Implementing Deductive Databases by Mixed Integer Programming

COLIN BELL
University of Iowa

ANIL NERODE
Cornell University

RAYMOND T. NG
University of British Columbia
and
V. S. SUBRAHMANIAN
University of Maryland

Existing and past generations of Prolog compilers have left deduction to run-time and this may account for the poor run-time performance of existing Prolog systems. Our work tries to minimize run-time deduction by shifting the deductive process to compile-time. In addition, we offer an alternative inferencing procedure based on translating logic to mixed integer programming. This makes available for research and implementation in deductive databases, all the theorems, algorithms, and software packages developed by the operations research community over the past 50 years. The method keeps the same query language as for disjunctive deductive databases, only the inferencing procedure changes. The language is purely declarative, independent of the order of rules in the program, and independent of the order in which literals occur in clause bodies. The technique avoids Prolog's problem of infinite looping. It saves run-time by doing primary inferencing at compile-time. Furthermore, it is incremental in nature. The first half of this article translates disjunctive clauses, integrity constraints, and database facts into Boolean
Implementing Deductive Databases

I,quatians. and develops procedures to use mixed integer programming methods to compute least models of definite deductive databases, and minimal models and the Generalized Closed World Assumption of disjunctive deductive databases.

These procedures are sound and complete. The second half of the article proposes a query processing system based on mixed integer programming compilation, and describes our (implemented) prototype compiler. Experimental results using this compiler are reported. These results suggest that our compilation-based mixed integer programming paradigm is a promising approach to practical implementation of query systems for definition and disjunctive databases.

Categories and Subject Descriptors: F.4.1 Mathematical Logic and Formal Languages
Mathematical Logic: Mathematical Logic: H.2.3 Database Management: Languages: Artificial Intelligence: Knowledge Representation Formalisms and Methods

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Deductive databases, disjunctive databases, logic programs, minimal models, negation and disjunction in deductive databases

1. INTRODUCTION

Using linear programming techniques as a basis for proving a propositional theory goes back to the earliest days of logic. Thus it far predates the usual history of linear programming, as it goes back to Boole (see Hailperin [1976]). This approach was revived and perfected by Jeroslow [Blair et al. 1988; Jeroslow 1988, 1989; Jeroslow and Wang 1990] whose work has been continued by Hooker [1988, 1990]. A quick summary of the fundamental idea is that logic deduction can be replaced by linear programming if logical conditions such as Horn Clauses are written first as Boolean inequalities

\[ c_1 x_1 + \cdots + c_n x_n \geq a, \]

where \( a, c_1, \ldots, c_n \) are integers. This gives a linear programming problem in the real domain that can be solved symbolically using the usual linear programming packages and then using techniques such as cutting-plane to find the 0–1 solutions for the original logic problem. (See Section 6 for a more detailed discussion.)

Here we extend this approach to compute models of deductive databases. In the past decade, significant progress has been made in bottom-up computation of deductive databases. This includes works by Bancilhon, Ramakrishnan, Beeri, and others, [Bancilhon et al. 1986; Bancilhon and Ramakrishnan 1986; Beeri and Ramakrishnan 1987], and Ullman [1989]. Several of these paradigms have been implemented; the LDL system at MCC [Naqvi and Tsur 1989; Shmueli et al. 1988; Zaniolo 1988], the CORAL system at Wisconsin-Madison [Ramakrishnan et al. 1992], the NAIL system [Morris et al. 1986] at Stanford, and the XSB system at Stonybrook [Warren 1992; Dietrich 1987]. There have also been attempts to develop methods that include both top-down and bottom-up computations [Bry 1989; Warren 1992].

One fundamental difference between our approach and bottom-up approaches developed thus far is that we rely on numerical optimization.
techniques for computing deductive databases. In other words, we solve a symbolic logic program by treating it as a numerical problem. One major advantage of our approach is that it provides a unified framework for computing different semantics of deductive databases. In this paper, we focus on computing the least model of definite deductive databases, minimal models, and the Generalized Closed World Assumption of disjunctive deductive databases [Minker 1982]. In Bell et al. [1994], and Nerode et al. [1995], we show how to use our approach to compute nonmonotonic and classical negations in deductive databases, and how to support circumscriptive deductive databases. Another advantage of our approach is that the language support is purely declarative, independent of the order rules in databases, and independent of the order in which literals occur in rule bodies. Moreover, our approach is incremental in nature. Techniques for adding and deleting constraints have a long history in the linear programming literature, and incremental algorithms are well developed. Thus the transition to this new computational paradigm supports efficient updating and can be used in conjunction with the existing body of work on updating databases. Finally, parallel algorithms to solve linear programming problems have already been developed [Hillier and Lieberman 1974; Luo and Reijns 1992]. Thus our approach has benefited from advances in parallel computing.

To a large extent, models of deductive databases computed in our framework are analogous to materialized views in relational databases. In these databases, the intensional database (IDB) is a collection of views and the extensional database (EDB) is a collection of data or facts. In forming a materialized view, the IDB and EDB are compiled together. The ultimate goal is to maximize run-time performance by minimizing computation (except simple operations such as retrievals) at run-time. The intention is the same for our approach. By combining the IDB and EDB at compile-time, we hope to provide effective support for such applications as control systems that require real-time responses to queries [Kohn 1991; Kohn and Nerode 1993]. See Section 5 for a description of how our system interacts with a relational database system. Note that like the materialized view community, we do not assume that the IDB does not change over time. Indeed, changes can, and are supposed to, occur often. Thus one important issue for approaches combining IDB and EDB at compile-time to address is how to update the compiled structures to reflect the changes. See Blakeley et al. [1986] and Roussopoulos [1991] for updating materialized views of relational databases. As mentioned before, updating a computed model in our framework can be readily supported by incremental linear programming techniques. Section 6 provides some experimental evidence on how effective linear programming techniques can support incremental processing.

It is well known that the Generalized Closed World Assumption (GCWA) for propositional deductive databases is co-NP-hard [Cadoli and Lenzerini 1990]. Thus computing the minimal models of even propositional deductive databases is a complex task. The only way to determine the efficiency of a proposed algorithm for computing the GCWA is via exhaustive experimentation on typical expected cases. In this paper we develop a technique for
computing the minimal models of a disjunctive program. We cannot yet make any definite statement of efficiency of our mixed integer programming approach vis-a-vis other approaches. What we can report is that our method performs well, shows some promise, and deserves further study. Based on our experiments, it seems certain that more sophisticated techniques will emerge based on a deeper understanding of how mixed integer programming handles query problems. The area is in an exploratory phase of development.

