Hypothetical Datalog: Complexity and Expressibility*

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

bonner@db.toronto.edu

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. To establish expressibility, we augment the logic with *negation-by-failure* and view it as a query language for relational databases. The logic of hypothetical additions then expresses all database queries which are computable in PSPACE. When deletions are included, the logic expresses all database queries computable in EXPTIME.

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 [13, 12, 22, 26, 19, 4]. Others have done as much by investigating the intuitionistic semantics of embedded implications [20, 21]. However, the complexity and expressibility of these logics has remained an open question. 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 additions, hypothetical deletions have received little attention in the logic-programming literature, one exception being the work of Manchanda and Warren [19]. 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).

Hypothetical queries are particularly useful in the context of rule-based systems and deductive databases. For example, one might ask, "If I were to take *eng*423,

*This paper appears in the *Proceedings of the Second International Conference on Database Theory (ICDT)*, pages 144–160. Volume 326 of Lecture Notes in Computer Science. Springer Verlag. Conference held August 1988 at Bruges, Belgium.

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

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

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. In particular, the act contains rules such as the following: "You are eligible for citizenship if your father would be eligible if he were still alive" [13]. Rules such as these are called *embedded implications* [20].

Embedded implications have an intuitionistic semantics. That is, their models are Kripke structures, consisting of many possible substates [11]. McCarty has developed an intuitionistic fixpoint-semantics for embedded implications and has shown that they have interesting semantical properties analogous to the unique minimal-model property of Horn clauses [20, 3]. In particular, he has shown that they have a unique maximal Kripke-model, which in turn, has a unique minimal substate. It is also known that in the propositional case, intuitionistic validity is PSPACE-complete [23, 18]. To our knowledge, however, we are the first to address the questions of the data-complexity and expressibility of intuitionistic reasoning.

In contrast to additions, hypothetical deletions do not appear to have an intuitionistic semantics. The problem is that, like most logics, intuitionistic logic has no concept of database; it sees only the logical closure of the database and the rulebase. Thus, deletions from the database make no sense in intuitionistic logics. One alternative is to view our logic as a dynamic logic of database updates. In fact, our notation for hypothetical updates is reminiscent of dynamic logic, with additions and deletions being the primitive actions [14, 19]. This paper introduces a semantics for hypothetical updates which in some sense is "midway" between that of dynamic and intuitionistic logic.

Semantics are introduced, however, mainly as a vehicle for establishing upper complexity bounds. We define a notion of *rulebase interpretation* and introduce a *fixpoint operator*, much like the T-operator of Horn-clause logic [3, 24]. A bottom-up, iterative procedure is developed for computing the "least fixpoint" of a database and a set of rules. It is easy to show that this procedure runs in exponential time.

For the restricted logic of hypothetical additions, a mixed top-down/bottom-up algorithm is developed. One way to view this algorithm is as a top-down, recursive algorithm in which bottom-up, least-fixpoint computations occur at each level of recursion. There are polynomially many levels of recursion and each level takes polynomial time. Thus, we have an algorithm which runs in polynomial space.

In this way, EXPTIME and PSPACE upper-bounds on the data-complexity of query processing are established for the full logic and its restriction, resp. To establish these as lower bounds, we use the logics to encode the computations of alternating Turing-machines [9]. 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 possible only to encode alternating PTIME machines. This is why hypothetical additions have a lower complexity than hypothetical additions and deletions combined.

The issue of expressibility is also addressed. In particular, we view the two logics as query languages for relational databases; and we ask, "what class of database queries do these logics express?" Because they are monotonic, there are many simple queries that these logics cannot express; so we first augment them with *negation-by-failure* [17]. When augmented in this way, the full logic expresses all database queries which are computable in exponential time; and the restricted logic expresses all database queries computable in polynomial space.

The proofs rely on our simulation of alternating Turing-machines. In this respect they are similar to other expressibility proofs in the literature [16, 25]. One difference, however, is that we do not require the data domain to be linearly ordered. Linearly-ordered domains are used to simulate counters, which in turn, are used to simulate tape-head movements. Our approach is to start with unordered domains and assert linear orders hypothetically. This, however, involves some novel technical problems.

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 rule-based system $[R, DB]$. The next section will be more precise about this. This section provides examples of hypothetical queries and how they are expressed in our logic.

The examples are centered on a rule-based 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

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

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]

[3] See [17] for a description of meta-level and object-level reasoning.

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:

$$R, DB + take(tony, cs452) \vdash grad(tony)$$

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

Example 2. "If Tony had not taken eng101, would 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:

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

At the object-level, it is represented by the expression $grad(tony)[del : take(tony, eng101)]$.

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

Example 4. "Retrieve those students who have taken $eng201$ but who could have graduated if they had taken $eng250$ instead." i.e., at the meta-level,

$$R, DB \vdash take(s, eng201) \quad \text{and}$$
$$R, DB + take(s, eng250) \sim take(s, eng201) \vdash grad(s)$$

and at the object-level,[4]

$$take(s, eng201) \ \wedge$$
$$grad(s)\ [add : take(s, eng250)]\ [del : take(s, eng201)]$$

[4] Here we use the convention that $A[B][C]$ means $(A[B])[C]$.

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

Example 5. 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 hypothetical rule can be represented by adding the following two rules to the rulebase:[5]

$$aid2(s) \leftarrow \sim aid1(s),\ grad1(s).$$
$$grad1(s) \leftarrow \exists c, grad(s)[add : take(s,c)]$$

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.

3 Hypothetical Inference

Having provided examples of hypothetical queries and rules, this section now defines a logic for expressing them. First its syntax is defined, and then its meaning is given in terms of an inference system.[6] We use the syntax and terminology of Horn-clause logic augmented with with modal-like operators. In particular, an expression is called *atomic* iff it is an atomic formula of predicate logic. Although the rest of this paper focusses on the function-free case, the definitions of this section do not require this restriction.

