J. LOGIC PROGRAMMING 1997, 33(1):1–47

INTUITIONISTIC DEDUCTIVE DATABASES AND THE POLYNOMIAL TIME HIERARCHY

ANTHONY J. BONNER

▷ Deductive databases are poor at tasks such as planning and design, where one must explore the consequences of hypothetical actions and possibilities. To address this limitation, we have developed a deductive database language in which a user can create hypotheses and draw inferences from them. In earlier work, we established initial results on the complexity and expressibility of this language. In this paper, we establish more comprehensive results by exploring the interaction of *negation-as-failure* with a natural syntactic restriction called *linearity*. The main result is a tight connection between intuitionistic logic, database queries, and the polynomial time hierarchy. A tight connection with second-order logic follows as a corollary. First, we show that rulebases in our language fit neatly into a well-established logical framework—*intuitionistic logic*. Second, we show that linearity reduces their data complexity from PSPACE to NP. Third, we show that negation-as-failure increases their complexity from NP to some level in the polynomial time hierarchy (PHIER). Specifically, linear rulebases with k strata are data complete for Σ_k^{P}, the k^{th} level in the hierarchy. Fourth, we show that linear rulebases express *exactly* the generic database queries in PHIER. Finally, we characterize the generic queries in Σ_k^{P} in terms of rulebases with k strata. Unlike many other expressibility results in the literature, these results do *not* depend on the artificial assumption that the data domain is linearly ordered. We thus establish a strong link between two well-established, but previously unrelated areas: intuitionistic logic and the polynomial time hierarchy. ◁

Address correspondence to University of Toronto, Department of Computer Science, Toronto, Ontario, Canada M5S 3G4. Email: bonner@db.toronto.edu

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

1. INTRODUCTION

Researchers from several areas have recognized the need for computer systems that reason hypothetically. Decision support systems (DSS) are a good example, especially in domains like financial planning where many "what if" scenarios must be considered [29, 43]. A typical example is an analyst who must predict a company's deficit for the upcoming year assuming that employee salaries are increased by a given percentage. Or he might want a table of deficit predictions for a number of hypothetical salary increases [52, 53]. Similar problems occur in computer aided design (CAD). Here, one must evaluate the effect on the overall design of local design alternatives and of various external factors [18, 45]. For example, an engineer may need to know how much the price of an automobile would increase if supplier X raised his prices by Y percent [18]. The number of hypothetical scenarios multiplies quickly when several factors are varied simultaneously, such as prices, interest rates, tax rates, etc. One may also need to consider variations in more complex factors, such as government regulations, company policy, tax laws, etc.

1.1. Background

The database community has addressed some of these needs by developing systems that integrate query processing with hypothetical updates. Such systems allow a user to pose queries not only to the real database, but also to hypothetical databases. Hypothetical databases are derived from a real database by a series of hypothetical assumptions, or updates. Early work in this area was done by Stonebraker, who showed that hypothetical databases can be efficiently implemented by slight extensions to conventional database mechanisms [49, 48]. He pointed out that hypothetical databases are useful for debugging purposes, for generating test data, and for carrying out a variety of simulations. He also argued that "there are advantages to making hypothetical databases central to the operation of a database management system" [48].

The logic-programming community has taken these ideas one step further, integrating hypothetical updates not just with query processing, but with logical inference as well. Since the premise of a logical rule is just a query, several researchers have developed *hypothetical rules*, in which the premise can query not only a real database, but hypothetical databases as well. Vieille, et al, for instance, have implemented a deductive database along these lines [52, 53], and Warren and Manchanda have used hypothetical rules to reason about database updates [54, 33]. In [39], Miller shows that hypothetical insertions can structure the runtime environment of logic programs, resulting in programs that are more elegant, more efficient, and easier to maintain. In [40], he develops a theory of lexical scoping based on the hypothetical creation of constant symbols during inference.

These logical systems are well-suited to solving problems in Artificial Intelligence, especially problems that involve reasoning about alternative courses of action. For example, an AI program may need to infer that *if* the pawn took the knight, *then* the rook would be threatened. The program may also have to consider sequences of possible moves, exploring hypothetical possibilities to great depth. Hypothetical inference has also extended the capabilities of expert systems. Gabbay and Reyle, for instance, have reported a need to augment Prolog with hypothetical rules in order to encode the British Nationality Act, because the act contains rules such as, "You are eligible for citizenship if your father *would* be eligible *if* he were still alive" [20]. And McCarty, also motivated by legal applications, has developed a wide class of hypothetical rules for computer-based consultation systems, especially systems for reasoning about corporate tax law and estate tax law [36, 38, 44].

In [6, 5], we investigated hypothetical reasoning in deductive databases. We considered rules that can hypothetically insert or delete facts from a database. In [6], we showed that with both insertion and deletion, the data complexity of hypothetical inference is complete for EXPTIME, while with insertion alone it is complete for PSPACE. Without hypothetical operations, the language reduces to Datalog (function-free Horn logic), whose data complexity is complete for PTIME. We also proved completeness results on the logic's ability to express databases queries. In [5], we developed a logical semantics for hypothetical reasoning with insertion and deletion, including a proof theory, model theory and fixpoint theory. We then extended the semantics to account for negation-as-failure.

Other researchers in the logic programming community have investigated the semantics of hypothetical inference. This work has focussed on the hypothetical insertion of facts into a database, since such updates fit neatly into a well-established logical framework—*intuitionistic logic* [17]. In intuitionistic logic, hypothetical insertion arises from formulas called *embedded implications* [36]. These are rules of the form $A \leftarrow (B \leftarrow C)$, which informally means that A can be inferred if assuming C allows B to be inferred. The assumption of C is a hypothetical insertion into the database. Gabbay first showed that embedded implications are intuitionistic [20]. Working independently, McCarty and Miller extended embedded implications to a wider class of formulas, and developed fixpoint semantics based on intuitionistic logic [36, 39]. Recently, researchers have begun investigating the semantics of negation-as-failure for embedded implications [19, 8, 24, 42, 23, 15], which presents interesting technical challenges because of the intuitionistic setting.

The related phenomenon of *counterfactual* reasoning has been explored extensively in the context of belief revision and knowledge base updates (*e.g.*, [22, 30, 16, 41]). In this work, a database (or knowledge base) is treated as a set of arbitrary logical formulas, and the issue is what to do when a formula inserted into the database contradicts formulas already there. The focus is therefore on the resolution of logical contradictions. In contrast, logical contradiction is not an issue in the kind of hypothetical reasoning described above, since it deals with databases for which logical contradiction is impossible. For example, deductive databases and definite Horn programs are always satisfiable since they always have at least one model (the minimal model). Adding atoms or Horn rules to such databases does not lead to logical contradictions. Nevertheless, hypothetical reasoning of this kind has a profound effect on complexity, expressibility and semantics.

1.2. Intuitionistic Deductive Databases

There has been considerable work on the semantics of intuitionistic rulebases in general. However, there has been little work on the case of greatest interest to database systems, the function-free predicate case. Such rulebases might be called *intuitionistic deductive databases.* Several questions naturally arise. For instance, what is their data complexity, and what database queries can they express? Initial results along these lines were provided by Statman [46] for the propositional case, and by our own PSPACE results for hypothetical insertion [6]. In this paper, we provide new results on semantics, and a comprehensive set of results on complexity and expressibility. The main result is a tight connection between intuitionistic logic, database queries, and the polynomial time hierarchy. A tight connection with second-order logic follows as a corollary. To establish these results, we explore a natural syntactic restriction called *linear recursion* and its interaction with *negation-as-failure*. In addition, we provide examples demonstrating the utility of intuitionistic logic as a database query language.

Our first result is a new and simplified proof that hypothetical insertion is intuitionistic. We present the proof theory for hypothetical insertion, and show that it is sound and complete with respect to intuitionistic model theory. Unlike other proofs in the literature [20, 36, 39], ours is based on the Henkin constructions of modal logic [14, 26].

Next, we develop the notion of *linear* intuitionistic rulebases, generalizing the notion of linear Horn rulebases [4]. Intuitively, a rule is linear if recursion occurs through only one premise. In classical Horn logic, "linear rules play an important role because, (*i*) there is a belief that most 'real life' recursive rules are linear, and (*ii*) algorithms have been developed to handle them efficiently" [4]. We show that linearity reduces the data complexity of intuitionistic rulebases from PSPACE to NP. The lower bound is proved by encoding the computations of an arbitrary NP-machine as a linear rulebase. The upper bound is more difficult and is proved by showing that for any linear rulebase, all proof trees are of polynomial size.

Third, we augment intuitionistic rulebases with negation-as-failure. Following [5], we extend the notion of stratification [2, 12] from Horn rules to intuitionistic rules. We then show that the data complexity of a linear rulebase depends crucially on the number of strata it has. Each stratum increases the complexity by one level in the polynomial time hierarchy, so that rulebases with k strata are data complete for Σ_k^{P}, the k^{th} level in the hierarchy. We also show that strata composed entirely of Horn rules have little effect on the complexity and can largely be ignored. The lower complexity bounds are proved by encoding the computations of a cascade of oracle Turing machines. The upper bounds are proved by a procedure that combines bottom-up, deterministic inference with top-down, non-deterministic inference.

Fourth, we characterize the generic database queries expressible by linear intuitionistic rulebases. A query is *generic* if it satisfies the consistency criterion of Chandra and Harel [10, 11]. Genericity captures the idea that the constant symbols in a database are uninterpreted. We first show that stratified linear rulebases are expressively complete for the polynomial time hierarchy (PHIER); that is, any generic database query in PHIER can be expressed as a stratified linear rulebase. If the rulebases do not contain any constant symbols, then they express *exactly* the generic queries in PHIER, no more and no less. Using rulebases with a fixed number of strata, we characterize the generic queries in each individual level of the hierarchy, Σ_k^{P}, for any $k \geq 1$. This establishes a tight connection between two previously unrelated, but well-established areas: intuitionistic logic and the polynomial time hierarchy.

Unlike many expressibility results in the literature (*e.g.*, [27, 51, 12]), these results do *not* assume that the data domain is linearly ordered. The assumption of ordered domains is a technical device that is often used to achieve expressibility results, but it is not an intrinsic feature of databases [1]. Intuitionistic rulebases do not need this assumption, since they can generate linear orders on a domain hypothetically [6].

Finally, we recall that the generic queries in PHIER are precisely the queries definable in second-order logic [47, 28]. We have therefore characterized the second-order definable queries in terms of intuitionistic deductive databases.

1.3. Introductory Examples

A deductive database consists of a set of atomic formulas (the database) and a set of logical rules (the rulebase). This section gives examples in which the database is changed during inference by hypothetically inserting facts into it. The examples are based on a deductive database for a university. In the examples, the formula $take(s, c)$ intuitively means that student s has taken course c, and the formula $grad(s)$ means that student s is eligible to graduate. Also, R denotes a rulebase, DB denotes a database, and $R + DB$ denotes their union (*i.e.*, their logical conjunction). Given a query, ϕ, the expression $R + DB \vdash \phi$ means that ϕ can be inferred from the rules in R and the facts in DB. The examples below are selected and adapted from [6]. Additional examples can be found in [6, 7].

Example 1.1. [Hypothetical Queries] Consider the following query: *Could Ester graduate if she took csc452?* In other words, if $take(ester, cs452)$ were added to the database, could we infer $grad(ester)$? This query can be formalized at the meta-level as follows:

$$R + DB + take(ester, cs452) \ \vdash \ grad(ester) \tag{1.1}$$

We represent this query by the implication $grad(ester) \leftarrow take(ester, cs452)$. That is $R + DB \ \vdash \ grad(ester) \leftarrow take(ester, cs452)$ is true if and only if condition (1.1) is satisfied.

Example 1.2. [Non-Determinism] Consider the following query: *Retrieve those students who could graduate if they took (at most) one more course.* At the meta-level, we want those students s such that for some course c,

$$R + DB + take(s, c) \ \vdash \ grad(s) \tag{1.2}$$

We represent this query by the expression $\psi(s) \ = \ \exists c\, [grad(s) \leftarrow take(s, c)]$. That is $R, DB \vdash \psi(s)$ is true if and only if condition (1.2) is satisfied. Observe that the hypothetical update is non-deterministic, since it depends on the non-deterministic choice of c, as reflected by the existential quantifier.

Example 1.3. [Hypothetical Rules] Suppose the university has the following policy:

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

This policy is represented by the following two rules:

$$within1(s, d) \ \leftarrow \ \exists c\, [grad(s, d) \leftarrow take(s, c)].$$
$$grad(s, math\&phy) \ \leftarrow \ within1(s, math),\ within1(s, phy).$$

where the predicate $grad(s, d)$ means that student s is eligible for a degree in discipline d, and $within1(s, d)$ means that s is within one course of a degree in discipline d. Note that the premise of the first rule is a hypothetical query similar to the query in Example 1.2.

2. SYNTAX AND PROOF THEORY

This section describes a logical inference system for hypothetical rules. Similar systems have been developed by other researchers [20, 36, 39, 34]. This section defines a simplified system, one that retains many of the essential properties of the more elaborate systems while admitting a clean theoretical analysis. The system is an extension of classical Horn logic, both syntactically and proof-theoretically. The language of the logic includes three infinite, enumerable sets: a set of variables $x, y, z, ...$, a set of constant symbols $a, b, c, ...$, and a set of predicate symbols $A, B, C, ...$. Since this paper focuses on databases and data complexity, function symbols are not included in the language. (However, because the set of constant symbols is infinite, it should be possible to extend our development to include function symbols, by treating each ground Herbrand term as a distinct constant symbol.) The material presented in this section is adapted from [6], to which the reader is referred for additional details.

Definition 2.1. [Hypothetical Queries] A hypothetical query is a formula of the form $B \leftarrow C_1, C_2, \ldots, C_k$ where B and each C_i are atomic formulas, and $k \geq 0$.

Definition 2.2. [Hypothetical Rules] An embedded implication is a formula of the form $A \leftarrow \phi_1, \phi_2, \ldots, \phi_k$ where A is an atomic formula, each ϕ_i is a hypothetical query, and $k \geq 0$.

The following three formulas are all embedded implications:

$$A \leftarrow (B \leftarrow C, D) \qquad A \leftarrow (B \leftarrow C), (D \leftarrow E) \qquad A(x) \leftarrow [B(x, y) \leftarrow C(x, y)]$$

When a hypothetical query has an empty premise, we shall write B instead of $B \leftarrow$. Likewise, we shall write $A \leftarrow B$ instead of $A \leftarrow (B \leftarrow)$. Embedded implications thus include Horn rules as a special case, and hypothetical queries include atomic formulas as a special case. We will often use the terms *embedded implication* and *hypothetical rule* interchangeably. Likewise for the terms *Horn rule* and *hypothetical query*, depending on the context.

Given a hypothetical query $B \leftarrow C_1, \ldots, C_k$, we call the predicate symbol of B the *goal predicate* of the query. Thus, in the rule $A(x) \leftarrow B_1(x, y),\ [B_2(y) \leftarrow C(y)]$, both B_1 and B_2 are goal predicates. The predicate symbol A appearing in the rule head is called the *head predicate.* In a top-down, Prolog-style proof procedure, goal predicates would become subgoals and would be resolved against head predicates. (McCarty has developed such a proof procedure for embedded implications [37].)

When establishing the data complexity of hypothetical inference in Section 4, we will focus on specific sets of constant and predicate symbols. These sets will be finite, but of unbounded size. It is convenient to treat these sets as parameters of the proof theory. This motivates the following definition.

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

The symbol $\mathcal{DB}(\mathcal{R})$ shall denote the set of all ground atomic formulas constructible from the predicate symbols in *pred* and the constant symbols in *dom*. Since *dom* and *pred* are finite, so is $\mathcal{DB}(\mathcal{R})$. A database for $\mathcal{R}$ is any subset of $\mathcal{DB}(\mathcal{R})$, and during hypothetical inference, only those formulas in $\mathcal{DB}(\mathcal{R})$ will be inserted into a database. When $\mathcal{R}$ is understood, we shall write $\mathcal{DB}$ for short. For convenience, we shall often refer to $\mathcal{R}$ simply as a hypothetical system or as a rulebased system.

Definition 2.4. [Inference] Given a hypothetical rulebased system $[R, dom, pred]$, hypothetical inference is defined by the axioms and inference rules below. All formulas are constructed from dom and pred, and $DB \subseteq \mathcal{DB}$.

Axioms: $R + DB \vdash A$ for every ground atomic formula $A \in DB$.
Inference Rules:

1. If $A \leftarrow \phi_1, \ldots, \phi_m$ is a ground instantiation of a rule in R, then
$$\frac{R + DB \vdash \phi_i \text{ for each } i}{R + DB \vdash A}$$
2. For any ground hypothetical query $B \leftarrow C_1, \ldots, C_k$,
$$\frac{R + DB + \{C_1, \ldots, C_k\} \vdash B}{R + DB \vdash B \leftarrow C_1, \ldots, C_k}$$

In this inference system, each inference rule has the following interpretation: If the expressions above the horizontal line can be inferred, then those below the line can also be inferred. We shall soon see that inference can be performed in a "top-down" manner by inverting these rules. Note that hypothetical inference is defined with respect to a particular rulebased system. Thus, whenever we write $R + DB \vdash \phi$, some system $\mathcal{R} = [R, dom, pred]$ is understood.