The organization of the paper is as follows. In the next section, we present first a translation of deductive databases to sets of linear constraints. We study the relationship between models of deductive databases and solutions of the constraints. In Section 3, we give a description of how to compute the least models of definite deductive databases by optimizing an appropriate objective function subject to the constraints associated with the definite deductive database. In Section 4, we give sound and complete algorithms to compute all the minimal models of disjunctive programs. We also describe an optimization algorithm that reduces the number of constraints and variables. In Section 5, we outline a system design that implements the idea of query processing based on mixed integer programming compilation. We also describe our implemented prototype compiler. We then discuss how models can be stored and accessed using relational database management systems. In Section 6, we summarize experimental results on the performance of our prototype compiler and compare our approach with related work. Finally, we conclude with a discussion on future work. The Appendix lists the animal database we experimented with in Section 6.

2. TRANSLATING LOGIC PROGRAMS TO LINEAR CONSTRAINTS

In this section first we present a translation of disjunctive deductive databases to sets of linear constraints. Then we establish results relating solutions of the linear constraints to Herbrand models of the programs.

2.1 The Constraint Version of Programs

Let $\mathcal{L}$ be a language that contains finitely many constant and predicate symbols, but has no function symbols. Thus the Herbrand based of $\mathcal{L}$, denoted by $B_{\mathcal{L}}$, is finite.

**Definition 2.1.1.** (1) A normal clause is of the form:

$$A \leftarrow L_1 \land \cdots \land L_n,$$

where $A$ is an atom and $L_1, \ldots, L_n$ are literals, not necessarily ground, and $n \geq 0$.

(2) A normal deductive database is a finite set of normal clauses.

Even though we have only defined normal clauses and deductive databases in the preceding, for the purpose of this paper, we in essence have also covered the case of disjunctive clauses and deductive databases. This is because for any disjunctive program there is an easily constructed normal program having the same minimal models, and computing these models is the goal of our computation. For notational ease, we generally use only
normal clauses. Thus, throughout this paper, we do not distinguish between "normal" and "disjunctive."\(^1\)

**Definition 2.1.2.** (1) Let \( C \) be a disjunctive clause. A **ground** instance of \( C \) is any clause obtained by replacing all variables in \( C \) by constants of \( \mathcal{L} \). Different occurrences of the same variable must be replaced by the same constant.

(2) Let \( P \) be a disjunctive deductive database. The ground version of \( P \), denoted by \( \text{grd}(P) \), is the set of all ground instances of clauses in \( P \).

Because \( \mathcal{L} \) does not contain any function symbols, \( \text{grd}(P) \) is a finite set of ground clauses. Moreover, because \( \text{grd}(P) \) does not contain any variable symbols from language \( \mathcal{L} \), when no confusion arises, the expression "variables" will refer to variables appearing in constraints, not to variable symbols of \( \mathcal{L} \). We also adopt the convention that \( Xs \) and \( Ys \), often subscripted, are variables occurring in constraints.

**Definition 2.1.3.** (1) Let \( X \) be a variable. \( X \) is called a **binary** variable if it takes on only the values 0 or 1.

(2) For all \( A \in B_\mathcal{L} \), we let \( X_A \) be a binary variable corresponding to \( A \). The set \( \{X_A \mid A \in B_\mathcal{L}\} \) will be called a **binary variable representation** of \( B_\mathcal{L} \).

The binary variable \( X_A \) will represent the truth value of \( A \). Thus the truth value of \( \bar{A} \), the negation of \( A \), is \( 1 - X_A \). When no confusion will arise we abuse notation by using \( X_A \) as the shorthand representation for \( 1 - X_A \) even though, according to the preceding definition, \( X_A \) is not defined for \( \bar{A} \not\in B_\mathcal{L} \).

**Example 2.1.1.** Consider the following ground clauses defining \( A \):

\[
A \leftarrow L_1 \land L_2
\]

\[
A \leftarrow L_3 \land L_4 \land L_6
\]

From the first clause, \( A \) must be true, if \( L_1 \) and \( L_2 \) are true. Otherwise, \( A \) may or may not be true depending on other clauses. This is the intuition behind the following constraint for clause (1):

\[
X_A \geq X_{L_1} \cdot X_{L_2}.
\]

Similarly, the constraint

\[
X_A \geq X_{L_3} \cdot X_{L_4} \cdot X_{L_6}
\]

corresponds to clause (2).

The following definition formalizes the transformation of clauses in \( \text{grd}(P) \) into constraints.

**Definition 2.1.4.** (1) Let \( P \) be a normal deductive database and \( C = A \leftarrow L_1 \land \cdots \land L_n \) be a clause in \( \text{grd}(P) \). Then the **constraint** version of \( C \),

\(^1\) Note that for the purpose of this paper, negations are considered to be classical. In Bell et al. [1994] we develop a general framework with both classical and nonmonotonic negation in deductive databases.
denoted by \( if(C) \), is the constraint:

\[
X_A \geq \prod_{j=1}^{n} X_{L_j}.
\]

(2) The constraint version of \( P \), denoted by \( if(P) \), is the set of constraint versions of all clauses in \( \text{grd}(P) \).

An important aspect of \( if(P) \) is that it may be very large due to the fact that ground clauses are used to generate it. This issue is addressed in Section 4.4.

It is easy to see that in our translation the order in which literals occur in clause bodies is irrelevant. From the point of view of solving a set of constraints, the order of clauses is also irrelevant. However, in the form presented in Definition 2.1.4, the constraints are nonlinear, and thus are generally more difficult to solve than linear constraints (cf. see Hillier and Lieberman [1974] for a more extensive discussion). To remedy this we use the following lemma. It linearizes the constraints in \( if(P) \).

**Definition 2.1.5.** Let \( S \) be a set/domain. Two sets \( X \) and \( Y \) of constraints are \( S \)-equivalent if and only if every solution of \( X \) that assigns values from \( S \) to the variables of \( X \) is a solution of \( Y \), and vice versa.

