Hypothetical Reasoning with Intuitionistic Logic

Anthony J. Bonner

University of Toronto
Department of Computer Science
Toronto, Ontario M5S 1A4
Canada

bonner@db.toronto.edu

Abstract

This paper addresses a limitation of most deductive database systems: They cannot reason hypothetically. Although they reason effectively about the world as it is, they are poor at tasks such as planning and design, where one must explore the consequences of hypothetical actions and possibilities. To address this limitation, this paper presents a logic-programming language in which a user can create hypotheses and draw inferences from them. Two types of hypothetical operations are considered: the *insertion* of tuples into a database, and the *creation* of new constant symbols. These two operations are interesting, not only because they extend the capabilities of database systems, but also because they fit neatly into a well-established logical framework, namely *intuitionistic logic*. This paper presents the proof theory for the logic, outlines its intuitionistic model theory, and summarizes results on its complexity and on its ability to express database queries. Our results establish a strong link between two previously unrelated, but well-developed areas: intuitionistic logic and computational complexity. This, in turn, leads to a strong link with classical second-order logic. Moreover, unlike many expressibility results in the literature, our results do not depend on the artificial assumption that the data domain is linearly ordered.

Appears in R. Demolombe and T. Imielinski, editors, *Non-Standard Queries and Answers*, Studies on Logic and Computation, Chapter 8, pages 187–219. Oxford University Press, October 1994.

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

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 [34, 44]. 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 [54]. 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 [24, 47]. 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 [24]. The number of hypothetical scenarios multiplies quickly when several factors are varied simultaneously, such as prices, interest rates, tax rates, etc. One may also need to consider variations in more complex factors, such as government regulations, company policy, tax laws, etc.

The database community has 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 [51, 50]. 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" [50].

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 developed a deductive database along these lines [54], and Warren and Manchanda have used hypothetical rules to reason about database updates [55, 36]. In [41], 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 [42], 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" [26]. 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 [38, 40, 46].

Theoretical work on hypothetical inference has also been carried out, largely by the logic-programming community. Most of this work focuses on the hypothetical insertion of atoms into a database. These updates, it turns out, fit neatly into a well-known logical system, *intuitionistic logic* [23]. Most of this work has been semantic, first showing that hypothetical insertion is indeed intuitionistic, and then casting the semantics in terms of a least fixpoint theory, in the logic-programming tradition. Gabbay first showed that hypothetical insertion is intuitionistic [25]. Working independently, McCarty and Miller extended this result to operations that create new constant symbols during inference, and they developed fixpoint semantics based on intuitionistic logic [38, 41]. Earlier, Statman showed that intuitionistic logic is PSPACE-complete in the propositional case [48]. Several researchers have also investigated the the semantics of negation-as-failure for hypothetical insertions [25, 14, 31, 43, 30, 22]. We discuss this work in Section 6.

This paper adds to the picture by summarizing some of our own results on hypothetical inference. Our work has considered a variety of hypothetical operations, including insertion, deletion and bulk updates [11, 10, 15, 12, 13, 14, 9, 8]. This paper, however, focuses on two specific operations: (i) adding formulas to a database hypothetically ("insertion"), and (ii) creating new constant symbols hypothetically ("creation"). The ability to create new constants is achieved *without* using function symbols. In fact, because of its importance to database systems, the entire paper focuses on the function-free case. Since the language has an *infinite* set of constant symbols, however, it may be possible to generalize our development to include function symbols, in which case the Herbrand universe would be infinite.

The operations of hypothetical insertion and creation promise new capabilities and applications for deductive databases and logic-programming systems. For example, in testing a circuit design, we might ask, "*If* input X is high at time t, *will* output Y be high at time $t + \Delta$?" This simple hypothetical query can be represented by the formula $high(X, t) \rightarrow high(Y, t + \Delta)$. We might also want to know whether this query is true for *any* time t, not just for a specific time. This would be the case if we wanted a guarantee that the circuit will always behave in a certain way. The query then becomes universally quantified and has the following logical form:

$$\forall_t [high(X, t) \rightarrow high(Y, t + \Delta)]$$

Notice that this formula is a *query*, not a rule. It is very different from the queries normally found in the deductive-database and logic-programming literature, since it contains an implication and is universally quantified. Numerous examples of such queries can be constructed, including sophisticated rulebases in which such queries appear in the premises of rules. Such rules are called *embedded implications* [38]. As we shall see, the implication sign in queries and rule bodies corresponds to hypothetical insertion, and the universal quantifier corresponds to the hypothetical creation of new constant symbols. It is in this way that the two operations of insertion and creation fit neatly into a logical, first-order framework.

In addition to applications, these two operations have important practical and theoretical properties. For instance, since they can be handled entirely within first-order intuitionistic logic, they have a well-established semantics. In addition, they have an efficient Prolog-style inference procedure, one based on resolution and unification, in the logic-programming tradition [39]. Lastly, these hypothetical operations have important complexity-theoretic properties. For instance, as we show in Section 7, they characterize the database queries in many well-known complexity classes, such as NP, PSPACE, *re*, and the polynomial-time hierarchy. One outcome of this paper, then, is to develop a strong link between two previously unrelated, but well-established areas: intuitionistic logic and computational complexity. As a corollary, Section 7.3 links intuitionistic logic and classical second-order logic.

2 Introductory Examples

This section gives several examples of hypothetical queries and rules. The examples are centered on a deductive database that describes university policy. In these examples, the atomic formula $take(s, c)$ intuitively means that student s has taken course c, and $grad(s)$ means that s is eligible for graduation.

Conceptually, the deductive database has two parts, extensional and intensional. The extensional part consists of atomic facts, such as $take(tony, cs250)$; and the intensional part consists of rules, such as

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

Formally, the deductive database is a set, R, of rules and facts, and the notation $R \vdash \phi$ means that formula ϕ can be inferred from the rules and facts in R.[1] In the examples, each query is described in three ways: (i) informally in English, (ii) formally at the meta-level, and (iii) formally at the object-level.

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

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

At the object level, the expression $grad(tony) \leftarrow take(tony, cs452)$ represents this query. That is, $R \vdash grad(tony) \leftarrow take(tony, cs452)$ iff meta-level condition (1) is satisfied. $\Box$

Example 2.2 "Retrieve those students who could graduate if they took one more course." i.e., at the meta-level, we want those s for which the following statement is true for some c:

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

[1] Later, when we study complexity, we will need to distinguish more carefully between rules and facts. We will then adopt the notation $R + DB \vdash \phi$, where R denotes a set of rules, and DB denotes a database of atomic facts. Until then, however, there is no need to be so discriminating, so we shall simply assume that R contains both rules and facts.

The expression $\exists_c[grad(s) \leftarrow take(s,c)]$ represents this query at the object-level. That is, for each value of s, the statement $R \vdash \exists_c[grad(s) \leftarrow take(s,c)]$ is true iff the meta-level condition is satisfied. □

Queries such as these can be used in the premises of rules. Such rules turn the query language into a logic for building rulebases.

Example 2.3 Consider the following university 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 can be represented as two rules:

$$within1(s,d) \leftarrow \exists_c[grad(s,d) \leftarrow take(s,c)].$$
$$grad(s, mathphys) \leftarrow within1(s, math),\ within1(s, phys).$$

Here, $grad(s,d)$ 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 d. Note that the premise of the first rule is a query similar to the one in Example 2.2. □

In [8], it is shown that the university policy in Example 2.3 cannot be expressed in Datalog or in any query language based on classical logic. Hypothetical rules thus increase the capabilities of deductive database systems. The next two examples demonstrate the use of universal quantification in queries and rule bodies.

