Hypothetical Datalog: Complexity and Expressibility

Anthony J. Bonner
Rutgers University
Department of Computer Science
New Brunswick, NJ 08903
United States

bonner@db.toronto.edu

Appears in *Theoretical Computer Science (TCS)*, 76:3–51, 1990.

Abstract

We present an extension of Horn-clause logic which can hypothetically add and delete tuples from a database. Such logics have been discussed in the literature, but their complexities and expressibilities have remained an open question. This paper examines two such logics in the function-free, predicate case. It is shown, in particular, that augmenting Horn-clause logic with hypothetical addition increases its data-complexity from PTIME to PSPACE. When deletions are added as well, complexity increases again, to EXPTIME. We then augment the logic with *negation-as-failure* and develop the notion of *stratified* hypothetical rulebases. It is shown that negation does not increase complexity. To establish expressibility, we view the logic as a query language for relational databases. It is shown that any typed generic query that is computable in PSPACE can be expressed as a stratified rulebase of hypothetical additions. Similarly, any typed generic query that is computable in EXPTIME can be expressed as a stratified rulebase of hypothetical additions and deletions. Neither of these results assumes the data domain is linearly ordered. In this way, we establish the expressive completeness of our logics for queries in PSPACE and EXPTIME, resp.

This paper is available at the following URL:
ftp://db.toronto.edu/pub/bonner/papers/hypotheticals/tcs90.ps.gz

Contents

1 Introduction

Several researchers in the logic programming community have pointed out the utility of augmenting Horn-clause logic with the ability to hypothetically add facts to a database [11, 10, 20, 24, 16, 4]. Others have done as much by investigating the intuitionistic semantics of embedded implications [18, 19]. The complexity and expressibility of these logics has remained an open question however. This paper addresses these issues from a database point of view. In particular, results on data-complexity and expressibility are established for the function-free, predicate case.

This paper also examines hypothetical deletion. In contrast to addition, hypothetical deletion has received little attention in the logic-programming literature, one exception being the work of Manchanda and Warren [16]. As in their work, this paper focuses on deletions from the database, and not from the least fixpoint. Although relatively simple, such deletions nevertheless increase the complexity and expressibility of the logic from PSPACE to EXPTIME.[1,2]

Results are established for two distinct extensions of Horn-clause logic: (i) Horn logic augmented with hypothetical additions (called the *restricted* logic), and (ii) Horn logic augmented with both hypothetical additions and deletions (called the *full* logic). The restricted logic consists of Horn rules plus rules of the form $A \leftarrow B[add : C]$, which intuitively means, "Infer A if adding C to the database allows the inference of B." The full logic consists of the restricted logic plus rules of the form $A \leftarrow B[del : C]$, which intuitively means, "Infer A if deleting C from the database allows the inference of B." These additions and deletions are entirely hypothetical, so that the database is in its original state after all inferences have been completed.

Hypothetical queries are particularly useful in the context of rulebased systems and deductive databases. For example, one might ask, "If I were to take $eng423$, would I be able to graduate?", or perhaps, "Retrieve those students who could graduate if they took one more course." Gabbay and Reyle have reported a need to augment Prolog with hypothetical rules in order to encode the British Nationality Act. The act contains hypothetical rules such as the following: "You are eligible for citizenship if your father *would be* eligible *if* he were still alive" [11]. Such rules are called *embedded implications*, and they can be implemented as rules of hypothetical addition [18].

The formal properties of hypothetical additions are still being explored. Gabbay has shown that they have an intuitionistic semantics [10], and Miller has developed fixpoint semantics for the predicate case [20]. McCarty has extended this work to a larger class of formulas, and has established interesting semantic results analogous to the unique minimal-model property of Horn-clause logic [18, 19]. It is also known that intuitionistic validity is PSPACE-complete in the propositional case [21, 15]. To our knowledge, however, we are the first to address the questions of the data-complexity and expressibility of this kind of reasoning.

In order to establish upper bounds on the data complexity of the restricted logic of hypothetical addition, a mixed iterative/recursive algorithm is developed. One way to view

[1] In this paper, the terms EXPTIME and "exponential time" denote DTIME($2^{n^{O(1)}}$).

[2] The increase in complexity is strict iff $PSPACE \neq EXPTIME$, which although considered likely, is an open question.

this algorithm is as a recursive procedure in which bottom-up, least-fixpoint computations occur at each level of recursion. At each recursive call, the procedure applies the rulebase to a different hypothetical database. There are polynomially many levels of recursion and each level takes polynomial time; so the entire computation runs in polynomial space.

In order to establish upper complexity bounds for the full logic of hypothetical additions and deletions, we define a notion of *rulebase interpretation* and introduce a T-operator. This operator is similar to, but more complex than the T-operator of Horn logic [3, 22]; and it applies not to first-order structures, but to mappings. The development of a fixpoint semantics is not our goal however, and for this reason, the presentation is entirely proof theoretic. Indeed, the T-operator is introduced for purely pragmatic reasons: It is a convenient vehicle for establishing an upper complexity bound. First, a bottom-up, iterative procedure is developed for computing the "least fixpoint" of the operator. It is then shown that this procedure runs in exponential time.

In this way, EXPTIME and PSPACE upper-bounds on the data-complexity of query processing are established for the full and restricted logics, resp. To establish these as lower bounds as well, we use the logics to encode the computations of alternating Turing-machines [8]. In the full logic, the database represents the machine tape and hypothetical updates represent machine transitions: hypothetical deletion simulates tape erasure, and hypothetical addition simulates tape writing. In this way, alternating PSPACE machines can be encoded. In the restricted logic, the approach is more familiar: time is represented explicitly and the database encodes an entire computation path. By making time explicit, however, it is only possible to encode alternating PTIME machines. Hypothetical additions by themselves thus have a lower complexity than hypothetical additions and deletions together.

Having established tight complexity bounds for the two hypothetical logics, we extend the language by allowing negated premises. As in Horn logic, negation is interpreted as the failure to prove a goal. A notion of *stratified* hypothetical rulebases is then defined, and its proof theory is developed. It is shown that hypothetical inference with negation is independent of how the database is stratified. It is also shown that negation does not increase the data complexity of either the full or the restricted logic.

Although negation-as-failure does not increase the complexity, it does increase expressibility of hypothetical rules. Without negation, only monotonic queries can be expressed, that is, queries in which the answer gets larger as the database gets larger. With negation, non-monotonic queries can be expressed as well, such as relational algebra queries involving complementation. Indeed, it is shown that negation makes hypothetical rules expressively complete for PSPACE and EXPTIME. That is, any typed generic query which is computable in PSPACE can be expressed as a stratified rulebase of hypothetical additions. Similarly, any typed generic query which is computable in EXPTIME can be expressed as a stratified rulebase of hypothetical additions and deletions.

The proofs of expressive completeness rely on our encodings of alternating Turing-machines. In this respect they are similar to other expressibility proofs in the literature [13, 23]. One difference, however, is that we do not assume that the data domain is linearly ordered. Typically, linear orders are used to simulate counters, which in turn are used to simulate tape-head movements. Stratified hypothetical rules do not need an *a priori* domain ordering since they can generate a linear 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 machine simulations are senstive to which linear order is used; so that different "guesses" can lead to different query answers. It is shown, however, that generic queries are robust in that they return the same answer no matter which linear order is used. In this way, genericity is central to our completeness results.

2 Examples

This section gives several examples of hypothetical queries and rules. In each example, tuples are hypothetically added or deleted from the database before a least-fixpoint query is made. These queries are expressed with modal-like operators; in particular, $Q[add : P]$ means, "If P were added to the database, then Q would be true"; and $Q[del : P]$ means, "If P were deleted from the database, then Q would be true." The notation $R, DB \vdash \phi$ means that the query ϕ is true when applied to the rulebase R and the database DB. Before providing precise definitions in the next section, this section gives examples of hypothetical queries and rules and how they are expressed.

The examples are centered on a rulebased system which describes university policy. For instance, the atomic formula $take(s, c)$ means that student s has taken course c, and $grad(s)$ means that s is eligible for graduation. Thus, the database DB contains facts such as $take(tony, cs250)$, and the rulebase R contains rules such as

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

2.1 Hypothetical Queries

In the following examples, each query is described in three ways: (i) informally, in English, (ii) formally, at the meta-level, and (iii) formally, at the object-level with operators of addition and deletion.[3] In each example, a hypothetical query posed to the database DB is converted to an atomic query over another database, obtained by hypothetical additions or deletions.

Example 1. Consider the query, "If Tony took cs452, would he be eligible to graduate?" That is, 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:

$$R, DB + take(tony, cs452) \vdash grad(tony) \tag{1}$$

In our language of hypotheticals, this query is represented by the expression $\psi = grad(tony)[add : take(tony, cs452)]$. That is, ψ is an object-level level expression such that $R, DB \vdash \psi$ iff meta-level condition (1) is satisfied. □

Example 2. "If Tony had not taken eng101, could he have graduated?" That is, if $take(tony, eng101)$ were deleted from the database, could we infer $grad(tony)$? This query can be formalized at the meta-level as follows:

[3] See [14] for a description of meta-level and object-level reasoning.

$$R, DB \sim take(tony, eng101) \vdash grad(tony)$$

At the object-level, the expression $grad(tony)[del : take(tony, eng101)]$ represents this query. □

Example 3. "Retrieve those students who could graduate if they took (at most) one more course." i.e., at the meta-level, we want those s such that

$$\exists_c\, [R, DB + take(s, c) \vdash grad(s)]$$

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

2.2 Hypothetical Rules

Having introduced hypothetical queries, we can also use them in the premises of rules. These rules have the form $A \leftarrow B[add : C]$, which intuitively means, "A is true if inserting C into the database makes B true." Such rules turn our query language into a logic for building rulebases.

Example 4. Suppose the university wishes to change its policy on financial aid by enacting the following rule:

> "If a student is within one course of graduation and is not eligible for primary aid, then he is eligible for secondary aid."

This policy can be represented with two rules:[4]

$$\begin{aligned} aid2(s) &\leftarrow \sim aid1(s),\ grad1(s). \\ grad1(s) &\leftarrow \exists_c\, grad(s)[add : take(s, c)]. \end{aligned}$$

Here, $aid1(s)$ and $aid2(s)$ mean that student s is eligible for primary and secondary aid, resp., and $grad1(s)$ means that s is within one course of graduation. Note that the premise of the second rule is identical to the query $\psi(s)$ in example 3. Thus, if a rulebase R contains this rule, then for any database DB and any student s,

$$\begin{aligned} R, DB \ \vdash\ & grad1(s) \\ & \textit{if} \quad R, DB \ \vdash\ \exists_c\, grad(s)[add : take(s, c)] \\ & \textit{if} \quad \exists_c \ \ R, DB \ \vdash\ grad(s)[add : take(s, c)] \\ & \textit{if} \quad \exists_c \ \ R, DB + take(s, c) \ \vdash\ grad(s) \end{aligned}$$

□

Of course, hypothetical rules can be cascaded, so that in order to answer one hypothetical query, we must first answer another hypothetical query.