**Lemma 2.1.1.** Let \( X, X_1, \ldots, X_n \) be binary variables. The constraint \( X \geq \prod_{i=1}^{n} X_i \) is \( \{0, 1\} \)-equivalent to the constraint: \( X \geq 1 - \sum_{i=1}^{n} (1 - X_i) \).

**Proof Outline.** As each \( X_i \) is a binary variable, the product \( \prod_{i=1}^{n} X_i \) is \( \{0, 1\} \)-equivalent to \( X \geq 0 \). But because \( \prod_{i=1}^{n} X_i = 0 \), there must exist some \( 1 \leq j \leq n \) such that \( X_j = 0 \). In other words, it is true that \( 1 - \sum_{i=1}^{n} (1 - X_i) = 1 - (1 - 0) - \sum_{i=1}^{n} (1 - X_i) \leq 0 \).

Because \( X \) is a binary variable, \( X \geq 1 - \sum_{i=1}^{n} (1 - X_i) \) is \( \{0, 1\} \)-equivalent to \( X \geq 0 \).

**Case 1.** \( \prod_{i=1}^{n} X_i = 0 \).
In other words, \( X \geq \prod_{i=1}^{n} X_i \) is \( \{0, 1\} \)-equivalent to \( X \geq 0 \). But because \( \prod_{i=1}^{n} X_i = 0 \), there must exist some \( 1 \leq j \leq n \) such that \( X_j = 0 \). In other words, it is true that \( 1 - \sum_{i=1}^{n} (1 - X_i) = 1 - (1 - 0) - \sum_{i=1}^{n} (1 - X_i) \leq 0 \).

Because \( X \) is a binary variable, \( X \geq 1 - \sum_{i=1}^{n} (1 - X_i) \) is \( \{0, 1\} \)-equivalent to \( X \geq 1 \).

**Case 2.** \( \prod_{i=1}^{n} X_i = 1 \).
In other words, it is the case that for all \( 1 \leq i \leq n \), \( X_i = 1 \) or \( 1 - X_i = 0 \). Hence, it is necessary that \( X \geq 1 - \sum_{i=1}^{n} (1 - X_i) \) is \( \{0, 1\} \)-equivalent to \( X \geq 1 \).

**2.2 Models Versus Solutions**

We now study the relationship between solutions of \( if(P) \) and Herbrand models of \( P \). When we use the terms “interpretation” and “model,” they are synonymous and refer to Herbrand interpretations and models unless explicitly stated otherwise.

**Definition 2.2.1.** Let \( \{X_A | A \in B_J\} \) be a binary variable representation of \( B_J \).

(1) A binary variable assignment is a mapping \( S: \{X_A | A \in B_J\} \to \{0, 1\} \).
Let $I$ be an interpretation. Define the binary variable assignment $S_I$ corresponding to $I$ as follows:

$$S_I(X_A) = \begin{cases} 1 & \text{if } A \in I, \\ 0 & \text{otherwise}. \end{cases}$$

The following theorem is now easy to establish. It proves a 1–1 correspondence between solutions of $\text{id}(P)$ and models of $P$. In essence, the theorem shows that the procedure of using $\text{id}(P)$ to compute the (Herbrand) models of $P$ is sound and complete.

**Theorem 2.2.1.** Let $P$ be a disjunctive deductive database, $I$ be an interpretation, and $S_I$ be the binary variable assignment corresponding to $I$. $S_I$ is a solution of $\text{id}(P)$ if and only if $I$ is a model of $P$.

**Proof Outline.**

**Case 1.** ("if" part) $I$ is a model of $P$.

Suppose $S_I$ is not a solution of $\text{id}(P)$. Then there exists a constraint $C$ in $\text{id}(P)$ such that $S_I$ does not satisfy. In particular, $C$ is of the form: $X_A \geq \bigwedge_{j=1}^m X_{L_j}$. By Definition 2.1.4, $C$ corresponds to the ground clause $A \leftarrow L_1 \land \cdots \land L_m$ in $\text{grd}(P)$. But because $S_I$ does not satisfy $X_A \geq \bigwedge_{j=1}^m X_{L_j}$, it must be the case that $S_I(X_A) = 0$ and $\bigwedge_{j=1}^m S_I(X_{L_j}) = 1$. Thus it is necessary that for all $1 \leq j \leq m$, $S_I(X_{L_j}) = 1$. Then by Definition 2.2.1, it is true that $A \not\in I$, but for all $1 \leq j \leq m$, $L_j \in I$. Therefore $I$ cannot satisfy $A \leftarrow L_1 \land \cdots \land L_m$, and thus cannot be a model of $\text{grd}(P)$ and $P$, a contradiction!

**Case 2.** ("only-if" part) $S_I$ is a solution of $\text{id}(P)$.

Suppose $I$ is not a model of $P$ and $\text{grd}(P)$. Then there must exist a clause in $\text{grd}(P)$ that $I$ does not satisfy. By an argument similar to the preceding one, $S_I$ cannot satisfy the corresponding constraint in $\text{id}(P)$, a contradiction! Hence it is necessary that $I$ is a model of $P$. □

Deductive databases often have a set of integrity constraints associated with them. For integrity constraints of the form:

$$\leftarrow L_1 \land \cdots \land L_n,$$

they can be translated to constraints in exactly the same way as in Definition 2.1.4. So $\text{id}(P)$ can be used to support integrity checking.

**Corollary 2.2.1.** Let $P$ be a disjunctive deductive database and $IC$ be a set of integrity constraints in the form given previously. Let $\text{id}(IC)$ denote the constraints corresponding to the integrity constraints in $IC$. Then: $P \cup IC$ is consistent if and only if the solution set of $\text{id}(P) \cup \text{id}(IC)$ is non empty.

**Example 2.2.1.** Let $P$ be the following deductive database:

$$A \leftarrow C \land \neg B$$

$$C \leftarrow.$$

Suppose $IC$ is the following set of integrity constraints:

$$\leftarrow A \land C$$

$$\leftarrow B \land C.$$
It is easy to see that \( P \cup IC \) is inconsistent. Corresponding to \( P \cup IC \) is the following set of constraints:

\[
\begin{align*}
X_A + X_B &\geq X_C \\
X_c &\geq 1 \\
X_A + X_C &\leq 1 \\
X_B + X_C &\leq 1.
\end{align*}
\]

No solution exists for this set of constraints.