Example 2.4 "Retrieve those departments for which any student can graduate by taking just one course." That is, at the meta level, we want those departments d such that

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

Or, using the notation of the previous example, we want those departments d for which any student is within one course of graduation. i.e.,

$$\forall_s\ [R \vdash within1(s,d)] \tag{2}$$

At the object level, this query is represented by the expression $\forall_s\ within1(s,d)$. That is, for each value of d, the statement $R \vdash \forall_s\ within1(s,d)$ is true iff condition (2) is satisfied. □

Example 2.5 The following rule defines a department to be "easy" if any student can obtain a degree from the department by taking history 100 and english 100.

$$easy(d) \leftarrow \forall_s[grad(s,d) \leftarrow take(s, his100), take(s, eng100)]$$

□

The last two examples illustrate a subtle point about our use of universal quantification: Quantification occurs not just over those students currently mentioned in the database, but over all *possible* students, including those who *might* enroll in the future. In Example 2.4, for instance, we are asking if a hypothetical student (who might have taken no courses) can graduate by taking just one course. The fact that every student in the database may have already taken several courses is irrelevant. For this reason, the idea of hypothetical objects is central to our interpretation of universal quantification, and this is what gives it an intuitionistic semantics, as we shall see in Section 4. Proof theoretically, the creation of hypothetical objects means creating new constant symbols to represent them. The next section describes an inference system that does exactly this.

3 Proof Theory

This section describes a proof theory for hypothetical inference, one based on a generalization of Horn rules called *embedded implications*. Embedded implications are first-order formulas that can hypothetically insert formulas into a database, and can hypothetically create new constant symbols. Such systems have been developed by several researchers [26, 38, 41, 10, 15, 37]. This section defines a simplified version of these systems, one that retains many of the essential properties of the more elaborate systems while admitting a clean theoretical analysis. One notable feature of this system is that it allows universal quantifiers in rule bodies. In the logic programming context, such rules were first treated by McCarty. In [38], McCarty develops an intuitionistic fixpoint semantics for embedded implications, and in [39], he describes a practical, tableau proof procedure. Such rules have also been extensively investigated by Miller. In [41], Miller develops an intuitionistic fixpoint semantics for embedded implications, and in [42], he develops an interpreter for them.

The language of our 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, $C, ...$ Note that the language, as defined here, does not include function symbols.

Definition 3.1 (Embedded Implications)

1. *An atomic formula is an embedded implication.*
2. *If A is atomic and if $\phi_1, ..., \phi_k$ are embedded implications, then $A \leftarrow \phi_1, ..., \phi_k$ is also an embedded implication.*
3. *If $\phi(x)$ is an embedded implication, then $\forall_x \phi(x)$ is also an embedded implication.*

Embedded implications thus include formulas of the form A, $A \leftarrow B$, $A \leftarrow (B \leftarrow C)$, $A \leftarrow (B \leftarrow (C \leftarrow D)$, etc. They also include formulas with embedded universal-quantifiers, such as $A \leftarrow \forall_x B(x)$ and $A \leftarrow \forall_x [B(x) \leftarrow C(x), D(x)]$. They do not, however, include formulas with explicit disjunction or existential quantification.

This paper is concerned primarily with embedded implications that are *closed*, i.e., that have no free variables. A set of closed embedded implications shall be called a *hypothetical deductive database*, or simply a *rulebase*. In a rulebase, universal quantifiers at the top level may be implicit and need not be written. Thus, the following set:

$$\{ \ A(x) \leftarrow B(x), \qquad C \leftarrow \forall_x (D(x,y) \leftarrow E(x,y)) \ \}$$

is an abbreviation for the following rulebase:

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

Definition 3.2 (Hypothetical Inference) *The five items below define an inference system for closed embedded implications. The first item specifies axioms, and the other four items specify rules of inference. Here, R is a hypothetical deductive database, ϕ and each ϕ_i are closed embedded implications, and $\phi(x)$ is an embedded implication whose only free variable is x. b is a constant symbol.*

1. $R \vdash \phi \quad \textit{if} \quad \phi \in R$
2. $R \vdash \phi(b) \quad \textit{if} \quad R \vdash \forall_x \phi(x)$
3. $R \vdash \phi \quad \textit{if} \quad R \vdash \phi \leftarrow \phi_1 ... \phi_n$ *and* $R \vdash \phi_i$ *for each* i. *(Modus Ponens).*
4. $R \vdash \phi \leftarrow \phi_1, ..., \phi_k \quad \textit{if} \quad R + \{\phi_1, ..., \phi_k\} \vdash \phi$
5. $R \vdash \forall_x \phi(x) \quad \textit{if} \quad R \vdash \phi(b)$ *provided b is new, i.e., does not appear in R or $\phi(x)$.*

This inference system derives expressions of the form $R \vdash \phi$, where both R and ϕ are closed. When this system is used for query processing, ϕ represents a yes/no query. That is, given ϕ and R, the system returns "yes" iff $R \vdash \phi$ can be derived. In databases and logic programming, it is common to ask queries that return a set of tuples as answers. Such queries can be defined using embedded implications with free variables [10, 15, 9, 26, 27, 39]. We address this issue in Section 7, where we discuss database queries in general. Until then, however, we shall assume that a query is a closed formula. Indeed, as far as the above inference system is concerned, a query (or goal) is *always* a closed formula.

Notice that the first three items in Definition 3.2 form an inference system for definite Horn logic (e.g., Prolog and Datalog). The last two rules extend this to an inference system for hypothetical reasoning by formalizing two hypothetical operations: insertion and creation. Like all inference systems, this one can be run in either a top-down (backward) or a bottom-up (forward) mode. When run top-down, the fourth rule inserts formulas $\phi_1 \ldots \phi_k$ into the database, and the fifth rule creates a new constant symbol, b. Although these two operations might seem procedural, the inference system provides a simple, logical semantics, since the order in which the rules are applied is unimportant. Moreover, the next section provides that other essential ingredient of logical systems, a model theory.

Inference rule *5*, above, is somewhat unusual in that it creates new constant symbols during inference. It therefore deserves some discussion. Since the constant symbol b is new, it does not appear in any of the rules, and so is not given any special treatment by them. That is, if $R \vdash \phi(b)$ is derivable when b is new, then it will be derivable for any b. Intuitively, this is why we can replace the universal quantifier in rule *5* by a single constant symbol. Alternatively, we can view b as a kind of skolem constant. That is, if R were interpreted according to classical semantics,[2] then we could reason as follows, where $\vdash_c$ denotes classical

[2] In fact, when R consists entirely of Horn rules, the inference system of Definition 3.2 is classical; i.e., it is sound and complete with respect to classical model theory.

inference:

$$
\begin{array}{lll}
 & R \vdash_c \forall_x \phi(x) & \\
\textit{iff} & R \wedge \neg\forall_x \phi(x) & \text{is unsatisfiable.} \\
\textit{iff} & R \wedge \exists_x \neg\phi(x) & \text{is unsatisfiable.} \\
\textit{iff} & R \wedge \neg\phi(b) & \text{is unsatisfiable, where } b \text{ is a skolem constant (i.e., } b \text{ is new).} \\
\textit{iff} & R \vdash_c \phi(b) &
\end{array}
$$

Notice that b is a skolem *constant*, not a skolem function. This is always the case, since according to Definition 3.2, both R and $\forall_x \phi(x)$ are closed. Rule *5* is therefore a sound rule of classical logic. It is also a sound rule of intuitionistic logic.[3] Normally, skolemization is a preprocessing phase prior to resolution. This is not necessary, however, and in the inference system of Definition 3.2, "skolemization" is effectively integrated into the inference process. This kind of run-time skolemization is central to the tableau proof procedure developed in [39], and to the interpreter presented in [42]. Run-time skolemization is treated in a more general setting by Gabbay in [27]. Because Gabbay's language includes existential quantifiers, his analogue of rule *5* uses skolem functions, not just skolem constants.