Example 5. Using the predicate $grad1(s)$ of the previous example, we can define the predicate $grad2(s)$, which means that student s is within 2 courses of graduation. Similarly, we can use $grad2(s)$ to define $grad3(s)$.

[4] The first rule uses negation-by-failure [14].

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

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. Thus, if the rulebase R contains these three rules, then for all databases DB and all students s,

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

In this way, in answering one query, one may need to answer other queries over many different hypothetical databases. □

Recursive hypothetical rules can also be defined. The next example suggests how recursion can be used to perform complex hypothetical transactions on a database.

Example 6. The predicates $grad1(s)$, $grad2(s)$ and $grad3(s)$ of the previous example 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:

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

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. Thus, if the rulebase R contains these two rules, then for any student s and any whole number n,

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

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

3 Hypothetical Inference

This section defines a logical inference system for hypothetical queries and rules.[5] Such systems have been developed by several researchers [11, 10, 20, 17, 18]. The one presented

[5]The semantics of this inference system is touched upon only briefly in this paper. In [4, 18, 20], however, it is shown that the restricted logic of hypothetical addition has an intuitionistic semantics.

here is a simplified version which retains many of the essential properties of the more elaborate systems while admitting a clean theoretical analysis. It is an extension of Horn logic, both syntactically and proof theoretically, and some of the terminology is borrowed from first-order predicate logic. In this paper, all logical expressions are function-free.

Definition 1 *A premise (or query) is an expression having one of the following forms:*

- A *where* A *is atomic.*
- $A[add : B]$ *where* A *and* B *are atomic.*
- $A[del : B]$ *where* A *and* B *are atomic.*

Definition 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 premise.*

Definition 3 *A hypothetical rulebased-system is a triple* $\mathcal{R} = [\mathrm{R}, \mathrm{dom}, \mathrm{pred}]$ *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, *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 4 *(Hypothetical Inference) Suppose* [R,dom,pred] *is a hypothetical rulebased-system. Then, for any database* $DB \subseteq \mathcal{DB}$, *and for all ground atomic formulas* $A, B \in \mathcal{DB}$,

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. $R, DB \vdash A[add : B]$ *if* $R, DB + \{B\} \vdash A$
4. $R, DB \vdash A[del : B]$ *if* $R, DB \sim \{B\} \vdash A$

Note that hypothetical inference is defined with respect to a particular rulebased system. Thus, whenever we write $R, DB \vdash \phi$, some system $\mathcal{R} = [R, dom, pred]$ will be understood.[6]

It is implicit in definition 4, that all the variables in a rule are universally quantified. In Horn logic, a universal quantifier that does not appear in the head of a rule can be converted to an existential quantifier in the body. This extends to hypothetical rules. For example, in the rule $A(x) \leftarrow B(x, y)[add : C(x, y)]$, the variable y does not appear in the head. If this rule were to appear in the rulebase R, then the inference rules above would imply that for any ground term x,

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

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

[6] In sections 3 through 5, *dom* and *pred* are theoretical conveniences; however, the domain *dom* becomes crucial when negation-by-failure is introduced in section 7.

$$\forall_x \forall_y \; A(x) \leftarrow B(x,y)[add : C(x,y)]$$
$$\forall_x \; A(x) \leftarrow \exists_y \; B(x,y)[add : C(x,y)]$$

It is the latter interpretation which enables our logic to represent the examples given in section 2. In particular, the rule $grad1(s) \leftarrow grad(s)[add : take(s,c)]$ defines those students who are "within one course of graduation."

Example 7. Suppose R consists of rules defining a predicate D plus the following $n+1$ rules:

$$A_1 \leftarrow A_2[add : B_1]$$
$$A_2 \leftarrow A_3[add : B_2]$$
$$\ldots$$
$$A_n \leftarrow A_{n+1}[add : B_n]$$
$$A_{n+1} \leftarrow D$$

Then $R, DB \vdash A_1$ *if* $R, DB + \{B_1, ..., B_n\} \vdash D$. Note that if R contains no other rules defining the predicates $A_1, ..., A_{n+1}$, and if they are not in the database DB, then the *if* becomes an *iff*. □

The next two examples not only illustrate hypothetical inference, they also introduce, in simplified form, the kind of rules used in section 6 to simulate Turing machines. In these two examples, the database DB is assumed to include the following atomic formulas, which define a finite sequence $a_1, a_2, ..., a_n, a_{n+1}$:

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

The first example is a recursive version of example 7.

Example 8. Suppose R consists of rules defining the predicate D plus the following three rules:

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

Then $R, DB \vdash E$ *if* $R, DB + \{B(a_1), ...B(a_n)\} \vdash D$. If R contains no other rules defining the predicates E and $A(x)$, and if they do not appear in the database, then the the *if* becomes an *iff*. □

Example 9. Suppose R consists of rules defining the predicate D plus the following four rules:

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

Then $R, DB \vdash E$ *if*

$$
\begin{array}{ll}
 & R, DB + \{B(a_1), B(a_2), ..., B(a_n)\} \vdash D \\
\textit{or} & R, DB + \{C(a_1), B(a_2), ..., B(a_n)\} \vdash D \\
\textit{or} & R, DB + \{B(a_1), C(a_2), ..., B(a_n)\} \vdash D \\
\textit{or} & R, DB + \{C(a_1), C(a_2), ..., B(a_n)\} \vdash D \\
 & \quad \cdots \\
\textit{or} & R, DB + \{C(a_1), C(a_2), ..., C(a_n)\} \vdash D
\end{array}
$$

If R contains no other rules defining the predicates E and $A(x)$, and if they do not appear in the database, then the *if* becomes an *iff*. □

The last example illustrates that a fixed set of hypothetical rules can express $O(2^n)$ meta-level queries, where n is the database size. Such "blowups" can be viewed as the source of the EXPTIME and PSPACE data-complexity bounds for our hypothetical logics.

4 Bottom-Up Inference

In this section, hypothetical inference is formulated in terms of a bottom-up, iterative operator, much like the T-operator of Horn-clause logic [3, 22]. Although operators of this kind are commonly associated with semantics, the development presented here is entirely proof theoretic and forms the theoretical basis for the algorithms developed in the next section, algorithms whose complexities are easily determined. This section develops enough theory to verify that these algorithms are correct with respect to the inference system defined in the last section.

A set of Horn rules R can be viewed as a mapping that takes a database DB and returns a larger database $T_R(DB)$, obtained by applying the rules of R to DB. From this, one can proceed to define T_R^n and the "limit" T_R^*. The meaning of R can then be taken to be the mapping that associates a database DB with the fixpoint $T_R^*(DB)$.

The situation for hypothetical rules is more complex: To apply a hypothetical rule to DB, one may need to compute $T_R^*(DB')$ first, where DB' is a hypothetical database. This computation, in turn, may require the computation of T_R^* on other databases. One might even wonder whether deadlock is possible, in which the fixpoint computations of several databases "wait" for each other. The answer to that is no: if T_R^n has been computed for all databases, then T_R^{n+1} can be computed for any database. The idea of computing T_R^n on all databases "in parallel" is nicely captured by considering T_R as operating not on databases, but on mappings. Given the identity mapping $\mathcal{I}$, we apply T_R repetitively, to obtain $T_R^n(\mathcal{I})$. The rest of this section elaborates upon this idea.

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 $\mathcal{DB}$. In the following definition, we consider mappings $\mathcal{M} : 2^{\mathcal{DB}} \rightarrow 2^{\mathcal{DB}}$.

Definition 5

- *A mapping* $\mathcal{M}$ *is inflationary iff* $DB \subseteq \mathcal{M}(DB)$ *for each database* DB.

- $\mathcal{M}_1 \leq \mathcal{M}_2$ *iff* $\mathcal{M}_1(DB) \subseteq \mathcal{M}_2(DB)$ *for each database* DB.
- $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$ *iff* $\mathcal{M}(DB) = \mathcal{M}_1(DB) \cup \mathcal{M}_2(DB)$ *for each database* DB.
- $\mathcal{I}$ *is the identity mapping. That is,* $\mathcal{I}(DB) = DB$ *for each database* DB.

Definition 6 *An interpretation is an inflationary mapping* $\mathcal{M} : 2^{\mathcal{DB}} \rightarrow 2^{\mathcal{DB}}$.

Definition 7 *(Satisfaction) Suppose* $\mathcal{M}$ *is an interpretation and* DB *is a database. If* A *and* B *are ground atomic formulas, then*

- $\mathcal{M}, DB \models A$ *iff* $A \in \mathcal{M}(DB)$
- $\mathcal{M}, DB \models A[add : B]$ *iff* $\mathcal{M}, DB + \{B\} \models A$
- $\mathcal{M}, DB \models A[del : B]$ *iff* $\mathcal{M}, DB \sim \{B\} \models A$

The following basic result about interpretations is straightforward.

Lemma 1 *If* $\mathcal{M}_1 \leq \mathcal{M}_2$ *then* $\mathcal{M}_1, \mathcal{DB} \models \phi \;\Rightarrow\; \mathcal{M}_2, \mathcal{DB} \models \phi$.

Definition 8 *Suppose* $\mathcal{R} = [R, dom, pred]$ *is a hypothetical rulebased-system. Then* T_R *is the operator that takes an interpretation as input, and returns another interpretation as output, defined as follows:* $T_R(\mathcal{M}), DB \models A$ *iff*

- $\mathcal{M}, DB \models A$ *or*
- *there is a rule* $A' \leftarrow \phi_1, ..., \phi_k$ *in* R *and a ground substitution* θ *such that* $A = A'\theta$ *and* $\mathcal{M}, DB \models \phi_i\theta$ *for each* i.

for any ground atomic formula A.

Intuitively, a mapping $\mathcal{M}$ represents a set of inferences, and $T_R(\mathcal{M})$ represents the same set of inferences extended with a single application of each rule in R. The following basic results about T_R follow immediately from lemma 1.

Lemma 2 T_R *is inflationary and monotonic. i.e.,*

- $\mathcal{M} \leq T_R(\mathcal{M})$
- *If* $\mathcal{M}_1 \leq \mathcal{M}_2$, *then* $T_R(\mathcal{M}_1) \leq T_R(\mathcal{M}_2)$.

Definition 9

- $T_R^0(\mathcal{M}) = \mathcal{M}$
- $T_R^{n+1}(\mathcal{M}) = T_R[T_R^n(\mathcal{M})]$
- $T_R^*(\mathcal{M}) = \cup_{n=0}^{n=\infty} T_R^n(\mathcal{M})$

In our hypothetical logic, the interpretation $T_R^*(\mathcal{I})$ plays a role similar to that of the least fixpoint in Horn-clause logic. In particular, hypothetical inference is equivalent to satisfaction in $T_R^*(\mathcal{I})$, as the following theorem shows.