In this section, we have presented a transformation of a disjunctive program \( P \) into a set \( if(P) \) of linear constraints. We have demonstrated the equivalence between models of \( P \) and solutions of \( if(P) \). Thus among the solutions of \( if(P) \), there is one solution corresponding to each minimal model of \( P \). The question we raise now is how to compute these minimal solutions. In the next section, we first describe how to compute the least models of definite deductive databases. Then in Section 4, we show how to find all the minimal models of general disjunctive deductive databases.

3. COMPUTING LEAST MODELS OF DEFINITE PROGRAMS

In this section we consider the computation of models for a special group of disjunctive deductive databases—namely, the class of definite deductive databases. The main result of this section is Theorem 3.1 which says that the least models of such deductive databases can be computed by solving linear constraints over the domain of the reals, not over the domain of \([0, 1]\). The significance of this theorem is that any linear programming algorithm, such as the simplex or dual-simplex algorithm, can therefore be applied to find least models with at least some efficiency. In the next section, we see how this real-valued computation facilitates the processing of minimal models.

**Definition 3.1.** (1) Let \( \mathbb{R} \) denote the set of all real numbers. A real variable \( Y \) can take on any value in \( \mathbb{R} \).

(2) For all \( A \in B_J \), let \( Y_A \) be a real variable corresponding to the binary variable \( X_A \). The set \( \{ Y_A \mid A \in B_J \} \) is called a real variable representation of \( B_J \).

**Definition 3.2.** Let \( P \) be a definite deductive database, \( \{ X_A \mid A \in B_J \} \) be a binary variable representation of \( B_J \), and \( \{ Y_A \mid A \in B_J \} \) be the corresponding real variable representation of \( B_J \).

(1) Let \( C \) be a constraint in \( if(P) \). Then \( real(C) \) is the constraint obtained by replacing every occurrence of \( X_A \) in \( C \) with \( Y_A \) for all \( A \in B_J \).

(2) Let \( real(P) \) be the following set of constraints:

\[
real(P) = \{ real(C) \mid C \in if(P) \} \cup \{ 0 \leq Y_A \leq 1 \mid A \in B_J \}.
\]

The preceding definition shows how to obtain a set \( real(P) \) of linear constraints involving real variables. This process involves replacing each binary variable with its corresponding real variable. Furthermore, con-
straints ensuring that all real variables can only range from 0 to 1 are added to \( \text{real}(P) \). We adopt the convention of using \( X \)'s to denote binary variables and \( Y \)'s to denote real variables.

Theorem 3.1 shows that the optimal solution of \( \text{real}(P) \) that minimizes \( \sum_{A \in B} Y_A \) is equivalent to the least model of \( P \). See also Chandrasekaran [1984] and Chandru and Hooker [1991] for results that could have been applied here. Our proof requires the following definitions and lemmas.

**Definition 3.3.** Let \( \{Y_A | A \in B_\varphi\} \) be a real variable representation of \( B_\varphi \). A real variable assignment is a mapping \(^2\) \( R : \{Y_A | A \in B_\varphi\} \rightarrow [0, 1] \).

**Definition 3.4.** ([van Emden and Kowalski 1976]) Let \( P \) be a definite deductive database, and \( I \) be a Herbrand interpretation.

1. Define the operator \( T_P \) associated with \( P \) that maps Herbrand interpretations to Herbrand interpretations as follows:
   
   \[
   T_P(I) = \{A \in B_\varphi | A \leftarrow B_1 \land \cdots \land B_n \text{ is in } \text{grd}(P) \text{ and for all } 1 \leq i \leq n, B_i \in I\}.
   \]

2. Define the upward iterative of \( T_P \) as follows:
   
   (i) \( T_P \uparrow 0 = \emptyset \);
   
   (ii) \( T_P \uparrow \alpha = T_P(T_P \uparrow (\alpha - 1)) \) if \( \alpha > 0 \).

**Lemma 3.1.** Let \( P \) be a definite deductive database, \( \{X_A | A \in B_\varphi\} \) be a binary variable representation of \( B_\varphi \), and \( \{Y_A | A \in B_\varphi\} \) be the corresponding real variable representation of \( B_\varphi \). For all \( n \geq 1 \), if \( A \in T_P \uparrow n \), then for all real variable assignments \( R \) satisfying \( \text{real}(P) \), \( R(Y_A) = 1 \).

**Proof Outline.** Proceed by induction on \( n \).

- **Base case.** \( n = 1 \).

Consider any \( A \in T_P \uparrow 1 \). Then by Definition 3.4, \( A \leftarrow \) is a clause in \( \text{grd}(P) \). By Definitions 2.1.4 and 3.2, the constraint \( Y_A \geq 1 \) is in \( \text{real}(P) \). By Definition 3.2 again, the constraint \( 0 \leq Y_A \leq 1 \) is also in \( \text{real}(P) \). Therefore, it is necessary that for all real variable assignments \( R \) satisfying \( \text{real}(P) \), \( R(Y_A) = 1 \).

- **Inductive case.** \( n > 1 \).

Consider any \( A \in T_P \uparrow n \). Then by Definition 3.4, there exists a clause \( A \leftarrow B_1 \land \cdots \land B_k \) in \( \text{grd}(P) \) such that for all \( 1 \leq i \leq k \), \( B_i \in T_P \uparrow (n - 1) \). By Definitions 2.1.4 and 3.2, the constraint \( X_A \geq \Pi_{i=1}^k X_{B_i} \) is in \( \text{if}(P) \). Then by Definition 3.2, the constraint \( Y_A \geq \Pi_{i=1}^k Y_{B_i} \) is in \( \text{real}(P) \). But by the induction hypothesis, because for all \( 1 \leq i \leq k \), \( B_i \in T_P \uparrow (n - 1) \), it is necessary that for all real variable assignments \( R \) satisfying \( \text{real}(P) \), \( R(Y_{B_i}) = 1 \). Thus the constraint can only be satisfied if \( R(Y_A) \geq 1 \). Now by Definition 3.2, the constraint \( 0 \leq Y_A \leq 1 \) is in \( \text{real}(P) \). Therefore in order for \( R \) to satisfy

---

\(^2\) For our purpose it is sufficient to define \( R \) as \( \{Y_A | A \in B_\varphi\} \rightarrow [0, 1] \), because of the constraints \( 0 \leq Y_A \leq 1 \) for \( A \in B_\varphi \) in \( \text{real}(P) \).

implementing deductive databases

real(P), it is necessary that R must satisfy the two constraints to give R(Y_A) = 1. This completes the proof of the induction and the lemma. □

Lemma 3.2 follows immediately from the fact that T_P is monotonic and the Herbrand base is finite.