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.*

Notice that hypothetical rules include atomic formulas and definite Horn-rules as special cases. This syntax is therefore an extension of that of Horn logic[17]. The inference system too is an extension of Horn logic, as the following definition shows.

Definition 3 *Suppose* R *is a set of hypothetical rules and* DB *is a database. Then hypothetical inference is defined as follows, where* A *and* B *are ground atomic formulas:*

1. $R, DB \vdash A$ *if for some rule* $A' \leftarrow \phi_1, ..., \phi_k$ *in* R, *and for some substitution* θ *such that* $A = A'\theta$, *it is the case that* $R, DB \vdash \phi_i\theta$ *for each* i.
2. $R, DB \vdash A$ *if* $A \in DB$
3. $R, DB \vdash A[add : B]$ *if* $R, DB + \{B\} \vdash A$
4. $R, DB \vdash A[del : B]$ *if* $R, DB \sim \{B\} \vdash A$

It is implicit in the above definition, that all variables in a rule are universally quantified; however, variables that appear in the premise, but not the head of a rule admit a special interpretation. 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 inference rule 1 would imply that for any ground term x, it is the case that $R, DB \vdash A(x)$ if $R, DB + \{C(x,y)\} \vdash B(x,y)$ for some ground term y. For this reason, this rule can be interpreted in two equivalent ways:[7]

- $\forall x \forall y\ A(x) \leftarrow B(x,y)[add : C(x,y)]$

[5] The first rule uses negation-by-failure [17].

[6] Semantics are touched upon only briefly in this paper; but see [4] for a proof that the hypothetical logic of additions has an intuitionistic semantics.

[7] Similar interpretations also exist in Horn-clause logic.

- $\forall x\ A(x) \leftarrow \exists y B(x,y)[add : C(x,y)]$

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 6. Suppose R contains rules for defining a predicate D and also contains the following rules:

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

Then $R, DB \vdash A_1$ if $R, DB + \{B_1, ..., B_n\} \vdash D$.

In the next two examples, we assume the database DB includes the following atomic formulas, which define a finite sequence $a_1, a_2, ..., a_n, a_{n+1}$:

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

Example 7. Suppose R contains rules defining the predicate D and also contains the following three rules:

$$\begin{array}{l} E \leftarrow FIRST(x),\ A(x). \\ A(x) \leftarrow NEXT(x,y),\ A(y)[add : B(x)]. \\ A(x) \leftarrow LAST(x),\ D. \end{array}$$

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

Example 8. Suppose R contains rules defining the predicate D and also contains the following four rules:

$$\begin{array}{l} E \leftarrow FIRST(x),\ A(x). \\ A(x) \leftarrow NEXT(x,y),\ A(y)[add : B(x)]. \\ A(x) \leftarrow NEXT(x,y),\ A(y)[add : C(x)]. \\ A(x) \leftarrow LAST(x),\ D. \end{array}$$

Then $R, DB \vdash E$ if

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

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, 24]. This view of inference forms the basis of algorithms developed in the next section, algorithms whose complexities are easily determined. This section develops enough theory to verify that these algorithms are sound and complete with respect to the inference system defined in the last section.

We first introduce a rudimentary semantics in which a rulebase is viewed as a mapping $\mathcal{M}$ which takes a database DB as input and returns a larger database $\mathcal{M}(DB)$ as output. For instance, the identity mapping $\mathcal{I}$ corresponds to the empty rulebase, which when given a database DB, does not add any inferences to it. More generally, a set of Horn rules is naturally interpreted as a mapping which takes a database and returns a least fixpoint. Justify this semantics on philosophical grounds, however, is not the purpose of this paper. Our motive is purely pragmatic: this semantics is a convenient vehicle for establishing upper complexity-bounds.

Given a rulebase R, we define an operator T_R which represents "one step" inferences. This operator takes a mapping $\mathcal{M}$ as input and returns another mapping $T_R(\mathcal{M})$ as output. For instance, $T_R(\mathcal{I})$ is the mapping which when given a database DB, adds those atomic formulas which can be derived from R and DB in one step. Similarly, $T_R^k(\mathcal{I})$ is the mapping which when given a database DB, adds those atomic formulas which can be derived from R and DB in k or fewer steps.

The rest of this section makes these ideas precise. We shall assume that the databases DB are built from a universal domain of constant and predicate symbols and that all mappings $\mathcal{M}$ are defined on these databases.

Definition 4 *A rulebase interpretation is a mapping $\mathcal{M}$ which takes a database DB and returns a larger database $\mathcal{M}(DB)$. That is, $DB \subseteq \mathcal{M}(DB)$. In addition,*

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

Definition 5 *Suppose $\mathcal{M}$ is a rulebase interpretation and DB is a database. If A and B are atomic, then*

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

Definition 6 *Suppose R is a set of hypothetical rules. Then T_R is an operator which takes a rulebase interpretation $\mathcal{M}$ as input and returns a rulebase interpretation $T_R(\mathcal{M})$ as output. Furthermore, if DB is a database, then $A \in [T_R(\mathcal{M})](DB)$ iff $A \in \mathcal{M}(DB)$ or*

> *for some rule $A' \leftarrow \phi_1, ..., \phi_k$ in R, and for some substitution θ such that $A = A'\theta$, it is the case that $\mathcal{M}, DB \models \phi_i$ for each i.*

Loosely speaking, $\mathcal{M}$ represents a set of inferences, and $T_R(\mathcal{M})$ adds to $\mathcal{M}$ those inferences which can be derived from R in one step. Of course, the exact inferences performed depend on the database, which is why $\mathcal{M}$ and $T_R(\mathcal{M})$ are functions of DB.

Lemma 1 *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 7 *If $\mathcal{M}$ is a rulebase interpretation, then*

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

From these definitions, the following theorem is straightforward (see [5] for details). It is the basis of the algorithms introduced in the next section.

Theorem 1 *$R, DB \vdash \phi$ iff $T_R^*(\mathcal{I}), DB \models \phi$.*

Example 9. Suppose R contains exactly two rules, $B \leftarrow C$ and $A \leftarrow B[add : C]$. The following table shows that it takes two applications of 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 form left to right, atoms are added to each column, but never subtracted, reflecting the inflationary character of T_R.