Theorem 1 *Suppose* [R,dom,pred] *is a hypothetical rulebased-system. Then for any database $DB \subseteq \mathcal{DB}$ and any query ϕ over $\mathcal{DB}$,*

$$R, DB \vdash \phi \quad \textit{iff} \quad T_R^*(\mathcal{I}), DB \models \phi$$

This theorem is the main result of this section and is the basis of the algorithms introduced in the next section. It is proved in the following two lemmas, which establish sufficiency and neccessity, resp.

Lemma 3 *(Sufficiency) Suppose* [R,dom,pred] *is a hypothetical rulebase-system. Then for any database $DB \subseteq \mathcal{DB}$, and for any query ϕ over $\mathcal{DB}$,*

$$\textit{if} \;\; T_R^n(\mathcal{I}), DB \models \phi \;\; \textit{then} \;\; R, DB \vdash \phi$$

Proof: (by induction) There are three cases, depending on whether ϕ has the form A, $A[add : B]$ or $A[del : B]$, where A and B are atomic formulas in $\mathcal{DB}$. We prove only the last case, the other two being similar.

Basis: Suppose that $T_R^0(\mathcal{I}), DB \models A[del : B]$.

Then	$\mathcal{I}, DB \models A[del : B]$	by definition 9
	$\mathcal{I}, DB \sim \{B\} \models A$	by definition 7
	$A \in \mathcal{I}(DB \sim \{B\})$	by definition 7
	$A \in DB \sim \{B\}$	by definition 6
	$R, DB \sim \{B\} \vdash A$	by definition 4
	$R, DB \vdash A[del : B]$	by definition 4

Induction: Suppose the lemma is true for n. We show that it must then be true for $n+1$.

if	$T_R^{n+1}(\mathcal{I}), DB \models A[del : B]$	
then	$T_R^{n+1}(\mathcal{I}), DB \sim \{B\} \models A$	by definition 7
	$T_R[T_R^n(\mathcal{I})], DB \sim \{B\} \models A$	by definition 9

and according to definition 8, there are two cases:

Case (i): $T_R^n(\mathcal{I}), DB \sim \{B\} \models A$.

Thus	$T_R^n(\mathcal{I}), DB \models A[del : B]$	by definition 7.
	$R, DB \vdash A[del : B]$	by induction hypothesis.

Case (ii): There is a rule $A' \leftarrow \phi_1, ..., \phi_k$ in R and a ground substitution θ such that $A = A'\theta$ and $T_R^n(\mathcal{I}), DB \sim \{B\} \models \phi_i\theta$ for each i.

Thus	$R, DB \sim \{B\} \vdash \phi_i\theta$ for each i,	by induction hypothesis.
	$R, DB \sim \{B\} \vdash A$	by definition 4.
	$R, DB \vdash A[del : B]$	by definition 4.

□

Lemma 4 *(Neccessity) Suppose* [R,dom,pred] *is a hypothetical rulebased-system. Then for any database* $DB \subseteq \mathcal{DB}$*, and for any query* ϕ *over* $\mathcal{DB}$*,*

$$\textit{if } R, DB \vdash \phi \textit{ then } T_R^n(\mathcal{I}), DB \models \phi \textit{ for some finite } n.$$

Proof: Define $R, DB \overset{n}{\vdash} \phi$ to mean that "$R, DB \vdash \phi$" can be derived in n applications (or less) of the inference rules of definition 4. Clearly, $R, DB \vdash \phi$ iff $R, DB \overset{n}{\vdash} \phi$ for some finite n. We show by induction on n that *if* $R, DB \overset{n}{\vdash} \phi$ *then* $T_R^{n-1}(\mathcal{I}), DB \models \phi$.

Basis: If $R, DB \overset{1}{\vdash} \phi$ then ϕ is atomic and $\phi \in DB$, by inference rule 1. Therefore $\phi \in \mathcal{I}(DB) = [T_R^0(\mathcal{I})](DB)$, and so $T_R^0(\mathcal{I}), DB \models \phi$, by definition 7.

Induction: If $R, DB \overset{n+1}{\vdash} \phi$, we must show that $T_R^n(\mathcal{I}), DB \models \phi$. There are three cases, depending on whether rule 2, 3 or 4 is the last inference rule used in the derivation of "$R, DB \vdash \phi$".

Case (i): If $R, DB \overset{n+1}{\vdash} \phi$ and rule 2 is the last inference rule, then ϕ is atomic. Furthermore, there is a rule $\phi' \leftarrow \phi_1, ..., \phi_k$ in R and a ground substitution θ such that $\phi = \phi'\theta$ and $R, DB \overset{n}{\vdash} \phi_i\theta$ for each i.

Thus	$T_R^{n-1}(\mathcal{I}), DB \models \phi_i\theta$ for each i,	by induction hypothesis.
	$T_R[T_R^{n-1}(\mathcal{I})], DB \models \phi$	by definition 8.
	$T_R^n(\mathcal{I}), DB \models \phi$	by definition 9.

Case (ii): If $R, DB \overset{n+1}{\vdash} \phi$ and rule 3 is the last inference rule, then $\phi = A[add : B]$ and $R, DB + \{B\} \overset{n}{\vdash} A$.

Thus	$T_R^{n-1}(\mathcal{I}), DB + \{B\} \models A$	by induction hypothesis.
	$T_R^n(\mathcal{I}), DB + \{B\} \models A$	by definition 8.
	$T_R^n(\mathcal{I}), DB \models A[add : B]$	by definition 7.

Case (iii): If $R, DB \overset{n+1}{\vdash} \phi$ and rule 4 is the last inference rule, then $\phi = A[del : B]$ and $R, DB \sim \{B\} \overset{n}{\vdash} A$. In this case, the proof is parallel to case (ii).

□

Corollary 1 *Suppose* [R,dom,pred] *is a hypothetical rulebased-system. Then for any database* $DB \subseteq \mathcal{DB}$*, and for any query* ϕ *over* $\mathcal{DB}$*,*

$$R, DB \vdash \phi \textit{ iff } T_R^n(\mathcal{I}), DB \models \phi \textit{ for some finite } n.$$

This corollary follows immediately from the preceeding two lemmas. From this corollary, theorem 1 is straightforward.

Example 10. 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 mapping $T_R^k(\mathcal{I})$ for some k. As k increases from left to right, atoms are added to each column but never subtracted, reflecting the inflationary character of T_R. 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	AD
CD	CD+B	BCD+A	ABCD

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 therefore 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 also that to compute $T_R^2(\mathcal{I})(\{D\})$, we must compute $T_R(\mathcal{I})$ on other databases first.

Note that A is eventually inferred for all databases containing D. That is, if $D \in DB$ then $T_R^*(\mathcal{I}), DB \models A$. Thus, by theorem 1, it should also be the case that if $D \in DB$ then $R, DB \vdash A$. Using the inference rules of definition 4, it is not hard to see that this is indeed the case. □

5 Data Complexity: Upper Bounds

This section establishes upper complexity bounds for both hypothetical logics, i.e., for the full logic of additions and deletions and for the restricted logic of additions. In particular, the data complexity of the full logic is shown to be in EXPTIME, and for the restricted logic, in PSPACE.

Data complexity is the complexity of evaluating a database query when the query is fixed and the database is regarded as input. In particular, if ψ is a database query, then the data-complexity of ψ is defined to be the complexity of the language $\{\langle \overline{x}, DB \rangle \mid \overline{x} \in \psi(DB)\}$ where $\overline{x}$ is a tuple and DB is a database [6, 23].

A rulebase can be viewed as defining a database query if one of its predicates is reserved as an output relation. Let R be the rulebase, and let $OUT(\overline{x})$ be the output predicate. Then R and OUT define a database query whose data-complexity is given by the language

$$\{\langle \overline{x}, DB \rangle \mid R, DB \vdash OUT(\overline{x})\} \tag{2}$$

To be precise, hypothetical inference must be defined with respect to a particular inference system $[R, dom, pred]$, where dom and $pred$ are finite sets of constant and predicate symbols, resp. Since R is fixed, the predicates involved in inference are also fixed. A database DB may contain other predicates, but 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.

The set of constant symbols dom is not fixed, however, and depends on the database DB. Indeed, in order to reason about DB, dom must include all the constant symbols appearing in both DB and R. It is convenient to use $dom = dom_R + dom(DB)$ where $dom(DB)$ is the data domain of DB and dom_R is a fixed set of constant symbols including all those which appear in R.

Thus, a rulebase R and a predicate $OUT(\overline{x})$ define a database query. The data complexity of this query is the complexity of the language (2). For each database DB, the inferences $R, DB \vdash OUT(\overline{x})$ are defined with respect to the hypothetical inference system $[R, dom, pred]$ where $pred$ is fixed and $dom = dom_R + dom(DB)$.

5.1 EXPTIME Upper-Bound

Using the analysis of section 4, it is straightforward to show that query evaluation in the full logic can be done in exponential time. The key points are (i) that an interpretation $\mathcal{M}$ can be represented in exponential space, (ii) that each application of operator T_R requires exponential time, and (iii) that at most exponentially many applications of T_R are needed to compute $T_R^*(\mathcal{I})$.

Let $\mathcal{R} = [R, dom, pred]$ be a hypothetical rulebased-system. dom and $pred$ are finite sets of constant and predicate symbols, resp. 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. If $dom = dom_R + dom(DB)$, where dom_R is fixed, then "exponential in $|dom|$" implies "exponential in $|dom(DB)|$".

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

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

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

5.2 PSPACE Upper-Bound

This section shows that for the restricted logic of hypothetical additions, query evaluation can be done in polynomial space (polynomial in the size of the database). Essentially, one computes $[T^*_R(\mathcal{I})](DB)$ by an iterative computation much like the bottom-up, least-fixpoint computation of Horn logic. However, whenever a hypothetical rule is applied, so that one needs an answer to a query on a hypothetical database DB', one computes $[T^*_R(\mathcal{I})](DB')$ recursively. Thus, the computations of $T^n_R(\mathcal{I})$ 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 bottom-up, 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 bottom-up 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' in a bottom-up fashion 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 bottom-up 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 section 3 in the propositional case without hypothetical deletions.