The rest of this section gives examples illustrating the basic properties of the inference system above. These examples use the following lemma, which is an immediate consequence of inference rules 1, 2 and 3. This lemma is useful when the top-most universal quantifiers in a rulebase are implicit, e.g., when we write $A(x,y) \leftarrow B(x,y)$ instead of $\forall_x \forall_y [A(x,y) \leftarrow B(x,y)]$.

Lemma 3.1 *Let $\phi \leftarrow \phi_1 ... \phi_n$ be a rule in R, and let θ be a substitution such that $\phi\theta$ and each $\phi_i\theta$ are closed. If $R \vdash \phi_i\theta$ for each i, then $R \vdash \phi\theta$.*

3.1 Examples of Inference

The examples below view hypothetical inference as a top-down process that starts from the goal, as in Prolog. The first example illustrates the basic idea.

Example 3.1 [Propositional Inference] Suppose that R consists of the following three rules:

$$
\begin{array}{l}
A \leftarrow (B \leftarrow D). \\
B \leftarrow C. \\
C \leftarrow D.
\end{array}
$$

Then $R \vdash A$. This can be proved by a straightforward, top-down argument:

[3] The proof is an adaptation of a proof given in [23, Chapter 5] on the correctness of Beth Tableaus, which also create new constant symbols.

$$\begin{array}{lll} & R \vdash A & \\ \textit{if} & R \vdash B \leftarrow D & \text{by Lemma 3.1, using the rule } A \leftarrow (B \leftarrow D). \\ \textit{if} & R + D \vdash B & \text{by inference rule } \textit{4}. \\ \textit{if} & R + D \vdash C & \text{by Lemma 3.1, using the rule } B \leftarrow C. \\ \textit{if} & R + D \vdash D & \text{by Lemma 3.1, using the rule } C \leftarrow D. \end{array}$$

But the last line is trivially true, by inference rule *1.* □

The next two examples shows how simple, hypothetical updates can be cascaded to produce more-complex, hypothetical transactions. The first example is propositional. The second example extends the idea to recursive, predicate inference.

Example 3.2 [Cascaded Hypotheticals] Suppose R consists of rules defining a predicate A_n plus the following n rules:

$$\begin{array}{l} A_0 \leftarrow (A_1 \leftarrow B_1) \\ A_1 \leftarrow (A_2 \leftarrow B_2) \\ \quad \cdots \\ A_{n-1} \leftarrow (A_n \leftarrow B_n) \end{array}$$

Then $R \vdash A_0$ *if* $R + B_1 + B_2 + \ldots + B_n \vdash A_n$. This can be proved in a top-down fashion as follows:

$$\begin{array}{ll} & R \vdash A_0 \\ \textit{if} & R \vdash A_1 \leftarrow B_1 \\ \textit{if} & R + B_1 \vdash A_1 \\ \textit{if} & R + B_1 \vdash A_2 \leftarrow B_2 \\ \textit{if} & R + B_1 + B_2 \vdash A_2 \\ & \quad \cdots \\ \textit{if} & R + B_1 + B_2 + \ldots + B_n \vdash A_n \end{array}$$

□

Example 3.3 [Recursion] Suppose R contains the following rule and atomic formulas:

$$\begin{array}{l} A(x) \leftarrow NEXT(x, y),\ [A(y) \leftarrow B(y)]. \\ NEXT(c_0, c_1), \quad NEXT(c_1, c_2) \ \ldots \ NEXT(c_{n-1}, c_n) \end{array}$$

Then, for $0 \leq i \leq n$,

$$R \vdash A(c_0) \quad \textit{if} \quad R + B(c_1) + B(c_2) + \ldots + B(c_n) \vdash A(c_n) \tag{3}$$

To prove this, first note that $R \vdash NEXT(c_{i-1}, c_i)$. In fact, $R + S \vdash NEXT(c_{i-1}, c_i)$ for any set of formulas S. Hence,

$$
\begin{array}{ll}
 & R+S \ \vdash \ A(c_{i-1}) \\
\textit{if} & R+S \ \vdash \ NEXT(c_{i-1}, c_i) \quad \text{and} \quad R+S \ \vdash \ A(c_i) \leftarrow B(c_i) \\
\textit{if} & R+S \ \vdash \ A(c_i) \leftarrow B(c_i) \\
\textit{if} & R+S+B(c_i) \ \vdash \ A(c_i)
\end{array}
$$

Thus, to prove $A(c_{i-i})$, we can add $B(c_i)$ to the database and then try to prove $A(c_i)$. Carrying out this process recursively leads to statement (3). i.e.,

$$
\begin{array}{ll}
 & R \ \vdash \ A(c_0) \\
\textit{if} & R+B(c_1) \ \vdash \ A(c_1) \\
\textit{if} & R+B(c_1)+B(c_2) \ \vdash \ A(c_2) \\
 & \ldots \\
\textit{if} & R+B(c_1)+B(c_2)+\ldots+B(c_n) \ \vdash \ A(c_n)
\end{array}
$$

□

The next example extends Example 3.1 to include universal quantification in a rule premise. The quantifier is dealt with by creating a new, hypothetical constant symbol during inference.

Example 3.4 [Creating New Constants] Suppose R consists of the following rules:

$$
\begin{array}{l}
A \leftarrow \forall_x[B(x) \leftarrow D(x)] \\
B(x) \leftarrow C(x) \\
C(x) \leftarrow D(x)
\end{array}
$$

Then $R \vdash A$, as the following top-down argument shows:

$$
\begin{array}{lll}
 & R \ \vdash \ A & \\
\textit{if} & R \ \vdash \ \forall_x[B(x) \leftarrow D(x)] & \text{by lemma 3.1.} \\
\textit{if} & R \ \vdash \ B(c) \leftarrow D(c) & \text{where } c \text{ is new, by inference rule } \textit{5}. \\
\textit{if} & R+D(c) \ \vdash \ B(c) & \text{by inference rule } \textit{4}. \\
\textit{if} & R+D(c) \ \vdash \ C(c) & \text{by lemma 3.1, using the rule } B(x) \leftarrow C(x). \\
\textit{if} & R+D(c) \ \vdash \ D(c) & \text{by lemma 3.1, using the rule } C(x) \leftarrow D(x).
\end{array}
$$

But the last line is trivially true, by inference rule *1.* □

In this example, the constant symbol c is created during inference, when inference rule *5* is invoked in a top-down mode. The ability to create constants in this way is crucial to the inference system, and it is the basis of the Prolog-style proof procedure developed in [39]. In [15], this ability is used to simulate the computations of arbitrary Turing machines. Without the ability to create new constant symbols, it is only possible to simulate Turing machines that use a polynomial amount of space, i.e., PSPACE-machines (see Section 5). The next example uses recursive rules to generate an unbounded number of new constant symbols and to build a counter of unbounded range.