DB	$[T_R(\mathcal{I})](DB)$	$[T_R^2(\mathcal{I})](DB)$	$[T_R^3(\mathcal{I})](DB)$
–	–	*A*	A
A	A	A	A
B	*A*B	AB	AB
C	*B*C	*A*BC	ABC
AB	AB	AB	AB
AC	A*B*C	ABC	ABC
BC	*A*BC	ABC	ABC
ABC	ABC	ABC	ABC

Atoms added by T_R have been highlighted. For instance, the B in the second column/fourth row was added by the Horn rule $B \leftarrow C$. This in turn caused the addition of A to the third column/first row by the rule $A \leftarrow B[add : C]$. Formally, the first application of T_R establishes that $T_R(\mathcal{I}), \{\} \models B[add : C]$, and the second establishes that $T_R^2(\mathcal{I}), \{\} \models A$.

Note that A is eventually inferred for all databases; that is, $T_R^*(\mathcal{I}), DB \models A$ for all DB. Thus, by theorem 1, it should also be the case that $R, DB \vdash A$ for all DB. Using the inference rules of section 3, it is not hard to see that this is indeed the case.

5 Complexity: Upper Bounds

This section establishes upper complexity bounds both for the full logic of hypothetical additions and deletions and for the restricted logic of hypothetical additions. In particular, the data complexity for the full logic is in EXPTIME, and for the restricted logic, it is in PSPACE. *Data complexity* is the complexity of query processing when the database is regarded as input but the arity of all relations is bounded and the rulebase is fixed [25]. For purposes of presentation, we restrict the analysis to the propositional case, though the arguments generalize to the function-free predicate case in a straightforward way.

5.1 EXPTIME Upper-Bound

Using the analysis of section 4, it is straightforward to show that query processing in the full logic can be done in exponential time. The key points are (i) that a hypothetical 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 $T_R^*(\mathcal{I})$.

Suppose that R is a set of hypothetical rules and DB is a database. Let $\mathcal{L}$ be the set of atoms appearing in R and DB. We represent an interpretation $\mathcal{M}$ of R by the set $\{< DB', \mathcal{M}(DB') > \mid DB' \subseteq \mathcal{L}\}$. Indeed, 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 $\mathcal{M} \leftarrow \{< DB', DB' > \mid DB' \subseteq \mathcal{L}\}$. 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. Since there are $2^{|\mathcal{L}|}$ subsets of $\mathcal{L}$, our representation of $\mathcal{M}$ uses exponential space (exponential in the size of $DB \cup R$). Thus, each application of the operator T_R can be done in exponential time. By lemma 1, each such application makes $\mathcal{M}$ larger. Thus, during each application, except the last, there must be some element $< DB', S >$ of $\mathcal{M}$ in which S acquires a new atom. But, for each DB', this can happen at most n times, where $n = |\mathcal{L}|$. And since there are 2^n values of DB', no more changes will be possible after $n \cdot 2^n$ operator applications. Thus $T_R^*(\mathcal{I})$ is computed after exponential time.

Given $T_R^*(\mathcal{I})$ and a query ϕ, one can then use definition 5 to determine in exponential time whether $T_R^*(\mathcal{I}), DB \models \phi$. Thus, for the full logic of hypothetical additions and deletions, the complexity of query processing is in EXPTIME in the propositional case.

This argument is easily generalized to the function-free, predicate case. In this case, however, we are interested in data-complexity, in which the arity of all predicates is bounded. The hypothetical databases DB' are built from the predicate symbols and constant symbols appearing in DB and R. Because of the bounded arity, the number of possible tuples is polynomial in the size of the data domain; so the number of hypothetical databases DB' is still exponential. Hence, the data-complexity of the full logic is in EXPTIME in the function-free, predicate case.

5.2 PSPACE Upper-Bound

This section shows that for the restricted logic of hypothetical additions, query processing can be done in polynomial space. In particular, because there are no hypothetical deletions, a query ϕ to a rulebase system $[R, DB]$ is answered by referring only to hypothetical databases which contain DB. We exploit this fact to write a mixed top-down/bottom-up procedure $LFP(R, DB)$ which computes $[T_R^*(\mathcal{I})](DB)$ and which runs in polynomial space. This procedure has a single top-down, recursive phase; and at each level of recursion, a bottom-up, iterative computation is performed. Each bottom-up phase computes $LFP(R, DB')$ for some database DB' containing DB. Because there are no hypothetical deletions, DB' grows larger with each level of recursion.

Each bottom-up phase of LFP is analogous to the bottom-up, least-fixpoint computations of Horn-clause logic. 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 head of the rule is added to S iff the premise of the rule is satisfied.

To test the premise of a hypothetical rule, LFP calls itself recursively, invoking another bottom-up phase. For instance, to apply 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 bottom-up fashion. If B is inferred during this phase, then $R, DB \vdash B[add : C]$, and the premise of the hypothetical rule is satisfied.

At each level of recursion, the bottom-up phase takes polynomial time (as in Horn-clause logic). Furthermore, the depth of recursion is polynomial, since only polynomially-many distinct tuples can be added to DB. The entire computation, therefore, is performed in polynomial space.

This description is made precise by the three procedures given below, which for clarity, are restricted to the propositional case. They 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 for some rule $A \leftarrow \phi_1, ..., \phi_n$ in R, it is the case that $R, DB \vdash \phi_i$ for each i.