Example 2.1. [Propositional Inference] Suppose R contains the following rules:

$$A \leftarrow (B \leftarrow D) \qquad B \leftarrow C \qquad C \leftarrow D$$

Then $R + DB \vdash A$ for any database DB. This is proved by a simple, top-down argument:

$$\begin{array}{llll} & R + DB & \vdash A & \\ \textit{if} & R + DB & \vdash B \leftarrow D & \textit{by inference rule 1}, \\ \textit{if} & R + DB + D & \vdash B & \textit{by inference rule 2}, \\ \textit{if} & R + DB + D & \vdash C & \textit{by inference rule 1}, \\ \textit{if} & R + DB + D & \vdash D & \textit{by inference rule 1}. \end{array}$$

But the last line is trivially true, since it is an axiom.

It is implicit in Definition 2.4 that the variables in a rule are universally quantified. As in classical logic, if a variable appears in the body of a rule but not the head, then the universal quantifier can be moved inside the rule body and converted to an existential. For example, if the rule $A(x) \leftarrow [B(x,y) \leftarrow C(x,y)]$ appears in a rulebase R, then Definition 2.4 implies that for *any* constant symbol e,

$$R \vdash A(e) \quad \textit{if} \quad R \vdash [B(e,f) \leftarrow C(e,f)] \quad \textit{for some constant } f.$$

Thus, with an abuse of notation, this rule can be read in two equivalent ways:

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

The latter interpretation enables the logic to represent the examples given in Section 1.3. In particular, the rule $within1(s) \leftarrow [grad(s) \leftarrow take(s,c)]$ defines those students who are "within one course of graduation."

The next example illustrates, in simplified form, the kind of rules used in the Turing-machine encodings of Section 6.2. Additional examples are given in [6, 5, 7].

Example 2.2. [Recursion] Suppose R is a rulebase containing the following rules:

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

and suppose DB is a database containing the following atomic formulas:

$$FIRST(0),\quad NEXT(0,1),\quad NEXT(1,2)\ \ldots\ NEXT(n-1,n),\quad LAST(n).$$

Then $\quad R + DB \vdash A \quad$ if $\quad R + DB + P(1) + P(2) + \cdots + P(n) \vdash C$

Finally, we state a basic result about hypothetical inference. It can be proved by a straightforward induction over the length of derivations.

Lemma 2.1. [Monotonicity] *Suppose* $R_1 \subseteq R_2$ *and* $DB_1 \subseteq DB_2$*. If* $R_1 + DB_1 \vdash \phi$ *then* $R_2 + DB_2 \vdash \phi$

3. MODEL THEORY

The inference system of Definition 2.4 is not classical. Instead, it is intuitionistic, as several researchers have shown [20, 36, 39]. This section reviews intuitionistic semantics and develops a new and simpler proof that the inference system is sound and complete intuitionistically. This proof is inspired by the Henkin constructions of modal logic [14, 26]. Given a rulebased system $\mathcal{R}$, we define an intuitionistic structure, $M_{\mathcal{R}}$, called the *canonical model* of $\mathcal{R}$. This structure, defined proof-theoretically, provides the necessary link between inference and semantics, and leads to a simple proof of completeness.

Intuitionistic logic makes finer distinctions than classical logic. For instance, the formulas $A \leftarrow B$ and $A \vee \sim B$ are equivalent classically, but not intuitionistically. Similarly, the formulas $G \leftarrow A, B$ and $(G \leftarrow A) \vee (G \leftarrow B)$ are equivalent classically, but not intuitionistically [7, 5]. Intuitionistic logic also admits fewer inferences than classical logic, such as the above equivalences. This can be seen in the inference system of Definition 2.4. Each inference rule in the system is a classically *sound*, but the system as a whole is classically *incomplete*: there are some classical theorems that it cannot (and should not!) prove.

Example 3.1. [Classical Incompleteness] Suppose R consists of the following rules:

$$A \leftarrow (B \leftarrow C) \qquad D \leftarrow A \qquad D \leftarrow C$$

If these rules are interpreted classically, then $R \models_c D$, where $\models_c$ denotes entailment in classical logic. To see this, note that from the classical definition of implication, the first rule can be expanded in terms of disjunction and negation to give $(A \vee \sim B) \wedge (A \vee C)$. Thus $R \models_c A \vee C$. Furthermore, the last two rules can be combined to give $D \leftarrow (A \vee C)$. Thus $R \models_c D$. However, the expression $R \vdash D$ *cannot* be derived using inference rules in Definition 2.4. To see this, note that there are only two lines of top-down inference, both of which fail. First,

$$R \vdash D \quad \textit{if} \quad R \vdash C$$

which fails, since are no rules for inferring C. Second,

$$R \vdash D \quad \textit{if} \quad R \vdash A \quad \textit{if} \quad R \vdash B \leftarrow C \quad \textit{if} \quad R + C \vdash B$$

which fails, since there are no rules for inferring B. Thus, $R \not\vdash D$ in our inference system, so it is not complete with respect to classical semantics. Instead, it is sound and complete with respect to intuitionistic semantics, as shown in this section.

3.1. Intuitionistic Logic

This section defines the semantics of first-order, intuitionistic logic in the function-free case. It has a modal-like semantics that can be interpreted as a semantics of incomplete knowledge. Informally, an intuitionistic structure is a set of substates, each representing a state of our "knowledge." The substates are organized into a partial order, and as we "climb" the partial order from one substate to another, our knowledge increases; that is, we know about more objects, and we know more true formulas. The language of intuitionistic logic is that of first-order predicate calculus. It includes three infinite, enumerable sets: a set of variables $x, y, z, \ldots$; a set of constant symbols $a, b, c, \ldots$; and a set of predicate symbols $A, B, C, \ldots$. More extensive treatment and discussion of intuitionistic logic can be found in [17, 32].

Definition 3.1. [Structures] A first-order intuitionistic structure is a quadruple $M = \langle S, \leq, \pi, d \rangle$, where

1. S is a non-empty set.
2. $\leq$ is a partial order on S.
3. π is a monotonic mapping from elements of S to sets of ground atomic formulas. Thus, if $s_1 \leq s_2$ then $\pi(s_1) \subseteq \pi(s_2)$.
4. d is a monotonic mapping from elements of S to sets of constant symbols. $d(s)$ must contain all the constant symbols in $\pi(s)$.

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

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

Definition 3.2. If dom is a set of constant symbols, then $\mathcal{F}(dom)$ denotes the set of first-order formulas containing only those constant symbols in dom. If s is a substate, then $\mathcal{F}(s)$ means $\mathcal{F}[d(s)]$.

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

Definition 3.3. [Satisfaction] Suppose M is an intuitionistic structure and s is a substate of M. Then,

$s, M \models A$	iff	$A \in \pi(s)$ when A is atomic.
$s, M \models \psi_1 \wedge \psi_2$	iff	$s, M \models \psi_1$ and $s, M \models \psi_2$
	and	$\psi_1 \wedge \psi_2$ is in $\mathcal{F}(s)$.
$s, M \models \psi_1 \vee \psi_2$	iff	$s, M \models \psi_1$ or $s, M \models \psi_2$
	and	$\psi_1 \vee \psi_2$ is in $\mathcal{F}(s)$.
$s, M \models \sim \psi$	iff	$r, M \not\models \psi$ for all $r \geq s$
	and	$\sim \psi$ is in $\mathcal{F}(s)$.
$s, M \models \psi_2 \leftarrow \psi_1$	iff	$r, M \models \psi_2$ whenever $r, M \models \psi_1$, for all $r \geq s$
	and	$\psi_2 \leftarrow \psi_1$ is in $\mathcal{F}(s)$.
$s, M \models \forall x\ \psi(x)$	iff	$r, M \models \psi(b)$ for all $b \in d(r)$ and all $r \geq s$
	and	$\forall x\ \psi(x)$ is in $\mathcal{F}(s)$.
$s, M \models \exists x\ \psi(x)$	iff	$s, M \models \psi(b)$ for some $b \in d(s)$
	and	$\exists x\ \psi(x)$ is in $\mathcal{F}(s)$.

Note that unlike classical logic, intuitionistic implication is not defined in terms of disjunction and negation. Instead, it has an independent semantic definition. It is this definition that captures hypothetical insertion. The following basic result is an immediate consequence of the above definitions [17]. It reflects the idea that knowledge increases monotonically as we climb from one substate to a higher one.

Lemma 3.1. $s, M \models \phi$ *if and only if* $r, M \models \phi$ *for all* $r \geq s$.

Definition 3.4. [Models] $M \models \psi$ if $s, M \models \psi$ for all substates s of M such that $\psi \in \mathcal{F}(s)$. In this case, we say that M is a model of ψ, or that ψ is true in M.

Definition 3.5. [Validity] A formula ψ is valid if $M \models \psi$ for all intuitionistic structures M.

Definition 3.6. [Entailment] Suppose ψ_1 and ψ_2 are formulas. Then $\psi_1 \models \psi_2$ if the formula $\psi_2 \leftarrow \psi_1$ is valid. In this case, we say that ψ_1 entails ψ_2.

3.2. Soundness and Completeness

This section shows that the inference system in Definition 2.4 is sound and complete with respect to intuitionistic semantics. This result assumes several conventions about how to interpret rules and rulebases as first-order formulas, conventions that are common in the logic-programming literature. First, we interpret a rule $A \leftarrow \phi_1, \ldots, \phi_k$ as the first-order formula $A \leftarrow (\phi_1 \wedge \ldots \wedge \phi_k)$. Second, we interpret free variables in a rule as universally quantified. Thus, the rule $A(x) \leftarrow B(x)$ is interpreted as the formula $\forall x\, [A(x) \leftarrow B(x)]$. Third, we treat a finite set of formulas, such as a rulebase or a database, as a conjunction. Also, recall that hypothetical inference is defined with respect to a particular rulebased system $[R, dom, pred]$, where R is a finite set of embedded implications, dom is a finite set of constant symbols, and $pred$ is a finite set of predicate symbols. Given such a system, $\mathcal{DB}$ denotes the set of ground atomic formulas constructible from the constant symbols in dom and the predicate symbols in $pred$. With this in mind, we can now state the following theorem, which is the main result of this section.

Theorem 3.2. Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system, and let ϕ be a ground hypothetical query constructed from the symbols in dom and pred. Then, $R + DB \models \phi$ if and only if $R + DB \vdash \phi$.

In this theorem, and throughout this section, we assume that $DB \subset \mathcal{DB}$, the set of all ground atomic formulas constructed from the symbols in dom and $pred$.

The *if* direction of Theorem 3.2 (*i.e.*, soundness) follows immediately because the axioms and rules of hypothetical inference given in Definition 2.4 are intuitionistically valid. The axioms are trivially valid, and inference rules *1* and *2* correspond to modus ponens and the deduction theorem, respectively. The rest of this section proves the *only if* direction (*i.e.*, completeness). The approach taken here is inspired by the Henkin constructions of modal logic [14, 26]. Given a rulebased system $\mathcal{R}$, we define an intuitionistic structure $M_{\mathcal{R}}$ called the *canonical model* of $\mathcal{R}$. This structure, defined proof-theoretically, provides the necessary link between inference and semantics.

Definition 3.7. Let $\mathcal{R} = [R, dom, pred]$ be a rulebased system. The canonical model of $\mathcal{R}$ is the intuitionistic structure $M_{\mathcal{R}} = \langle S, \leq, \pi, d\rangle$ where

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

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

The canonical model is a legitimate intuitionistic structure. That is, it satisfies the four conditions of Definition 3.1. Condition *1* is satisfied since S will always contain the empty database as a substate. Condition *2* is satisfied since set containment is a partial order. Condition *3* is satisfied since hypothetical inference is monotonic, by Lemma 2.1. Condition *4* is satisfied since $d(DB)$ is the domain of constant symbols from which all atomic formulas in the rulebased system are constructed.

The proof of completeness involves three main steps:

1. Show that if $DB, M_{\mathcal{R}} \models \phi$ then $R + DB \vdash \phi$
2. Show that $M_{\mathcal{R}}$ is an intuitionistic model of R.
3. Conclude that the inference system is intuitionistically complete.

We prove each item in turn. The following two lemmas prove item *1*. The first lemma is a basic result about canonical models which follows immediately from the definitions.

Lemma 3.3. Let $\mathcal{R} = [R, dom, pred]$ *be a rulebased system, and let* A *be an atom in* $\mathcal{DB}$*. Then,* $DB, M_{\mathcal{R}} \models A$ *if and only if* $R + DB \vdash A$*.*

Lemma 3.4. Let $\mathcal{R} = [R, dom, pred]$ *be a rulebased system, and let* ϕ *be a ground hypothetical query constructed from the symbols in dom and pred. If* $DB, M_{\mathcal{R}} \models \phi$ *then* $R + DB \vdash \phi$*.*

Proof. ϕ has the form $B \leftarrow C_1, \ldots, C_k$ where B and each C_i are atomic formulas in $\mathcal{DB}$. Hence, if $DB, M_{\mathcal{R}} \models B \leftarrow C_1, \ldots, C_k$, then

$$\begin{array}{lll}
DB', M_{\mathcal{R}} \models B \;\Leftarrow\; DB', M_{\mathcal{R}} \models C_1, \ldots, C_k & & \text{for all } DB' \geq DB, \text{ by Def. 3.3,} \\
R + DB' \vdash B \;\Leftarrow\; R + DB' \vdash C_1, \ldots, C_k & & \text{for all } DB' \geq DB, \text{ by Lemma 3.3,} \\
R + DB' \;\vdash\; B & & \text{for all } DB' \geq DB + \{C_1, \ldots, C_k\}, \\
R + DB + \{C_1, \ldots, C_k\} \;\vdash\; B & & \text{using } DB' = DB + \{C_1, \ldots, C_k\}, \\
R + DB \;\vdash\; B \leftarrow C_1, \ldots, C_k & & \text{by inference rule 2 of Def. 2.4.}
\end{array}$$

□

The next two lemmas prove item *2*. They both exploit the fact that in the canonical model, each substate has the same domain, *dom*. Thus, a formula is defined on one substate if and only if it is defined on all the substates.

Lemma 3.5. Let $\mathcal{R} = [R, dom, pred]$ *be a rulebased system, and let* $\psi(x)$ *be a first-order formula whose constant symbols are in dom. Then,* $DB, M_{\mathcal{R}} \models \forall x\ \psi(x)$ *if and only if* $DB, M_{\mathcal{R}} \models \psi(b)$ *for all* $b \in dom$*.*

Proof. First, note that if $b \in dom$, then the formulas $\psi(b)$ and $\forall x\ \psi(x)$ contain only those constant symbols in *dom*. These formulas are therefore defined at every substate of $M_{\mathcal{R}}$, since *dom* is the domain of each substate. Thus, in Definition 3.3, the conditions "$\forall x\ \psi(x)$ *is in* $\mathcal{F}(s)$" and "$\psi(b)$ *is in* $\mathcal{F}(s)$" are always satisfied, and can therefore be ignored. Keeping this in mind,

$$\begin{array}{lrcll}
 & DB, M_{\mathcal{R}} & \models & \forall x\ \psi(x) & \\
\textit{iff} & DB', M_{\mathcal{R}} & \models & \psi(b) & \text{for all } b \in d(DB') \text{ and all } DB' \geq DB, \text{ by Def. 3.3,} \\
\textit{iff} & DB', M_{\mathcal{R}} & \models & \psi(b) & \text{for all } b \in dom \text{ and all } DB' \geq DB, \\
\textit{iff} & DB, M_{\mathcal{R}} & \models & \psi(b) & \text{for all } b \in dom, \text{ by Lemma 3.1.}
\end{array}$$

□

Lemma 3.6. If $\mathcal{R} = [R, dom, pred]$ *then* $DB, M_{\mathcal{R}} \models (R + DB)$.

PROOF. $DB, M_{\mathcal{R}} \models DB$ trivially, so we need only show that $DB, M_{\mathcal{R}} \models R$. Let ψ be any rule in R. The constant symbols in ψ are therefore all in *dom*, so ψ is defined at every substate of $M_{\mathcal{R}}$. By convention, the free variables in ψ are universally quantified. Thus, to show that $DB, M_{\mathcal{R}} \models \psi$, it is sufficient to examine the ground instances of ψ (where instances are generated by replacing the variables in ψ by constants from *dom*). Suppose, therefore, that $A \leftarrow \phi_1, \ldots, \phi_k$ is a ground instance of ψ. Thus, if $DB' \geq DB$ and $DB', M_{\mathcal{R}} \models \phi_1 \wedge \ldots \wedge \phi_k$, then

$DB', M_{\mathcal{R}} \models \phi_j$	for each j, by Definition 3.3,
$R + DB' \vdash \phi_j$	for each j, by Lemma 3.4,
$R + DB' \vdash A$	by inference rule 1 of Definition 2.4,
	since $A \leftarrow \phi_1, \ldots, \phi_k$ is a ground instance of a rule in R,
$DB', M_{\mathcal{R}} \models A$	by Lemma 3.3.

Since this is true for any $DB' \geq DB$, it follows that $DB, M_{\mathcal{R}} \models A \leftarrow \phi_1, \ldots, \phi_k$, by Definition 3.3. Since this is true for any ground instance of ψ, it follows that $DB, M_{\mathcal{R}} \models \psi$, by Lemma 3.5. Since this is true for any rule in R, it follows that $DB, M_{\mathcal{R}} \models R$. □

Finally, the following corollary proves item *3*.