Example 3.5 [Unbounded Counters] Suppose R contains the following rule:

$$B(x) \leftarrow \forall_y [B(y) \leftarrow NEXT(x,y)]$$

Then, for any $n \geq 0$,

$$\begin{array}{ll} & R \vdash B(c_0) \\ \textit{if} & R + NEXT(c_0, c_1) + NEXT(c_1, c_2) + ... + NEXT(c_{n-1}, c_n) \ \vdash \ B(c_n) \end{array} \tag{4}$$

where $c_1, ..., c_n$ are new and distinct constant symbols. To see this, first suppose that that S is a set of formulas that mentions the constants $c_0....c_{i-1}$, but not c_i. Then,

$$\begin{array}{ll} & R + S \ \vdash \ B(c_{i-1}) \\ \textit{if} & R + S \ \vdash \ \forall_y [B(y) \leftarrow NEXT(c_{i-1}, y)] \\ \textit{if} & R + S \ \vdash \ B(c_i) \leftarrow NEXT(c_{i-1}, c_i) \\ \textit{if} & R + S + NEXT(c_{i-1}, c_i) \ \vdash \ B(c_i) \end{array}$$

Thus, to prove $B(c_{i-1})$, we can create a new constant symbol, c_i, add $NEXT(c_{i-1}, c_i)$ to the database, and then try to prove $B(c_i)$. Carrying out this process recursively leads to statement (4). i.e.,

$$\begin{array}{ll} & R \ \vdash \ B(c_0) \\ \textit{if} & R + NEXT(c_0, c_1) \ \vdash \ B(c_1) \\ \textit{if} & R + NEXT(c_0, c_1) + NEXT(c_1, c_2) \ \vdash \ B(c_2) \\ & \quad \cdots \\ \textit{if} & R + NEXT(c_0, c_1) + NEXT(c_1, c_2) + ... + NEXT(c_{n-1}, c_n) \ \vdash \ B(c_n) \end{array}$$

□

4 Model Theory

The inference system of Definition 3.2 is not classical. That is, it is not sound and complete with respect to classical model theory. This might seem surprising, since each of the five inference rules *is* classical. Indeed, the first three are familiar classical inference rules, the fourth is the deduction theorem, and the fifth was shown to correspond to classical skolemization. Thus, the entire inference system is classically sound, so every inference is a classical theorem. However, the system is not classically complete. That is, there are some classical theorems that it cannot (and should not!) prove.

To see this, consider a deductive database R consisting of the following three rules:

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

If these rules are interpreted classically, then $R \models_c D$. 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 $(A \vee \neg B) \wedge (A \vee C)$. 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 from inference rules in Definition 3.2. In particular, there are only two lines of top-down inference, both of which fail. First,

$$\begin{array}{ll} & R \vdash D \\ \textit{if} & R \vdash C \end{array}$$

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

$$\begin{array}{ll} & R \vdash D \\ \textit{if} & R \vdash A \\ \textit{if} & R \vdash B \leftarrow C \\ \textit{if} & R + C \vdash B \end{array}$$

which fails, since there are no rules for inferring B. Our inference system is therefore not complete with respect to classical semantics. i.e., Hypothetical inference is not classical.

Instead, hypothetical inference is intuitionistic. That is, the inference system of Definition 3.2 is sound and complete with respect to intuitionistic model theory, and therefore forms a fragment of intuitionistic logic. (It is not full intuitionistic logic since embedded implications do not include disjunction or negation.) The intuitionistic nature of embedded implications was first established by Gabbay [25]. Working independently, McCarty and Miller extended this work to include rules with universal quantifiers in their premises [38, 41]. The work of McCarty also includes formulas with negations in rule heads. Recently, a greatly simplified proof of soundness and completeness was developed by Bonner, first for the propositional case [8], and then for the full predicate case [15]. The proof is not based on the fixpoint constructions of logic programming, but on the canonical-model constructions of Modal logic [21].

The rest of this section outlines intuitionistic model-theory, specializing the presentation for the special case of embedded implications. Note that the syntax of the logic is first-order, but its model theory is modal, i.e., is based on a set of possible worlds. A complete development of intuitionistic logic can be found in [23, 35].

Definition 4.1 (Structures) *An intuitionistic structure is a triple $M = \langle S, \leq, \pi \rangle$, where*

- *S is a non-empty set,*
- *$\leq$ is a transitive, reflexive relation on S,*
- *π is a mapping from elements of S to sets of ground atomic formulas,*
- *for any two elements s_1 and s_2 in S, if $s_1 \leq s_2$ then $\pi(s_1) \subseteq \pi(s_2)$.*

The elements of S are called the substates of M.

In a complete development of intuitionistic model theory, each substate may have its own distinct domain of constant symbols. We have assumed here, however, that the domain of each substate is equal to the universe of all constant symbols. In [15], we show that for embedded implications, these are the only kind of structures that one needs to consider. We use these simplified structures since they simplify the theoretical development.

Truth in an intuitionistic structure M is defined relative to its substates. Thus, one can ask whether a formula ψ is true at a particular substate s of M, written $s, M \models \psi$. The following definition makes this idea precise.

Definition 4.2 (Satisfaction) *Suppose M is an intuitionistic structure and s is a substate of M. Then,*

$$
\begin{array}{lll}
s, M \models A & \textit{iff} & A \in \pi(s) \quad \textit{when } A \textit{ is atomic.} \\
s, M \models \psi_1 \wedge \psi_2 & \textit{iff} & s, M \models \psi_1 \quad \textit{and} \quad s, M \models \psi_2 \\
s, M \models \forall_x \psi(x) & \textit{iff} & r, M \models \psi(b) \quad \textit{for all } r \geq s, \textit{ and all constant symbols } b. \\
s, M \models \psi_2 \leftarrow \psi_1 & \textit{iff} & r, M \models \psi_1 \ \textit{ implies } \ r, M \models \psi_2 \quad \textit{for all } r \geq s
\end{array}
$$

Note that unlike classical logic, intuitionistic implication is not defined in terms of disjunction and negation. Rather, it has an independent semantic definition. In effect, intuitionistic implication is a binary modal operator. An intuitive interpretation may be found in [35] and [23].

Definition 4.3 (Models) *$M \models \psi$ iff $s, M \models \psi$ for all substates s of M. If $M \models \psi$, then M is a model of ψ.*

Definition 4.4 (Validity) *A formula ψ is valid iff $M \models \psi$ for all intuitionistic structures M.*

Definition 4.5 (Entailment) *Suppose ψ_1 and ψ_2 are formulas. Then $\psi_1 \models \psi_2$ iff the formula $\psi_2 \leftarrow \psi_1$ is valid.*

The following theorem is a central result of [15]:

Theorem 4.1 (Soundness and Completeness) *If ϕ is an embedded implication, and R is a set of embedded implications, then $R \models \phi$ iff $R \vdash \phi$.*

5 Data Complexity

This section establishes the data complexity of embedded implications. To do this, we formally divide a deductive database into two parts: (i) a set of embedded implications, R, called the rulebase, and (ii) a set of ground atomic formulas, DB, called the database. Derivations thus have the form $R + DB \vdash \psi$. Informally, the *data complexity* of this system is the complexity of inference when the rulebase is fixed and the database varies (acts as input). More formal definitions are given in [18, 53, 11]. We first show that inference with embedded implications is semi-decidable, and more specifically, that its data complexity is complete for re. We then show that the data complexity can be reduced—first to PSPACE, and then to NP— by imposing natural restrictions on the syntax of the rulebase.

Theorem 5.1 *The data complexity of embedded implications is complete for re.*

To prove completeness, we use embedded implications to encode the computations of an arbitrary Turing machine. The main step is to generate new constant symbols during inference, and to use them to construct a counter, as in Example 3.5. The counter provides a way of keeping track of computation time and tape position. Since embedded implications can generate an unbounded number of new constant symbols, we can construct a counter of unbounded range, and thereby use as much time and tape as the Turing machine needs. A detailed proof is given in [15].