2. $R, DB \vdash A$ if $A \in DB$

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 1 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 until $[T^*_R(\mathcal{I})](DB)$ has been computed. Together, LFP and T execute a single bottom-up phase. Inference rules 2 and 3 are implemententby the procedure $MATCH(R, DB, \phi)$, 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)
  S1 ← DB;
  S2 ← T(R, DB);
  do until S1 = S2
    S1 ← S2;
    S2 ← T(R, S1);
  return(S1);
  end

Procedure: T(R, DB)
  S ← DB;
  for each rule A ← φ1, ..., φn in R do
    if MATCH(R, DB, φi) is true for each i
      then S ← S + {A};
  return(S);
  end

Procedure: MATCH(R, DB, φ)
  if φ is atomic and φ ∈ DB
    then return(true);
  if φ = A[add : B] and B ∈ DB
                    and A ∈ DB
    then return(true);
  if φ = A[add : B] and B ∉ DB
                    and A ∈ LFP(R, DB + {B})
    then return(true)
    else return(false);
  end
```

It is straightforward to show that if $LFP(R, DB')$ are correctly computed for all $DB' > DB$, then $LFP(R, DB)$ will return the correct answer. 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 the Horn-clause case. In fact, $MATCH$ calls LFP recursively; but this recursion can go only to depth n, where n is the number of distinct atoms in R. Thus, there are $O(n)$ levels of recursion, each taking polynomial time; so the entire computation is in PSPACE.

6 Complexity: Lower Bounds

This section establishes lower complexity bounds both for the full logic of hypothetical additions and deletions and for the restricted logic of additions. In particular, the data-complexity for the full logic is EXPTIME-hard, and for the restricted logic, it is PSPACE-hard. Recall that data-complexity is the complexity of query processing when the database is regarded as input but the arity of all relations is bounded and the rulebase is fixed [25].

The main idea is to use our 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 [9].

6.1 EXPTIME Lower-Bound

We first show that query processing for the full logic of hypothetical additions and deletions is EXPTIME-hard. Our strategy is to use the logic to encode the computations of an alternating PSPACE machine. Since APSPACE=EXPTIME, the result follows [9]. In addition, these machine encodings are central to the EXPTIME expressiveness result of section 7.

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

$$R(M), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s} \qquad (1)$$

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

The important point is that the rulebase $R(M)$ is independent of the input $\overline{s}$. This allows us prove that the data-complexity of query processing is EXPTIME-hard. Indeed, let M be an APSPACE machine which recognizes an EXPTIME-complete language[8]. The result then follows immediately. Combining this with the EXPTIME upper-bound of section 5, we have the following result:

Theorem 2 *For the full logic of hypothetical additions and deletions, the data-complexity of query processing is EXPTIME-complete in the function-free, predicate case.*

6.1.1 Simulating the Machine

The rest of this section establishes formula (1).

Since M is an APSPACE machine, its computations use n^k tape cells for some integer k, where n is the length of the input. A counter is needed to represent these n^k positions, which is easily implemented by placing the following entries in the database $DB(\overline{s})$:

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

Given this counter, we can represent machine *id*'s with the following predicates:

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

We need to specify the initial *id* of the machine. This means stating that the control is in state q_0, that the tape head is scanning the first tape cell, and that the input is the string $\overline{s} =< s_0, s_1, ..., s_{n-1} >$. We must, therefore, put the symbol s_i in the i^{th} tape cell and *blank* in the rest. This is done by placing the following entries in the database $DB(\overline{s})$:

$$\begin{array}{ll} HEAD(q_0, 0), & \\ CELL(s_0, 0), & CELL(blank, n), \\ CELL(s_1, 1), & CELL(blank, n+1), \\ \dots & \dots \\ CELL(s_{n-1}, n-1), & CELL(blank, n^k-1). \end{array}$$

[8] see [10] for examples of such machines.

We must also simulate the read, write and move operations of the machine's control head. This is done by hypothetically adding and deleting entries from the database. That is, we construct a logic program which in moving from one hypothetical database to another, simulates the transition of the machine M from one *id* to another. In particular, at any point in the inference, the database represents an *id* of the machine. This is comparable to the use by Abiteboul and Vianu, of a database update language to simulate non-deterministic Turing machines [1].

Every machine *id* is either *accepting* or *rejecting*. Furthermore, since M is an alternating Turing machine, it has three kinds of states: *accepting*, *universal* and *existential*. Whether an *id* is accepting or not is then determined 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.

This criterion for acceptance can be encoded as a set of hypothetical rules. In particular, we add rules to $R(M)$ so that when DB encodes an *id*,

$R(M), DB \vdash ACCEPT$ iff DB encodes an accepting *id*.

where $ACCEPT$ is a 0-ary predicate.

We define three types of rules, one for each type of machine state. The simplest type is for those *id*'s which have an accepting state. Specifically, for each accepting state, we introduce a 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 the machine M. In particular, 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 that the control head is reading. 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:

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

where c, q and j are variables.

$TRANS(c, q, j)$ is a predicate which completes the machine transition by adding the new formulas to the database. This depends on the details of the transition relation. Suppose that given c and q, the machine has k possible transitions. Suppose that the i^{th} such transition is given by the triple $< c_i,\ \Delta_i,\ q_i >$, which means *write the symbol* c_i*, move* Δ_i *squares to the right, and go into state* q_i, where Δ_i may be 1, 0 or -1.

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

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

where the c's and q's are constant symbols and the j's are variables.

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 $R(M)$:

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

We can write such rules for each pair of tape-symbols c and control-states q. In this way, we encode both the transition relation of the machine and the criterion for acceptance. That is, we can construct a set of hypothetical rules $R(M)$ such that $R(M), DB \vdash ACCEPT$ iff DB encodes an accepting *id*.

By definition, the machine M accepts its input iff the initial *id* is accepting. Since the initial *id* is represented by the database $DB(\overline{s})$, it follows that $R(M), DB(\overline{s}) \vdash ACCEPT$ iff M accepts the input $\overline{s}$. This establishes formula (1), thereby proving theorem 2.