Lemma 3.2. Let P be a definite deductive database. Then there exists an integer n such that T_P \uparrow n is the least Herbrand model of P.

Combining these two lemmas, it is necessary that for every solution R of real(P), R must assign 1 to every atom in the least model of P. Furthermore, the following theorem shows that the unique optimal solution of real(P) corresponds to the least model of P.

Theorem 3.1. Let P be a definition deductive database, and \{Y_A | A ∈ B_i\} be a real variable representation of B_i. Let I be an interpretation of P, and R_I be the real variable assignment defined as follows: for all A ∈ B_i, R_I(Y_A) is 1 if A ∈ I, 0 otherwise. Then: R_I is the unique optimal solution of real(P) that minimizes \sum_{A ∈ B_i} Y_A if and only if I is the least model of P.

Proof Outline. Case 1. ("if" part) I is the least model of P. From Theorem 2.2.1 it is easy to see that R_I is solution of real(P). Now by Lemma 3.2, there exists an integer n such that T_P \uparrow n = I. Let R' be any real variable assignment satisfying real(P). Then by Lemma 3.1, for all A ∈ I, it follows that R'(Y_A) = 1. Let m be the cardinality of I. In other words, it is necessary that \sum_{A ∈ B_i} R'(Y_A) ≥ m. But given the definition of R_I, it is the case that \sum_{A ∈ B_i} R_I(Y_A) = \sum_{A ∈ I} R_I(X_A) + \sum_{A \notin I} R_I(Y_A) = m. Thus it is necessary that \sum_{A ∈ B_i} R'(Y_A) ≥ \sum_{A ∈ B_i} R_I(Y_A). Hence, R_I is an optimal solution of real(P) that minimizes \sum_{A ∈ B_i} Y_A. Furthermore, recall that for any real variable assignment R' satisfying real(P), R'(Y_A) = 1 = R_I(Y_A) for all A ∈ I. So if R' ≠ R_I, there must exist an A ∉ I such that R'(Y_A) = 1 and R_I(Y_A) = 0. In that case, it is necessary that \sum_{A ∈ B_i} R'(Y_A) > m = \sum_{A ∈ B_i} R_I(Y_A). Hence R_I is the unique optimal solution.

Case 2. ("only-if" part) R_I is the unique optimal solution of real(P) that minimizes \sum_{A ∈ B_i} Y_A. Suppose I is not the least model of P. Let I' be the least model of P. Because I is a model from Theorem 2.2.1, it is necessary that I' ⊆ I. But based on the argument presented in Case 1, it is necessary that R_I is not the unique optimal solution of real(P), a contradiction! □

The preceding theorem shows that the least model of a definite deductive database P can be computed by minimizing the real constraints in real(P). In other words, instead of solving an integer linear program, it suffices to solve the corresponding linear program containing real variables. This is computationally desirable because linear programs with real variables can be solved more easily than linear programs with integer variables.

4. Computing Minimal Models of Disjunctive Programs
Thus far we have shown how to compute the least models of definite deductive databases. Here we extend this to computing the minimal models of
disjunctive programs that are not definite. Recall from Section 2 that we have established the equivalence between models of a disjunctive program $P$ and solutions of $\text{if}(P)$. But unlike the situation for a definition program, in which one objective function can be established to compute the unique least model, in this more general case it is unclear how all minimal models of $P$ can be computed by a single optimization of $\text{if}(P)$. Because of this, we first introduce the notion of card-minimal models and show how these models can be computed. Then, based on card-minimal models, we present an iterative algorithm that computes all minimal models.

4.1 Computing Card-Minimal Models

**Definition 4.1.1.** Let $P$ be a disjunctive deductive database, and $M$ be a model of $P$.

1. $M$ is a card-minimal model of $P$ if there is no model $M'$ of $P$ such that $\text{cardinality}(M') < \text{cardinality}(M)$.
2. $M$ is an $\subseteq$-minimal model if there exists no model $M'$ of $P$ such that $M' \subseteq M$.

A card-minimal model is a model of least possible cardinality. It is obvious that a card-minimal model is a $\subseteq$-minimal model. The converse, however, is false. When no confusion arises, because a $\subseteq$-minimal model is a minimal model in the usual sense, we refer to a $\subseteq$-minimal model simply as a minimal model.

Recall, from Theorem 2.2.1, that solutions of $\text{if}(P)$ and models of $P$ are equivalent. Furthermore, Theorem 4.1.1 shows that an optimal solution of $\text{if}(P)$ that minimizes $\Sigma_A \in B, X_A$ corresponds to a card-minimal model of $P$.

**Theorem 4.1.1.** Let $P$ be a disjunctive deductive database, $M$ be an interpretation of $P$, and $S_M$ be the binary variable assignment corresponding to $M$ as defined in Definition 2.2.1. Then: $S_M$ is an optimal solution of $\text{if}(P)$ that minimizes $\Sigma_A \in B, X_A$ if and only if $M$ is a card-minimal model of $P$.

**Proof Outline.** Case 1. ("if" part) $M$ is a card-minimal model of $P$.

By Theorem 2.2.1, because $M$ is a model of $P$, $S_M$ is a solution to $\text{if}(P)$. Suppose $S_M$ is not an optimal solution that minimizes $\Sigma_A \in B, X_A$. Thus there exists a solution $S'$ of $\text{if}(P)$ such that $\Sigma_A \in B, S'(X_A) < \Sigma_A \in B, S_M(X_A)$. Consider the following interpretation $M'$: for all $A \in B, A \in ' if and only if $S'(X_A) = 1$, and $A \not\in M'$ otherwise. By Theorem 2.2.1, because $S'$ is a solution of $\text{if}(P)$, $M'$ is a model of $P$. But based on the definitions of $S_M$, $S'$, and $M'$, it is necessary that the cardinality of $M'$ is strictly less than that of $M$, a contradiction! Hence $S_M$ is an optimal solution of $\text{if}(P)$ that minimizes $\Sigma_A \in B, X_A$.