1. $R, DB \vdash A$ *if* $A \in DB$
2. $R, DB \vdash A$ *if there is a rule* $A \leftarrow \phi_1, ..., \phi_n$ *in* R *such that* $R, DB \vdash \phi_i$ *for each* i.
3. *(a)* $R, DB \vdash A[add : B]$ *if* $B \in DB$ *and* $R, DB \vdash A$

   (b) $R, DB \vdash A[add : B]$ *if* $B \notin DB$ *and* $R, DB + \{B\} \vdash A$

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

Procedure: LFP(R, DB)
    $S_1 \leftarrow DB$;
    $S_2 \leftarrow$ T(R, DB);
    do until $S_1 = S_2$:
        $S_1 \leftarrow S_2$;
        $S_2 \leftarrow$ T(R, S_1);
    end;
    return(S_2);

Procedure: T(R, DB)
    $S \leftarrow DB$;
    for each rule $A \leftarrow \phi_1, ..., \phi_n$ in R do:
        if MATCH(R, DB, ϕ_j) is true for each $1 \leq j \leq n$
            then $S \leftarrow S + \{A\}$;
    end;
    return(S);

Procedure: MATCH(R, DB, ϕ)
    if ϕ is atomic and $\phi \in DB$
        then return(*true*);
    if $\phi = A[add : B]$ and $B \in DB$ and $A \in DB$
        then return(*true*);
    if $\phi = A[add : B]$ and $B \notin DB$ and $A \in$ LFP($R, DB + \{B\}$)
        then return(*true*);
        else return(*false*);

It is straightforward to show that if LFP(R, DB') are correctly computed for all $DB' > DB$, then LFP(R, DB) returns the correct answer. Furthermore, these procedures are trivially correct when DB is the largest database $\mathcal{DB}$, consisting of all possible atomic formulas. By induction, therefore, these three procedures are correct.

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 $DB = \mathcal{DB}$. 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.

6 Data Complexity: Lower Bounds

This section establishes lower complexity bounds for both hypothetical logics: the full logic of additions and deletions, and the restricted logic of additions. In particular, the data-complexity of the full logic is shown to be EXPTIME-hard, and for the restricted logic, PSPACE-hard. Recall that *data complexity* is the complexity of query evaluation when the rulebase is fixed and the database is regarded as input (section 5).

The main idea is to use each logic to encode the computations of an alternating Turing machine. Like nondeterministic machines, an alternating machine may have many possible transitions at any point in the computation. Alternating machines, however, may require that *all* machine transitions be successful, not just one. This capability gives alternating machines more power than nondeterministic machines [8].

6.1 EXPTIME Lower-Bound

This section shows that query processing for the full logic of additions and deletions is EXPTIME-hard. The idea is use the logic to encode the computations of an alternating PSPACE machine. Since APSPACE=EXPTIME, the result follows [8]. In addition, these machine encodings are central to the expressiveness result of section 8.2.

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 $R(M)$ such that

$$R(M), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s}. \tag{3}$$

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,[7] the result follows immediately. Combining this with the EXPTIME upper-bound of section 5.1 gives the following result.

Theorem 2 *The data-complexity of the full logic of hypothetical additions and deletions is EXPTIME-complete.*

The rest of this section describes the construction of the database $DB(\overline{s})$ and the rulebase $R(M)$, thereby establishing formula (3).

6.1.1 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, machine *id*'s can be represented with two predicates:

- $CELL(c, j)$: the tape cell at position j contains the symbol c.

[7] see [9] for examples of such machines.

- $HEAD(q, j)$: the control head is in state q and is scanning the cell at position j.

The initial *id* of the machine must also be specified. This means stating that the control head 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 $DB(\overline{s})$:

$$HEAD(q_0, 0),$$
$$CELL(s_0, 0),\ CELL(s_1, 1)\ ...\ CELL(s_{n-1}, n-1),$$
$$CELL(blank, n),\ CELL(blank, n+1)\ ...\ CELL(blank, n^k - 1).$$

6.1.2 Building the Rulebase $R(M)$

During inference, the rulebase $R(M)$ uses hypothetical operations to simulate the action of the control head. For instance, tape erasure is simulated with hypothetical deletion, and tape writing, with hypothetical addition.[8] 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. An alternating machine has three kinds of states: *accepting*, *universal* and *existential*. Whether an *id* is accepting or not 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

$$R(M), DB \vdash ACCEPT \quad \textit{iff} \quad DB \textit{ represents an accepting id}. \tag{4}$$

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

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

which proves theorem 2.

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

6.1.3 Defining the Predicate $ACCEPT$

This section defines the predicate $ACCEPT$ so that it satisfies formula (4). 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 introduce a distinct constant symbol q_a and add the following rule to $R(M)$:

$$ACCEPT \leftarrow HEAD(q_a, j).$$

That is, an *id* is accepting if it contains the accepting state q_a. (The variable j signifies that the head position 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 control head; j, the position of the control head; and c, the tape symbol at position j. This information is encoded in the database by the formulas $HEAD(q, j)$ and $CELL(c, j)$. For each machine transition, we delete these two formulas from the database, and add two new ones. The deletions are taken care of by a single rule:

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

where c, q and j are variables.

The rules defining the predicate $TRANS$ 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 is added to $R(M)$:

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

where the c's and q's are constant symbols 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 are added to $R(M)$:

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

where the c's and q's are constant symbols and the j's are variables. The expressions $j_1 = j + \Delta_1$ are abbreviations for $NEXT(j, j_1)$ and $NEXT(j_1, j)$, depending on whether Δ_1 is 1 or -1, resp.

With such rules, both the transition relation of the machine and the criterion for acceptance can be encoded. Thus, a set of hypothetical rules $R(M)$ can be constructed satisfying formula (4).

6.2 PSPACE Lower-Bound

This section shows that query evaluation for the restricted logic of hypothetical additions 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 [8]. The machine encodings are also central to the expressiveness result of section 8.1.

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 two predicates:

- $CELL(c, j, t)$: at time t, the tape cell at position j contains the symbol c.
- $HEAD(q, j, t)$: at time t, the control head 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 6.1.1, 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

$$R(M), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s}. \tag{5}$$

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,[9] the result follows immediately. Combining this with the PSPACE upper-bound of section 5.2 gives the following result.

Theorem 3 *The data-complexity of the restricted logic of hypothetical additions is PSPACE-complete.*

The database $DB(\overline{x})$ and the rulebase $R(M)$ are constructed much as they were in section 6.1 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.

[9] See [12] for examples of such languages.

6.2.1 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 control head 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 $DB(\overline{s})$:

$$HEAD(q_0, 0, 0),$$

$$CELL(s_0, 0, 0),\ CELL(s_1, 1, 0)\ ...\ CELL(s_{n-1}, n-1, 0),$$

$$CELL(blank, n, 0),\ CELL(blank, n+1, 0)\ ...\ CELL(blank, n^k - 1, 0).$$

This is identical to the encoding given in section 6.1.1 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.

As in section 6.1.1, $DB(\overline{s})$ also encodes a successor predicate $NEXT(i, j)$, defining a counter from 0 to $n^k - 1$.

$R(M)$ 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

$$R(M), DB \vdash ACCEPT(t) \quad \textit{iff} \quad \textit{the id at time } t \textit{ is accepting} \tag{6}$$

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 (5) can be established, thereby proving theorem 3.

6.2.2 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 6.1.3. Again, three types of rules are defined, one for each type of machine state: universal, existential and accepting.

For each accepting state, we introduce a distinct constant symbol q_a and add the following rule to $R(M)$:

$$ACCEPT(t) \leftarrow HEAD(q_a, j, t).$$

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 the variable j signifies that the head position 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 control head; j, the position of the control 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 $HEAD(q, j, t)$ and $CELL(c, j, t)$. The following rule extracts this information from the database and increments t by one:

$$ACCEPT(t) \quad \leftarrow \quad HEAD(q, j, t),\ CELL(c, j, t),\ NEXT(t, t'),\ TRANS(c, q, j, t').$$

where c, q and j are variables representing the control information, and the variable t' represents the time of the successor *id*. Notice that this rule is not hypothetical. In particular, it does not perform any deletions—in contrast to the corresponding rule of section 6.1.3.

The rules defining the predicate $TRANS$ 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 $R(M)$:

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

where the c's and q's are constant symbols, the j's are variables, and t is a variable.

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 $R(M)$:

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

where the c's and q's are constant symbols, the j's are variables, and t is a variable.

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

6.2.3 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 control head, the contents of all tape cells are unchanged. This is an instance of the *frame axiom* [14], 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.

The following rule identifies each tape cell which is not under the control head at time t_1 and passes its value on to the same cell at time $t_2 = t_1 + 1$.

$$CELL(c, j, t_2) \;\leftarrow\; NEXT(t_1, t_2),\ CELL(c, j, t_1),\ HEAD(q, j', t_1),\ NEQ(j, j').$$

This rule invokes 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 (6).

7 Negation as Failure

In this section, the language of hypothetical rules is extended by allowing negated premises. Thus rules of the form $A \leftarrow \sim B[add : C]$ are allowed. Operationally, the expression $\sim B[add : C]$ is interpreted as the failure to prove $B[add : C]$. Inference involving negation-as-failure is not always well-defined however. This is a familiar problem in Horn logic. 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. As in the Horn case, however, if there is no recursion through negation, then it is not difficult to provide an operational semantics for negation [2]. In particular, a rulebase with negation can be stratified or layered so that within each layer, a negated premise refers only to rules found in the layers below [2]. We therefore extend the familiar notion of stratification from Horn rulebases to hypothetical rulebases. The proof theory for a stratified rulebase R is developed and is shown to be independent of any particular stratification of R. It is then shown that stratified negation does not increase the data complexity of hypothetical rules.

7.1 Stratification

Let $A \leftarrow \phi_1 ... \phi_k$ be a hypothetical rule possibly with negated premises. Each premise ϕ_i thus has one of six possible forms: B, $B[add : C]$, $B[del : C]$, or the negation of any of these, where B and C 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. Goal predicates are central to the notion of a stratified rulebase. For convenience, the predicate symbol appearing in the head of a rule shall be called the head predicate.

Definition 10 *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 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.

Example 11. The following rulebase is stratified by R_0, R_1, R_2:

$$R_2 \begin{cases} A_2(x) \leftarrow B_2(x,y),\ A_2(y)[add : C_2(y)]. \\ A_2(x) \leftarrow D_2(x),\ \sim A_1(x). \end{cases}$$

$$R_1 \begin{cases} A_1(x) \leftarrow B_1(x,y),\ A_1(y)[add : C_1(y)]. \\ A_1(x) \leftarrow D_1(x),\ \sim A_0(x). \end{cases}$$

$$R_0 \begin{cases} A_0(x) \leftarrow B_0(x,y),\ A_0(y)[add : C_0(y)]. \\ A_0(x) \leftarrow D_0(x). \end{cases}$$

□

7.2 Operational Semantics

This section defines the meaning of the expression $R, DB \vdash \phi$ when R is a stratified, hypothetical rulebase. The definition is similar to that for stratified Horn rulebases; i.e., a series of bottom-up fixpoint computations are performed, one for each stratum; when each fixpoint computation is completed, negative literals are computed, and the next fixpoint computation is begun.