Corollary 3.7. [Completeness] *Let* $\mathcal{R} = [R, dom, pred]$ *be a rulebased system, and let* ϕ *be a ground hypothetical query constructed from the symbols in dom and pred. If* $R + DB \models \phi$ *then* $R + DB \vdash \phi$.

PROOF. First note that R, DB and ϕ are finite and are constructed from the constant symbols in *dom* and the predicate symbols in *pred*. They are therefore defined on every substate of $M_{\mathcal{R}}$. Thus, the formula $\phi \leftarrow (R + DB)$ is also defined everywhere. Keeping this in mind, suppose that $R + DB \models \phi$. Then,

$\phi \leftarrow (R + DB)$ is valid,	by Definition 3.6,
$M \models \phi \leftarrow (R + DB)$	for all intuitionistic structures, M,
$M_{\mathcal{R}} \models \phi \leftarrow (R + DB)$	since $M_{\mathcal{R}}$ is an intuitionistic structure,
$DB, M_{\mathcal{R}} \models \phi \leftarrow (R + DB)$	by Definition 3.4,
$DB, M_{\mathcal{R}} \models \phi \Leftarrow DB, M_{\mathcal{R}} \models R + DB$	by Definition 3.3,
$DB, M_{\mathcal{R}} \models \phi$	by Lemma 3.6,
$R + DB \vdash \phi$	by Lemma 3.4.

□

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

Corollary 3.8. Let $\mathcal{R} = [R, dom, pred]$ *be a rulebased system, and let* ϕ *be a ground hypothetical query constructed from the symbols in dom and pred. Then* $R + DB \vdash \phi$ *if and only if* $DB, M_{\mathcal{R}} \models \phi$

PROOF. The *if* direction is just Lemma 3.4. The *only if* direction follows from soundness: if $R + DB \vdash \phi$ then $R + DB \models \phi$ by soundness, so $DB, M_{\mathcal{R}} \models \phi$ as shown in the proof of Corollary 3.7. □

At this point, it is worth mentioning two properties of the canonical model: (*i*) The domain is constant, *i.e.*, does not vary from one substate to another; and (*ii*) every formula is defined at every substate. These properties helped simplify the proof of completeness. In addition, it follows from Corollary 3.8 that we can restrict the semantics of embedded implications to structures having these two properties. For instance, if R is a rulebase and ϕ is an embedded implication, then $R \models \phi$ if and only if $R \leftarrow \phi$ is true in all structures having *constant* domain. This is a more restrictive notion of entailment than that given by Definition 3.6. Thus, embedded implications can be viewed as having a kind of simplified intuitionistic semantics. This is true even for larger classes of embedded implications, such as those defined in [9].

4. LINEAR RECURSION

In [6], we showed that the data complexity of embedded implications is complete for PSPACE. In this section, we develop a syntactic restriction that reduces their complexity. The restriction is called *linearity*. Informally, a rule is linear if recursion occurs through only one premise. In classical Horn logic, "linear rules play an important role because, (*i*) there is a belief that most 'real life' recursive rules are linear, and (*ii*) algorithms have been developed to handle them efficiently" [4]. We first extend the notion of linearity from Horn clauses to embedded implications. We then show that linearity reduces their data complexity from PSPACE to NP. To prove the lower complexity bound, we use linear embedded implications to encode the computations of arbitrary NP-machines. To prove the upper bound, we show that if R is linear, and if $R + DB \vdash \psi$ can be derived, then it has a small derivation, *i.e.*, a derivation of polynomial size. Such derivations can be found in non-deterministic polynomial time. Besides establishing NP-completeness, the proofs in this section also form a model for the more complex proofs in Section 6, which involve negation-as-failure, oracle Turing machines, and the polynomial time hierarchy.

The first step is to define linearity precisely. We generalize the definition given in [4] for Horn rules. Central to this definition is the concept of mutually recursive predicates.

Definition 4.1. Let R be a rulebase. (*i*) A predicate P refers to a predicate Q if Q is a goal predicate of some rule in R with head predicate P. (*ii*) The reflexive, transitive closure of "refers" is denoted by the symbol "$\prec$". (*iii*) $P \equiv Q$ if $P \prec Q$ and $Q \prec P$.

If $P \prec Q$, then we say that P *depends* on Q; and if $P \equiv Q$, then we say that P and Q are *mutually recursive*. Note that mutual recursiveness is an equivalence relation.

Definition 4.2. [Linearity] Let r be a rule in a rulebase R. (*i*) r is recursive if it has at least one goal predicate Q that is mutually recursive with its head predicate P. (*ii*) r is linear if it has exactly one such goal predicate. (*iii*) The rulebase R is linear if every recursive rule in R is linear.

Example 4.1. [Linearity]

A Linear Rulebase	A Non-Linear Rulebase
$A \leftarrow (B \leftarrow D_1),\ (P \leftarrow D_2).$	$A \leftarrow (B \leftarrow D_1),\ (C \leftarrow D_2)$
$B \leftarrow (C \leftarrow E_1),\ Q.$	$B \leftarrow (C \leftarrow E_1),\ (A \leftarrow E_2)$
$C \leftarrow (A \leftarrow F_1)$	$C \leftarrow (A \leftarrow F_1),\ (B \leftarrow F_2)$

4.1. Data Complexity

Before establishing the data complexity of linear embedded implications, we first define the notion more precisely, especially in the context of rulebases.

Data complexity is the complexity of evaluating a database query when the query is fixed and the database is regarded as input. Formally, if ψ is a database query, then its data-complexity is the complexity of the language $\{\langle \overline{x}, DB \rangle \mid \overline{x} \in \psi(DB)\}$ where $\overline{x}$ is a tuple, DB is a database, and $\psi(DB)$ is the value of the query applied to DB [11, 51]. This language is called the *graph* of the query ψ. The data complexity of a set of queries is the complexity of their graphs. When studying the data complexity of a query language, one looks for the most complex query in the language. This motivates the following definition [11, 51].

Definition 4.3. A set of queries is data complete for a complexity class $\mathcal{C}$ if (1) the graph of each query is in $\mathcal{C}$, and (2) there is some query in the set whose graph is a complete language for $\mathcal{C}$.

A hypothetical rulebase defines a database query if one of its predicates is reserved as an output relation. For example, rulebase R and predicate OUT define the query whose graph is the language $\{\langle \overline{x}, DB \rangle \mid R + DB \vdash OUT(\overline{x})\}$. Actually, we must also define what inference system $[R, dom, pred]$ we are using. In particular, we must specify the set of constant symbols dom and the set of predicate symbols $pred$. For the purpose of expressing database queries, it is convenient to fix $pred$ independently of the database. It is also convenient to use $dom = dom(R) + dom(DB)$, where $dom(R)$ and $dom(DB)$ are the sets of constant symbols in R and DB, respectively. $dom(DB)$ is called the *data domain.* With this convention, a rulebase R and a predicate OUT uniquely define a database query. The language of embedded implications thus defines a set of queries. The following theorem, which is the main result of this section, establishes the data complexity of this set.

Theorem 4.1. The data complexity of linear embedded implications is NP-complete.

In [6], we showed that the data complexity of embedded implications is complete for PSPACE. The NP lower bound in Theorem 4.1 is implicit in the proof of the PSPACE lower bound. To establish this lower bound, [6] uses embedded implications to encode the computations of alternating Turing machines [13], which are a generalization of non-deterministic machines. To be more precise, let M be a one-tape Turing machine that runs in alternating polynomial time (an APTIME-machine), and let $\overline{s}$ be an input string for the machine. In [6], we encode $\overline{s}$ as a database $DB(\overline{s})$, and M as a rulebase $R(M)$ so that

$$R(M) + DB(\overline{s}) \vdash ACCEPT \quad \textit{if and only if} \quad M \textit{ accepts } \overline{s}. \tag{4.1}$$

where $ACCEPT$ is a 0-ary predicate symbol. The important point is that the rulebase $R(M)$ is independent of the input $\overline{s}$. This shows that the data-complexity of query processing is APTIME-hard. PSPACE-hardness follows immediately since PSPACE = APTIME [13].

As a special case, M may be an NP-machine. In this case, the rulebase $R(M)$ constructed in [6] is linear. (A generalization of these linear rulebases is constructed in Section 6.2, to encode NP oracle machines.) Thus, linear embedded implications can simulate the computations of arbitrary NP-machines. This proves the lower complexity bound of Theorem 4.1.

4.2. Upper Complexity Bound

This section establishes the upper bound of Theorem 4.1. That is, we show that the data complexity for linear embedded implications is in NP. Our strategy is to show that linear rulebases give rise to proof trees that are "narrow" and which therefore have a "small" number of nodes. More precisely, we show that if R is a linear rulebase, and if the expression $R + DB \vdash \psi$ is true, then the expression has a proof tree of polynomial size (polynomial in the size of the data domain). Most of this section is devoted to proving this result. Given this result, it is easy to show that inference is in NP: Given an expression $R + DB \vdash \psi$, we can non-deterministically "guess" a possible proof-tree of polynomial size, and then check in polynomial time whether it is a legal proof.

Proof Trees: This subsection gives a precise definition of proof trees for hypothetical inference and establishes basic results about them. The results apply to any rulebase of embedded implications, linear or not. The next subsection establishes stronger results that apply to linear rulebases only.

Definition 4.4. [Inference Expressions] If R is a rulebase, DB is a database, and ψ is a hypothetical query, then the expression $R, DB \vdash \psi$ is called an *inference expression.*

Definition 4.5. [Proof Trees] A *proof tree* for a rulebase R is a finite, rooted tree in which each node is labeled by an inference expression of the form $R, DB \vdash \psi$, where DB and ψ may vary from node to node. Each node in the tree must satisfy the following three conditions, where A, B and each C_i are ground atomic formulas:

- If the node is labeled by the expression $R, DB \vdash A$ where $A \in DB$, then the node has no children.
- If the node is labeled by the expression $R, DB \vdash A$ where $A \notin DB$, then for some ground instance $A \leftarrow \psi_1, \ldots, \psi_m$ of a rule in R, the node has exactly m children and the i^{th} child is labeled by the expression $R, DB \vdash \psi_i$.
- If the node is labeled by the expression $R, DB \vdash B \leftarrow C_1, \ldots, C_k$, then the node has exactly one child, and the child is labeled by the expression $R, DB + \{C_1, \ldots, C_k\} \vdash B$.

Definition 4.5 mimics the hypothetical inference system of Definition 2.4. In fact, it is not hard to see that $R, DB \vdash \psi$ is derivable in the inference system if and only if $R, DB \vdash \psi$ labels the root of some proof tree. In this case, we say that the tree is a *proof* of the expression $R, DB \vdash \psi$.

In general, a proof tree may be of arbitrary size, since the same inference expression may appear any number of times on a single branch of the tree, due to cycles in the proof. Such repetitions are redundant however. To show the existence of small proof trees, we focus on trees in which no expression occurs more than once on any given branch. This motivates the following definition.

Definition 4.6. [Minimal Proof Trees] A proof tree is *minimal* if no inference expression appears more than once on any branch of the tree.

The next lemma is a basic result about minimal proof trees. It shows that any proof tree can be reduced step-by-step to a minimal proof tree.

Lemma 4.2. The inference expression $R, DB \vdash \psi$ is true if and only if it has a minimal proof tree.

PROOF. $R, DB \vdash \psi$ is true if and only if it has a proof tree. We show that any proof tree can be transformed into a minimal proof tree. If a proof tree is not minimal, then it has a branch in which some inference expression ξ appears more than once. On this branch, let the occurrence of ξ which is closest to the root be called number 1, and let the next closest occurrence be called number 2. Now construct a new tree by replacing the subtree rooted at occurrence 1 by the subtree rooted at occurrence 2. We have thus replaced one subtree by a smaller subtree. Furthermore, the new tree is a proof tree. Because a proof tree is finite, we can apply this transformation repeatedly until a minimal proof tree is obtained. □

The next lemma shows that the depth of a minimal proof tree is polynomial. The crucial fact in the proof is that as one proceeds down a branch of a proof tree, the database never shrinks and can only get larger, because entries may be added to the database hypothetically, but never deleted.

Definition 4.7. [Paths] A path in a tree is a sequence of nodes $q_0, q_1, q_2, \ldots, q_m$ such that q_{i+1} is a child of q_i for $0 \leq i < m$.

Lemma 4.3. In a minimal proof tree, every path has length $O(n^{2k_0})$, where n is the size of the data domain and k_0 is the maximum number of distinct variables in any rule in the rulebase.

PROOF. Let $q_0, q_1, \ldots, q_m$ be a path in a minimal proof tree, and suppose that node q_i is labeled by the inference expression $R, DB_i \vdash \psi_i$. Because the proof tree is minimal, these expressions are all distinct. Moreover, as we traverse the path, the database is non-decreasing; that is, $DB_i \subseteq DB_{i+1}$. We can thus divide the path into segments such that within each segment, the database is the same, and between segments, it increases.

Within each segment, node labels have the form $R, DB \vdash \psi_i$ where the queries, ψ_i, are all distinct and ground. Thus, the length of a segment is no greater than the number of distinct ground queries. To estimate this number, observe that except for the root node, each query in a proof tree comes from the premise of a rule. Furthermore, each rule has $O(n^{k_0})$ ground instances, each giving rise to a fixed number of ground queries. For example, the ground rule $A \leftarrow (B \leftarrow C), (D \leftarrow E)$ gives rise to four queries: $B \leftarrow C$ and $D \leftarrow E$, as well as B and D. We thus have $O(n^{k_0})$ ground rules, each giving rise to $O(1)$ ground queries. A proof tree therefore contains only $O(n^{k_0})$ distinct ground queries. Each path segment thus has length $O(n^{k_0})$.

Lastly, note that in going from one path segment to the next, the database increases, that is, a set of ground atomic formula is added to it. This happens via ground hypothetical queries. For example, the ground query $B \leftarrow C, D$ adds the set $\{C, D\}$ to the database. However, because there are only $O(n^{k_0})$ distinct ground queries, there are only $O(n^{k_0})$ distinct sets of atoms that can be added to the database. Thus, on a given path, the database can increase only $O(n^{k_0})$ times before it saturates, *i.e.*, before the rulebase can no longer add new formulas to it. Thus, a path contains only $O(n^{k_0})$ segments, each of length $O(n^{k_0})$. The total length of a path is therefore $O(n^{2k_0})$. $\square$

Corollary 4.4. A minimal proof tree has polynomial depth (polynomial in the size of the data domain).

Proof Trees for Linear Rulebases: Corollary 4.4 assures us that for a rulebase of hypothetical additions, every minimal proof tree has polynomial depth. Such a tree may still have exponential size however, as the examples in [7, 5] show. However, these examples all involve non-linear rulebases. This section shows that such examples cannot be constructed from linear rulebases. In particular, we show that every minimal proof tree of a linear rulebase has polynomial size, polynomial in the size of the data domain.

Recall that linear recursion is defined in terms of mutual recursion, which is an equivalence relation on predicate symbols. Any maximal set of mutually-recursive predicates thus forms an equivalence class. These classes play a central role in our analysis, and for the purpose of this section, we refer to them as *clusters*. The clusters of a rulebase are therefore disjoint and exhaustive. Clusters divide a rulebase into recursive and non-recursive parts. Each cluster represents an island of recursion in which each predicate depends on every other predicate; whereas between clusters, there is no recursion. To study the non-recursive behavior of a rulebase, we look at the dependency of one cluster upon another.

Definition 4.8. [Dependency Graphs] The dependency graph of a rulebase is a directed graph whose nodes are the clusters of the rulebase. Furthermore, for any two distinct clusters, C_1 and C_2, there is an edge from C_1 to C_2 if and only if some predicate in C_1 depends on a predicate in C_2.

Because there is no recursion between clusters, the dependency graph is acyclic. This property allows us to assign a rank to each node in the graph based on the maximum distance from the node to a leaf (sink) in the graph. We then extend this idea from dependency graphs to proof trees in a straightforward way, allowing us to rank the nodes of a proof tree. The main result of this section is then proved by an induction over node rank.

Definition 4.9. [Rank] Let R be a rulebase of embedded implications.

- The rank of a cluster of R is the maximum distance in the dependency graph from the cluster to a sink (where sinks have rank 1).
- The rank of a hypothetical query, $A \leftarrow B_1, \ldots, B_k$, is the rank of the cluster to which A belongs.
- For a node in a proof tree labeled by the expression $R, DB \vdash \psi$, the rank of the node is the rank of ψ.

Intuitively, rank in a proof tree can be understood by dividing each branch of the tree into recursive phases, each dealing with predicates from a particular cluster. That is, within a recursive phase, nodes are labeled by expressions of the form $R, DB \vdash A \leftarrow B_1, \ldots, B_k$, where A always belongs to the same cluster. The rank of a node is then the maximum number of recursive phases from the node to a sink.

The following lemma states two basic facts about rank. The second statement is true because in a linear rulebase, at most one premise of a rule belongs to the same cluster as the head of the rule. It is this fact that limits the size of proof trees of linear rulebases.

Lemma 4.5. (i) In any proof tree, the rank of a node is no greater than the rank of its parent. (ii) In a proof tree for a linear rulebase, each node has at most one child of the same rank as itself.

We are now in a position to place an upper bound on the size of the proof trees of a linear rulebase. We do this by estimating the number of descendants of each node. The proof is by induction on node rank; *i.e.*, we first estimate the number of descendants for nodes of rank 1, then of rank 2, etc. The following lemma does precisely this and is the main result of this section.