6.2 PSPACE Lower-Bound

This section shows that query processing 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 [9]. These machine encodings are also central to the PSPACE expressiveness result of section 7.

The main difference between the encodings in this section and the last is that we now represent time explicitly. That is, because tape-erasure can no longer be simulated by database deletion, we introduce a time parameter t into many of the predicates. In particular, we represent machine *id*'s as follows:

- $CELL(c, j, t)$: 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.

As before, we store a successor predicate $NEXT(i, j)$ in the database so that we can count from 0 to $n^k - 1$, where n is the length of the machine input. This time, however, we count computation steps (time) as well as tape positions (space). Thus we can simulate only those machines which halt in at most n^k steps, that is, machines which run in polynomial *time*.

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

$$R(M), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s} \qquad (2)$$

Again, it is important that the rulebase $R(M)$ is independent of the machine input $\overline{s}$. This allows us to prove that the data-complexity of query processing is PSPACE-hard. Indeed, let M be an APTIME machine which recognizes a PSPACE-complete language.[9] The result then follows immediately. Combining this with the PSPACE upper-bound of section 5 gives the following result:

Theorem 3 *For the restricted logic of hypothetical additions, the data-complexity of query processing is PSPACE-complete in the function-free, predicate case.*

6.2.1 Simulating the Machine

Because we are representing time explicitly, we can now store more than one machine *id* in the database. In fact, we can store a sequence of *id*'s, that is, a computation *path*. In general, an alternating Turing machine has a computation *tree*, containing many such paths. The rulebase $R(M)$ explores this tree one path at a time, asserting each path hypothetically, and then retracting it.

[9] See [15] for examples of such languages.

As before, the transition relation of the machine M is encoded as a set of hypothetical rules. This time, however, there is one set of hypothetical additions for each transition, and no hypothetical deletions. For each transition, the rules add a new machine *id* to the database, so that the computation path is "grown" one *id* at a time.

The rules are a simple variation of those in section 6.1 (see [5] for details). They define a predicate $ACCEPT(t)$ which 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

$R(M), DB \vdash ACCEPT(t)$ iff the last *id* in the path is accepting.

As before, we encode the input string $\overline{s}$ in the database $DB(\overline{s})$ as the initial *id* of the machine. That is, $DB(\overline{s})$ represents a computation path of length 1 ending at time 0. Thus, machine M accepts input $\overline{s}$ iff $R(M), DB(\overline{s}) \vdash ACCEPT(0)$.

We can now establish formula (2) by adding the rule $ACCEPT \leftarrow ACCEPT(0)$ to $R(M)$. This, in turn, proves theorem 3.

7 Expressibility

We have shown that the data-complexity of the full logic of hypothetical additions and deletions is EXPTIME-complete, and that for the restricted logic of hypothetical additions, it is PSPACE-complete. This is the price we pay in worst-case performance, for the ability to reason about hypothetical database updates. What do we get for this price? In [4] it is shown that there are low-complexity, hypothetical queries that cannot be expressed in classical logic unless one assumes the user has full knowledge of the rulebase. Thus we get the ability to express certain hypothetical queries in a rulebase-independent way. This section, however, treats these logics as query languages for a relational database and establishs the class of queries that they express.

Section 5 showed that any database query expressible in the full logic is computable is EXPTIME. This section establishes a converse: by augmenting the full logic with *negation-by-failure* [17, 12], any database query computable in exponential time can be expressed. Similarly, by augmenting the restricted logic with negation-by-failure, any database query computable in polynomial space can be expressed.

The proofs rely on the simulations of Turing machines as developed in the section 6. In this respect, they are similar to other expressibility proofs in the literature (e.g., [16, 25]). One difference, however, is that we do not require the domain of the database to be linearly ordered. Linearly-ordered domains are used to simulate counters, which in turn, are used to simulate the movement of the control head. Our approach is to start with unordered domains and assert linear orders hypothetically. This, however, involves some interesting technical problems.

The main problem is that we cannot select and assert a particular linear order. There is no *a priori* ordering on the domain, and so there is no way for a rulebase to prefer one ordering over another. We can, however, assert all possible linear-orders, one after another, and simulate the Turing machine for each one. This is sufficient as long as the Turing machine simulation is insensitive to the particular linear-order being used. To prove this, we exploit the definition of generic "database queries" as given by Chandra and Harel [6, 7]. This definition requires that a query satisfy a consistency criterion: if the constants in the database are renamed in a consistent way, then the constants in the answer to a query must be renamed in the same way. In our machine simulations, changing the linear order is equivalent to renaming the database constants. Hence, it does not matter which linear order is used. In this way, the consistency criterion is central to our ability to use unorderd domains.

Before proceeding, we define the notions of relational database and of database query. The following definitions are esentially those given in [6] and [7].

Definition 8 *Let U be a countable set, called the universal data domain. A relational database DB of type $\overline{\alpha} = (\alpha_1, ..., \alpha_m)$ is a tuple $(D, R_1, ..., R_m)$ 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 9 *A database query of type $\overline{\alpha} \rightarrow \beta$ is a partial function ψ which takes a database DB of type $\overline{\alpha}$ and returns a relation $\psi(DB)$ over $dom(DB)$ of arity β. ψ must satisfy the following consistency criterion: if DB' can be derived from DB by renaming constant symbols,[10] then $\psi(DB')$ can be derived from $\psi(DB)$ by the same renaming.*

7.1 Negation by Failure

It is clear that in order to express all EXPTIME-computable queries, our logic must be augmented in some way. This is because the logic is *monotonic*: as entries are added to the database, inferences do not disappear. Clearly many low-complexity queries are non-monotonic, such as relational-algebra queries involving complementation. Adding negation-by-failure to a logic makes it non-monotonic as well.