Case 2. ("only-if" part) $S_M$ is an optimal solution of $\text{if}(P)$ that minimizes $\Sigma_A \in B, X_A$.

By Theorem 2.2.1, because $S_M$ is a solution of $\text{if}(P)$, $M$ is a model of $P$. Suppose $M$ is not a card-minimal model of $P$. Then there exists a model $M'$ of $P$ such that the cardinality of $M'$ is strictly less than the cardinality of $M$. Consider the binary variable assignment $S_{M'}$ corresponding to $M'$ as defined
Implementing Deductive Databases.

By Theorem 2.2.1, because \( M' \) is a model of \( P \), \( S_{M'} \) is a solution of \( \text{if}(P) \). But based on the definition of \( S_{M'} \), it is necessary that \( \sum_{A \in B} S_{M'}(X_A) \leq \sum_{A \in B} S_{M}(X_A) \). Thus it follows that \( S_{M'} \) is not an optimal solution that minimizes \( \sum_{A \in B} X_A \), a contradiction! Hence \( M \) is a card-

Example 4.1.1. Let \( P \) be the following set of clauses:

\[
\begin{align*}
A &\leftarrow \neg B \land \neg C \land \neg D \\
B &\leftarrow \neg A \land \neg C \land \neg D \\
C &\leftarrow D \\
D &\leftarrow C.
\end{align*}
\]

There are three minimal models: \( \{A\} \), \( \{B\} \), and \( \{C, D\} \). It is straightforward to verify that the objective function \( \min(X_A + X_B + X_C + X_D) \) yields optimal solutions: (i) \( X_A = 1 \) and \( X_B = X_C = X_D = 0 \), and (ii) \( X_B = 1 \) and \( X_A = X_C = X_D = 0 \). These two solutions correspond to the two card-minimal models of the given deductive database.

Theorem 4.1.1 shows that the objective function \( \min \sum_{A \in B} X_A \) computes the card-minimal models of \( P \). But, as illustrated by the preceding example, a disjunctive deductive database may have \( \subseteq \)-minimal models of many different cardinalities. \( \subseteq \)-minimal models are defined in terms of set inclusions, not in terms of set cardinalities. It is unclear how to set up an objective function so that all minimal models can be computed in a single optimization. Notwithstanding, in the second half of this section, we present an algorithm that computes all minimal models based on the algorithm described here. But before we do so, let us first consider the efficiency of computing card-minimal models.

In Section 3, we have shown that to compute the least model of a definite deductive database \( P \), it suffices to compute the optimal solution of \( \text{real}(P) \) that minimizes \( \sum_{A \in B} Y_A \), thus avoiding solving an integer linear program. We can ask if one wants to compute card-minimal models, whether it is always sufficient to use \( \text{real}(P) \) in the place of \( \text{if}(P) \), that is, replacing all integrality constraints of the form \( X_A = 0 \lor 1 \) with those of the form \( 0 \leq Y_A \leq 1 \). The answer is surely positive if \( P \) is a definite deductive database, because the card-minimal model is just the least model. However, as shown by the following example, \( \text{real}(P) \) cannot replace \( \text{if}(P) \), if \( P \) is a disjunctive deductive database.

Example 4.1.2. Let \( P \) be the following deductive database:

\[
\begin{align*}
A &\leftarrow \neg B \\
A &\leftarrow \neg C \\
B &\leftarrow \neg C.
\end{align*}
\]

Then, according to Definition 2.1.4, \( \text{if}(P) \) consists of the following constraints:

\[
\begin{align*}
X_A + X &\geq 1 \\
X_A + X_C &\geq 1 \\
X_B + X_C &\geq 1.
\end{align*}
\]
It is easy to check that there are three optimal solutions that minimize \( (X_A + X_B + X_C) \): (i) \( X_A = X_B = 1, X_C = 0 \), (ii) \( X_A = X_C = 1, X_B = 0 \), and (iii) \( X_A = X_B = 1, X_C = 0 \). In each of these cases, the optimal value of \( (X_A + X_B + X_C) \) is 2. But now suppose the integrality constraints on \( X_A, X_B, X_C \) are dropped, so that they can take on any value between 0 and 1. Then it is easy to see that \( X_A = X_B = X_C = 0.5 \) is a feasible solution that gives \( (X_A + X_B + X_C) = 1.5 \). Hence dropping the integrality constraints does not always work.

The preceding example shows that for disjunctive deductive databases, we cannot easily avoid dealing with integer linear programs. Fortunately, solving integer linear programs is a well-developed area in Operations Research. Two common and efficient techniques are branch-and-bound and cutting-plane. Both these methods are based on solving the given integer linear program as one with real variables. For our prototype implementation described in the next section, we adopt the cutting-plane technique which can be outlined as follows.\(^3\)

**Algorithm 1** (Cutting-plane [Gillett 1976; Hillier and Lieberman 1974]). Let \( LP \) be a linear program with integrality constraints, that is, constraints stipulating that every variable can only assume integral values.

1. Ignore the integrality constraints of \( LP \) and solve the corresponding linear program using the simplex method.
2. If the solution from Step 1 is an all-integer solution, the program is solved.
3. Otherwise, add a new constraint (cut) to the problem and return to Step 1.

Intuitively, Step 3 of Algorithm 1 adds a new constraint that prunes away some real-valued solutions. The precise constraint that is added differs from algorithm to algorithm, but the best known such technique is the Gomory technique [Gomory 1965]. We omit discussion of this technique here.

### 4.2 Computing Minimal Models

Theorem 4.1.1 shows that card-minimal models of a disjunctive program \( P \) can be computed by minimizing \( \sum_{A \in B_j} X_A \) subject to \( if(P) \). The following algorithm makes use of this result to compute all minimal models of \( P \) in an iterative fashion.

**Algorithm 2** (Minimal Models of \( P \)). Let \( P \) be a disjunctive deductive database and \( if(P) \) be constructed as described in Definition 2.1.4. In the following, \( S \) is intended to contain all the minimal models of \( P \) and \( AC \) is a set of addition constraints.

1. Set \( S \) and \( AC \) to \( \emptyset \).
2. Use Algorithm 1 to solve the integer linear program minimizing \( \sum_{A \in B_j} X_A \) subject to \( if(P) \cup AC \).