Lemma 4.6. In a minimal proof tree for a linear rulebase, a node of rank r has $O(n^{2k_0 r})$ descendants, where n is the size of the data domain and k_0 is the maximum number of distinct variables in any rule in the rulebase.

PROOF. (By induction on rank)

Basis: Let q_0 be a node of rank 1. Because the rulebase is linear, q_0 can have at most one child whose rank is 1, by Lemma 4.5. Every other child must have a rank which is strictly less than 1. But 1 is the smallest rank. Thus, q_0 has at most one child q_1, and its rank is 1. Similarly, q_1 has at most one child q_2, and its rank is also 1. Thus we have a path $q_0, q_1, q_2, \ldots, q_m$ in the proof tree, consisting of the descendants of q_0. By Lemma 4.3, this path has length $O(n^{2k_0})$. Node q_0 thus has $O(n^{2k_0})$ descendants.

Induction: First note that each node in the proof tree has at most m_0 children, where m_0 is the maximum number of premises of any rule in the rulebase. m_0 is therefore constant.

Now suppose that the lemma is true for nodes of rank $r-1$, and let q_0 be a node of rank r. By Lemma 4.5, q_0 has at most one child of rank r. Call it q_1. Similarly, q_1 has at most one child of rank r. Call it q_2. In this way, we define a path $q_0, q_1, q_2, \ldots, q_m$, where each q_i has rank r, and every child of q_i that is not on the path has rank less than r. By Lemma 4.3, this path has length $O(n^{2k_0})$. Moreover, each of the off-path children has $O[n^{2k_0(r-1)}]$

descendants, by induction hypothesis.

Every node in the proof tree has at most m_0 children, of whatever rank. Thus, there are $O(n^{2k_0})$ nodes on the path $q_0, \ldots, q_m$, and each of these nodes has at most m_0 other children, and each of these children has $O[n^{2k_0(r-1)}]$ descendants. Thus, the total number of descendants of q_0 is

$$O(n^{2k_0}) \; + \; O(n^{2k_0}) \times m_0 \times O[n^{2k_0(r-1)}] \quad = \quad O(n^{2k_0 r})$$

This equation shows that each application of the induction principle introduces a additional hidden factor of m_0. Thus, the term $O(n^{2k_0 r})$ has a hidden factor of m_0^r, which is constant since r is.

□

The rank of a node in a proof tree is just the rank of some cluster in the rulebase, R. Let r_0 be the maximum rank of any cluster in R. Then r_0 is an upper bound on the rank of every node in every proof tree for R. In particular, r_0 bounds the rank of the root node. Thus, by Lemma 4.6, the root of a proof tree has $O(n^{2k_0 r_0})$ descendants. Since k_0 and r_0 are independent of n, we have the following result.

Corollary 4.7. A minimal proof tree for a linear rulebase has polynomial size (polynomial in the size of the data domain).

By combining Corollary 4.7 and Lemma 4.2, we get the following result.

Theorem 4.8. Let R be a linear rulebase of embedded implications. Then the inference expression $R, DB \vdash \psi$ is true if and only if it has a proof tree of polynomial size (polynomial in the size of the data domain).

A Proof Procedure: It follows immediately from Theorem 4.8 that inference for linear rulebases is in NP. To see this, first non-deterministically "guess" a possible proof tree of polynomial size. Then check in polynomial time whether it represents a legal proof. This section presents a proof procedure based on this idea. This procedure is also a model for a more complex procedure developed in Section 6.1 for linear rulebases with negation-as-failure.

The procedure PROVE, below, attempts to generate a proof tree for the expression $R, DB \vdash \psi$, and it attempts to do so in a top-down, non-deterministic manner. The procedure effectively combines the guessing phase with the checking phase. Note that each of the conditions 1–3 in the procedure corresponds to a condition in Definition 4.5.

Procedure: PROVE(DB, ψ)
1. if $\psi \in DB$ then return *true*;
2. elseif ψ is atomic then
    choose a ground instance $A \leftarrow \psi_1, \ldots, \psi_m$ of a rule in R;
    if $A = \psi$ then return $\wedge_j$ PROVE(DB, ψ_j);
        else return *false*;
3. elseif $\psi = A \leftarrow B_1, \ldots, B_k$ then return PROVE($DB + \{B_1, \ldots, B_k\}, A$);

The choice point in condition 2 of this procedure is non-deterministic. It "guesses" a ground instance of a rule. The subsequent test, "if $A = \psi$," then determines whether the guess can be used to extend the proof tree in a top-down fashion. If the guess fails the test, then the procedure returns *false*; if it passes the test, then the procedure calls itself recursively, thereby extending the proof tree.

If, for a particular sequence of guesses, PROVE(DB, ψ) returns *true*, then during execution, the procedure effectively generates a proof tree for $R, DB \vdash \psi$ in pre-order (*i.e.*, depth first and left to right). Conversely, any proof tree for $R, DB \vdash \psi$ can be generated in this way by some sequence of guesses. These comments hold for any rulebase, R, of embedded implications, linear or not. We thus have the following result.

Lemma 4.9. The expression $R, DB \vdash \psi$ *has a proof tree iff* PROVE(DB, ψ) *returns true for some sequence of guesses (in which case, the procedure generates a proof tree in pre-order during execution).*

For any sequence of guesses, the execution time of PROVE(DB, ψ) is a polynomial function of the size of the tree that it generates. In the special case in which R is linear, Theorem 4.8 assures us that if $R, DB \vdash \psi$ is true, then it has a proof tree of polynomial size; so PROVE has an accepting computation of polynomial length. Thus, for linear rulebases, PROVE runs in NP-time [21]. This establishes the upper bound of Theorem 4.1.

Although this completes the main task of this section, we can use the development to produce a new proof that the data complexity of general embedded implications is in PSPACE.[1]The proof is similar to the one just given for linear rulebases, but it exploits the polynomial *depth* of minimal proof trees, instead of polynomial *size*. We first combine Lemma 4.2 and Corollary 4.4 to give the following result.

Lemma 4.10. Suppose R is a rulebase of embedded implications (not necessarily linear). Then, the expression $R, DB \vdash \psi$ *is true if and only if it has a proof tree of polynomial depth.*

We augment the procedure PROVE with a counter so that it halts (and fails) after polynomially-many levels of recursion. Call the augmented procedure PROVE$'$. This procedure generates only proof trees of polynomial depth. Conversely, any proof tree of polynomial depth can be generated by PROVE$'$ for some sequence of guesses. Thus, by Lemma 4.10, an inference expression $R, DB \vdash \psi$ is true if and only if PROVE$'$(DB, ψ) returns true for some sequence of guesses.

The complexity of PROVE$'$ is easily established. For any particular sequence of guesses, PROVE$'$ makes polynomially-many recursive calls, each taking polynomial time. This computation can be done in polynomial space. Thus, the entire computation, including guesses, runs in non-deterministic polynomial space. The data complexity of embedded implications is therefore in NPSPACE, and thus in PSPACE, since PSPACE = NPSPACE, by Savitch's Theorem [25].

5. NEGATION AS FAILURE

Section 4 showed that the data complexity of linear embedded implications is NP complete. However, despite this great computational power, there are some simple, low-complexity queries that embedded implications cannot express. This is

[1]This fact was originally proved in [6].

because, like all inference systems, embedded implications are *monotonic*: As the database expands, the answer to a query also expands. Such systems cannot express non-monotonic queries, such as retrieving those students who are *not* eligible to graduate. To capture such queries, this section augments embedded implications with a well-known non-monotonic operator: *negation-as-failure*. This is a natural extension to the logic, since any practical logic-programming language has to incorporate negation-as-failure in some form. This operator also allows the logic to express *all* the database queries in NP, both monotonic and non-monotonic. In fact, Sections 6 and 7 establish a strong relationship between linear embedded implications, negation-as-failure, and computational complexity.

In this section, we extend the inference system of Definition 2.4 to allow rules with negated premises. Thus, rules of the form $A \leftarrow\, \sim (B \leftarrow C)$ are allowed. The expression $\sim (B \leftarrow C)$ is interpreted as the failure to prove $B \leftarrow C$. Thus, A is inferred if $B \leftarrow C$ cannot be inferred. Unfortunately, as in Horn logic programming, the semantics of negation-as-failure is problematic when recursion occurs through negation. To avoid these problems, we focus on a class of *stratified* rulebases [2, 12]. We first extend the idea of stratification from Horn rulebases to hypothetical rulebases, and then develop the proof theory.

The material in this section is adapted from [5], where we develop the model theory and proof theory for hypothetical deletion as well as insertion. That proof theory is equivalent to the one developed in [6], but is simpler and more intuitive, and is a straightforward extension of negation-free inference. In [8], we develop an extension of intuitionistic model theory for which the proof theory presented here is sound and complete.

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

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

In applying the rule for $\mathit{near\_grad}(s)$, we ask if the student s is within one course of graduation. That is, is there some course c such that if $\mathit{take}(s,c)$ were assumed to be true, then $\mathit{grad}(s)$ would also be true? The student is eligible for a fellowship only if this hypothetical test fails. Conversely, he is eligible for a stipend only if it succeeds. Note that this hypothetical test is vacuously true if $\mathit{grad}(s)$ is true (as long as there exists a course somewhere in the database!). But we do not want to give a stipend to a student who has already graduated, so we include the test $\sim \mathit{grad}(s)$ in the rule for $\mathit{stipend}(s)$. This means that a stipend is available only to those students who need *exactly* one course to graduate.

5.1. *Stratified Hypothetical Rulebases*

This subsection gives a precise definition of stratified embedded implications, adapting the definitions of [6]. The first step is to extend some of the definitions given in Section 2.

Definition 5.1. [Hypothetical Queries] A positive hypothetical query is a formula of the form $B \leftarrow C_1, C_2, \ldots, C_k$ where B and each C_i are atomic formulas, and $k \geq 0$. A negative hypothetical query is a formula of the form $\sim (B \leftarrow C_1, C_2, \ldots, C_k)$.

Definition 5.2. [Hypothetical Rules] An embedded implication is a formula of the form $A \leftarrow \phi_1, \phi_2, \ldots, \phi_k$ where A is an atomic formula, each ϕ_i is a hypothetical query, and $k \geq 0$.

As in the negation-free case, when a hypothetical query has an empty premise, then we write B instead of $B \leftarrow$. In addition, goal predicates are extended to include negative queries. Thus, in the rule $A(x) \leftarrow B_1(x, y),\ \sim [B_2(y) \leftarrow C(y)]$, both B_1 and B_2 are goal predicates. For the time being, we allow infinite rulebases with infinitely many strata.

Definition 5.3. [Stratification] Let R be a set of embedded implications with possibly negated premises. A stratification is a sequence, $\mu = \langle \mu_1, \mu_2, \mu_3, \ldots, \rangle$, of sets of predicate symbols such that every predicate symbol in the language is in exactly one of the μ_i. The sequence μ stratifies R if R is a disjoint union of sets, $R_1, R_2, R_3, \ldots$, satisfying all of the following conditions:

1. each R_i contains rules defining predicates in μ_i only;
2. $P \in \cup_{j<i}\ \mu_j$ if P is the goal predicate of a negative premise of a rule in R_i;
3. $P \in \cup_{j\leq i}\ \mu_j$ if P is the goal predicate of a positive premise of a rule in R_i.

Definition 5.3 guarantees that if recursion occurs, then it occurs only within a single stratum and that it never occurs through negation. In this definition, we call R_i the i^{th} *stratum* of R. We shall sometimes refer to R_1 as the *bottom stratum*. We say that the predicate symbols in μ_i *belong* to the i^{th} stratum of R, and that a hypothetical query belongs to the same stratum as its goal predicate. We say that a rulebase is *stratified* if it has a stratification.

Example 5.2. [Stratification] The following rulebase is stratified and has three strata, R_1, R_2, R_3:

$$R_3 \begin{cases} A_2(x) \leftarrow B_2(x, y),\ [A_2(y) \leftarrow C_3(y)]. \\ A_2(x) \leftarrow D_2(x),\ \sim [A_1(x) \leftarrow C_2(x)]. \end{cases}$$

$$R_2 \begin{cases} A_1(x) \leftarrow B_1(x, y),\ [A_1(y) \leftarrow C_1(y)]. \\ A_1(x) \leftarrow D_1(x),\ \sim A_0(x). \end{cases}$$

$$R_1 \begin{cases} A_0(x) \leftarrow B_0(x, y),\ [A_0(y) \leftarrow C_0(y)]. \\ A_0(x) \leftarrow D_0(x). \end{cases}$$

Finally, we extend Definition 2.3 to include stratified rulebases.

Definition 5.4. A stratified rulebased system is a quadruple $\mathcal{R} = [R, \mu, dom, pred]$ where R is a set of embedded implications, μ is a stratification of R, dom is a set of constant symbols, and pred is a set of predicate symbols with associated arities. dom and pred must include all the constant and predicate symbols in R, respectively.

5.2. Stratified Inference

Given a stratified rulebased system, we define a sequence of inference systems, one for each stratum. These systems generalize the inference system of Definition 2.4 in two ways. First, each system corresponds to a particular stratum, so that the j^{th} system uses only those rules in the j^{th} stratum. Second, each inference system is provided with an unspecified set of axioms, $\mathcal{A}$. Each inference system thus defines a mapping, cl_j, that takes a set of axioms, $\mathcal{A}$, as input, and returns a set of inferred expressions, $cl_j(\mathcal{A})$, as output. The proof theory is defined stratum-by-stratum in terms of these mappings. Given the output from cl_j, we apply the closed world assumption to it, and use the resulting expressions as the input to cl_{j+1}.

To simplify the presentation, this section assumes that a stratified rulebased system $\mathcal{R} = [R, \mu, dom, pred]$ is given. All formulas are constructed from the constant symbols in *dom* and the predicate symbols in *pred*, and all ground instantiations are with respect to *dom*. $\mathcal{DB}$ denotes the set of all ground atomic formulas constructible from *dom* and *pred*. For convenience, we refer to ground hypothetical queries as *goals*.

Definition 5.5. [Inference Expressions] An inference expression for R is an expression of the form $R + DB \vdash \phi$ where DB is a database and ϕ is a goal. The inference expression is positive if ϕ is positive, and negative if ϕ is negative.

Definition 5.6. [Inference] Let $\mathcal{A}$ be a set of inference expressions for R. The axioms and inference rules below form an inference system for the j^{th} stratum of R, where $DB \subseteq \mathcal{DB}$. The set of inference expressions derivable in this system is denoted $cl_j(\mathcal{A})$.

Axioms:

1. $R + DB \vdash A$ is an axiom, for every atomic formula $A \in DB$.
2. Each expression in $\mathcal{A}$ is an axiom.

Inference Rules:

1. If $A \leftarrow \phi_1, \ldots, \phi_m$ is a ground instantiation of a rule in the j^{th} stratum of R, then

$$\frac{R + DB \vdash \phi_i \quad \text{for each } i}{R + DB \vdash A}$$

2. If $B \leftarrow C_1, \ldots, C_k$ is a goal belonging to the j^{th} stratum of R, then

$$\frac{R + DB + \{C_1, \ldots, C_k\} \vdash B}{R + DB \vdash B \leftarrow C_1, \ldots, C_k}$$

As in Definition 2.4, the inference system above is intuitionistic. As with all logical inference systems, this system is monotonic in the set of axioms, as well as idempotent and inflationary. We thus have the following basic result.

Lemma 5.1. *cl_j is a function that maps sets of inference expressions for R into sets of inference expressions for R. Moreover, it has the following properties:*

- *Monotonicity:* If $\mathcal{A} \subseteq \mathcal{B}$ *then* $cl_j(\mathcal{A}) \subseteq cl_j(\mathcal{B})$.
- *Idempotence:* $cl_j(\mathcal{A}) = cl_j(cl_j(\mathcal{A}))$
- *Inflation:* $\mathcal{A} \subseteq cl_j(\mathcal{A})$

Definition 5.7. [Closed World Assumption] Let $\mathcal{A}$ be a set of inference expressions for R. Then $\overline{\mathcal{A}}$ is another set of inference expressions for R, where for any database $DB \subseteq \mathcal{DB}$, and any positive goal ϕ,

- $R + DB \vdash \phi \ \in \ \overline{\mathcal{A}}$ *iff* $R + DB \vdash \phi \ \in \ \mathcal{A}$
- $R + DB \vdash \sim \phi \ \in \ \overline{\mathcal{A}}$ *iff* $R + DB \vdash \phi \ \notin \ \mathcal{A}$

Definition 5.8. [Inference with Negation]

- $\mathcal{A}^0 \ = \ \overline{\{\}}$
- $\mathcal{A}^{j+1} \ = \ \overline{cl_{j+1}(\mathcal{A}^j)}$ for $j \geq 0$.
- $\mathcal{A}^* \ = \ \overline{\bigcup_{j \geq 0} \mathcal{A}^j}$

Definition 5.9. Let DB be a database, and let ϕ be a goal. Then $R + DB \vdash \phi$ if the expression $R + DB \vdash \phi$ is in $\mathcal{A}^*$.

This completes the proof theory of stratified embedded implications. Additional details and discussion can be found in [5, 8]. Note that although the proof theory is defined in a bottom-up fashion, top-down inference is also possible, as the following examples show.