Let R be a hypothetical rulebase, and let DB be a database. *Negation-by-failure* is then defined by adding the following inference rule to those of definition 3:

$$R, DB \vdash \sim \phi \quad \textit{if} \quad R, DB \not\vdash \phi$$

Unfortunately, such inferences are not always well-defined when negation appears in the premise of rules which are mutually recursive. This is a familiar problem in Horn-clause logic [2]. For example, given the two rules $A \leftarrow \sim B$ and $B \leftarrow \sim A$ and an empty database, it is not clear whether A is to be inferred, or B, or both, or neither. However, as in the Horn-clause case, as long as there is no recursion through negation, there is no ambiguity. In this paper, we restrict our attention to such "stratified" rulebases.

In the following example, negation-by-failure is used to determine the parity of a monadic relation D. The first two rules select and remove elements of D from the database one-by-one. As elements are removed, the rulebase "flips back and forth" between two subqueries, $EVEN$ and ODD, until the relation D is empty. The third rule states that the empty relation has even parity.

It does not matter in which order the elements of D are removed. Every order will give the same answer: either every order will result in a proof of $EVEN$ or every order will result in a proof of ODD.

Example 10. Suppose R is the following collection of rules:

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

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

$EVEN$ is clearly a non-monotonic query, since as elements are added to D, the predicate $EVEN$ alternates between true and false. Thus, $EVEN$ cannot be expressed in our hypothetical logic without negation-by-failure. It is also known, however, that $EVEN$ cannot be expressed in Horn-clause logic even with negation-by-failure [7, 8].

Using a slightly more sophisticated use of negation-by-failure, $EVEN$ can be implemented in the restricted logic of 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

[10] In this case we say that DB and DB' are *isomorphic*. See [6] for a more precise definition.

addition and negation as follows: (i) instead of deleting the element $D(x)$, add the element $ND(x)$; and (ii) instead of selecting an element $D(x)$ which has not been deleted, select an element $D(x)$ for which $ND(x)$ has not been added. In this way, the deletion of $D(x)$ from the database can be simulated.

Example 11. Suppose R is the following collection of rules:

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

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

7.2 PSPACE Expressibility

This section shows that any database query that can be computed in polynomial space can be expressed in our logic of hypothetical additions augmented with negation-by-failure. i.e.,

Theorem 4 *Suppose that ψ is a database query of type $\overline{\alpha} \rightarrow \beta$ and that DB is a database of type $\overline{\alpha}$. Suppose also that DB has a monadic predicate D(x) which defines the data domain. If ψ is computable in polynomial space, then there is a rulebase $R(\psi)$ of hypothetical additions with negation-by-failure such that*

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

where Q is a predicate symbol of arity β.

This theorem assumes that the data domain is accessible via a predicate D. Although this is not necessary, it is a much weaker assumption than that of a linearly ordered domain. Furthermore, for many purposes, the data domain can be 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:

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

To prove theorem 4, suppose we have a database query ψ which is computable in in polynomial space. Then there is a PSPACE Turing-machine which computes it. That is, if DB is encoded on the input tape, then the machine generates an encoding of $\psi(DB)$ on its output tape after consuming (at most) a polynomial amount of tape.

This machine generates all tuples in $\psi(DB)$; that is, it solves the *generation problem* for ψ. Related to this is the *recognition problem*, recognizing whether a particular tuple is in $\psi(DB)$. It is easy to transform the generation problem into the recognition problem: given a tuple $\overline{x}$ and a database DB, generate the relation $\psi(DB)$, and return *true* iff $\overline{x} \in \psi(DB)$. Since the arity of the output relation $\psi(DB)$ is fixed, there are polynomially many tuples $\overline{x}$ that need to be tested (polynomial in $|dom(DB)|$). Thus, given a PSPACE machine which solves the generation problem for ψ, there is another PSPACE machine which solves the recognition problem. This machine recognizes the language $\{< \overline{x}, DB > \mid \overline{x} \in \psi(DB)\}$.

Since PSPACE=APTIME, there is an alternating PTIME machine M which also recognizes this language [9]. Our main result is to construct a rulebase $R(M)$ which solves the recognition problem for ψ by simulating the computations of M. Specifically, we construct $R(M)$ so that for all databases DB of type $\overline{\alpha}$,

$$\begin{aligned} R(M), DB + \{P_0(\overline{x})\} \vdash A \\ \textit{iff} \quad M \ \textit{accepts} \ < \overline{x}, DB > \end{aligned} \tag{3}$$

where P_0 is a predicate symbol of arity β.

Given this, it is straightforward to augment $R(M)$ with rules so that it can solve the generation problem for ψ. In particular, we add the following rule to $R(M)$ to form $R(\psi)$:

$$Q(x_1 ... x_\beta) \leftarrow D(x_1), ...\ D(x_\beta), A[add : P_0(x_1 ... x_\beta)].$$

This rule generates all possible β-tuples $\overline{x}$ over $dom(DB)$. For each such tuple, $P_0(\overline{x})$ is hypothetically added to the database and then $< \overline{x}, DB >$ is tested for acceptance by M. Thus, we have the following:

$$
\begin{aligned}
& R(\psi), DB \vdash Q(\overline{x}) \\
& \quad \textit{iff} \quad R(M), DB + \{P_0(\overline{x})\} \vdash A \\
& \quad \textit{iff} \quad M \; accepts \; < \overline{x}, DB > \\
& \quad \textit{iff} \quad \overline{x} \in \psi(DB)
\end{aligned}
$$

which proves theorem 4.

7.2.1 Constructing R(M)

The above proof is based on construction of the rulebase $R(M)$, which solves the recognition problem for a query ψ according to formula 3. The rest of this section describes this construction.

Recall that M is an alternating PTIME machine which accepts the language $\{< \overline{x}, DB > \mid \overline{x} \in \psi(DB)\}$. Section 6.2 showed how to simulate APTIME machines in the logic of hypothetical additions. To use this construction, two things must be done: (i) encode the pair $< \overline{x}, DB >$ onto M's input tape, and (ii) simulate a counter from 0 to $n^k - 1$, where n is the size of the input and n^k is an upper bound on the amount of time consumed during the computation. Actually, it will be convenient to choose n to be the size of the data domain $dom(DB)$. We can do this because the database size is at most a polynomial function of the domain size.

7.2.2 Counting