\(^3\) Note that there exist algorithms more efficient than the cutting-plane and the branch-and-bound algorithms. But for quick prototyping, we opted for the cutting-plane algorithm because of its simplicity. The focus here is to show that the mixed integer programming approach is feasible and promising. As for further improving the performance of our system, such as by incorporating more efficient algorithms, it is a task that we will undertake in future research.
3. If no (optimal) solution can be found, halt and return $S$ as the set of minimal models.
4. Otherwise, let $M$ be the model corresponding to the optimal solution found in Step 2. Add $M$ to $S$.
5. Add the constraint $\sum_{A \in M} X_A \leq (k - 1)$ to $AC$, where $k$ is the cardinality of $M$. Then go to step 2.

Example 4.2.1. Consider the program $P$ listed in Example 4.1.1. After simplification, $if(P)$ effectively consists of the following constraints:

$$X_A \geq 1 - X_B - X_C - X_D$$
$$X_D \geq X_C$$
$$X_C \geq X_D.$$ 

The first optimal solution obtained by performing Step 4 of Algorithm 2 corresponds to the model $\{A\}$. It is easy to verify that this is indeed a minimal model of $P$. Now based on this model, the following constraint is added to $AC$ in Step 5:

$$X_A \leq 0.$$ 

Thus in solving $lc(P) \cup AC$ in Step 2 in the next iteration, every solution must satisfy this additional constraint. This translates to the fact that no model obtained in this step can be a superset of $\{A\}$. In particular, the next optimal solution corresponds to $\{B\}$, for which the following constraint is added to $AC$:

$$X_B \leq 0.$$ 

Similarly, the model obtained in Step 4 of the next iteration cannot be a superset of $\{A\}$ and $\{B\}$. In this case, it is $\{C, D\}$. After adding the constraint:

$$X_C + X_D \leq 1,$$

in Step 5, the algorithm halts in Step 3 in the next iteration.

The following lemma shows that Algorithm 2 is sound and complete in computing all minimal models of $P$.

**Lemma 4.2.1.** Let $P$ be a disjunctive deductive database, and $S$ be computed as described in Algorithm 2. $S$ is the set of all minimal models of $P$.

**Proof Outline.** Observe that in each iteration of Algorithm 2, because of the constraints added in Step 5 in the previous iterations, the model $M$, if any, obtained in Step 4: (i) must not be supersets of any one of the models already added to $S$, and (ii) must be card-minimal among all models satisfying (i). Thus by a straightforward induction on the models added to $S$, the lemma can be established. □
algorithms. More details are given in the next section, which discusses the implementation. In addition, because all the minimal models of $P$ are computed, the Generalized Closed World Assumption of $P$, denoted as $GCWA(P)$, can also be obtained readily [Minker 1982]. This computation is described in detail in the next section.

4.3 Purging Extraneous Constraints and Variables

A weakness of the framework presented thus far is the large number of constraints and variables in $if(P)$, even when $P$ may contain only a few nonground clauses. This is due primarily to the fact that $if(P)$ corresponds to the ground version of program $P$. In particular, a large number of constant symbols can give rise to a large number of ground atoms in the Herbrand universe, and thus a large number of ground clauses and constraints. To ameliorate the situation, we designed an optimization method that purges “extraneous” constraints and variables. Before presenting the details of the optimization, we first illustrate the method with an example.

Example 4.3.1. Let $P$ be the following set of ground clauses:

$$
\begin{align*}
A & \leftarrow B \land \neg D \\
A & \leftarrow \\
B & \leftarrow C \\
B & \leftarrow E \\
E & \leftarrow C \land \neg A.
\end{align*}
$$

$C$ is the only atom that does not appear in the head of any clause and does not appear negated in the body of any clause. (Note that $D$ occurs negatively in the body of Clause (1). But as far as computation of minimal models is concerned, $D$ in effect appears in the head of (1).) It is then easy to see that $C$ cannot be in any minimal model of $P$. Thus Clauses (3) and (5) are never useful, and can therefore be thrown away. Similarly, $C$ can be discarded as well, reducing $P$ to Clauses (1), (2), and (4). We can repeat the same process again. However, instead of searching for atoms that do not appear in the heads of (remaining) clauses, it is generally more efficient to make use of clauses that have just been deleted. In our example, candidate atoms are $A$, $B$, and $E$, as they either occur in the heads or negatively in the bodies of Clauses (3) and (5). On further checking the candidate atoms, $E$ is the one that is dispensable, and Clause (4) can be removed. Finally, in the next iteration, $B$, Clause (1), and thus $D$ can be discarded. $A$ and Clause (2) are, respectively, the only variable and clause to remain intact.

Note that the preceding deductive database is not stratified, and one may wonder whether this optimization technique of purging constraints and variables is closely related to a stratification of atoms in the deductive database. However, we believe that the technique is independent of stratification. For instance, we can modify the preceding program $P$ to $Q$ by simply deleting all
negation operators, for example, Clause (1) becomes \( A \leftarrow B \land D \). Thus \( Q \) is a definite program. It is easy to verify that the same optimization technique can be applied to \( Q \) to delete all clauses except for Clause (2).

Strictly speaking, inasmuch as our optimization algorithm operates on \( \text{if}(P) \), we should present the previous example in terms of constraints and variables. But because of the 1–1 correspondences between constraints and ground clauses, and between variables and ground atoms, here we choose to present the example in the form of ground clauses and atoms for ease of understanding. The same comment applies to the description of the following optimization algorithm.

**Optimization 1 (Purging extraneous constraints and variables).** Let \( P \) be a ground disjunctive deductive database. There are two input parameters to this algorithm. \( N \) specifies the maximum number of times the algorithm iterates (\( N \) is set, a priori, to some integer), and \( S \) initially consists of atoms that neither occur in the head of any clause nor occur negatively in the bodies of any clauses. In the following, \( Q \) is intended to consist of clauses from \( P \) that remain after the optimization.