Example 5.3. [Stratified Inference] Suppose $B \notin DB$, and R is the rulebase consisting of the single rule $A \leftarrow \sim (B \leftarrow C)$. Then $R + DB \vdash A$. This can be proved by a straightforward top-down argument:

$$
\begin{array}{ll}
 & R + DB \ \vdash \ A \\
\textit{if} & R + DB \ \vdash \sim (B \leftarrow C) \\
\textit{if} & R + DB \ \not\vdash \ B \leftarrow C \\
\textit{if} & R + DB + C \ \not\vdash \ B
\end{array}
$$

But the last line is trivially true, since no rules in R infer B, and $B \notin DB + C$.

Example 5.4. [Hamiltonian Path] Suppose that DB is a database representing a directed graph. That is, $NODE(a) \in DB$ if and only if a is a node in the graph, and $EDGE(a, b) \in DB$ if and only if there is an edge in the graph from a to b. Suppose also that R is the following collection of rules:

$$
\begin{array}{l}
YES \ \leftarrow \ SELECT(x), \ [PATH(x) \leftarrow MARK(x)] \\
PATH(x) \ \leftarrow \ EDGE(x, y), \ SELECT(y), \ [PATH(y) \leftarrow MARK(y)] \\
PATH(x) \ \leftarrow \ \sim SOMELEFT. \\
SOMELEFT \ \leftarrow \ SELECT(x). \\
SELECT(x) \ \leftarrow \ NODE(x), \ \sim MARK(x).
\end{array}
$$

Then $R + DB \vdash YES$ if and only if the graph represented by DB has a directed Hamiltonian path.

This can be seen in terms of top-down inference. From this perspective, the rulebase tries to construct a Hamiltonian path incrementally. The first rule selects a node, x, to begin the path. The second rule then extends the path one node at a time. Recursively, it selects a node, y, connected to the last node in the path by a single edge. Each time a node is selected, it is marked so that it will not be selected again. In this way, no node is selected twice. The third and fourth rules say that a Hamiltonian path has been found when there are no unmarked nodes left in the graph, that is, when every node has been visited exactly once. Note that each node selection is non-deterministic, so in effect, the rulebase searches the graph for all possible Hamiltonian paths.

6. THE POLYNOMIAL TIME HIERARCHY

Section 4 showed that linear recursion reduces the data complexity of embedded implications from PSPACE to NP. Assuming that NP $\neq$ coNP, it should not be surprising that negation-as-failure affects the data complexity of linear rulebases. For instance, if $A(x)$ is an NP-complete query, then $\sim A(x)$ is a coNP-complete query.[2]This section shows that the data complexity of linear embedded implications corresponds to some level in the polynomial-time hierarchy, depending on the number and type of strata in the rulebase. Section 7 uses these results to characterize the expressibility of linear embedded implications, and to characterize the second-order definable queries in terms of embedded implications.

The polynomial-time hierarchy is a sequence of complexity classes between P and PSPACE. It is based on oracle Turing machines [25], and can be defined recursively as follows [47]:

- $\Sigma_0^{\mathrm{P}} = \Delta_0^{\mathrm{P}} = \mathrm{P}$.
- $\Delta_{k+1}^{\mathrm{P}} = \mathrm{P}^{\Sigma_k^{\mathrm{P}}}$ = those languages accepted in deterministic polynomial time by an oracle machine whose oracle is a language in Σ_k^{P}.
- $\Sigma_{k+1}^{\mathrm{P}} = \mathrm{NP}^{\Sigma_k^{\mathrm{P}}}$ = those languages accepted in non-deterministic polynomial time by an oracle machine whose oracle is a language in Σ_k^{P}.
- $\mathrm{PHIER} = \cup_k \Sigma_k^{\mathrm{P}} = \cup_k \Delta_k^{\mathrm{P}}$.

Note that $\Delta_1^{\mathrm{P}} = \mathrm{P}^{\mathrm{P}} = \mathrm{P}$. Likewise, $\Sigma_1^{\mathrm{P}} = \mathrm{NP}^{\mathrm{P}} = \mathrm{NP}$. It is well known that $\mathrm{P} \subseteq \Delta_k^{\mathrm{P}} \subseteq \Sigma_k^{\mathrm{P}} \subseteq \Delta_{k+1}^{\mathrm{P}} \subseteq \mathrm{PHIER} \subseteq \mathrm{PSPACE}$.[3]

This section proves several results linking the polynomial-time hierarchy to embedded implications. The simplest result to state is the following: The data complexity of linear rulebases with k strata is complete for Σ_k^{P}.[4]It turns out, however,

[2]For general embedded implications, the data complexity is complete for PSPACE [6]. In this case, negation has no effect on the data complexity, since PSPACE = coPSPACE [25].

[3]Although considered likely, it is an open question as to whether any of these containments are strict.

[4]It would be tempting to conclude that the set of all stratified rulebases is complete for the entire polynomial time hierarchy; but this would be incorrect. It is not known whether PHIER has any complete problems, and their existence is considered unlikely, since they would imply the

that additional strata can often be added to a rulebase with only a small cost in complexity, as long as the new strata contain only Horn rules (with possibly negated premises). We shall say that such strata are *non-hypothetical*, while other strata are *hypothetical*. The rulebase in Example 5.4 has one hypothetical stratum (the first three rules), and one non-hypothetical stratum (the last two rules). We show that for linear rulebases having k *hypothetical* strata, the data complexity is no more than $\Delta^{\mathrm{P}}_{k+1}$. Thus, no matter how many non-hypothetical strata are added, the data complexity will not exceed $\Delta^{\mathrm{P}}_{k+1}$. It turns out that non-hypothetical strata at the top of a rulebase account for the entire increase in complexity. Thus, if the top stratum is hypothetical, then there is no increase in data complexity, which remains complete for Σ^{P}_{k}. Thus, any number of non-hypothetical strata can be added *between* two hypothetical strata without affecting the data complexity. The following three theorems are a precise statement of these results. Each theorem is formulated in terms of a set of embedded implications $\mathcal{R}^i_k$ that are stratified and linear. The first theorem focuses on the number of strata, the second focuses on hypothetical strata, and the third assumes that the top stratum is hypothetical.

Definition 6.1. $R \in \mathcal{R}^1_k$ if R is linear and there is a stratification of R with at most k strata.

Theorem 6.1. For $k \geq 1$, the data complexity of $\mathcal{R}^1_k$ is complete for Σ^{P}_{k}.

Definition 6.2. $R \in \mathcal{R}^2_k$ if R is linear and there is a stratification of R with at most k hypothetical strata.

Theorem 6.2. For $k \geq 1$, the data complexity of $\mathcal{R}^2_k$ is in $\Delta^{\mathrm{P}}_{k+1}$ and is hard for Σ^{P}_{k} and coΣ^{P}_{k}

Definition 6.3. $R \in \mathcal{R}^3_k$ if R is linear and there is a stratification of R with at most k hypothetical strata and for which the top stratum is hypothetical.

Theorem 6.3. For $k \geq 1$, the data complexity of $\mathcal{R}^3_k$ is complete for Σ^{P}_{k}.

These three theorems remain true if in the definition of $\mathcal{R}^i_k$, the phrase "at most k" is replaced by "exactly k". Furthermore, the rulebases in $\mathcal{R}^i_k$ do not have to be entirely linear: The hypothetical strata must be linear, but the non-hypothetical strata may be non-linear. For succinctness sake, though, we shall refer simply to "linear rulebases".

6.1. Upper Complexity Bounds

This section establishes the upper bounds of Theorems 6.1, 6.2 and 6.3. Given a rulebase, we present a mixed top-down/bottom-up proof procedure in which each stratum is processed separately. Each non-hypothetical stratum is processed by a

collapse of the hierarchy. Suppose, for instance, that some problem, $\mathcal{P}$, in PHIER were complete for PHIER. Then $\mathcal{P}$ would be in Σ_k for some k. Moreover, any problem in PHIER would be reducible in polynomial time to $\mathcal{P}$. This means that all problems in levels higher than Σ_k would be reducible to a problem in Σ_k. All these problems would therefore be in Σ_k, so the hierarchy above Σ_k would collapse to a single level.

bottom-up procedure that runs in polynomial time; and each hypothetical stratum is processed by a top-down procedure that runs in non-deterministic polynomial time. To simplify the presentation, we assume that a stratified rulebased system $\mathcal{R} = [R, \mu, dom, pred]$ is given. Moreover, we assume that the system is finite, *i.e.*, that R, μ, *dom* and *pred* are all finite. All formulas are built from *dom* and *pred*, and all ground instantiations are with respect to *dom*.

We first remark that the upper bound in Theorem 6.1 is a straightforward consequence of the upper bounds in Theorems 6.2 and 6.3. To see this, suppose that a linear rulebase has k strata. Consider two cases. (i) If all k strata are hypothetical, then we are done, by Theorem 6.3. (ii) If at most $k-1$ strata are hypothetical, then the data complexity of the rulebase is in Δ_k^{P}, by Theorem 6.2. But $\Delta_k^{\mathrm{P}} \subseteq \Sigma_k^{\mathrm{P}}$, so again we are done. The rest of this section is therefore devoted to proving the upper bounds of Theorems 6.2 and 6.3.

Our approach is to define a series of proof procedures $\mathrm{PROVE}_1, \ldots, \mathrm{PROVE}_k$ one for each stratum. PROVE_i takes two arguments, a database DB, and a goal ψ. If ψ belongs at or below the i^{th} stratum, then $\mathrm{PROVE}_i(DB, \psi)$ returns *true* if and only if $R + DB \vdash \psi$ is true. Each PROVE_i invokes PROVE_{i-1} as a subroutine, *i.e.*, as an oracle. Thus, if the rulebase has a total of k strata, then the overall proof procedure is a cascade of oracle machines starting at PROVE_k, which invokes PROVE_{k-1}, which invokes PROVE_{k-2}, etc. For *any* goal, ψ, the procedure $\mathrm{PROVE}_k(DB, \psi)$ returns *true* if and only if $R + DB \vdash \psi$ is true.

Each procedure PROVE_i is one of two types, depending on the form of the i^{th} stratum of the rulebase. If the i^{th} stratum is non-hypothetical, then PROVE_i is a bottom-up, iterative procedure based on the bottom-up procedure for computing the least fixpoint of a Horn rulebase. This procedure can be viewed as a P-machine that uses PROVE_{i-1} as an oracle. On the other hand, if the i^{th} stratum is hypothetical, then PROVE_i is a top-down, recursive procedure based on the procedure PROVE developed in Section 4.2 for negation-free linear rulebases. PROVE_i can be viewed as an NP-machine that uses PROVE_{i-1} as an oracle. The overall proof procedure can thus be viewed as a cascade of oracle machines each of which is either a P-machine or an NP-machine. Moreover, if the rulebase has k *hypothetical* strata, then at most k of these machines are NP-machines. Thus, the overall procedure is in $\Delta_{k+1}^{\mathrm{P}}$. In addition, if the top stratum of the rulebase is hypothetical, then the top oracle-machine is an NP-machine, so the overall procedure is in Σ_k^{P}.

As an example, suppose that the rulebase has three strata all of which are hypothetical. Then the data complexity of the rulebase is $\mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}} = \Sigma_3^{\mathrm{P}}$. On the other hand, suppose that the middle stratum is non-hypothetical. Then the data complexity of the rulebase is in $\mathrm{NP}^{\mathrm{P}^{\mathrm{NP}}} = \mathrm{NP}^{\mathrm{NP}} = \Sigma_2^{\mathrm{P}}$, where the first equality follows because an oracle in P^{NP} can be simulated in polynomial time with an oracle in NP. Finally, suppose that the top two strata are non-hypothetical and the bottom stratum is hypothetical. Then the data complexity of the rulebase is in $\mathrm{P}^{\mathrm{P}^{\mathrm{NP}}} = \mathrm{P}^{\mathrm{NP}} = \Delta_2^{\mathrm{P}}$, where the first equality follows as before. In general, all the P-machines in a cascade can be ignored, except possibly the topmost one.

For convenience, we write $\mathrm{PROVE}_i^{\mathrm{P}}$ to indicate that PROVE_i is a P-machine, and $\mathrm{PROVE}_i^{\mathrm{NP}}$ to indicate that it is an NP-machine.

The Procedure $\mathrm{PROVE}_i^{\mathrm{P}}$*:* If the i^{th} stratum of the rulebase is non-hypothetical, then it is processed by the procedure $\mathrm{PROVE}_i^{\mathrm{P}}$. Assuming that ψ belongs at or below the i^{th} stratum, $\text{PROVE}_i^{\text{P}}(DB, \psi)$ returns *true* if and only if $R + DB \vdash \psi$. $\text{PROVE}_i^{\text{P}}$ is a bottom-up iterative procedure based on the bottom-up procedure for computing the least fixpoint of a Horn rulebase [50, 3]. The idea is to add inferred atoms to a set, S, until saturation is reached. Starting with $S = DB$, the procedure repeatedly applies ground instantiations of the rules in the i^{th} stratum. For each ground rule, the head of the rule is added to S if S satisfies the premises of the rule. This process is repeated until no new atoms can be inferred. Termination is guaranteed since for a finite rulebased system, $[R, \mu, dom, pred]$, the set of all ground atomic formulas is finite.

Although the i^{th} stratum is non-hypothetical, it may contain rules whose premises belong to lower strata. In this case, the procedure PROVE_{i-1} is invoked as a subroutine to determine whether these premises are true. If PROVE_{i-1} is treated as an oracle, which returns in constant time, then $\text{PROVE}_i^{\text{P}}$ runs in polynomial time (polynomial in the size of the data domain, dom), just like the least-fixpoint computation for a set of function-free Horn rules. Except for a brief initialization phase, the database, DB, is fixed during the entire computation.

The procedure $\text{PROVE}_i^{\text{P}}$ is given below. If ψ belongs below the i^{th} stratum, then it is passed directly to the oracle, PROVE_{i-1}; otherwise, it is reduced to a positive, atomic goal. A "least fixpoint" computation is then carried out; and *true* is returned if and only if the reduced goal is in the "least fixpoint."

Procedure: $\text{PROVE}_i^{\text{P}}(DB, \psi)$
    if ψ belongs below the i^{th} stratum, then return $\text{PROVE}_{i-1}(DB, \psi)$;
    elseif $\psi = \sim \psi_0$ then return $not[\text{PROVE}_i^{\text{P}}(DB, \psi_0)]$;
    elseif $\psi = A \leftarrow B_1, \ldots, B_k$ then return $\text{PROVE}_i^{\text{P}}(DB + \{B_1, \ldots, B_k\}, A)$;
    elseif $\psi \in \text{LFP}_i(DB)$ then return *true*;
    else return *false*;

The "least fixpoint" computation is invoked in the fifth line of this procedure by the expression $\text{LFP}_i(DB)$, and is implemented by the three procedures below. The procedures LFP_i and V_i apply the rules in the i^{th} stratum of R repeatedly until saturation, as in Horn clause logic. The procedure $\text{MATCH}_i(DB, S, \psi)$ determines whether a ground hypothetical query, ψ, is trivially true given a set of inferred atoms, S. There are two cases. If ψ belongs to the i^{th} stratum, then since the rulebase is stratified, ψ must be a positive literal. In this case, MATCH_i determines whether $\psi \in S$. If ψ belongs to a *lower* stratum, then MATCH_i invokes PROVE_{i-1} as an oracle (in which case, ψ may be positive or negative.) The database DB is simply passed to PROVE_{i-1} as an argument. Observe that $DB \subseteq S$ whenever MATCH_i is invoked.

Procedure: $\text{LFP}_i(DB)$
    $S_1 := DB$;
    $S_2 := \text{V}_i(DB, S_1)$;
    do until $S_1 = S_2$
        $S_1 := S_2$;
        $S_2 := \text{V}_i(DB, S_2)$;
        end do;
    return S_2;

Procedure: $\mathrm{V}_i(DB, S_1)$
  $S_2 := S_1$;
  for each ground instantiation, $A \leftarrow \psi_1, \ldots, \psi_m$,
  of each rule in the i^{th} stratum of R, do
    if $\mathrm{MATCH}_i(DB, S_1, \psi_j)$ is true for each $1 \leq j \leq m$
      then $S_2 := S_2 \cup \{A\}$;
    end do;
  return S_2;

Procedure: $\mathrm{MATCH}_i(DB, S, \psi)$
  if ψ belongs below the i^{th} stratum, then return $\mathrm{PROVE}_{i-1}(DB, \psi)$;
  elseif $\psi \in S$ then return *true*;
    else return *false*;

It is not hard to see that these procedures run in polynomial time relative to an oracle for PROVE_{i-1}. First recall that they operate on a finite rulebased system, $[R, \mu, dom, pred]$. Since we are estimating data complexity, only the data domain, *dom*, may vary; so let n be the size of *dom*. The set of all ground instantiations of all rules in R is of size $O(n^j)$, where j is the maximum number of variables in any rule in R. The set of all ground atomic formulas is of size $O(n^k)$, where k is the maximum arity of any predicate symbol in *pred*. The sets S, S_1, S_2 and DB are thus also of size $O(n^k)$. All sets used by the procedures are therefore of polynomial size. If PROVE_{i-1} is treated as an oracle, then MATCH_i runs in polynomial time. Hence V_i also runs in polynomial time, as does each iteration of the loop in LFP_i. The only remaining issue is to show that this loop undergoes only polynomially many iterations.