If there is a total linear order on the data domain, then by using predicates of arity k, one can construct a counter from 0 to $n^k - 1$, where $n = |dom(DB)|$. The assumption of a linearly ordered domain is common in the literature [16, 25], especially when expressiblity results are established in terms of complexity classes, as in theorem 4. For hypothetical logics, however, this assumption is unnecessary; if there is no linear order on the domain, then one can be asserted hypothetically.

The difficulty is in choosing which linear order to assert. Since there is nothing special about any ordering, the rules have no way of selecting one over another. The trick is to hypothetically assert every possible ordering, one at a time, and to note that the Turing-machine simulations will return the same answer for each one. A similar trick was used in expressing the query $EVEN$ in examples 10 and 11. In this way, no *a priori* domain ordering is needed and no distinguished ordering is selected.

The rules below use the techniques of example 11 to assert all possible linear orderings of the data domain. They select and remove domain elements one-by-one, where deletion is simulated using negation-by-failure. After the first selection, a ground instance of the predicate $FIRST_1(x)$ is added to the database; after each subsequent selection, an instance of $NEXT_1(x, y)$ is added; and after the last selection, an instance of $LAST_1(x)$ is added.

$$
\begin{aligned}
& A \leftarrow SEL(x),\ B(x)[add : FIRST_1(x),\ ND(x)] \\
& B(x) \leftarrow SEL(y),\ B(y)[add : NEXT_1(x, y),\ ND(y)] \\
& B(x) \leftarrow \sim SEL(y),\ ACCEPT[add : LAST_1(x)] \\
& SEL(y) \leftarrow D(y),\ \sim ND(y)
\end{aligned}
$$

After asserting a linear order in this way, the rules try to infer the atom $ACCEPT$, which invokes a Turing-machine simulation. These simulations are constructed so that either $ACCEPT$ is true for all linear orders, or $ACCEPT$ is false for all linear orders.

During each of these simulations, there is a (hypothetical) linear-order on the data domain $dom(DB)$. In particular, for the ordering $< a_1, a_2 ... a_n >$, the following entries have been added to the database:

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

This is in addition to the original database predicates $P_1, ... P_m$.

This ordering on $dom(DB)$ provides a way of counting from 0 to $n - 1$. In addition, it induces an ordering on $[dom(DB)]^k$, the space of k-tuples of the domain elements. This in turn provides a way of counting from 0 to $n^k - 1$. In particular, we can write Horn rules to define the following three predicates: $FIRST(\overline{x})$, $NEXT(\overline{x}, \overline{y})$, and $LAST(\overline{y})$, where $\overline{x}$ and $\overline{y}$ are k-tuples (see [5] for details). $FIRST(\overline{x})$ and $LAST(\overline{y})$ are true iff $\overline{x}$ and $\overline{y}$ represent the integers 0 and $n^k - 1$, respectively; and $NEXT(\overline{x}, \overline{y})$ is true iff $\overline{y}$ represents the integer $\overline{x} + 1$.

7.2.3 Representing the Machine

With the above counter, we can use k-tuples to represent n^k distinct points in time and n^k distinct positions on the tape. We can thus simulate APTIME machines much as we did in section 6.2. In particular, 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 $\overline{p}$.

We must also specify the initial *id* of the machine, that is, the initial tape contents, the initial head position and the initial state of the control head. The control head is initialized by the following rule, which says that at time 0, the control 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 next section deals with the tape.

7.2.4 Initializing the Machine Tape

Given a database DB and a tuple $\overline{x}$, we must encode them onto the tape of our machine. We are assuming that $\overline{x}$ is of arity β and that DB consists of ground instances of the predicates $P_1, P_2, ...P_m$, of arities $\alpha_1, ..., \alpha_m$, resp.[11] We are also assuming that DB is augmented by a single instance of a predicate P_0, to represent the tuple $\overline{x}$ (see formula 3). Thus, we are given the set $DB + \{P_0(\overline{x})\}$, and we must write rules which encode it onto the Turing-machine tape.

Any reasonable encoding will do, and it is convenient to use a bit-map representation. We divide an initial segment of the tape into $m+1$ blocks and put a bit-map of P_i in the i^{th} block. In particular, each tape cell in block i corresponds to a possible database entry $P_i(x_1, ..., x_{\alpha_i})$; that is, a cell contains a 1 iff the corresponding entry $P_i(x_1, ..., x_{\alpha_i})$ is in the database. The use of negation-by-failure is crucial to setting the tape cell to 0 when $P_i(x_1, ..., x_{\alpha_i})$ is *not* in the database.

It is straightforward to construct a set of Horn rules which set the input tape cells to 0 or 1 and which make all other tape cells blank. In particular, one can define a predicate $INIT(c, \overline{p})$ which is true iff the tape cell at position $\overline{p}$ should be initialized to the symbol c, where $c \in \{0, 1, blank\}$ (see [5] for details). The machine tape is then initialized by the following rule:

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

7.2.5 Order Independence

This machine representation depends on the existence of a total linear order on the data domain. If such an order is not provided *a priori*, then the rules of section 7.2.2 can assert one hypothetically. These rules assert every possible linear order, one after the other, simulating the machine M for each one. In most respects, the linear order is a mere implementation detail, and the particular ordering used does not effect the computations which M performs. In one respect, however, this is not true: the initialization of the machine tape depends crucially on the linear order. In particular, different linear orders will result in different bit-map representations of the database.

Which of these bit maps actually represents the database? They all do. M either accepts all these bit maps, or it rejects all these bit maps. This is because M is a special Turing machine: it solves the recognition

[11] That is, we are assuming that the database query ψ which we are trying to express is of type $\overline{\alpha} \rightarrow \beta$, where $\overline{\alpha} = (\alpha_1, ..., \alpha_m)$.

problem for a database query ψ. As a database query, ψ satisfies the *consistency criterion*: if the constant symbols in a database DB are renamed in a consistent way, then the constant symbols in the output relation $\psi(DB)$ are renamed in the same way. It is not hard to see that changing the linear order on the data domain is equivalent to renaming the constants symbols in DB, at least as far as the machine M is concerned. Thus, the different bit maps that M receives for each linear order represent isomorphic databases.