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 11 *Let R be a set of hypothetical rules, possibly containing negated premises. Then $\overline{R}$ is a negation-free rulebase constructed from R as follows: In every rule of R, replace each negated premise $\sim P(\overline{x})$, $\sim P(\overline{x})[add : Q]$, or $\sim P(\overline{x})[del : Q]$ by the non-negated premise $\overline{P}(\overline{x})$, $\overline{P}(\overline{x})[add : Q]$, or $\overline{P}(\overline{x})[del : Q]$, resp.*

Given a stratified rulebase R with strata $R_0, ..., R_k$, 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 section 4. In particular, given an interpretation $\mathcal{M}$, we can compute the fixpoint $T^*_{\overline{R}_i}(\mathcal{M})$.

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*; 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 4, it was defined as an inflationary mapping $\mathcal{M} : 2^{\mathcal{DB}} \rightarrow 2^{\mathcal{DB}}$. To take negative literals into account, we now define it to be an inflationary mapping $\mathcal{M} : 2^{\mathcal{DB}} \rightarrow 2^{\overline{\mathcal{DB}}}$. To capture the intended meaning of the new predicate symbols $\overline{P}$, we define the following operation on interpretations. Given an interpretation $\mathcal{M}$, 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 12 *Let $\mathcal{M}$ be an interpretation. Then $\overline{\mathcal{M}}$ is another interpretation such that for each database $DB \subseteq \mathcal{DB}$ and each ground atomic formula $P(\overline{x}) \in \mathcal{DB}$,*

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

Let R be a stratified hypothetical rulebase with stratification $R_0, ..., R_k$. The following procedure defines the operational semantics of this stratification. For brevity, we use T^*_i to stand for $T^*_{\overline{R}_i}$. 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 $\overline{\mathcal{M}}$ is used to enforce the closed world assumption between adjacent strata.

Procedure: FIXPOINT$(R_0, ..., R_k)$
  $\mathcal{M}_0 \leftarrow T^*_0(\mathcal{I})$
  $\mathcal{M}_1 \leftarrow T^*_1(\overline{\mathcal{M}}_0)$
  $\mathcal{M}_2 \leftarrow T^*_2(\overline{\mathcal{M}}_1)$
    ...
  $\mathcal{M}_k \leftarrow T^*_k(\overline{\mathcal{M}}_{k-1})$
  return$(\overline{\mathcal{M}}_k)$

This procedure has an obvious counterpart in the operational semantics of stratified Horn rulebases. In both cases, the procedure constructs a preferred fixpoint. Indeed, it is not hard to see that if $\mathcal{M}$ = FIXPOINT(R), then $\mathcal{M}$ is a fixpoint of $T_{\overline{R}}(\mathcal{M})$, i.e., $\mathcal{M} = T_{\overline{R}}(\mathcal{M})$.

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 $R_0, ..., R_k$. In the next section, however, it is shown that for a given rulebase, the value of FIXPOINT is independent of the way it is stratified. 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 13 *Let R be a stratified rulebase, and let ϕ be a hypothetical query. Then*

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

where $\mathcal{M} = \text{FIXPOINT}(R)$.

Example 12. Let R be the following stratified rulebase:

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

Note that the first stratum, R_0, is identical to the rulebase of example 10. The corresponding negation-free rulebase $\overline{R}$ then consists of the following rules, where $\overline{A}$ and $\overline{E}$ are new predicate symbols:

$$\begin{array}{ll} \overline{R}_2 & \left\{ F \leftarrow \overline{E}, E[add : C] \right. \\ \overline{R}_1 & \left\{ E \leftarrow \overline{A}, C. \right. \\ \overline{R}_0 & \left\{ \begin{array}{l} A \leftarrow B[add : C] \\ B \leftarrow C, D. \end{array} \right. \end{array}$$

The table below illustrates the computation of FIXPOINT(R). The first column shows the four possible databases DB constructible from the two predicates B and C. The remaining columns show the values of $\mathcal{M}_i(DB)$ and $\overline{\mathcal{M}}_i(DB)$ for $i = 0, 1, 2$. Thus, the second column shows the values of $[T^*_{\overline{R}_0}(\mathcal{I})](DB)$, the computation of which is shown in detail in example 10.

DB	$\mathcal{M}_0(DB)$	$\overline{\mathcal{M}}_0(DB)$	$\mathcal{M}_1(DB)$	$\overline{\mathcal{M}}_1(DB)$	$\mathcal{M}_2(DB)$	$\overline{\mathcal{M}}_2(DB)$
–	–	$\overline{A}\,\overline{E}$	$\overline{A}\,\overline{E}$	$\overline{A}\,\overline{E}$	$\overline{A}\,\overline{E} + F$	$\overline{A}\,\overline{E}F$
C	C	$\overline{A}C\overline{E}$	$\overline{A}C\overline{E} + E$	$\overline{A}CE$	$\overline{A}CE$	$\overline{A}CE$
D	$D + A$	$AD\overline{E}$	$AD\overline{E}$	$AD\overline{E}$	$AD\overline{E}$	$AD\overline{E}$
CD	$CD + AB$	$ABCD\overline{E}$	$ABCD\overline{E}$	$ABCD\overline{E}$	$ABCD\overline{E}$	$ABCD\overline{E}$

Atoms added by the rules in $\overline{R}$ are preceded by a plus sign. For instance, the $+E$ in the fourth column/second row is due to the rule $E \leftarrow \overline{A}, C$. In the columns headed $\overline{\mathcal{M}}_i(DB)$, negation is simulated. Thus, in each row of the table, the atom $\overline{A}$ is present in column 3 iff the atom A is not present in column 2. This process is to be carried out for all predicates, but for clarity we have shown it only for A and E (the only predicates which appear negated in R.) Note that in row 2, the atom $\overline{E}$ is present in columns 3 and 4 but is removed in column 5, after the atom E is inferred. This "removal" of barred predicates reflects the non-monotonic nature of negation-as-failure.

It is instructive to note the $+F$ in the sixth column/first row, which is due to the rule $F \leftarrow \sim E, E[add : C]$. This rule emphasizes the hypothetical nature of this kind of inference. The premise of the rule states that both $\sim E$ and E must be true, but not on the same database. In particular, to infer F on a database DB, one must be able to infer $\sim E$ on DB, and to infer E on $DB + C$. This is precisely the case in the table above when $DB = \{\}$. $\Box$

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 data domain: in this case, $A = dom \sim B$. It is because of unguarded rules that an explicit domain *dom* is needed in the definition of a rulebase system (section 3). The main results of this paper, however, do not depend upon a need to know *dom*, as the rules presented in this paper are all guarded.

7.3 Stratification Independence

In this section, it is shown that the value of the procedure FIXPOINT is independent of the stratification used to compute it. The development presented here is an adaptation of that given in [2] for the semantics of stratified Horn rulebases. First, we divide a stratified rulebase 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.

Definition 14 *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 15 *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. Thus, each stratum consists of a disjoint union of clusters. Similar results were established for Horn rules in [2].

Clusters are a convenient device for hiding recursion from our theoretical development. Indeed, 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 16 *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, then if R_j depends on R_i, then R_j is a higher stratum that R_i. The following basic result about dependency is immediate.

Lemma 5 *Let $S_0, S_1, ..., S_n$ be subsets of a stratified rulebase R. Then,*

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

Definition 17 *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 6 *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 which is closely related to the T-operator.

Definition 18 *Let R be a rulebase with stratification $R_0, ..., R_k$. If S is a subset of some stratum R_i, then $U_S(\mathcal{M}) = \overline{T^*_{\overline{S}}(\overline{\mathcal{M}})}$ for all interpretations $\mathcal{M}$.*

Given an interpretation $\mathcal{M}$, the operator U_S first applies the closed world assumption to $\mathcal{M}$; it then applies the rules in $\overline{S}$ to $\mathcal{M}$ in a bottom-up fashion to saturation; and finally it applies the closed world assumption to the result. U_S thus generalizes the treatment that the procedure FIXPOINT gives to a single stratum; i.e., U_S can be applied to an arbitrary interpretation $\mathcal{M}$, and the rule set S need not be an entire stratum. U_S is thus a convenient theoretical tool for describing the behavior of rule sets within a stratum. It is also closely related to the procedure FIXPOINT. In particular, if R is stratified by $R_0, R_1 ..., R_k$, then

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

The following basic results about U-operators are straightforward.

Lemma 7 *Let R be a rulebase with stratification $R_0, ..., R_k$, and let S_1 and S_2 be subsets of some stratum R_i. If $S_1 \not\prec S_2$ then $U_{S_1 \cup S_2} \;=\; U_{S_2} U_{S_1}$.*

Corollary 2 *Let R be a rulebase with stratification $R_0, ..., R_k$. Let S_1 be a subset of some stratum R_i, and let S_2 be a subset of some stratum R_j. If $S_1 \not\prec S_2$ and $S_2 \not\prec S_1$ then $U_{S_1} U_{S_2} \;=\; U_{S_2} U_{S_1}$.*

Lemma 7 allows us to break down 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 need a basic result about clusters.

Definition 19 *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.

Lemma 8 *A set of clusters can be arranged in a linear sequence.*

Proof: By lemma 6, the dependency graph of the clusters is acyclic. Thus we can 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 was printed first. We claim that 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 Q_i follows Q_j in the sequence (i.e., $i > j$). Thus, if Q_i precedes Q_j, then Q_i does not depend on Q_j; i.e., if $i < j$ then $Q_i \not\prec Q_j$. Hence the sequence is linear. □

Because of this lemma, each stratum can be replaced by a linear sequence of the clusters that make it up, where each each cluster is now viewed as a stratum in its own right. The following theorem and its corollary makes this idea precise.

Theorem 4 *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: $Q_0 \not\succ Q_i$ for each $1 \leq i \leq n$, since $Q_0, Q_1, ..., Q_n$ is linear. Thus

$$\begin{array}{ll} Q_0 \not\succ Q_1 \cup \cdots \cup Q_n & \text{by lemma 5.} \\ U_{Q_0 \cup Q_1 \cup \cdots \cup Q_n} \; = \; U_{Q_1 \cup \cdots \cup Q_n} U_{Q_0} & \text{by lemma 7.} \\ U_{Q_0 \cup Q_1 \cup \cdots \cup Q_n} \; = \; U_{Q_n} \cdots U_{Q_1} U_{Q_0} & \text{by induction.} \\ U_{R_i} \; = \; U_{Q_n} \cdots U_{Q_1} U_{Q_0} & \text{since } R_i = Q_0 \cup Q_1 \cup \cdots \cup Q_n. \end{array}$$

□

Corollary 3 *Let R be a rulebase with stratification $R_0, ..., R_k$. For each i, let $Q_i^1 ... Q_i^{n_i}$ be the clusters of stratum R_i arranged in a linear order. Then $Q_1^1 ... Q_1^{n_1}, ..., Q_k^1 ... Q_k^{n_k}$ is a stratification of R. Furthermore,*

$$\text{FIXPOINT}(R_0, ..., R_k) \; = \; U_{Q_k^{n_k}} \cdots U_{Q_1^1}(\mathcal{I})$$

Thus, every stratification of a rulebase R can be reduced to a cluster stratification, in which each stratum consists of a single cluster. Furthermore, this cluster stratification is equivalent to the original stratification in the sense that they both result in the same value of FIXPOINT. Thus, to show that any two stratifications of R are equivalent, it suffices to show that any two cluster stratifications are equivalent.