To see this, note that $V_i(DB, S)$ adds atoms to S, but never removes any. Thus $S \subseteq V_i(DB, S)$. Hence, with each iteration of the loop in LFP_i, the set S_2 increases monotonically. The loop terminates when S_2 does not change; so with each iteration, except the last, at least one atom is added to S_2. Hence, there can be at most $O(n^k)$ iterations before S_2 saturates and equals the set of all ground atomic formulas, at which point the loop would terminate immediately. The loop therefore undergoes polynomially many iterations, each taking polynomial time; so the entire procedure, LFP_i, runs in polynomial time. Since $\mathrm{PROVE}_i^{\mathrm{P}}$ merely carries out simple preprocessing before invoking LFP_i, it too runs in polynomial time relative to an oracle for PROVE_{i-1}.

The Procedure $\mathrm{PROVE}_i^{\mathrm{NP}}$*:* If the i^{th} stratum of the rulebase is hypothetical, then it is processed by the procedure $\mathrm{PROVE}_i^{\mathrm{NP}}$, which is based on the procedure PROVE developed in Section 4.2 for negation-free linear rulebases. Assuming that ψ belongs at or below the i^{th} stratum, $\mathrm{PROVE}_i^{\mathrm{NP}}(DB, \psi)$ returns *true* if and only if the expression $R + DB \vdash \psi$ is true. Unlike $\mathrm{PROVE}_i^{\mathrm{P}}$, the database may change frequently during the execution of $\mathrm{PROVE}_i^{\mathrm{NP}}$.

In a brief pre-processing phase, $\mathrm{PROVE}_i^{\mathrm{NP}}$ reduces ψ to a positive goal, and then passes it to the procedure TEST_i. TEST_i is a top-down, recursive procedure based on the procedure PROVE developed in Section 4.2 for negation-free linear rulebases. TEST_i non-deterministically chooses a rule that concludes ψ, and then tries to prove each of the rule premises. In this way, TEST_i expands goals into subgoals non-deterministically. This expansion continues recursively until either (i) a subgoal is trivially satisfied by the database, or (ii) no appropriate rules can be found, or (iii) the appropriate rules belong to a lower stratum, in which case the procedure PROVE_{i-1} is invoked as a subroutine. If PROVE_{i-1} is treated as an oracle, which returns in constant time, then TEST_i runs in non-deterministic polynomial time, just like the procedure PROVE of Section 4.2. The value of $\mathrm{TEST}_i(DB, \psi)$ is *true* if for some sequence of non-deterministic choices, the returned value is true.

```
Procedure: PROVE_i^NP(DB, ψ)
  if  ψ = ~ ψ0  then return  not[TEST_i(DB, ψ0)];
                else return  TEST_i(DB, ψ);


Procedure: TEST_i(DB, ψ)
  if ψ belongs below the i^th stratum, then return  PROVE_{i-1}(DB, ψ);
  elseif  ψ = A ← B1, ..., Bk  then return  TEST_i(DB + {B1, ..., Bk}, A);
  elseif  ψ ∈ DB  then return true;
  elseif the i^th stratum is empty, then return false;
  else choose a ground instance,  A ← ψ1, ..., ψm,
                    of a rule in the i^th stratum of R;
       if  A = ψ  then return  ∧_j TEST_i(DB, ψj);
                  else return false;
```

To show that $\mathrm{PROVE}_i^{\mathrm{NP}}$ runs in NP time, we invoke the analysis of proof trees given in Section 4.2. Actually, this analysis must be modified slightly to take oracle invocations into account. The first step is to extend the notion of proof trees from negation-free rulebases to stratified rulebases. Since the procedure $\mathrm{PROVE}_i^{\mathrm{NP}}$ applies only to the i^{th} stratum, it is sufficient to define proof trees for a single stratum (instead of the entire rulebase).

In Section 4.2, the nodes of a proof tree are labeled by expressions of the form $R + DB \vdash \psi$. Furthermore, at leaf nodes, ψ is atomic and $\psi \in DB$. To define proof trees for the i^{th} stratum, two minor changes are necessary. First, only rules in the i^{th} stratum may be used to construct the proof tree. Second, at leaf nodes, the formula ψ may be any goal defined *below* the i^{th} stratum. In this case, the leaf is labeled by the expression $R + DB \vdash \psi$ if $\mathrm{PROVE}_{i-1}(DB, \psi)$ returns *true*. This extension takes oracle invocations into account, representing them as leaves of a proof tree. Moreover, an oracle invocation is treated like a database access, adding just a single node to the proof tree.

Assuming that PROVE_{i-1} is correct, all the results of Section 4.2 are true with respect to this modified definition of proof trees. In particular, if the i^{th} stratum of R is linear, and if ψ belongs at or below the i^{th} stratum, then the expression $R + DB \vdash \psi$ is true if and only if it has a proof tree of polynomial size. Thus, if $R + DB \vdash \psi$ is true, then using PROVE_{i-1} as an oracle, $\mathrm{PROVE}_i^{\mathrm{NP}}$ has an accepting computation of polynomial length. Thus, for linear rulebases, $\mathrm{PROVE}_i^{\mathrm{NP}}$ runs in NP-time relative to an oracle for PROVE_{i-1}.

This establishes the upper bounds of Theorems 6.1, 6.2 and 6.3.

6.2. Lower Complexity Bounds

This section proves the lower bounds of Theorems 6.1, 6.2 and 6.3. We construct a stratified linear rulebase whose data complexity is Σ_k^{P} hard. This rulebase has exactly k strata, each of which is hypothetical. It therefore establishes the lower bound of Σ_k^{P} in Theorems 6.1, 6.2 and 6.3. Besides providing a lower complexity bound, this rulebase is also crucial to the expressibility results of Section 7.

In the rulebase we construct, each stratum encodes an arbitrary NP oracle machine. The machine encoded in one stratum uses the machine in the stratum below as its oracle. By using hypothetical insertion, the higher machine provides input to the lower one. That is, M_i hypothetically writes into the database, and M_{i-1} reads this data. Likewise, M_{i-1} provides input to M_{i-2}. In this way, a rulebase having k strata can encode a cascade of k distinct NP oracle machines, *i.e.*, a Σ_k^{P}-machine. This establishes the Σ_k^{P} lower-bounds.

Hypothetical insertion is central to oracle invocation, since it allows a machine to communicate with its oracle, *i.e.*, to pass input to it through the database. Intuitively, this is why non-hypothetical strata do not cause a significant increase in data complexity, since they cannot provide input to an oracle. They can make a single oracle invocation, which provides some complexity increase, but they cannot invoke the oracle repeatedly with different inputs, which limits the complexity increase. To see that non-hypothetical strata provide some complexity increase, choose a rulebase in $\mathcal{R}_k^2$ for which the data complexity of inferring atom A is complete for Σ_k^{P}. Then the data complexity of inferring $\sim A$ is complete for $co\Sigma_k^{\mathrm{P}}$. If we add the rule $B \leftarrow \sim A$ to the rulebase, where B is a new atom, then the data complexity of inferring B is complete for $co\Sigma_k^{\mathrm{P}}$. Since the new rule adds a single non-hypothetical stratum on top of the rulebase, the number of hypothetical strata does not increase, so the rulebase is still in $\mathcal{R}_k^2$. The $co\Sigma_k^{\mathrm{P}}$ lower-bound in Theorem 6.2 therefore follows immediately from the Σ_k^{P} lower-bound. Establishing tighter complexity bounds for $\mathcal{R}_k^2$ is an open problem.

The rest of this section proves the Σ_k^{P} lower-bound in Theorems 6.1, 6.2 and 6.3. We choose an arbitrary language, L, in Σ_k^{P}, and an arbitrary string, $\overline{s}$. We then encode $\overline{s}$ as a database, $DB(\overline{s})$, and construct a rulebase, $R(L)$, with k strata so that

$$R(L) + DB(\overline{s}) \vdash ACCEPT \quad \textit{if and only if} \quad \overline{s} \in L \tag{6.1}$$

where $ACCEPT$ is a 0-ary predicate. The important point is that the rulebase $R(L)$ is independent of the string $\overline{s}$. We can therefore conclude that the data-complexity of $R(L)$ is Σ_k^{P} hard. In particular, let L be a language which is Σ_k^{P} complete.[5]The hardness result then follows immediately. The rest of this section describes the construction of $DB(\overline{s})$ and $R(L)$. The notion of a Σ_k^{P}-machine is central to these constructions.

A Σ_k^{P}-machine is a cascade of NP oracle machines, $M_k, \ldots, M_1$. An oracle machine has two tapes, a work tape and an oracle tape. Because $M_k, \ldots, M_1$ are cascaded, the oracle tape of M_i is the work tape of M_{i-1}. In this way, M_i can ask questions of M_{i-1}. To do this, M_i first writes a string on the work tape of M_{i-1}, and then asks M_{i-1} whether it accepts the string. M_i suspends its own computations while M_{i-1} is computing. In effect, M_k invokes M_{k-1} as a subrou-

[5] See [11] for examples of such languages.

tine. Likewise, M_{k-1} invokes M_{k-2} as a subroutine, etc. The language accepted by this compound machine is the language accepted by M_k when it is the root of this cascade of machines.

Because M_k runs in (non-deterministic) polynomial time, it can only write a string of polynomial length. M_{k-1} therefore runs in (non-deterministic) polynomial time both in terms of its own input and in terms of the input to M_k, that is, in terms of the string $\overline{s}$. By a simple induction, it follows that each invocation of machine M_i runs in time which is polynomial in the length of $\overline{s}$. More specifically, suppose that M_i runs in non-deterministic time $O(m^{l_i})$, where m is the length of the input to M_i. Then, as part of the compound machine, each invocation of M_i runs in non-deterministic time $O[n^{l_i * \cdots * l_k}]$, where n is the length of $\overline{s}$. But $l_i * \cdots * l_k \leq l$, where l is the product $l_1 * l_2 * \cdots * l_k$. Thus, any invocation of any of the M_i runs in non-deterministic time $O(n^l)$.

Building the Database $DB(\overline{s})$: As discussed above, each oracle machine runs in (non-deterministic) time n^l for some integer l, where n is the length of the input string $\overline{s}$. A counter is therefore needed to represent n^l points in time and n^l positions on tape. This counter is represented by the following atomic formulas in the database $DB(\overline{s})$:

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

Given this counter, we can represent the configuration of the Σ_k^{P}-machine at each point in its computation. To represent oracle machine M_i, we introduce the predicates below, one predicate for each symbol c in the tape alphabet, and one for each state q in the finite control. This representation assumes that M_i has exactly two tape heads, one for its work tape and one for its oracle tape.

- $CELL_i^c(j,t)$ means that at time t, the work tape of machine M_i has the symbol c in the cell at position j.
- $CONTROL_i^q(j_1,j_2,t)$ means that at time t, the finite control of machine M_i is in state q, its work head is over cell j_1 of the work tape, and its oracle head is over cell j_2 of the oracle tape.

For each value of t, these atomic formulas define k distinct *id*'s, one for each oracle machine. Note that we do not need a separate predicate for oracle tapes, since the oracle tape of machine M_i is the work tape of machine M_{i-1}. The bottom-most machine, M_1, does not use its oracle head and does not invoke an oracle.

Note that we need not have introduced separate predicates for each tape symbol and control state. Instead, we might have introduced new constant symbols c and q and defined just two predicates for each machine, $CELL_i(c,j,t)$ and $CONTROL_i(q,j_1,j_2,t)$. By using separate predicates, however, we can construct the rulebase $R(L)$ so that it contains no constant symbols. This property will be important in Section 7, where we use the constructions of this section to characterize the generic queries in Σ_k^{P}.

The database $DB(\overline{s})$ describes the initial tape contents of machines $M_k, \ldots, M_1$. Since $M_{k-1}, \ldots, M_1$ act as oracles, their work tapes are initially blank. This is represented by the following atomic formulas, which we put in $DB(\overline{s})$, for $1 \leq i \leq k-1$:

$$CELL_i^b(0,0),\ CELL_i^b(1,0),\ \ldots,\ CELL_i^b(n^l-1,0)$$

where b denotes a blank. These formulas state that at time 0, the tape cell at position j contains a blank, for $0 \leq j \leq n^l - 1$.

For the top-level machine, M_k, the work tape initially contains the input string $\overline{s} = \langle s_0, s_1, \ldots, s_{n-1}\rangle$ followed by blank tape cells. This information is represented by the following atomic formulas, which we put in $DB(\overline{s})$:

$$CELL_k^{s_0}(0,0),\ CELL_k^{s_1}(1,0)\ \cdots\ CELL_k^{s_{n-1}}(n-1,0),$$
$$CELL_k^{b}(n,0),\ CELL_k^{b}(n+1,0)\ \cdots\ CELL_k^{b}(n^l-1,0).$$

This states that at time 0, the symbol s_j appears in cell j for $0 \leq j \leq n-1$, and a blank appears in cell j for $n \leq j \leq n^l - 1$.

The database also supplies initial control information for machine M_k. It states that when computation begins, the finite control is in its initial state, q_0, and both tape heads are at the beginning of their respective tapes (*i.e.*, at time 0, they are at position 0). This information is encoded by the atom $CONTROL_k^{q_0}(0,0,0)$, which we put in $DB(\overline{s})$:[6]

This completes our construction of the database $DB(\overline{s})$. It defines a counter from 0 to $n^l - 1$ and specifies the initial configuration of the entire Σ_k^{P}-machine. Note that this construction can be performed in polynomial time and space (polynomial in n, the length of $\overline{s}$). The predicate symbols introduced above, $FIRST$, $NEXT$, $LAST$, $CELL_i^c$ and $CONTROL_i^q$, all belong to the bottom stratum. They can therefore appear in the premises of rules from any stratum.

Building the Rulebase $R(L)$: This section constructs the rulebase $R(L)$, which encodes the oracle machines $M_k, \ldots, M_1$. The rulebase has exactly k strata, where the i^{th} stratum encodes machine M_i. Central to the encoding of M_i is the unary predicate $ACCEPT_i$, which determines whether M_i accepts its input. The i^{th} stratum of $R(L)$ consists mainly of linear rules for defining $ACCEPT_i$. From the perspective of top-down inference, these rules generate a computation path for M_i, that is, a sequence of machine *id*'s. As each *id* is generated, it is inserted into the database hypothetically. In this way, a computation path for M_i is "grown" one *id* at a time. Since M_i is a non-deterministic machine, it may have many computation paths. The rulebase generates each path non-deterministically during top-down inference. If any path reaches an accepting state, then $ACCEPT_i(t)$ is true, where t is the time at which the computation began.

More formally, consider a particular computation path of machine M_i. Let $DB_i(t)$ be a database encoding the *id* at time t on this path. $R(L)$ is constructed so that $ACCEPT_i$ has the following property:

$$\begin{array}{ll} & R(L) + DB_i(t) \ \vdash\ ACCEPT_i(t) \\ \textit{if} & R(L) + DB_i(t) + DB_i(t+1) \ \vdash\ ACCEPT_i(t+1) \\ \textit{if} & R(L) + DB_i(t) + DB_i(t+1) + DB_i(t+2) \ \vdash\ ACCEPT_i(t+2) \\ & \quad \cdots \\ \textit{if} & R(L) + DB_i(t) + DB_i(t+1) + \cdots + DB_i(t+k) \ \vdash\ ACCEPT_i(t+k) \\ \textit{if} & \textit{the id represented by } DB_i(t+k) \textit{ has an accepting control state.} \end{array}$$

This represents a top-down line of inference. In trying to infer $ACCEPT_i(t)$, the

[6] Similar control information for the other machines, $M_{k-1}, \ldots, M_1$, will be hypothetically added to the database when the machines are invoked as oracles.

rules in the i^{th} stratum of $R(L)$ first examine the database $DB_i(t)$ to determine whether the finite control is in an accepting state. If not, then $DB_i(t+1)$ is generated and added to the database hypothetically. The process then repeats: In trying to infer $ACCEPT_i(t+1)$, the rules in $R(L)$ examine the database $DB_i(t+1)$ to determine whether the finite control is in an accepting state. If not, then $DB_i(t+2)$ is generated and added to the database hypothetically. The process is repeated until an *id* with an accepting control state is generated. In this way, as inference proceeds, the database is hypothetically expanded and represents a growing computation path.

The database $DB_i(t)$ represents an *id* of machine M_i. Since the machine is non-deterministic, it may have many computation paths which begin at this *id*. For each such path, $R(L)$ generates a top-down line of inference like the one just described. $ACCEPT_i(t)$ is true if and only if at least one of these paths reaches an accepting state. $ACCEPT_i$ thus has the following important property:

$$R(L) + DB_i(t) \vdash ACCEPT_i(t) \quad \textit{iff} \quad DB_i(t) \textit{ represents an accepting id.}$$

Recall that an *id* is accepting if and only if some computation path leads from this *id* to an *id* with an accepting control state. Machine M_i thus accepts its input if and only if its initial *id* is accepting.

The rulebase $R(L)$ encodes not just a single machine M_i, but a composite machine made up of $M_k, \ldots, M_1$. At any point during its computations, several of the M_i may have been started and may be in the middle of a computation. Top-down inference simulates these computations; and so at any point during inference, the database may contain many computation paths, one for each M_i. When the composite machine has invoked oracles to a depth of j, machines $M_k, \ldots, M_{k-j}$ have been started, and the database contains $j+1$ computation paths. At this point, only machine M_{k-j} is actually running and only its computation path is growing, the others being in a state of suspended computation, waiting for their oracles to return.