For example, suppose that ψ is a database query which takes a database of two monadic predicates P and Q, and returns a binary relation. Consider the recognition problem, $(a, b) \in \psi[\{P(a),\ Q(b)\}]$. In this case the data domain is $\{a, b\}$ and there are two linear orders on the domain, $a < b$ and $b < a$. The bit-map representations of this problem under these two linear orders are shown in diagrams 1 and 2 below. In each diagram, the sequence of 1's and 0's represents the machine tape, and the tuples beneath represent the interpretation given to the tape cells.

1. Encoding the problem $(a, b) \in \psi[\{P(a),\ Q(b)\}]$ under the linear order $a < b$:

0	1	0	0	1	0	0	1
(a,a)	(a,b)	(b,a)	(b,b)	$P(a)$	$P(b)$	$Q(a)$	$Q(b)$

2. Encoding the problem $(a, b) \in \psi[\{P(a),\ Q(b)\}]$ under the linear order $b < a$:

0	0	1	0	0	1	1	0
(b,b)	(b,a)	(a,b)	(a,a)	$P(b)$	$P(a)$	$Q(b)$	$Q(a)$

3. Encoding the problem $(b, a) \in \psi[\{P(b),\ Q(a)\}]$ under the linear order $a < b$:

0	0	1	0	0	1	1	0
(a,a)	(a,b)	(b,a)	(b,b)	$P(a)$	$P(b)$	$Q(a)$	$Q(b)$

Diagrams 1 and 2 show clearly that changing the linear order changes the input to the machine. Diagrams 1 and 3 show that renaming the constant symbols changes the input in exactly the same way.

In general, a re-ordering of the data domain is equivalent to a renaming of the constant symbols. Thus, either M accepts $< \overline{x}, DB >$ under all linear orderings, or M rejects $< \overline{x}, DB >$ under all linear orderings.

7.3 EXPTIME Expressibility

This section shows that any database query which is computable in exponential time can be expressed in the hypothetical logic of additions and deletions augmented with negation-by-failure. 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 focusses on initializing the machine.

With hypothetical deletions, initializing the machine is less straightforward than with hypothetical additions. If we were to use the technique of section 7.2, we would simply write Horn rules such as the following:

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

These rules invoke the predicate $INIT(c, \overline{x})$, which is true iff the cell at position $\overline{x}$ should be initialized to c. The main point, however, is that with such rules, the atomic formulas $HEAD(q_0, \overline{p})$ and $CELL(c, \overline{p})$ are not part of the database, but are inferred.

The difficulty with this is that the rules of hypothetical deletion do not delete inferred facts; they only delete facts that are in the database. Since hypothetical deletion plays a crucial role in our simulation of APSPACE machines, the machine *id* must be stored in the database. In particular, the initial *id* cannot be inferred using Horn rules.

The alternative is to add the initial *id* to the database using hypothetical rules. The rules below do precisely this. In trying to prove the atom $ACCEPT$, they first initialize the control head, then they iterate over all k-tuples $\overline{x}$, placing the appropriate symbol in the tape cell at position $\overline{x}$, and finally they try to infer the atom $SIMULATE$, which is meant to invoke a machine simulation , as described in section 6.1.

$$ACCEPT \leftarrow FIRST(\overline{x}),\ E(\overline{x})[add : HEAD(q_0, \overline{x})]$$
$$E(\overline{x}) \leftarrow INIT(c, \overline{x}),\ F(\overline{x})[add : CELL(c, \overline{x})]$$
$$F(\overline{x}) \leftarrow NEXT(\overline{x}, \overline{y}),\ E(\overline{y}).$$
$$F(\overline{x}) \leftarrow LAST(\overline{x}),\ SIMULATE.$$

Given this, and the techniques of 7.2, the following theorem is straightforward.

Theorem 5 *Suppose that ψ is a database query of type $\overline{\alpha} \rightarrow \beta$ and DB is a database of type $\overline{\alpha}$. Suppose also that DB has a monadic predicate D(x) which defines the data domain. If ψ is computable in exponential time then there is a rulebase $R(\psi)$ of hypothetical additions and deletions with negation-by-failure such that*

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

where Q is a predicate symbol of arity β.

8 Conclusions

Two modal-like operators have been introduced, $A[add : B]$ and $A[del : B]$, which represent hypothetical addition and deletion, resp. $A[add : B]$ means, "adding B to the database allows A to be inferred," and $A[del : B]$ means, "deleting B from the database allows A to be inferred." An extension of Horn-clause logic was then defined in which these operators appear in the premises of rules. 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 an analysis of the complexity and expressibility of hypothetical reasoning in the function-free, predicate case. 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 query processing in 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, in that they would have to be "back propagated" through the rules in order to maintain consistency. We have completely avoided this complication by considering deletions from the database only. Nevertheless, these relatively simple deletions increase the complexity and expressibility of the logic from PSPACE to EXPTIME.

Because these two logics are monotonic, there are many queries which they cannot express. Adding *negation-by-failure* solves this problem. Indeed, with negation, the full logic can express any database query which is computable in exponential time; and the restricted logic can express any database query computable in polynomial space.

Acknowledgements

The work of Thorne McCarty on the intuitionistic semantics of embedded implications was the original stimulus for this work, and discussions with Tomasz Imielinski provided direction and insight. Thanks go to Eric Allender and Jan Chomicki for helpful and timely comments on the paper.

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. Technical Report DCS-TR-230, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, 1988.

[5] A.J. Bonner. Hypothetical Datalog: Complexity and Expressibility. Technical Report DCS-TR-231, Department of Computer Science, Rutgers university, New Brunswick, NJ 08903, 1988. Also published in volume 76 (1990) of *Theoretical Computer Science (TCS)*, pages 3–51.

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

[7] 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.

[8] 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.

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

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

[11] M.C. Fitting. *Intuitionistic Logic, Model Theory and Forcing*. North-Holland, 1969.

[12] 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.

[13] 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.

[14] D. Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

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

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

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

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

[19] 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.

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

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

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

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

[24] 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.

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

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