The computational power of embedded implications thus comes from their ability to generate new constant symbols during inference. This ability, in turn, comes from the presence of universal quantifiers in rule bodies, as in the rule $A(x) \leftarrow \forall_y B(x,y)$. By banning such quantifiers, we can eliminate the creation of new constants and thus reduce the complexity of inference considerably. To do this, we simply ignore item *3* in Definition 3.1. Note that in the resulting rules, all variables are free. As in Horn logic, such variables are assumed to be universally quantified at the top level. Thus, a rule such as $A(x) \leftarrow [B(x,y) \leftarrow C(x,y)]$ is taken to mean $\forall_x \forall_y A(x) \leftarrow [B(x,y) \leftarrow C(x,y)]$.

Theorem 5.2 (No New Constants) *If universal quantifiers are not used in rule bodies, then the data complexity of embedded implications is complete for PSPACE.*

Detailed proofs of this theorem are given in [11] and [10]. Briefly, inference can be viewed as a top-down search through an and/or proof tree of polynomial depth. Because of the depth bound, the search can be carried out in polynomial space, which establishes the complexity upper-bound. In addition, because it is an and/or tree, the search can be highly complex. In fact, the search can simulate the computations of an arbitrary Alternating PTIME machine. This establishes the complexity lower-bound, since APTIME = PSPACE [20].

By imposing another syntactic restriction, called *linearity*, we can reduce the complexity of inference again, from PSPACE to NP. When a rulebase is linear, a proof tree has very few "and" nodes; so inference is simplified, becoming largely a process of searching an or-tree. Such searches can be done in non-deterministic polynomial time.

Informally, a rule is linear iff recursion occurs through only *one* premise. In Horn-clause logic, "linear rules play an important role because, (i) there is a belief that most 'real life' recursive rules are linear, and (ii) algorithms have been developed to handle them efficiently" [6]. A precise definition of linear recursion for embedded implications is given in [11, 9]. The following example illustrates the idea.

Example 5.1

A Linear Rulebase	A Non-Linear Rulebase
$A \leftarrow (B \leftarrow D_1), (G \leftarrow D_2)$	$A \leftarrow (B \leftarrow D_1), (C \leftarrow D_2)$
$B \leftarrow (C \leftarrow E_1), (H \leftarrow E_2)$	$B \leftarrow (C \leftarrow E_1), (A \leftarrow E_2)$
$C \leftarrow (A \leftarrow F_1), (K \leftarrow F_2)$	$C \leftarrow (A \leftarrow F_1), (B \leftarrow F_2)$

$\square$

The following theorem is the main result about linear embedded implications. Detailed proofs are given in [11, 9].

Theorem 5.3 (Linear Recursion) *If universal quantifiers are not used in rule bodies, and if recursion is linear, then the data complexity of embedded implications is complete for NP.*

6 Negation as Failure

This section extends embedded implications by allowing negated premises. Thus, rules of the form $A \leftarrow \neg(B \leftarrow C)$ are now allowed. Operationally, the expression $\neg(B \leftarrow C)$ is interpreted as the failure to prove $B \leftarrow C$. Thus, A is inferred if $B \leftarrow C$ cannot be inferred. This semantics is similar to the semantics of negation as commonly defined in Horn logic programming [5].

Example 6.1 The rules below are part of a rulebase that defines a student's eligibility for financial aid.[4] 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.

$$\begin{array}{l} stipend(s) \leftarrow admitted(s),\ near\_grad(s),\ \neg grad(s). \\ fellowship(s) \leftarrow admitted(s),\ \neg near\_grad(s),\ \neg grad(s). \\ near\_grad(s) \leftarrow [grad(s) \leftarrow take(s,c)] \end{array}$$

In applying the rule for $near\_grad(s)$, we ask if the student s is within one course of graduation. That is, is there some course c such that, if $take(s,c)$ were assumed to be true, then $grad(s)$ 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 $grad(s)$ 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 $\neg grad(s)$ is the rule for $stipend(s)$. This means that a stipend is available only to those students who need *exactly* one course to graduate. $\square$

A major difficulty with negation-as-failure is that it is not always well-defined. This is true both for Horn rules and for embedded implications. For instance, given the two rules $A \leftarrow \neg B$ and $B \leftarrow \neg A$, it is unclear whether A is to be inferred, or B, or both, or neither. Numerous approaches to this problem have been proposed. One approach has been to identify classes of Horn rulebases in which such problems do not arise. Perhaps the best known of these classes is the *stratified rulebases*. These rulebases are layered, and within each layer, a negated premise refers only to rules found in the layers below [4]. In this way, recursion never occurs through a negated premise, so the semantics of negation is

[4]This example, taken from [14], is an improvement by McCarty on an example originally due to Bonner [10].

always well-defined. Another approach has been to allow limited recursion through negation. This approach generalizes the notion of stratification to *local stratification* [45]. Intuitively, a rulebase is locally stratified if its ground instantiation is stratified (with possibly infinitely many strata). Locally stratified rulebases are perhaps the largest class of logic programs for which the semantics of negation is uncontroversial. A third approach to negation-as-failure attempts to define semantics for arbitrary Horn programs with negation. There have been numerous attempts here too. Perhaps the best known are the well-founded semantics [28, 52] and the stable model semantics [29]. Although, these semantics offer different interpretations of recursion through negation, they are equivalent when rulebases are stratified or locally stratified.

Several researchers have extended these ideas to embedded implications with negation. For instance, [14, 10, 11] develop a semantics of stratified rulebases, including a proof theory and model theory. Examples 6.1, 7.1 and 7.2 show stratified rulebases and describe their semantics informally. In the special case in which all the embedded implications are Horn, this semantics is equivalent to that of [4]. As in Horn logic, stratified rulebases act as a kind of benchmark in the semantics of negation. One reason is that most natural examples of negation are stratified. Another is that the semantics of non-stratified rulebases is likely to remain unsettled for some time. In addition, stratified embedded implications have important complexity-theoretic properties, as shown in Section 7.

Recently, there have been several proposals for a semantics of non-stratified embedded implications [31, 43, 30, 22]. Following [41], these proposals each define a model as a mapping from a set of logic programs to a set of values. These proposals are therefore not model theories in the traditional sense, since they only consider models of a certain syntactic form. (In contrast, [14] takes the traditional position that a model is a mapping from *any* set to a set of first-order semantic structures.) In [31], Harland develops a semantics in which predicates are labelled as "completely defined" or "incompletely defined". Unfortunately, restrictions on the labelling preclude many stratified programs, such as those in Examples 7.1 and 7.2. They also preclude the rulebases used to achieve the expressibility results of Section 7. This point is discussed at greater length in [14]. In [43], Olivetti and Terracini present a 3-valued modal semantics for the propositional case, and in [30], Giordano and Olivetti develop a 3-valued fixpoint semantics for the predicate case. In both cases, the semantics is of finite failure in arbitrary rulebases. [30] also develops a top-down SLD-style proof procedure. Under a syntactic restriction call *allowedness*, this procedure is sound and avoids floundering. Unfortunately, this restriction seems to preclude many stratified programs, such as those in Examples 6.1, 6.2 and 7.2. Finally, in [22], Dung proposes a stable model semantics that allows arbitrary rulebases, stratified or unstratified, though no proof theory is presented.