Actually, instead of dealing with cluster stratifications, we shall deal with linear sequences of clusters. This is justified by the next lemma. Linearity is a more convenient concept to deal with theoretically because unlike stratification, its definition does not involve negation.

Lemma 9 *Let $Q_0, ..., Q_n$ be the clusters of a stratified rulebase R. Then $Q_0, ..., Q_n$ is a stratification of R iff it is a linear sequence.*

Theorem 5 *Let $\mathcal{L} = Q_0, ..., Q_n$ and $\mathcal{L}' = Q'_0, ..., Q'_n$ be two linear orderings of a set of clusters. 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+1} occurs before Q_i in the ordering $\mathcal{L}'$. Then $Q_{i+1} \not\succ Q_i$ since $\mathcal{L}'$ is linear. Thus, if Q_i and Q_{i+1} are swapped in $\mathcal{L}$, then the resulting sequence is linear. In addition, because $\mathcal{L}$ is linear, $Q_i \not\succ Q_{i+1}$. Thus $U_{Q_{i+1}} U_{Q_i} \; = \; U_{Q_i} U_{Q_{i+1}}$ by corollary 2. Thus, after swapping Q_i and Q_{i+1}, the resulting sequence is equivalent to $\mathcal{L}$; 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. □

The following is the main result of this section. It is an immediate consequence of corollary 3 and theorem 5, and it shows that the procedure FIXPOINT is a function only of R, and not of its stratification. We can therefore write FIXPOINT(R) instead of FIXPOINT($R_0, ... R_k$).

Corollary 4 *Let R be a stratified rulebase, and let $R_0, ..., R_k$ and $R'_0, ..., R'_{k'}$ be any two stratifications of R. Then*

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

7.4 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. Ditto for computing the negation-free rule sets $\overline{R}_0, ..., \overline{R}_k$.

Next, recall from section 5.1 that an interpretation $\mathcal{M}$ 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 $\overline{\mathcal{M}}_i$ from $\mathcal{M}_i$ can be performed in exponential time. Furthermore, the argument given in section 5.1 shows that given $\overline{\mathcal{M}}_i$, $T^*_{\overline{R}_i}(\overline{\mathcal{M}}_i)$ can also be computed in exponential time. 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 5.2. These procedures are defined below for the propositional case, the predicate case being a straightforward generalization. In comparing these procedures to those in section 5.2, note that each procedure has a new argument i, and that a fourth procedure, COMP(S), has been defined. Note also that the procedures use not the rules in R, but the negation-free rules in $\overline{R}$.

```
Procedure: LFP(i, R, DB)
   if  i = 0  then  S1 ← DB;
               else  S1 ← LFP(i − 1, R, DB);
   S2 ← T(i, R, DB);
   do until S1 = S2:
      S1 ← S2;
      S2 ← T(i, R, S1);
   end;
```

return[COMP(S_2)];

Procedure: T(i, R, DB)
$S \leftarrow DB$;
for each rule $A \leftarrow \phi_1, ..., \phi_n$ in $\overline{R}_i$ do:
if MATCH(i, R, DB, ϕ_j) is true for each $1 \leq j \leq n$
then $S \leftarrow S + \{A\}$;
end;
return(S);

Procedure: MATCH(i, R, DB, ϕ)
if ϕ is atomic and $\phi \in DB$
then return(*true*);
if $\phi = A[add : B]$ and $B \in DB$ and $A \in DB$
then return(*true*);
if $\phi = A[add : B]$ and $B \notin DB$ and $A \in$ LFP($i, R, DB + \{B\}$)
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 5.2. There are three 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 definition of T(i, R, DB), which uses the negation-free rules in $\overline{R}_i$. Second, the initialization of the computation is different. In the definition of LFP(i, R, DB), the variable S_1 is initialized not to DB, but to LFP($i-1, R, DB$). Only in the base case ($i = 0$) is S_1 initialized to DB. Third, the procedure LFP returns COMP(S_2) instead of S_2. COMP is a simple procedure which when given $\mathcal{M}(DB)$, returns $\overline{\mathcal{M}}(DB)$.

For each value of i, LFP(i, R, DB) $= \overline{\mathcal{M}}_i(DB)$, where $\mathcal{M}_i$ is the i^{th} intermediate interpretation in the computation of FIXPOINT(R). In particular, LFP(k, R, DB) $= \mathcal{M}(DB)$, where $\mathcal{M} =$ FIXPOINT(R). Thus, for all atoms A,

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

by definition 13.

It is easy to see that LFP(i, R, DB) is correct. For the first stratum R_0, the computations are virtually the same as for non-stratified rulebases (section 5.2). Thus LFP($0, R, DB$) $= \overline{\mathcal{M}}_0(DB)$ for all DB. Now suppose that LFP($i-1, R, DB'$) $= \overline{\mathcal{M}}_{i-1}(DB')$ for all databases DB'. Then given an oracle for these values, the computations for evaluating LFP(i, R, DB)

are virtually the same as for non-stratified rulebases. Thus, LFP$(i, R, DB) = \overline{\mathcal{M}}_i(DB)$ for any database DB. Hence by induction, LFP(i, R, DB) is correct for all i and all DB.

To check the data complexity, we extend the argument used in section 5.2. As before, LFP is a recursive function in which each level of recursion takes polynomial time. Now, however, there are two ways in which LFP(i, R, DB) invokes itself recursively: (i) through the function MATCH, and (ii) through its own procedure body, by calling LFP$(i-1, R, DB)$. With each recursive call, the database DB and the integer i change monotonically, and in particular, DB is non-decreasing and i is non-increasing. Furthermore, with each recursive call, exactly one of them changes: either i is decremented by 1, or a new tuple is added to DB. However, i can change at most k times before it reaches 0, and DB can change at most $|\mathcal{DB}|$ times before it saturates and equals $\mathcal{DB}$. Thus, there are at most $k + |\mathcal{DB}|$ 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. Thus, 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. Thus, in combination with our earlier results, we have established the following two theorems.

Theorem 6 *The data-complexity of the restricted stratified logic of hypothetical additions is PSPACE-complete.*

Theorem 7 *The data-complexity of the full stratified logic of hypothetical additions and deletions is EXPTIME-complete.*

8 Expressibility

In this section, we treat our hypothetical logic as a query language for relational databases, and we ask, "what relational queries can it express?" Without negation, the logic is *monotonic*; i.e., the set of inferences increases monotonically as entries are added to the database. There are therefore many simple queries which the negation-free logic cannot express, such as relational algebra queries involving complementation. Adding negation-as-failure to the logic makes it non-monotonic and capable of expressing many of these queries. In fact, this section shows that stratified hypothetical rulebases are expressively complete for EXPTIME and PSPACE. That is, any typed generic query which can be computed in exponential time can be expressed as a stratified rulebase of hypothetical additions and deletions. Similarly, any typed generic query computable in polynomial space can be expressed as a stratified rulebase of hypothetical additions. Furthermore, this expressibility is achieved without assuming that the domain of the database is linearly ordered: If there is no *a-priori* order on the domain, then an order can be chosen non-deterministically and asserted hypothetically. Because the queries are generic, it does not matter which linear order is chosen.

No matter how complex a monotonic logic may be, there are many simple queries which it cannot express. Consider for instance the query $EVEN$ which determines the parity of a database relation D: It 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. Thus, there is no 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 if R is a set of hypothetical rules, a set of Datalog rules, or even a set of Horn rules with function symbols, as long as it is negation-free.[10] It is in this sense that the expressibility of negation-free rulebases is limited.

Adding negation-as-failure to a logic increases its expressibility, but only to a certain extent. For instance, although Datalog is more expressive with negation, it still cannot express the query $EVEN$ [6, 7]. Thus, although its data-complexity is P-complete, there are some simple queries in P which it cannot express. This limitation can be overcome, however, by assuming that the data domain is linearly ordered. Indeed, under this assumption, stratified Datalog expresses all the typed generic queries in P [13, 23, 7]. This section establishes comparable results for stratified hypothetical rulebases, but for higher complexity classes and without the assumption of a linearly ordered domain.

The following example shows a stratified hypothetical rulebase which expresses the query $EVEN$. 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 rulebase "flips back and forth" between the two subqueries $EVEN$ 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 negation is essential for detecting an empty relation. Also note that it does not matter in which order the elements of D are removed; every order gives the same answer: either every order results in a proof of $EVEN$ or every order results in a proof of ODD. These ideas of order independence and of detecting an empty relation are central to the proofs of expressibility later in this section.

Example 13. Suppose R is the following collection of rules:

$$\begin{aligned}
&EVEN \leftarrow D(x),\ ODD[del: D(x)].\\
&ODD \leftarrow D(x),\ EVEN[del: D(x)].\\
&EVEN \leftarrow \sim SOMELEFT.\\
&SOMELEFT \leftarrow D(x).
\end{aligned}$$

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

$EVEN$ can also be expressed in the restricted logic of stratified hypothetical additions. The trick is to notice that the use of deletion above is particularly simple. In fact, it can be simulated by a combination of addition and negation as follows: (i) instead of deleting elements from the relation D, they are inserted into a new relation ND; and (ii) instead of selecting undeleted elements from D, elements are selected from the difference relation $D \sim ND$. In this way, the deletion of D from the database is simulated. The following rules do exactly this.

Example 14. Suppose R is the following collection of rules:

[10] In the case of Horn rules, the expression $R, DB \vdash EVEN$ should be read as $R + DB \vdash_c EVEN$, where $\vdash_c$ denotes classical inference.

$$\begin{aligned}
&EVEN \leftarrow SELECT(x),\ ODD[add: ND(x)].\\
&ODD \leftarrow SELECT(x),\ EVEN[add: ND(x)].\\
&EVEN \leftarrow \sim SOMELEFT.\\
&SOMELEFT \leftarrow SELECT(x).\\
&SELECT(x) \leftarrow D(x),\ \sim ND(x).
\end{aligned}$$

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

In this example, it was possible to simulate hypothetical deletion by a combination of negation and hypothetical addition. In general, however, this is not possible (unless PSPACE=EXPTIME).