If at some point, M_{k-j} invokes its oracle, then M_{k-j} will be suspended, M_{k-j-1} will begin computing, and a new computation path will begin in the database. This new path is completely hypothetical: After M_{k-j-1} returns its answer, the database is restored to the state it was in just before M_{k-j-1} was invoked. In this way, no trace is left in the database of the computations performed by M_{k-j-1}. Machine M_{k-j} then continues its own computations, as if it were given the correct answer by an oracle that performed no computations at all. Intuitively, oracle computations are performed in a "hypothetical world" and do not consume any "real time".

This outlines the way in which the predicates $ACCEPT_i$ are computed. Given rules defining these predicates, a single Horn rule defines the predicate $ACCEPT$ of statement (6.1). This rule initiates the computations of the entire composite machine starting at time 0:

$$ACCEPT \leftarrow ACCEPT_k(0).$$

Actually, this rule is not quite what we need, since our goal is to construct a rulebase that has no constant symbols. For this reason, we use the following equivalent rule instead:

$$ACCEPT \leftarrow FIRST(t),\ ACCEPT_k(t).$$

We add this rule to the rulebase $R(L)$ in statement (6.1), putting the predicate

$ACCEPT$ into the top stratum (stratum k). The next step is to provide rules defining the predicates $ACCEPT_k, \ldots, ACCEPT_1$.

Implementing the Predicate $ACCEPT_i$ The predicate $ACCEPT_i$ is defined by three types of rules: (i) those that detect accepting states, (ii) those that encode the transition relation of machine M_i, and (iii) those that invoke the oracle machine M_{i-1}. Negation-as-failure is used only in rules of type (iii), to detect those cases in which the oracle returns *no*. We treat each of the three types of rule in turn.

(i) Suppose that q_a is an accepting state of machine M_i. Then any id containing q_a is an accepting id. This is easily encoded with the following rule:

$$ACCEPT_i(t) \;\leftarrow\; CONTROL_i^{q_a}(j_1, j_2, t).$$

The variable t records the time at which acceptance occurs, and the variables j_1 and j_2 signify that the tape-head positions are unimportant.

(ii) For each element of the machine's transition relation, we write a single hypothetical rule. For example, suppose that M_i has the following transition:

> If the finite control is in state q and the work head is scanning symbol b, then (1) write symbol c onto the work tape and move the work head one cell to the left, (2) write symbol d onto the oracle tape and move the oracle head one cell to the right, and (3) put the finite control into state q'.

This transition is encoded by the following rule:

$$\begin{aligned} ACCEPT_i(t) \;\leftarrow\;& CONTROL_i^q(j_1, j_2, t),\; CELL_i^b(j_1, t), \\ & NEXT(t, t'),\; NEXT(j_1', j_1),\; NEXT(j_2, j_2'), \\ & [ACCEPT_i(t') \leftarrow CONTROL_i^{q'}(j_1', j_2', t'),\; CELL_i^c(j_1, t'),\; CELL_{i-1}^d(j_2, t')]. \end{aligned} \tag{6.2}$$

The body of this rule does several things. The first line determines whether the finite control of machine M_i is in state q and whether its work head is scanning the symbol b. If so, then the second line computes the next point in time, t', the next position of the work head, j_1', and the next position of the oracle head, j_2'. The third line then inserts the new control information into the database hypothetically. Notice that the symbol d is inserted into the work tape of machine M_{i-1}, which is also the oracle tape of M_i. These hypothetical insertions specify the next id in the computation path of machine M_i. The third line then continues the simulation of M_i's computations by asking whether the new id is accepting.

Since M_i is a non-deterministic machine, it may have many successor id's. Each possibility is represented by a distinct rule. From the perspective of top-down inference, it is the non-deterministic choice of which rule to invoke which determines which computation path is followed.

(iii) Finally, we encode the mechanism for invoking oracles. When machine M_i invokes its oracle, the oracle replies either *yes* or *no*, depending on whether it accepts or rejects the string that M_i wrote onto its oracle tape. Because M_i is an oracle machine, its finite control has three special states: $q_?$, q_y and q_n. When M_i enters state $q_?$, the oracle is invoked and M_i is suspended until the oracle returns an answer. If the oracle returns *yes*, then M_i enters state q_y; and if it returns *no*, then M_i enters state q_n. In either case, we can assume that the tape heads of M_i do not move, and the tape contents do not change.

We encode this mechanism with the two rules below. These rules invoke the predicate $ORACLE_{i-1}(t)$, which is true if and only if the oracle returns *yes*. $ORACLE_{i-1}$ is easily defined in terms of $ACCEPT_{i-1}$, and its definition will be given shortly.

$$\begin{aligned}
ACCEPT_i(t) \leftarrow\ & CONTROL_i^{q_?}(j_1, j_2, t),\ ORACLE_{i-1}(t),\ NEXT(t, t'), \\
& [ACCEPT_i(t') \leftarrow CONTROL_i^{q_y}(j_1, j_2, t')]. \\
ACCEPT_i(t) \leftarrow\ & CONTROL_i^{q_?}(j_1, j_2, t),\ \sim ORACLE_{i-1}(t),\ NEXT(t, t'), \\
& [ACCEPT_i(t') \leftarrow CONTROL_i^{q_n}(j_1, j_2, t')].
\end{aligned} \tag{6.3}$$

In both of these rules, the body has four premises. Consider the first rule. The first premise determines whether the finite control of M_i is in state $q_?$. If so, the second premise invokes machine M_{i-1} as an oracle. If the oracle returns *yes*, then the third premise computes the next point in time, t'. The fourth premise then inserts new control information into the database hypothetically. This information specifies a new *id* in which the finite control is in state q_y. Note that the position of the work head, j_1, and the position of the oracle head, j_2, remain unchanged. (Rules defined in the next subsection ensure that the tape contents also remain unchanged.) Lastly, the fourth premise continues the simulation of M_i's computations by asking whether the new *id* is accepting.

The second rule is similar to the first except that it determines whether the oracle returns *no*. If so, it puts the finite control of M_i into state q_n and continues the computations. Notice that an oracle answer of *no* is represented by the failure to prove the subgoal $ORACLE_{i-1}(t)$. This is the only place in the rulebase where negation-as-failure is used, but its use here is essential. Without it, the data complexity would not climb from one level in the polynomial time hierarchy to the next.

The following rule defines the predicate $ORACLE_{i-1}$:

$$ORACLE_{i-1}(t) \leftarrow FIRST(j),\ [ACCEPT_{i-1}(j) \leftarrow CONTROL_{i-1}^{q_0}(j, j, j)].$$

This rule puts machine M_{i-1} into its initial configuration and begins the simulation of its computations. The hypothetical insertion positions the two tape heads at the beginning of their tapes and puts the finite control into its initial state, q_0. The attempt to infer $ACCEPT_{i-1}(t)$ then succeeds if and only if M_{i-1} accepts the string which is on its work tape. Note that until this rule is invoked, there are no formulas in the database of the form $CONTROL_{i-1}^{q}(j_1, j_2, t)$. This rule thus defines a unique starting point for the computations of M_{i-1}.

The predicate symbol $ACCEPT_i$ belongs to the i^{th} stratum, and $ORACLE_{i-1}$ belongs to the $i-1^{st}$ stratum. Note that the bottom stratum does not invoke an oracle (*i.e.*, machine M_1 does not have a state $q_?$), so there is no need for an even lower stratum to define a predicate $ORACLE_0$.

The Frame Problem: The above rules determine the *changes* to an *id* caused by a machine transition. However, the greater part of an *id* remains unchanged by these transitions: except for those cells under the tape heads, the contents of the machine tapes remain unchanged. This is an instance of the *frame problem* [35], and we must write linear rules to account for it. Such rules are necessary because we are representing time explicitly; *i.e.*, the database represents a sequence of *id*'s, and rules are needed to copy the unchanged portion of an *id* from one instant of time to the next.

In Horn logic programming, negation-as-failure normally plays a central role in any solution to the frame problem [31]. We do not have this luxury however. The rulebase $R(L)$ which we are constructing must have no more than k strata, and we have already created k of them, one for each oracle machine. Any attempt to use negation-as-failure would add new strata to the rulebase. We therefore go to some pains to keep our solution to the frame problem negation-free.

First, recall that exactly one machine, M_i, is computing at any given time. The machines above M_i are suspended, while those below M_i have not been started. We say that M_i is the *active* machine. We encode this idea with the rules below, which define two predicates for each machine, $ACTIVE_i$ and $INACTIVE_i$. $ACTIVE_i(t)$ means that machine M_i is active at time t, and $INACTIVE_i(t)$ means that M_i is not active at time t. We could use negation-as-failure to define $INACTIVE_i$ in terms of $ACTIVE_i$, but this would introduce a new stratum into the rulebase. In these rules, t, j_1 and j_2 are variables.

$$\begin{array}{ll} ACTIVE_i(t) \leftarrow CONTROL_i^q(j_1, j_2, t) & \textit{for all } q \neq q_? \\ INACTIVE_i(t) \leftarrow CONTROL_i^{q_?}(t) & \\ INACTIVE_i(t) \leftarrow ACTIVE_j(t) & \textit{for all } j > i \end{array} \tag{6.4}$$

The first rule says that M_i is active if it has been started but not suspended. The second rule says that M_i is inactive if it has been started and suspended. The third rule says that M_i is inactive if it has not yet been started, *i.e.*, if some higher machine, M_j, is active.

Second, recall that machines M_i and M_{i-1} share a tape, since the oracle tape of M_i is the work tape of M_{i-1}. A tape may thus be modified by two different heads (though at most one head will be writing at any given time). To deal with this situation, we need the rules below, which identify the positions of the work head and the oracle head of machine M_i at time t. These rules define two new predicates. $HEAD_i^w(j,t)$ means that at time t, the work head of machine M_i is positioned over cell j. Likewise for $HEAD_i^o(t,j)$ and the oracle head.

$$\begin{array}{l} HEAD_i^w(j,t) \leftarrow CONTROL_i^q(j,j',t) \\ HEAD_i^o(j,t) \leftarrow CONTROL_i^q(j',j,t) \end{array}$$

where t, j and j' are variables. We write a pair of rules like this for every control state, q.

Third, without loss of generality, we can assume that if a machine is active, then it writes something on both its tapes; otherwise, it writes nothing.[7] We therefore need to identify those tape cells that are *not* beneath a tape head of an active machine. These are exactly the cells to which the frame problem applies. To identify these cells, we introduce two predicates. The first predicate is $NWRITING_i^w(j,t)$, which means that the work head of machine M_i does *not* writing anything at position j at time t.

$$\begin{array}{l} NWRITING_i^w(j,t) \leftarrow INACTIVE_i(t) \\ NWRITING_i^w(j,t) \leftarrow ACTIVE_i(t),\ HEAD_i^w(j',t),\ NEQ(j',j). \end{array}$$

The first rule handles the case when M_i is *inactive*. In this case it does not write

[7] As seen in Rules (6.3), at the moment of oracle invocation, a machine is in state $q_?$, and is thus technically inactive, according to Rules (6.4). Thus, by assumption, a machine does not write anything at this moment. For this reason, unlike Rule (6.2), Rules (6.3) do not update any $CELL$ predicates.

anything on any cell of its work tape. The second rule handles the case when M_i is *active*. In this case, the machine only writes at cell j', the position of the work head; so it does not write on any other tape cell, j.

In a similar fashion, we define a predicate $NWRITING_i^o(j,t)$, which means that the oracle head of machine M_i does *not* write anything at position j at time t.

$$NWRITING_i^o(j,t) \leftarrow INACTIVE_i(t)$$
$$NWRITING_i^o(j,t) \leftarrow ACTIVE_i(t),\ HEAD_i^o(j',t),\ NEQ(j',j).$$

These two predicates are defined in terms of the predicate NEQ, which determines whether two tape cells are not equal. This predicate can be defined without negation by noting that $j_1 \neq j_2$ if and only if $j_1 < j_2$ or $j_2 < j_1$. *i.e.*,

$$NEQ(j_1,j_2) \leftarrow BEFORE(j_1,j_2)$$
$$NEQ(j_1,j_2) \leftarrow BEFORE(j_2,j_1)$$
$$BEFORE(j_1,j_2) \leftarrow NEXT(j_1,j_2)$$
$$BEFORE(j_1,j_3) \leftarrow NEXT(j_1,j_2),\ BEFORE(j_2,j_3).$$

Given the two predicates $NWRITING_i^w(j,t)$ and $NWRITING_i^o(j,t)$, we can address the frame problem in a straightforward way. The rule below propagates the contents of all tape cells that do not have a tape head writing on them. In this rule, j is a position on the work tape of machine M_i. Only two tape heads can write on this cell, the work head of M_i and the oracle head of M_{i+1}. If neither of these heads is in fact writing on the cell, then its contents are propagated forward from time t to time $t+1$. We create a rule like this for each tape symbol, c.

$$\begin{aligned} CELL_i^c(j,t') \leftarrow\ & NEXT(t,t'),\ CELL_i^c(j,t), \\ & NWRITING_i^w(j,t),\ NWRITING_{i+1}^o(j,t). \end{aligned}$$

This completes our solution to the frame problem. The predicate symbols introduced in this subsection all belong to the bottom stratum (stratum 1). This is necessary since $CELL_i^c$ already belongs to this stratum.

We add the rules of this subsection to the rulebase $R(L)$ that we are constructing. The resulting rulebase, which includes all the rules of the previous subsections, satisfies statement (6.1). Since this rulebase has k strata, all hypothetical, it establishes the lower complexity bound of Σ_k^{P} in Theorems 6.1, 6.2 and 6.3.

7. EXPRESSIBILITY

This section characterizes the expressibility of linear embedded implications with negation. The results establish a tight relationship between embedded implications, computational complexity, and generic queries. Informally, a query is *generic* if it treats all constant symbols equally [10, 11]. We first show that stratified linear rulebases are expressively complete for PHIER. That is, *any* generic query in the polynomial time hierarchy can be expressed as a stratified linear rulebase of embedded implications. Second, by considering only rulebases that are constant-free, we show that stratified linear rulebases express *exactly* the generic queries in PHIER. This result provides a characterization of the generic queries in PHIER in terms of intuitionistic logic. It also provides a new characterization of the second-order definable queries, since the generic queries in PHIER are exactly the queries definable in second-order logic [47, 28]. In addition, we characterize the queries in each level, Σ_k^{P}, of the hierarchy.

Unlike many expressibility results in the literature (*e.g.*, [27, 51, 12]), the results in this section do *not* assume that the data domain is linearly ordered. The assumption of ordered domains is a technical device that is often used to achieve expressibility results, but it is not an intrinsic feature of databases [1]. Embedded implications do not need this artificial assumption, since they can generate linear orders on the domain hypothetically [6]. Thus, the results of this section are for arbitrary databases, ordered or not.

Before continuing, we comment briefly on the difference between complexity and expressibility. Section 4 showed that without negation, the data complexity of linear embedded implications is NP-complete. However, there are many simple queries that the negation-free logic cannot express, despite its great computational power. This is because the logic is *monotonic*. This is illustrated in Example 7.1, where without negation, embedded implications cannot even retrieve the leaves of a tree, simply because this is a non-monotonic query. This is true not just of embedded implications, but of all monotonic logics, including Datalog (without negation), Prolog (without negation), full classical logic, and modal logics. However, when augmented with negation-as-failure, many non-monotonic queries can easily be expressed, as also illustrated in Example 7.1.

Example 7.1. [Non-Monotonic Queries] Consider a database that stores a tree as a binary relation $child(x, y)$. *i.e.*, $child(x, y) \in DB$ if and only if x and y are nodes in the tree and y is a child of x. Now consider the query that retrieves all leaves of the tree. Given the database $DB_1 = \{child(a, b), child(a, c)\}$, the query answer is $Q_1 = \{b, c\}$; and given the database $DB_2 = \{child(a, b), child(a, c), child(c, d)\}$, the query answer is $Q_2 = \{b, d\}$. This query is therefore non-monotonic, since $DB_1 \subseteq DB_2$ but $Q_1 \not\subseteq Q_2$. There is therefore no negation-free rulebase R, such that for all databases DB,

$$R + DB \vdash leaf(x) \quad \text{iff} \quad x \text{ is a leaf of the tree encoded in } DB,$$

where *leaf* is a unary predicate symbol. This is true even for rulebases of great computational complexity. However, if negation is allowed, then this query can be expressed by an almost trivial rulebase. Only two non-recursive, non-hypothetical rules are needed:

$$leaf(y) \leftarrow child(x, y), \sim internal(y) \qquad internal(y) \leftarrow child(y, z)$$

The first rule says that node y is a leaf if it is the child of some other node, x, and it is not an internal node. The second rule says that y is an internal node if it has a child, z.

7.1. Generic Queries

This section makes precise the ideas discussed above. The first two definitions are due to Chandra and Harel [10, 11].

Definition 7.1. [Relational Databases] Let U be a countable set, called the universal data domain. A relational database DB of type $\overline{\alpha} = \langle \alpha_1, \ldots, \alpha_m \rangle$ is a tuple $\langle D, R_1, \ldots, R_m \rangle$ where D is a finite subset of U and R_i is a relation over D of arity α_i, *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. The universal domain U is the set of constant symbols in our logical language. For each relation R_i, there is a predicate symbol P_i whose ground atomic formulas represent R_i. The domain of DB is simply the set of constant symbols appearing in DB.

Informally, a database query is *generic* if renaming the constants in the database causes the constants in the query answer to be renamed in the same way. This captures the idea that constant symbols are uninterpreted. To be more precise, we define a *renaming* to be a one-to-one mapping f of the universal domain U onto itself. A renaming can be extended to tuples, relations and databases in the obvious way. If DB is a database, then fDB denotes the renamed database.