None of the expressibility results in Section 7 depends on recursion through negation. For the purpose of this paper, then, we need only consider stratified rulebases. As described above, [14, 10, 11] extend the notion of stratification from Horn programs to embedded implications. The main new idea is that in defining strata, hypothetical assumptions can be ignored. For example, in the rule $A \leftarrow (B \leftarrow C)$, the atom C may be defined in any stratum, including strata that are *above* the strata in which A and B are defined. In contrast, the atom B must be defined in the same stratum as A or in lower strata, exactly as with stratified Horn rulebases. Intuitively, the atom B is "executed" in order to prove A, while the atom C is "assumed." This is why C does not affect the stratification; i.e., since C is not "executed," it cannot lead to recursion through negation.

Example 6.2 The following rulebase is stratified and has three strata, R_0, R_1, and R_2.

$$R_2 \begin{cases} A_2(x) \leftarrow B(x,y),\ [A_2(y) \leftarrow D(y)]. \\ A_2(x) \leftarrow C(x,y),\ \neg A_1(y). \end{cases}$$

$$R_1 \begin{cases} A_1(x) \leftarrow B(x,y),\ [A_1(y) \leftarrow D(y)]. \\ A_1(x) \leftarrow C(x,y),\ \neg A_0(y). \end{cases}$$

$$R_0 \begin{cases} A_0(x) \leftarrow B(x,y),\ [A_0(y) \leftarrow D(y)]. \\ A_0(x) \leftarrow C(x,y). \end{cases}$$

□

As originally pointed out by Gabbay [25], negation-as-failure displays curious paradoxes when extended from Horn rules to embedded implications. This is true even for very simple, non-recursive rulebases. In fact, if implicational queries are allowed, then these paradoxes arise even for *Horn* rules. The next example shows, for instance, that for stratified Horn rules, implication is not transitive. Other apparent paradoxes, including a failure of Modus Ponens, are described in [14].

Example 6.3 Suppose that R consists of the following two rules:

$$A \leftarrow B,\ \sim C.$$
$$B \leftarrow C.$$

Then $R \vdash A \leftarrow B$ and $R \vdash B \leftarrow C$, but $R \not\vdash A \leftarrow C$. Each of these points is straightforward to prove. For instance, to infer $A \leftarrow C$ from R, we would have to add C to the rulebase and then infer A; that is, we would have to derive the expression $R + C \vdash A$. However, once C is added the rulebase, the first rule is permanently blocked, so there is no way to derive A. Thus, $R + C \not\vdash A$. Hence $R \not\vdash A \leftarrow C$. □

These paradoxes are not hard to resolve, and as the examples in this paper suggest, they do not affect the practice of logic programming. As first pointed out in [14], embedded implications really have two distinct kinds of implication—a distinction that one naturally makes when writing logic programs. In the rule $A \leftarrow (B \leftarrow C)$, for instance, the outer implication and the inner implication are of a different kind. As noted above, the outer implication means that B is *executed*, in order to prove A, while the inner implication means that C is *assumed*. The paradoxes arise from an attempt to treat these two kinds of implication as one and the same thing. Even the inference system of Definition 3.2 distinguishes these two kinds of implication: inference rule *3* deals with "execution," and inference rule *4* deals with "assumption." It should not be surprising, then, that these two types of implication have different semantics. Indeed, the real surprize is that this difference does not make itself apparent until the proof theory is augmented with negation-as-failure. In the rest of this paper, we distinguish implicitly between these two kinds of implication. Thorough developments of them can be found in [11] and in [14]. [11] develops a relatively simple semantics, while [14] develops a more complex semantics, but one that is much closer to intuitionistic model theory. These works address the paradoxes due to hypothetical insertion. Additional paradoxes arise, however, when universal quantifiers are allowed in rule bodies. For instance, it is possible that a rulebase may entail $\forall_x A(x)$, and yet it may *not* entail $A(b)$ for any constant, b. Such issues have received little attention in the literature, and we address them in [7].

7 Expressibility

This section studies the impact of negation-as-failure on the ability of embedded implications to express database queries. We first show that with negation, embedded implications are expressively complete, i.e., that they can express *any* computable database query. We then focus on the syntactic restrictions of Section 5 and the impact that negation has on them. We show, for instance, that if new constants are not created during inference, then the logic is expressively complete for PSPACE, so that it can express *any* database query computable in polynomial space. Based on this result, we develop an *exact* characterization of the queries in PSPACE in terms of embedded implications. If we add the additional restriction that recursion is linear, then negation has a subtle and interesting effect on the logic. In this case, the complexity and expressibility depend on the number of strata in the rulebase: with each additional strata, the complexity and expressibility climb one level in the polynomial-time hierarchy. As a corollary, we conclude that stratified linear rulebases express exactly the second-order definable queries.

Unlike other results in the literature [32, 53, 19], 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 a linear order on the domain hypothetically. Thus, the results of this section are for arbitrary databases, ordered or not. These results characterize the queries in well-known complexity classes based on ordinary Turing machines. In particular, our results are unrelated to the relational complexity classes introduced in [1], which are based on so-called relational Turing machines. Although relational Turing machines capture many of the important features of database programming, they are less expressive than ordinary Turing machines. For instance, the query *EVEN* discussed below is in the complexity class P, but is not in the relational version of P. As we shall see, this query is expressible in embedded implications, although it is not expressible in many other relational query languages.

Before continuing, we must define what it means to express a database query using embedded implications. Complete definitions are given in [10, 11, 15]. Here, we summarize the main ideas. Formally, a database query is a mapping that takes a database as input and returns a set of ground tuples as output [17]. We can use embedded implications with free variables to define such mappings. Some care is required though, since the inference system of Definition 3.2 is defined only for embedded implications that are closed, i.e., that have no free variables. This presents no conceptual difficulties however, since we can focus on ground instantiations of the free variables. Formally, let $\phi(x_1, ..., x_k)$ be an embedded implication with free variables $x_1, ..., x_k$. Also, given a rulebase, R, and a database, DB, let $dom(DB)$ be the set of constant symbols in $R + DB$. Together, R and $\phi(x_1, ..., x_k)$ define a database query. When applied to database DB, the answer to this query is the following set of tuples:

$$\{ \langle c_1, ..., c_k \rangle \mid c_i \in dom(DB) \quad and \quad R + DB \vdash \phi(c_1, ..., c_k) \}$$

The inferences in this definition are all well defined since each $\phi(c_1, ..., c_k)$ is closed. Furthermore, if $dom(DB)$ has n elements, then there are exactly n^k ground instances of $\phi(x_1, ..., x_k)$. Thus, computing the answer to a query requires checking n^k inferences. This adds only a polynomial factor to the complexity bounds established in Section 5. Thus, the bounds are the same for yes/no queries and for queries that return tuples as answers. Moreover, in practice, it is not necessary to actually check n^k ground inferences, since unification-based proof procedures exist for embedded implications with free variables. One of the first such procedures was developed by McCarty [39] for the negation-free case. Having defined what it means to express queries with embedded implications, the rest of this section characterizes the exact sets of queries that can be expressed.

As shown in Section 5, without negation-as-failure, embedded implications can simulate arbitrary Turing machines. Nevertheless, there are some simple, low-complexity queries that they cannot express. This is because the negation-free logic is *monotonic*: As the database expands, the answer to a query also expands. This behavior is typical of many logical inference systems, such as Datalog, Horn logic, full classical logic, and modal logics. Such systems cannot express non-monotonic queries, such as relational algebra queries involving complementation. Retrieving those students "who are *not* eligible to graduate" is an example of a non-monotonic query. Even with enormous computational power, monotonic query languages cannot express this simple query, simply because it is non-monotonic.