The rest of this section shows that the two stratified hypothetical logics are expressively complete for EXPTIME and PSPACE, resp. The proofs rely on the simulations of Turing machines developed in the section 6. In this respect, they are similar to other expressibility proofs in the literature (e.g., [13, 23]). 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 control 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.

Informally, a query is *generic* iff renaming the constants in the database causes the constants in the answer to be renamed in the same way [5, 6]. This is called the *consistency criterion*, and it formalizes the idea that the constant symbols in a database are uninterpreted. In our machine simulations, changing the linear order is equivalent to renaming the database constants. This is why our machine encodings are insensitive to the particular linear order being used. In this way, the consistency criterion is central to our ability to use unordered domains.

Before establishing these expressibility results, we give a precise definition of what it means for a hypothetical rulebase R to express a database query. This in turn requires precise definitions of "relational database" and "database query". The first two definitions below are essentially those of [5] and [6].

Definition 20 *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.

Definition 21 *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: if DB' can be derived from DB by renaming constant symbols,[11] then $\psi(DB')$ can be derived from $\psi(DB)$ by the same renaming.*

With these two definitions, we can give a precise definition of expressibility:

Definition 22 *Suppose that ψ is a generic 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.

By this definition, negation-free rulebases cannot express the query $EVEN$, whereas stratified rulebases can.

8.1 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, we proof the following theorem.

Theorem 8 *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.*

The proof of this theorem involves an 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.

The rulebase $R(\varphi)$ that we construct will contain constant symbols that are not necessarily in the domain of the database. In order to isolate the constants in the data domain from these extra constants, we assume that the data domain is accessible via a predicate D. This assumption, which is common in the literature, also permits us to strengthen the statement of theorem 8 somewhat. In particular, if a domain predicate is available, then we can guarantee that the rulebase $R(\varphi)$ consists entirely of guarded rules. In addition, by using a domain predicate, we can construct $R(\varphi)$ so that theorem 8 is valid even if a naive top-down inference procedure is used, one which has no knowledge of the domain, such as the Prolog-style interpreter developed by McCarty [19].

Even so, the assumption of a data-domain predicate is not always needed, since the data domain is often defined as the set of constant symbols appearing in the database. In this case, D can be constructed from the database predicates $P_1, ..., P_m$ by adding the following rules to $R(\psi)$ for each i:

[11] In this case we say that DB and DB' are *isomorphic*. See [5] for a more precise definition.

$$\begin{aligned} D(x_1) &\leftarrow P_i(x_1, x_2 ... x_{\alpha_i}). \\ D(x_2) &\leftarrow P_i(x_1, x_2 ... x_{\alpha_i}). \\ &\cdots \\ D(x_{\alpha_i}) &\leftarrow P_i(x_1, x_2 ... x_{\alpha_i}). \end{aligned}$$

With this said, we prove theorem 8 by first reducing it to the problem of expressing yes/no queries, that is, queries which return either *true* or *false*. In fact, the following lemma is the main result of this section.

Lemma 10 *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 $\overline{\alpha}$,*

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

where YES is a predicate symbol of arity zero.

Given this lemma, theorem 8 can be proved in a straightforward way as follows:

Proof: (of theorem 8) 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 $\langle\alpha_1, ..., \alpha_m\rangle$,

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

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 10. The rulebase $R(\varphi)$ of theorem 8 is then constructed by adding to $R(\psi)$ the following rule:

$$Q(\overline{x}) \leftarrow D_{\alpha_0}(\overline{x}),\ YES[add : P_0(\overline{x})].$$

where $D_{\alpha_0}(\overline{x})$ is an abbreviation for $D(x_1), ..., D(x_{\alpha_0})$ and D is the domain predicate. This rule generates all possible α_0-tuples $\overline{x}$ over $dom(DB)$. 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 $\overline{x} \in [dom(DB)]^{\alpha_0}$,

$$\begin{aligned} R(\varphi), DB \vdash Q(\overline{x}) \quad &\textit{iff} \quad R(\psi), DB + P_0(\overline{x}) \vdash YES \\ &\textit{iff} \quad \psi[DB + P_0(\overline{x})] = \textit{true} \\ &\textit{iff} \quad \overline{x} \in \varphi(DB) \end{aligned}$$

□

Thus, theorem 8 is true if lemma 10 is true. The rest of this section is thus devoted to the proof of lemma 10.

If ψ is a yes/no query computable in PSPACE, then there is an APTIME machine M which accepts the language $\{DB \mid \psi(DB) = \textit{true}\}$. Section 6.2 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 6.2 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 $R'(M)$. The rest of this section describes the construction of $R'(M)$.

8.1.1 Asserting 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 common in the literature [13, 23], especially when expressibility results are established in terms of complexity classes, as in lemma 10. With hypothetical rules, however, this assumption is unnecessary. Indeed, this section provides a set of hypothetical rules which "guess" a linear order and hypothetically add it to the database. 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 13 and 14.

The rules below use the techniques of example 14 to guess and hypothetically assert a linear ordering of the data domain. From the perspective of top-down inference, the rules non-deterministically select elements from the domain, one after another. As each element is selected, it is used to hypothetically extend a linear order. Each element is also hypothetically "removed" from the domain, where "removal" is simulated by a combination of insertion and negation-as-failure. 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:

$$FIRST_1(a_1),\ NEXT_1(a_1, a_2),\ NEXT_1(a_2, a_3)\ \ldots\ NEXT_1(a_{n-1}, a_n),\ LAST_1(a_n). \quad (7)$$

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

$$YES \leftarrow SELECT(x),\ ORDER(x)[add: FIRST_1(x),\ ND(x)].$$

$$ORDER(x) \leftarrow SELECT(y),\ ORDER(y)[add: NEXT_1(x, y),\ ND(y)].$$

$$ORDER(x) \leftarrow \sim SOMELEFT,\ ACCEPT[add: LAST_1(x)].$$

$$SOMELEFT \leftarrow SELECT(y).$$

$$SELECT(y) \leftarrow D(y),\ \sim ND(y).$$

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 6.2. As we shall see, if M is generic, then either $ACCEPT$ is inferred for all linear orders, or for none. Therefore, YES is inferred iff M accepts its input.

8.1.2 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.

8.1.3 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 6.2, we use the following two predicates to represent the machine, where $\overline{p}$ and $\overline{t}$ are k-tuples:

- $CELL(c, \overline{p}, \overline{t})$: at time $\overline{t}$, the tape cell at position $\overline{p}$ contains the symbol c.
- $HEAD(q, \overline{p}, \overline{t})$: at time $\overline{t}$, the control head 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 control head. Of these two tasks, initializing the control head is by far the easiest. It can be accomplished by a single rule which says that at time 0, the head is in state q_0 and is scanning the tape cell at position 0:

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

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 a predicate $INITIAL(c, \overline{p})$, of arity $k+1$, which is true iff the tape cell at position $\overline{p}$ is to be initialized to symbol c, where $c \in \{0, 1, blank\}$. For clarity, this is done in several steps, utilizing several rule sets.

The first rule set computes the bit-map representation of each database relation P_i, defining $m+1$ new predicates $BITMAP_i$, of arity $\alpha_i + 1$, resp.

$$BITMAP_i(1, \overline{x}) \leftarrow P_i(\overline{x}).$$
$$BITMAP_i(0, \overline{x}) \leftarrow D_{\alpha_i}(\overline{x}),\ \sim P_i(\overline{x}).$$

where $D_{\alpha_i}(\overline{x})$ is an abbreviation for $D(x_1), ..., D(x_{\alpha_i})$. Thus, D_{α_i} extracts all possible α_i-tuples of the domain elements.

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 control head.

Each bit-map is stored at the low end of the 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 rule defines the predicate $BLOCK_i$ of arity $\gamma + 1$, which represents the i^{th} block of tape:

$$BLOCK_i(z, u, ..., u, \overline{x}) \leftarrow FIRST_1(u),\ BITMAP_i(z, \overline{x}).$$

The blocks are placed in consecutive order at the beginning of the machine tape. That is, 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.[12] The following rules encode the blocks onto the machine tape. There is one rule for each block, and they all contribute to defining the predicate *INPUT*, of arity $k+1$, which represents the tape.

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

The literal $DOM_i(y)$ is true iff 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:

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

The relation *INPUT* encodes 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 predicate *INITIAL*, of arity $k+1$, which represents the complete initialization of the machine tape. $INITIAL(c,\overline{p})$ is true iff the tape cell at position $\overline{p}$ initially contains the symbol c, where $c \in \{0,1,blank\}$.

$$INITIAL(z,\overline{p}) \leftarrow INPUT(z,\overline{p}).$$
$$INITIAL(blank,\overline{p}) \leftarrow D_k(\overline{p}),\ \sim ZERO\_OR\_ONE(\overline{p}).$$
$$ZERO\_OR\_ONE(\overline{p}) \leftarrow INPUT(0,\overline{p}).$$
$$ZERO\_OR\_ONE(\overline{p}) \leftarrow INPUT(1,\overline{p}).$$

where $D_k(\overline{p})$ is an abbreviation for $D(p_1),...,D(p_k)$.

Finally, the initial tape contents are just the tape contents at time 0:

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

Given an APSPACE machine M, we add the rules of this section and those of sections 8.1.1 and 8.1.2 to the rulebase $R(M)$ described in section 6.2. The collection of all these rules is called $R'(M)$. It is a stratified rulebase of hypothetical additions. We now show that for an appropriate choice of M, $R'(M)$ can act as the rulebase $R(\psi)$ of lemma 10.

8.1.4 Behavior under Renaming

Section 8.1.3 described a scheme for encoding a database DB as the initial *id* of a Turing machine. In this section, we focus not on the initial *id* per se, but on the string of 1's, 0's and *blank*'s which is encoded on the machine tape. In particular, we look at how this string changes when the linear order is changed and the constants in the database are renamed.

Given a database DB, let L be a set of atomic formulas representing a linear order on the data domain, such as formulas (7) in section 8.1.1. The rules of the previous two sections define a mapping s which takes a set $DB+L$ as input and returns a string $s(DB+L)$ as output. It is not hard to see that this mapping is generic. That is, if the constant symbols

[12] This encoding assumes that the data domain contains at least $m+1$ elements. If the number i is encoded in binary, however, 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.

in $DB + L$ are renamed, then the string returned by s does not change. (Note that constant symbols from the data domain do not appear in the string.) Indeed, the rules of sections 8.1.2 and 8.1.3 were carefully designed to have this property.