Definition 7.2. [Genericity] A *generic* query of type $\overline{\alpha} \rightarrow \beta$ is a partial function ψ that takes a relational database DB of type $\overline{\alpha}$ and returns a relation $\psi(DB)$ over $dom(DB)$ of arity β. In addition, ψ must satisfy the following *consistency criterion*: $\psi(fDB) = f\psi(DB)$ for any renaming f, and any database DB of type $\overline{\alpha}$.

Recall from Section 5.2 that inference is defined with respect to a stratified rulebased system $\mathcal{R} = [R, \mu, dom, pred]$, where dom and $pred$ are sets of constant and predicate symbols, respectively, and μ is a stratification of the rulebase, R. In the rest of this section, we assume that a stratified rulebase system is given, and we call dom the *domain of inference*. Recall that this domain is not fixed, but depends on the database, DB. As in Section 4.1, we take dom to be the set of constant symbols in $R + DB$. Thus, in the special case in which R is constant-free, the domain of inference is equal to the domain of the database, $dom(DB)$.

Definition 7.3. [Expressibility] Let ψ be a database query of type $\overline{\alpha} \rightarrow \beta$. A rulebase R expresses ψ if for all databases DB of type $\overline{\alpha}$, and all tuples $\overline{x}$ of arity β over $dom(DB)$,

$$R + DB \vdash OUT(\overline{x}) \quad \textit{if and only if} \quad \overline{x} \in \psi(DB)$$

where OUT is a predicate symbol of arity β.

If a rulebase contains no constant symbols, then it is guaranteed to be generic, as the next two lemmas show. In these lemmas, and in the rest of this section, the term "rulebase" means a finite set of stratified embedded implications.

Lemma 7.1. Suppose f is a renaming, and ψ is a ground hypothetical query. Then $R + DB \vdash \psi$ if and only if $fR + fDB \vdash f\psi$.

PROOF. Follows by straightforward inductions over the length of derivations and the number of strata. In particular, each of the inference rules in Definition 5.6 is invariant under a renaming of the constant symbols, as is the negation operation in Definition 5.7. $\square$

Lemma 7.2. Any query expressed by a constant-free rulebase is generic.

PROOF. Let R be a constant-free rulebase, and let ψ be a query expressed by R. Thus, $R + DB \vdash OUT(\overline{x})$ if and only if $\overline{x} \in \psi(DB)$. Since R is constant-free, the domain of inference is just the domain of the database $dom(DB)$, as discussed above. $\psi(DB)$ is therefore a relation over $dom(DB)$. ψ thus satisfies the first condition of genericity. To show that ψ also satisfies the consistency criterion, first note that since R is constant-free, $R = fR$ for any renaming f. Thus,

$$\begin{array}{lll} & \overline{x} \in \psi(DB) & \\ \textit{iff} & R + DB \vdash OUT(\overline{x}) & \text{since } R \text{ expresses } \psi, \\ \textit{iff} & fR + fDB \vdash OUT(f\overline{x}) & \text{by Lemma 7.1,} \\ \textit{iff} & R + fDB \vdash OUT(f\overline{x}) & \text{since } R = fR, \\ \textit{iff} & f\overline{x} \in \psi(fDB) & \text{since } R \text{ expresses } \psi. \end{array} \tag{7.1}$$

This is true for any database, DB, and any renaming, f. In particular, we can use fDB instead of DB, and f^{-1} instead of f.[8] Moreover, if S is a set of tuples, then $\overline{x} \in S$ if and only if $f\overline{x} \in fS$. Keeping these points in mind, we get the following:

$$\begin{array}{lll} & \overline{x} \in \psi(fDB) & \\ \textit{iff} & f^{-1}\overline{x} \in \psi(f^{-1}fDB) & \text{by equivalence (7.1),} \\ \textit{iff} & f^{-1}\overline{x} \in \psi(DB) & \\ \textit{iff} & ff^{-1}\overline{x} \in f\psi(DB) & \\ \textit{iff} & \overline{x} \in f\psi(DB) & \end{array}$$

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

7.2. Expressive Completeness

From Lemma 7.2 and from the upper complexity bounds of Section 6, it follows that any stratified linear rulebase that is constant-free expresses a generic query in PHIER. We now prove the converse, that any generic query in PHIER can be expressed as a stratified linear rulebase that is constant-free. We prove similar results for each individual level Σ_k^{P} of the polynomial time hierarchy. In particular, we prove Theorem 7.3 below, which is the main result of this section. This theorem is based on the set $\mathcal{R}_k^3$ defined in Section 6. Recall from Definition 6.3 that $\mathcal{R}_k^3$ is the set of linear rulebases with exactly k hypothetical strata, including the top stratum.

Theorem 7.3. [Completeness for Σ_k^{P}] *Let ϕ be a generic database query of type $\overline{\alpha} \to \beta$ whose graph is in Σ_k^{P} for some $k \geq 1$. Then there is a constant-free rulebase $R(\phi)$ in $\mathcal{R}_k^3$ that expresses ϕ. That is, for any database DB of type $\overline{\alpha}$,*

$$R(\phi) + DB \vdash OUT(\overline{x}) \quad \textit{if and only if} \quad \overline{x} \in \phi(DB)$$

where OUT is a predicate symbol of arity β.

This theorem leads immediately to a series of Corollaries. By the theorem, any generic query in Σ_k^{P} is expressible as a constant-free rulebase in $\mathcal{R}_k^3$. Conversely, any such rulebase expresses a generic query in Σ_k^{P}, by Lemma 7.2 and Theorem 6.3. Hence,

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

Corollary 7.4. [Characterization of Σ_k^{P}] *Constant-free rulebases in $\mathcal{R}_k^3$ express exactly the typed generic queries in Σ_k^{P}, for $k \geq 1$.*

We get results analogous to Theorem 7.3 and Corollary 7.4 for the entire polynomial time hierarchy. These results impose no restrictions on the structure of strata other than linearity. First note that any generic query in PHIER is in Σ_k^{P} for some k. Thus, according to Theorem 7.3, it is expressed by a rulebase in $\mathcal{R}_k^3$. This is a stratified linear rulebase of embedded implications. Hence,

Corollary 7.5. [Completeness for PHIER] *Any typed generic query in the polynomial time hierarchy can be expressed as a stratified linear rulebase of embedded implications.*

By considering only constant-free rulebases, we turn this completeness result into an exact characterization. First note that the data complexity of any stratified linear rulebase is in PHIER, by Theorem 6.1. If the rulebase is constant-free, then it also expresses a generic query. Hence,

Corollary 7.6. [Characterization of PHIER] *Stratified linear rulebases of embedded implications that are constant-free express exactly the typed generic queries in the polynomial time hierarchy.*

The typed generic queries in PHIER are precisely the queries definable in second-order logic [47, 28]. This gives another characterization of the queries expressed by stratified linear rulebases.

Corollary 7.7. [Second-Order Queries] *Stratified linear rulebases of embedded implications that are constant-free express exactly the second-order definable queries.*

Proof Sketch of Theorem 7.3: In [6], we proved that embedded implications are expressively complete for PSPACE; *i.e.*, they can express any generic query computable in polynomial space. The proof is easily adapted to Theorem 7.3. This section outlines the proof. Details can be found in [6].

Let ϕ be a generic database query of type $\overline{\alpha} \to \beta$ whose graph is in Σ_k^{P}. There is a Σ_k^{P}-machine that recognizes this graph, *i.e.*, that accepts the language $L = \{\langle \overline{x}, DB\rangle \mid \overline{x} \in \phi(DB)\}$. Recall from Section 6.2 that this machine is a cascade of NP oracle machines $M_k, \ldots, M_1$, where M_k receives the input, and each machine M_i invokes M_{i-1} as an oracle. Section 6.2 showed how to construct a rulebase that encodes the computations of this cascaded machine. Call this rulebase $R_1(\overline{M})$, where $\overline{M} = \langle M_k, \ldots, M_1\rangle$ denotes the cascaded machine. This rulebase is linear and has exactly k strata, all of which are hypothetical. $R_1(\overline{M})$ is therefore in $\mathcal{R}_k^3$. However, this rulebase is not exactly the rulebase $R(\phi)$ required in Theorem 7.3. $R_1(\overline{M})$ assumes that the initial configuration of machine $\overline{M}$ is encoded as a database $DB(\overline{s})$, where $\overline{s}$ is the input string for machine M_k. $DB(\overline{s})$ is also assumed to contain a counter. $DB(\overline{s})$ is therefore different from the database DB in Theorem 7.3, which is an arbitrary database. We must provide rules which when given DB, will construct $DB(\overline{s})$. We defined such rules in [6] for use with PSPACE-machines. These rules are linear, and can be applied to cascaded oracle machines with very few changes. The main change is a few straightforward rules for initializing the oracle machines.

These rules perform four main tasks. (i) They hypothetically generate a linear order of the data domain. This provides a counter. (ii) They generate every possible tuple $\overline{x}$ of arity β over the data domain. This provides a set of candidate query answers. (iii) For each $\overline{x}$, they encode DB and $\overline{x}$ in terms of the predicates $CELL_k^c(j,t)$ used in Section 6.2. This encodes the database and a candidate query answer onto the input tape of machine $\overline{M}$. (iv) They invoke the machine, and infer $OUT(\overline{x})$ if and only if the machine accepts its input. This determines whether $\overline{x}$ is an answer to the query. Because the query is generic, the result of this step is independent of the specific linear order generated in step (i) [6].

The rules involved in these steps are added to $R_1(\overline{M})$, to give a new rulebase called $R_2(\overline{M})$. The rules involved in steps (i), (ii) and (iv) are hypothetical, and are added to the top stratum of $R_1(\overline{M})$. This does not increase the number of strata. The rules involved in step (iii) consist of several non-hypothetical strata, which are placed underneath the bottom stratum of $R_1(\overline{M})$. This increases the total number of strata, but not the number of hypothetical strata. Thus, like $R_1(\overline{M})$, $R_2(\overline{M})$ is linear and constant-free, and has k hypothetical strata, including the top stratum. $R_2(\overline{M})$ is therefore in $\mathcal{R}_k^3$, as required by Theorem 7.3. By choosing the machine $\overline{M}$ appropriately, we get the rulebase $R(\phi)$ in Theorem 7.3.

Acknowledgments

The work of Thorne McCarty on the intuitionistic semantics of embedded implications was the original stimulus for this work. Discussions with Tomasz Imielinski were invaluable in giving the work a database perspective. Eric Allender, Jan Chomicki, Ron van der Meyden and Kumar Vadaparty also provided valuable feedback during the development of this work. The comments of the anonymous reviewers did much to improve and shorten the presentation.

REFERENCES

1. S. Abiteboul, M.Y. Vardi, and V. Vianu. Fixpoint Logics, Relational Machines, and Computational Complexity. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pages 156–168, Boston, MA, June 22–25 1992. IEEE Computer Society Press.
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. F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., May 28–30 1986.
5. A.J. Bonner. A logical semantics for hypothetical rulebases with deletion. To appear in the *Journal of Logic Programming (JLP)*.
6. A.J. Bonner. Hypothetical Datalog: Complexity and Expressibility. *Theoretical Computer Science (TCS)*, 76:3–51, 1990. Special issue on the 2^{nd} International Conference on Database Theory (ICDT'88).

7. A.J. Bonner. *Hypothetical Reasoning in Deductive Databases.* PhD thesis, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, USA, October 1991. Published as Rutgers Technical Report DCS-TR-283.
8. A.J. Bonner and L.T. McCarty. Adding Negation-as-Failure to Intuitionistic Logic Programming. In *Proceedings of the North American Conference on Logic Programming (NACLP)*, pages 681–703, Austin, Texas, Oct 29–Nov 1 1990. MIT Press.
9. A.J. Bonner, L.T. McCarty, and K. Vadaparty. Expressing Database Queries with Intuitionistic Logic. In *Proceedings of the North American Conference on Logic Programming (NACLP)*, pages 831–850, Cleveland, Ohio, October 16–20 1989. MIT Press.
10. A.K. Chandra and D. Harel. Computable Queries for Relational Databases. *Journal of Computer and System Sciences (JCSS)*, 21(2):156–178, 1980.
11. 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.
12. A.K. Chandra and D. Harel. Horn Clause Queries and Generalizations. *Journal of Logic Programming (JLP)*, 2(1):1–15, 1985.
13. A.K. Chandra, D. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.
14. B.F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, 1980.
15. P.M. Dung. Declarative Semantics of Hypothetical Logic Programming with Negation as Failure. In *Proceedings of the Third International Workshop on Extensions of Logic Programming*, pages 45–58, Bologna, Italy, February 1992. Springer-Verlag. Published as volume 660 of *Lecture Notes in Artificial Intelligence*, 1993.
16. T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactuals. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 261–273, San Diego, CA, June 2–4 1992.
17. M.C. Fitting. *Intuitionistic Logic, Model Theory and Forcing.* North-Holland, 1969.
18. The Committee for Advanced DBMS function. Third-Generation Database System Manifesto. *SIGMOD Record*, 19(3):31–44, September 1990. Also published as Memorandum No. UCB/ERL M90/28, Electronics Research Laboratory, College of Engineering, University of California, Berkeley.
19. 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.
20. 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.
21. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.
22. M.L. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(1):35–79, 1986.
23. L. Giordano and N. Olivetti. Negation as Failure in Intuitionistic Logic Programming. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*, pages 431–445, Washington, D.C., 1992. MIT Press.
24. J. Harland. A Kripke-like model for negation as failure. In *Proceedings of the North American Conference on Logic Programming (NACLP)*, pages 626–642, Cleveland, Ohio, October 1989. MIT Press.
25. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
26. G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic.* Methuen and Co. Ltd., London, 1968.
27. N. Immerman. Relational Queries Computable in Polynomial Time. In *Proceedings*

of the ACM Symposium on Theory of Computing (STOC), pages 147–152, 1982.
28. N. Immerman. Languages that Capture Complexity Classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.
29. R.H. Sprague Jr. and H.J. Watson, editors. *Decisions Support Systems: Putting Theory into Practice.* Prentice Hall, Englewood Cliffs, NJ, 1989.
30. H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge database and revising it. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 387–394, Boston, Mass., April 1991.
31. R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.
32. S. Kripke. Semantical Analysis of Intuitionistic Logic. I. In J.N. Crossley and M.A.E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North Holland, Amsterdam, 1965.
33. 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.
34. 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.
35. J.M. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969. Reprinted in *Readings in Artificial Intelligence*, 1981, Tioga Publ. Co.
36. L.T. McCarty. Clausal Intuitionistic Logic. I. Fixed-Point Semantics. *Journal of Logic Programming (JLP)*, 5(1):1–31, 1988.
37. L.T. McCarty. Clausal Intuitionistic Logic. II. Tableau Proof Procedures. *Journal of Logic Programming (JLP)*, 5(2):93–132, 1988.
38. L.T. McCarty. A Language for Legal Discourse. I. Basic Features. In *Proceedings of the Second International Conference on Artificial Intelligence and Law*, pages 180–189. ACM Press, June 1989.
39. D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming (JLP)*, 6:79–108, 1989.
40. D. Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Logic Programming: Proceedings of the Sixth International Conference*, pages 268–283, Cambridge, MA, 1989. MIT Press.
41. B. Nebel. Belief revision and default reasoning: Syntax-based approach. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 417–428, Cambridge, MA, April 11–25 1991.
42. N. Olivetti and L. Terracini. N-Prolog and Equivalence of Logic Programs (Part 1). *Journal Of Logic, Language and Information*, 1(4):253–340, 1992.
43. R.L. Olson and R.H. Sprague Jr. Financial Planning in Action. In R.H. Sprague Jr. and H.J. Watson, editors, *Decisions Support Systems: Putting Theory into Practice*, pages 373–381. Prentice Hall, Englewood Cliffs, NJ, 1989.
44. D.A. Schlobohm and L.T. McCarty. EPS-II: Estate planning with prototypes. In *Proceedings of the Second International Conference on Artificial Intelligence and Law*, pages 1–10. ACM Press, June 1989.
45. Editor: A.P. Sheth. Database Research at Bellcore. *SIGMOD Record*, 19(3):45–52, September 1990.
46. R. Statman. Intuitionistic Propositional Logic is Polynomial-Space Complete. *Theoretical Computer Science (TCS)*, 9(1):67–72, 1979.
47. L.J. Stockmeyer. The Polynomial Time Hierarchy. *Theoretical Computer Science*

(TCS), 3(1):1–22, 1976.
48. M. Stonebraker. Hypothetical databases as views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 224–229, 1981.
49. M. Stonebraker and K. Keller. Embedding expert knowledge and hypothetical databases into a data base system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 58–66, Santa Monica, CA, 1980.
50. 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.
51. M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 137–146, 1982.
52. L. Vieille, P. Bayer, V. Kuchenhoff, and A. Lefebvre. EKS-V1, A Short Overview. Presented at the AAAI-90 Workshop on Knowledge Base Management Systems, July 1990, Boston, USA.
53. L. Vieille, P. Bayer, V. Kuchenhoff, A. Lefebvre, and R. Manthey. The EKS-V1 system. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, number 624 in Lecture Notes in Computer Science, pages 504–506. Springer-Verlag, 1992.
54. D.S. Warren. Database Updates in Pure Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 244–253, 1984.