As another example, consider the query $EVEN$, which determines the parity of a database relation D: $EVEN$ returns *true* if D has an even number of entries, and *false* otherwise. $EVEN$ is non-monotonic since as entries are added to D, the value of $EVEN$ "flips back and forth" between *true* and *false*; i.e., once $EVEN$ becomes true, it does not remain true, as a monotonic query would. Without negation, our hypothetical logic is monotonic and cannot express the query $EVEN$. To be precise, there is no set of (negation-free) embedded implications, R, that makes the following statement true for *every* database, DB:

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

where $EVEN$ is a 0-ary predicate symbol. Again, this is true not just of embedded implications, but of all monotonic logics.

Augmenting a logic with negation-as-failure gives it the power to express non-monotonic queries. Often, however, this increased power is still limited. For instance, Datalog with negation-as-failure still cannot express $EVEN$ [18]. In such cases, a logic is often augmented with other devices, to guarantee expressive completeness. One typical device is to assume that the data domain is linearly ordered [32, 53, 19]. Datalog, for instance, is expressively complete for PTIME if augmented with *both* negation-as-failure and a linear order. Embedded implications do not need to make this latter assumption, however, since they can generate a linear order for themselves, and insert it into the database hypothetically [11, 10]. Indeed, negation-as-failure is the only device needed to achieve expressive completeness for embedded implications. Furthermore, negation-as-failure is a natural extension to the logic, since any practical logic-programming system must have it.

7.1 Examples

This section gives two examples of the expressiveness of embedded implications augmented with negation-as-failure. The first example shows how to express $EVEN$. The second example shows a simple encoding of an NP-complete problem.

The rulebase in Example 7.1 expresses $EVEN$. To understand this rulebase, we view its operation as top-down. From this perspective, the first two rules select and mark elements one-by-one from the database relation D. As elements are marked, query processing "flips back and forth" between the two subqueries $EVEN$ and ODD until all the elements in D are marked. In effect, these subqueries ask for the parity of the unmarked tuples in D. Thus, when there are no unmarked tuples left, the parity is even. The third and fourth rules handle this situation, terminating the recursion. Note that a tuple x is "marked" by hypothetically adding the formula $MARK(x)$ to the database. The fifth rule uses this fact to select an unmarked tuple from D.

Example 7.1 Suppose R is the following collection of rules:

$$\begin{array}{l}
EVEN \leftarrow SELECT(x), [ODD \leftarrow MARK(x)] \\
ODD \leftarrow SELECT(x), [EVEN \leftarrow MARK(x)] \\
EVEN \leftarrow \neg SOMELEFT. \\
SOMELEFT \leftarrow SELECT(x). \\
SELECT(x) \leftarrow D(x), \neg MARK(x).
\end{array}$$

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

With negation-as-failure, it is possible to write simple hypothetical rulebases that solve complex problems. The next example illustrates this by showing how to find a Hamiltonian path in a directed graph. Since this is an NP-complete problem, the example provides a simple demonstration of why the data complexity of hypothetical reasoning is not in P (assuming that P $\neq$ NP).

Example 7.2 Suppose that DB is a database representing a directed graph. That is, $NODE(a) \in DB$ iff a is a node in the graph, and $EDGE(a, b) \in DB$ iff there is an edge in the graph from a to b. Suppose also that R is the following collection of rules:

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

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

To understand the rulebase in Example 7.2, we view its operation as top-down. From this perspective, the rulebase tries to construct a Hamiltonian path one node at a time. The first rule selects a node x at which the path is to begin. The second rule is then applied repeatedly, selecting a node y connected 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.

7.2 Expressive Completeness

Negation-as-failure increases the expressibility of embedded implications. In fact, with negation, they are expressively complete, and can therefore express any computable generic query. In database theory, the ability to express generic queries is the normal measure of expressiveness for database query languages [2, 3, 17, 18, 32, 33].

Genericity captures the idea that constant symbols are uninterpreted, i.e., that they have no innate meaning. To make this idea more precise, consider the effect on a query of renaming the constant symbols in the database. If a query is generic, then this should have no effect other than to rename the constants in the output of the query. As a special case, the output of a yes/no query should not be affected at all. Thus, the query $EVEN$ in Example 7.1 is generic, since its output is not affected by a renaming of the constant symbols in the database. Likewise, the query in Example 7.2 is generic. More precise definitions of genericity are given in [17, 11]. The following theorem is our first result about the ability of embedded implications to express generic queries.

Theorem 7.1 *Stratified embedded-implications are expressively complete; that is, they express all the generic queries in re.*

Although they express all the generic queries in re, stratified embedded implications can also express queries that are *not* in re. For example, suppose the rulebase R defines a query $A(x)$ that is complete for re. Then the query $\neg A(x)$ is complete for *co-re*, and thus is not in re. Thus, negation-as-failure increases the data complexity of embedded implications. In fact, by using multiple strata, it provides enough power to express queries at any level in the arithmetic hierarchy. It would be interesting, however, to provide an exact, logical characterization of the generic queries in re. To this end, [15] presents simple syntactic restrictions that do precisely this. Any query satisfying these restriction is guaranteed to be in re, and all queries in re can be expressed by a rulebase satisfying these restrictions.

Section 5 showed that under certain syntactic restrictions, the data complexity of embedded implications is PSPACE complete. Unlike re, however, PSPACE is closed under complementation; that is, PSPACE = coPSPACE. Thus, if $A(x)$ is a PSPACE-complete query, then so is $\neg A(x)$. For this reason, negation-as-failure does not increase the complexity of these restricted embedded implications. It does, however, increase their expressiveness, since with negation, they can express non-monotonic queries. In fact, the following result shows that negation-as-failure increases their expressiveness as much as possible.

Theorem 7.2 *Under the following restrictions, embedded implications are expressively complete for PSPACE, that is, they express all the generic queries in PSPACE:*

1. *The rulebase is stratified.*
2. *Universal quantifiers do not appear in rule bodies.*

Although the rulebases in Theorem 7.2 can express every generic query in PSPACE, they can also express some non-generic queries. Non-genericity is caused by constant symbols in a rulebase, which cause the rulebase to give special treatment to some elements of the data domain. For example, a rulebase consisting of the single rule $A \leftarrow B(c)$ will infer A iff $B(c)$ is in the database. Since renaming c in the database will stop the inference of A, this rulebase defines a non-generic query. On the other hand, if the rulebase were $A \leftarrow B(x)$, where x is a variable, then a renaming of the database would not affect the inference of A at all. In this case, the rulebase defines a generic query. In general, queries defined by constant-free rulebases are guaranteed to be generic. In fact, such rulebases provide an exact characterization of the generic queries in PSPACE.

Theorem 7.3 *Under the following restrictions, embedded implications express exactly the generic queries in PSPACE:*

1. *The rulebase is stratified.*
2. *Universal quantifiers do not appear in rule bodies.*
3. *The rulebase contains no constant symbols.*

Theorem 7.3 is particularly interesting because it establishes a strong link between an important complexity class, polynomial space, and a well-known logic, intuitionistic logic. Other database query languages have been developed that are expressively complete for PSPACE [16, 18, 2, 3], but the language of stratified embedded implications is the only one based on a model theoretic semantics, and the only one to have an Prolog-style proof procedure, one based on resolution and unification in the logic-programming tradition [39]. Furthermore, unlike other languages, our results on PSPACE can be extended, as shown below, to include the polynomial time hierarchy and its various levels, thereby linking intuitionistic logic to other important complexity classes.