Formally, a renaming is a one-to-one mapping f of the data domain onto itself. A renaming can be extended to tuples, relations and databases in the obvious way. Since s is a generic mapping,

$$s(DB + L) = s(fDB + fL)$$

for any renaming f. Thus, 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 database $DB = \{P(a, b),\ P(b, b),\ Q(b)\}$. In this case the data domain is $\{a, b\}$ and there are two possible linear orderings, $a < b$ and $b < a$. The encodings of this database under these two linear orders are shown in diagrams 1 and 2 below. In each diagram, the sequence of 1's, 0's and *blank*'s represents the encoding of the database as a string, and the entries beneath show the interpretation given to each position in the string.[13] Each string is a possible input for Turing machine M.

1. Encoding the database $\{P(b, a),\ P(b, b),\ Q(b)\}$ under the linear order $a < b$:

0	0	1	1	0	1	*blank*	*blank*
$P(a,a)$	$P(a,b)$	$P(b,a)$	$P(b,b)$	$Q(a)$	$Q(b)$	–	–

2. Encoding the database $\{P(b, a),\ P(b, b),\ Q(b)\}$ under the linear order $b < a$:

1	1	0	0	1	0	*blank*	*blank*
$P(b,b)$	$P(b,a)$	$P(a,b)$	$P(a,a)$	$Q(b)$	$Q(a)$	–	–

3. Encoding the database $\{P(a, b),\ P(a, a),\ Q(a)\}$ under the linear order $b < a$:

0	0	1	1	0	1	*blank*	*blank*
$P(b,b)$	$P(b,a)$	$P(a,b)$	$P(a,a)$	$Q(b)$	$Q(a)$	–	–

Diagrams 1 and 2 show clearly that changing just the linear order changes the encoding of the database. Diagrams 1 and 3 show that renaming both the database and the linear order in the same way does not change the encoding.

8.1.5 Expressive Completeness

This section uses the above observation to complete the proof of lemma 10. That is, it is shown that a rulebase of the form $R'(M)$ can be constructed to express any generic yes/no query which is computable in polynomial space.

Recall that the rules in $R'(M)$ first "guess" a linear order L and insert it into the database hypothetically; then they encode the database DB as a string $s(DB + L)$ suitable

[13] According to section 8.1.3, each block should contain eight tape cells, the final four being blank. For brevity, we have shown only four tape cells per block.

for input to a Turing machine; finally they simulate machine M on this input. The atom $ACCEPT$ is inferred iff M eventually halts and accepts this input. That is,

$$\begin{array}{rll} R'(M), DB + L \vdash ACCEPT & \textit{iff} & M \textit{ accepts } s(DB + L) \\ & \textit{iff} & M[s(DB + L)] = \textit{true} \end{array}$$

where, in the second line, we are treating M as a mapping from strings to boolean values. This *iff* statement is true for any APTIME machine M.

The rules in $R'(M)$ do not guess a single linear order; rather, they guess every possible linear order, and they infer the atom YES iff $ACCEPT$ is inferred for at least one of them. That is, they infer YES iff $M[s(DB + L)]$ is true for some L. Formally,

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

where L ranges over the set of all linear orders.

Now pick a particular linear order L_0. Given any yes/no query ψ computable in polynomial space, there is an APTIME machine M for which $M[s(DB + L_0)] = \psi(DB)$. That is, $\psi(DB)$ is true iff machine M accepts DB (when encoded onto the input tape using the linear order L_0). Furthermore, it is not difficult to see that

$$\{L \mid L \textit{ is a linear order}\} \;=\; \{fL_0 \mid f \textit{ is a renaming}\}$$

Therefore,

$$\begin{array}{lll} & R'(M), DB \vdash YES & \\ \textit{iff} & \vee_L\, M[s(DB + L)] & \text{where } L \text{ ranges over all linear orders,} \\ \textit{iff} & \vee_f\, M[s(DB + fL_0)] & \text{where } f \text{ ranges over all renamings,} \\ \textit{iff} & \vee_f\, M[s[f^{-1}(DB + fL_0)]] & \text{since } s \text{ is generic,} \\ \textit{iff} & \vee_f\, M[s(f^{-1}DB + L_0)] & \\ \textit{iff} & \vee_f\, \psi(f^{-1}DB) & \text{by the definition of } M. \end{array}$$

Notice that $R'(M)$ does not express the query ψ; rather, it expresses the query $\vee_f\, \psi(f^{-1}DB)$. However, if ψ is generic, then its boolean output value does not change when the constants in the database are renamed; that is, $\psi(fDB) = \psi(DB) = \psi(f^{-1}DB)$ for any renaming f. Therefore, $\vee_f\, \psi(f^{-1}DB) \;=\; \vee_f\, \psi(DB) \;=\; \psi(DB)$. Hence,

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

Thus, if ψ is a generic yes/no query, then it is expressed by the stratified hypothetical rulebase $R'(M)$. Setting $R(\psi) = R'(M)$ thus establishes lemma 10.

8.2 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. 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 complicated. Section 6.1 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 8.1, we would simply write Horn rules such as the following:

$$HEAD(q_0, \overline{p}) \leftarrow FIRST(\overline{p}).$$
$$CELL(c, \overline{p}) \leftarrow INITIAL(c, \overline{p}).$$

These rules invoke the predicate $INITIAL(c, \overline{p})$, which is true iff the cell at position $\overline{p}$ is to be initialized to c. The problem with such rules is that the formulas $HEAD(q_0, \overline{p})$ and $CELL(c, \overline{p})$ are not part of the database, but are inferred. Unfortunately, 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 8.1.1 have generated a linear order. They first initialize the control head. 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 section 6.1.

$$\begin{aligned}
ACCEPT &\leftarrow FIRST(\overline{x}),\ INITIAL(c, \overline{x}), \\
&\qquad TAPE(\overline{x})[add : CELL(c, \overline{x}),\ HEAD(q_0, \overline{x})]. \\
TAPE(\overline{x}) &\leftarrow NEXT(\overline{x}, \overline{y}),\ INITIAL(c, \overline{y}), \\
&\qquad TAPE(\overline{y})[add : CELL(c, \overline{y})]. \\
TAPE(\overline{x}) &\leftarrow LAST(\overline{x}),\ SIMULATE.
\end{aligned}$$

These rules combined with the constructions of section 8.1 give the following result.

Theorem 9 *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.*

9 Conclusions

Two modal-like operators have been introduced, $B[add : C]$ and $B[del : C]$, which represent hypothetical addition and deletion, resp. Intuitively, $B[add : C]$ means, "inserting C into the database allows B to be inferred," and $B[del : C]$ means, "deleting C from the database allows B to be inferred." An extension of Horn-clause logic was then defined in which these operators appear in the premises of rules. For instance, the rule $A \leftarrow B[add : C]$ intuitively means, "infer A if inserting C allows the inference of B." Several examples were given of the use of this logic in formulating queries to a rulebase system.

The bulk of the paper was devoted to defining hypothetical inference in the function-free, predicate case and establishing its complexity and expressibility. Two cases were examined: (i) the full logic of hypothetical additions and deletions, and (ii) a restricted logic of hypothetical additions only. It was shown that the data-complexity of the full logic is EXPTIME-complete, and that for the restricted logic, it is PSPACE-complete. The complexity comes from the ability of these logics to encode alternating Turing-machines; APSPACE and APTIME machines, in particular.

The form of hypothetical deletion that was considered is relatively simple in that it deletes entries from the database, not from the least fixpoint. Deletions from the least fixpoint are potentially more complex since they would have to be "back propagated" through the rules in order to maintain consistency. This paper has completely avoided this problem by considering deletions from the database only. Nevertheless, these relatively simple deletions increase the complexity of the logic from PSPACE to EXPTIME.

The rule language was then extended to allow negated premises. Since it is difficult to describe the meaning of such rules if recursion is allowed through negation, a notion of stratification was developed, comparable to that of stratified Horn rulebases. A proof theory was developed, and it was shown that hypothetical inference is independent of how the rulebase is stratified. It was also shown that stratified negation does not increase the data complexity of either the full or restricted logics.

Although negation-as-failure does not increase the complexity of hypothetical rules, it does increase their expressibility. For one thing, it allows non-monotonic queries to be expressed. More significantly, it was shown that stratified hypothetical rulebases are expressively complete for PSPACE and EXPTIME. That is, stratified rulebases of hypothetical additions can express any typed generic query in PSPACE; and stratified rulebases of hypothetical additions and deletions can express any typed generic query in EXPTIME.

Acknowledgements

The work of Thorne McCarty on the intuitionistic semantics of embedded implications was the original stimulus for this work. Discussions with Tomasz Imielinski also provided direction and insight. Thanks go to Eric Allender and Jan Chomicki for helpful comments on the original version of this paper. Thanks also go to the anonymous reviewers of this paper for their thoughtful and detailed comments.

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] 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.

[3] K.R. Apt and M.H. Van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[4] A.J. Bonner. A Logic for Hypothetical Reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 480–484, Saint Paul, MN, August 21–26 1988. Published in expanded form as Technical Report TR-DCS-230, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

[5] A.K. Chandra and D. Harel. Computable Queries for Relational Databases. *Journal of Computer and System Sciences (JCSS)*, 21(2):156–178, 1980.

[6] 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.

[7] 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.

[8] A.K. Chandra, D. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[9] A.K. Chandra and L.J. Stockmeyer. Alternation. In *Proceedings of the Symposium on the Foundations of Computer Science (FOCS)*, pages 98–108, 1976.

[10] D.M. Gabbay. N-Prolog: an Extension of Prolog with Hypothetical Implications. II. Logical Foundations and Negation as Failure. *Journal of Logic Programming (JLP)*, 2(4):251–283, 1985.

[11] 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.

[12] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[13] N. Immerman. Relational Queries Computable in Polynomial Time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 147–152, 1982.

[14] R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[15] R.E. Ladner. The Computational Complexity of Provability in Systems of Propositional Modal Logic. *SIAM Journal of Computing*, 6(3):467–480, 1977.

[16] 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.

[17] Sanjay Manchanda. *A Dynamic Logic Programming Language for Relational Updates.* PhD thesis, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York, December 1987. Also published as Technical Report

TR 88-2, Department of Computer Science, The University of Arizona, Tuscon, Arizona 85721, January, 1988.

[18] L.T. McCarty. Clausal Intuitionistic Logic. I. Fixed-Point Semantics. *Journal of Logic Programming (JLP)*, 5(1):1–31, 1988.

[19] L.T. McCarty. Clausal Intuitionistic Logic. II. Tableau Proof Procedures. *Journal of Logic Programming (JLP)*, 5(2):93–132, 1988.

[20] D. Miller. A Logical Analysis of Modules in Logic Programming. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 106–114, Sept 1986.

[21] R. Statman. Intuitionistic Propositional Logic is Polynomial-Space Complete. *Theoretical Computer Science (TCS)*, 9(1):67–72, 1979.

[22] 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.

[23] M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 137–146, 1982.

[24] D.S. Warren. Database Updates in Pure Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 244–253, 1984.