. Maksym, A.J. Bonner, C.A. Dent, and G.L. Hemphill. Machine Analysis of Acoustical Signals. *Pattern Recognition*, 16(6):615–625. Also published in the *Proceedings of the Workshop on Issues in Acoustic Signal/Image Processing and Recognition*, San Miniato, Italy, 1982.
1983	Teaching Assistant, Department of Computer Science, Rutgers, The State University, New Brunswick, New Jersey.
1984–1986	Research Assistant, Department of Computer Science, Rutgers, The State University.
1986–1989	Teaching Assistant, Department of Computer Science, Rutgers, The State University.
1988	A.J. Bonner. A Logic for Hypothetical Reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Saint Paul, Minnesota.
1989	A.J. Bonner. Hypothetical Datalog: Negation and Linear Recursion. In *Proceedings of the ACM Symposium on the Principles of Database Systems*, pages 286–300, Philadelphia, Pennsylvania.
1989	A.J. Bonner, L.T. McCarty, and K. Vadaparty. Expressing Database Queries with Intuitionistic Logic. In *Proceedings of the North American Conference on Logic Programming*, pages 831–850, Cleveland, Ohio.
1990	A.J. Bonner and T. Imielinski. The Reuse and Modification of Rulebases by Predicate Substitution. In *Proceedings of the International Conference on Extending Database Technology*, pages 437–451, Venice, Italy.

1990	M.S. in Computer Science, Rutgers, The State University.
1990	A.J. Bonner and L.T. McCarty. Adding Negation-as-Failure to Intuitionistic Logic Programming. In *Proceedings of the North American Conference on Logic Programming*, pages 681–703, Austin, Texas.
1990	Post-Doctoral Fellow, Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, France.
1990	A.J. Bonner. Hypothetical Datalog: Complexity and Expressibility. *Theoretical Computer Science*, 76:3–51. Also appears, in abridged form, in Volume 326 of *Lecture Notes in Computer Science*, pages 144–160. Springer Verlag, 1988.
1991	Assistant Professor, Computer Science Department, Indiana University, Bloomington, Indiana.
1991	S. Abiteboul and A.J. Bonner. Objects and Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 238–247, Denver, Colorado.
1991–	Assistant Professor, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada.
1991	Ph.D. in Computer Science, Rutgers, The State University.