7.3 The Polynomial-Time Hierarchy

Section 5 showed that if embedded implications do not create new constant symbols, then the complexity of inference decreases from re to PSPACE. In addition, if recursion is linear, then complexity decreases again, from PSPACE to NP. Since NP is not closed under complementation (i.e., NP $\neq$ coNP), it should not be surprising that negation-as-failure effects the data complexity of these linear rulebases. For instance, if $A(x)$ is an NP-complete query, then $\neg A(x)$ is a coNP-complete query. In general, the data complexity will correspond to some level in the polynomial-time hierarchy, depending on the number and type of strata in the rulebase. This result eventually leads to a characterization of the second-order definable queries in terms of (first-order) embedded implications.

The polynomial-time hierarchy is a sequence of complexity classes between P and PSPACE. It is based on the idea of an oracle Turing-machine and can be defined recursively as follows [49]:

- $\Sigma_0^{\mathrm{P}} = \Delta_0^{\mathrm{P}} = \mathrm{P}$.
- $\Delta_{k+1}^{\mathrm{P}} = \mathrm{P}^{\Sigma_k^{\mathrm{P}}}$ = 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 $\mathrm{P} \subseteq \Delta_k^{\mathrm{P}} \subseteq \Sigma_k^{\mathrm{P}} \subseteq \Delta_{k+1}^{\mathrm{P}} \subseteq \mathrm{PHIER} \subseteq \mathrm{PSPACE}$.[5] This section provides several results linking the polynomial-time hierarchy to embedded implications. The simplest one to state is the following.

Theorem 7.4 *Under the following restrictions, the data complexity of embedded implications is complete for Σ_k^{P}, for $k \geq 1$:*

1. *The rulebase has k strata.*
2. *Recursion is linear.*
3. *Universal quantifiers do not appear in rule bodies.*

It turns out, however, 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 say that such strata are *Horn*. Other strata shall be called *hypothetical*. A hypothetical stratum thus contains at least one rule with an implication in its rule body, e.g., $A \leftarrow (B \leftarrow C)$. The rulebase in Example 7.1 has one Horn stratum (the last two rules) and one hypothetical stratum (the first three rules). The following theorem shows that if a rulebase has k hypothetical strata, then no matter how many Horn strata are inserted into it, the data complexity will not exceed $\Delta_{k+1}^{\mathrm{P}}$.

Theorem 7.5 *Under the following restrictions, the data complexity of embedded implications is complete for $\Delta_{k+1}^{\mathrm{P}}$, for $k \geq 0$:*

1. *The rulebase has k hypothetical strata.*
2. *Recursion is linear.*
3. *Universal quantifiers do not appear in rule bodies.*

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

As discussed at the beginning of Section 7, it does not follow from these complexity results that stratified linear rulebases are expressively complete. That is, although their data complexity may be complete for some class, there may still be many queries in that class that they cannot express. This is in fact the case for the rulebases described in Theorems 7.4 and 7.5. They do not appear to be expressively complete for Σ_k^{P} or Δ_k^{P}, respectively. Technically, we can obtain expressive completeness for Σ_k^{P} by assuming that the top stratum of the rulebase is hypothetical, as in Examples 7.1 and 7.2. This result leads to a characterization of the queries in Σ_k^{P} [11]. It also leads to an elegant characterization of the entire polynomial-time hierarchy.

Theorem 7.6 *Under the following restrictions, embedded implications express exactly the generic queries in PHIER:*

1. *The rulebase is stratified.*
2. *Recursion is linear.*
3. *Universal quantifiers do not appear in rule bodies.*
4. *The rulebase contains no constant symbols.*

This corollary immediately gives rise to a characterization of the second-order definable queries, since Immerman has shown that the generic queries in PHIER are precisely the queries definable in second-order logic [33]. This gives us an alternate characterization of the queries expressed by linear embedded-implications. It also gives the interesting result that the second-order queries can be characterized by a first-order logic.

Corollary 7.7 *The rulebases described in Theorem 7.6 express exactly the second-order definable queries.*

Acknowledgments

The work of Thorne McCarty on the intuitionistic semantics of embedded implications was the original stimulus for this work; and discussions with Tomasz Imielinski were invaluable in giving the work a database perspective. Many of the results relating to the creation of new constant symbols were developed in collaboration with Thorne McCarty and Kumar Vadaparty [15]. They also provided useful comments on the present paper. Thanks go to Eric Allender for answering all my questions about computational complexity, and to Ashok Chandra for providing valuable comments. Jan Chomicki and Ron van der Meyden also provided useful feedback during the development of the work presented herein. The anonymous reviewers of this paper also provided valuable references.

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] S. Abiteboul and V. Vianu. A Transaction Language Complete for Database Update Specification. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 260–268, 1987. Published in expanded form as Rapports de Recherche no. 715, INRIA, 78153 Le Chesnay Cedex, France.

[3] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. Technical Report 900, Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay Cedex, France, 1988.

[4] K.R. Apt, H.A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann, 1988.

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

[6] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., May 28–30 1986.

[7] A.J. Bonner. Adding Negation-as-Failure to Intuitionistic Logic Programming: Part II. In preparation.

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

[9] A.J. Bonner. Hypothetical Datalog: Negation and Linear Recursion. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 286–300, Philadelphia, PA, March 29–31 1989.

[10] A.J. Bonner. Hypothetical Datalog: Complexity and Expressibility. *Theoretical Computer Science (TCS)*, 76:3–51, 1990. Special issue on the 2^{nd} International Conference on Database Theory (ICDT).

[11] 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.

[12] A.J. Bonner. The complexity of reusing and modifying rulebases. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 316–330, San Diego, CA, June 2–4 1992.

[13] A.J. Bonner and T. Imielinski. The Reuse and Modification of Rulebases by Predicate Substitution. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 437–451, Venice, Italy, March 26–30 1990. Springer-Verlag. Published as volume 416 of *Lecture Notes in Computer Science*.

[14] 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.

[15] 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.

[16] A.K. Chandra. Theory of Database Queries. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 1–9, Austin, Texas, March 1988.

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

[18] A.K. Chandra and D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences (JCSS)*, 25(1):99–128, 1982.

[19] A.K. Chandra and D. Harel. Horn Clause Queries and Generlizations. *Journal of Logic Programming (JLP)*, 2(1):1–15, 1985.

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

[21] B.F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, 1980.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] D.M. Gabbay and U. Reyle. Computation with Run Time Skolemization. *Journal of Applied Non-Classical Logics*, 3:93–134, 1993.

[28] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[29] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the Fifth Logic Programming Symposium*, pages 1070–1080. MIT Press, 1988.

[30] 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.

[31] 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.

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

[33] N. Immerman. Languages that Capture Complexity Classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[34] R.H. Sprague Jr. and H.J. Watson, editors. *Decisions Support Systems: Putting Theory into Practice*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming (JLP)*, 6:79–108, 1989.

[42] 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.

[43] 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.

[44] 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.

[45] T. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann, 1988.

[46] 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.

[47] Editor: A.P. Seth. Database Research at Bellcore. *SIGMOD Record*, 19(3):45–52, September 1990.

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

[49] L.J. Stockmeyer. The Polynomial Time Hierarchy. *Theoretical Computer Science (TCS)*, 3(1):1–22, 1976.

[50] M. Stonebraker. Hypothetical databases as views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 224–229, 1981.

[51] 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.

[52] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 221–230, March 1988.

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

[54] 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.

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