1. Set \( Q \) to \( P \) and \( i \) to 0.
2. If \( i \geq N \), halt and return \( Q \) as the optimized version of \( P \).
3. Set \( R \) to \( \emptyset \).
4. For all atoms \( A \) in set \( S \), and for all clauses \( C \) in which \( A \) occurs (positively in the body of \( C \)), perform the following steps.
   a. Add the head of clause \( C \) and all atoms that occur negatively in the body of \( C \) to \( R \).
   b. Discard \( C \) from \( Q \).
5. Set \( S \) to \( \emptyset \).
6. For all atoms \( A \) in set \( R \), if \( A \) does not occur in the head of any clause nor does it occur negatively in the body of any clause, add \( A \) to \( S \).
7. If \( S \) is empty, halt and return \( Q \) as the optimized version of \( P \).
8. Increment \( i \) and go to Step 2.

Note that in order to compute the input parameter \( S \), instead of searching through the entire program, we can obtain it when the program is read. As for the other parameter \( N \), it is meant to be set by the program. The optimization algorithm described can potentially go on and on, therefore the purpose of \( N \) is to limit the amount of time spent on such optimization. In Section 6, we present experimental results on the effectiveness of this optimization algorithm and the choice of \( N \). Step 4(b) of the algorithm is executed on each iteration, thus at least one clause is discarded on each iteration. It might well be preferable in practice to set the value of \( N \) to be greater than or equal to the original number of clauses in \( P \). In this case, the algorithm will terminate only when no more clauses can be discarded. If the value of \( N \) is set sufficiently small that the algorithm halts when it would be possible to
discard more clauses and variables, then we suspect that additional time spent on further iterations of Optimization 1 would be more than paid back by savings in time spent on linear programming computations. The following lemma shows that the optimization preserves minimal models.

**Lemma 4.3.1.** Let $P$ be a disjunctive deductive database and $Q$ be obtained after $n$ iterations of Optimization 1. Then for all $n \geq 0$, the minimal models of $P$ and $Q$ are identical.

**Proof Outline.** First observe that in each iteration of Optimization 1, for any atoms $A$ in $S$, $A$ does not occur in the heads or negatively in the bodies of any clauses. Further observe that for any such atoms $A$, it cannot be contained in any minimal models of $P$. Let a program $P'$ be obtained by deleting all clauses in which $A$ occurs (positively in the bodies). Then it is obvious that an interpretation $I$ is a minimal model of $P'$ if and only if $I$ is a minimal model of $P$. Based on these observations, the lemma can be established by a simple induction on $n$. □

Purging extraneous constraints and variables is also advocated by Hooker [1988, p. 54]. Our procedure Optimization 1 corresponds to his “monotone variable fixing subroutine” for those variables that appear “negated everywhere.” Because we seek card-minimal models, it is inappropriate to use the monotone variable fixing subroutine for those variables that appear “un-negated everywhere.” Hooker also recommends a “chaining subroutine.” We have not implemented this in our current research. Optimization 1 could be modified to include chaining without altering the validity of the preceding lemma. In future research we will investigate whether the further reduction in the number of variables and constraints in the resulting linear program justifies the computation effort of the chaining subroutine.

### 4.4 Discussion: the Non-Ground Case

A valid critique of the work presented thus far is that it applies to ground programs. This restriction is necessary because mixed integer programming techniques do not have an operation analogous to unification in theorem proving. This is a drawback because the Herbrand base of a deductive database can be enormous, and the ground instantiation of a deductive database may be significantly “larger” than the original program. However, in Kagan et al. [1993], we have developed techniques that, given a definite (i.e., negation-free) program $P$, instantiate $P$ on an “instantiate by need” basis so that the set of atomic logical consequences of the nonground program $P$ can be computed.

In a nutshell, this partial instantiation method for evaluating logic programs proceeds as follows. First, a (nonground) definite deductive database $P$ is treated as if it were a propositional deductive database $P^*$ (i.e., an atom $A$ occurring in $P$ is considered to be a proposition $\rho_A$). Program $P^*$ may then be evaluated using the method presented in earlier sections. Assignments of true/false to different propositions $\rho_A$ and $\rho_B$ in $P^*$ may lead to “conflicts” when $A$ and $B$ are unifiable, but $\rho_A$ and $\rho_B$ are assigned different truth

values. If there are no such conflicts, then we are done. When such conflicts are present, Kagan et al. [1994a] articulates a precise strategy for removing such conflicts and shows that this strategy of Evaluate Propositional Program → Identify Conflicts → Partially Instantiate yields a soundness and completeness theorem for the computation of answer substitutions [Lloyd 1987].

In two separate attempts [Kagan et al. 1994a, 1994b; Gottlob et al. 1993], this partial instantiation strategy has been generalized from the computation of least models of definite programs to that of minimal, well-founded, and stable models. In view of these new results on partial instantiation, one may wonder whether the work presented here has been “subsumed.” The answer is certainly no, because all these partial instantiation methods do not know how to compute least models and minimal models (and stable and well-founded models) of propositional programs. Rather, they only know how to instantiate nonground programs on a “need-to-instantiate” basis, and simply feed the instantiated programs to the propositional methods, such as the ones presented in this paper. Consequently, the algorithms and optimization strategies developed so far are complementary with the partial instantiation strategies, and they can be used in conjunction to yield computational paradigms for nonground deductive databases.

Summarizing, in this section we have studied the notion of card-minimal models that can be obtained by minimizing if(P). Then based on this idea, we have presented an iterative algorithm that computes all minimal models of P. Furthermore, we have described an optimization algorithm that purges extraneous variables and constraints to reduce the size of if(P). In the next section, we describe how these algorithms can be implemented and supported in a system based on linear programming compilation.

5. DESIGN AND IMPLEMENTATION DETAILS
In this section, we first present an overview of an architecture for a query processing system that is based on mixed integer programming compilation. Then we describe in greater detail our prototype compiler which implements Algorithms 1, 2, and Optimization 1. Finally, we discuss how a relational database management system can be used to provide efficient run-time query processing.

5.1 Overview of a Complete System Architecture
A query processing system based on mixed integer programming compilation consists of three modules: the compilation, the run-time, and the update modules. Central to the compilation module is the minimal model solver which takes a linear program as input and produces all the minimal models of the disjunctive program corresponding to the linear program. The minimal model solver essentially performs Algorithms 1, 2, and Optimization 1. More details about our prototype compiler are given in Section 5.2. Furthermore, the compilation module includes a preprocessor. Its primary function is to
translate a given disjunctive program into a linear program, based on the approach described in Section 2.

The run-time module of a query processing system based on mixed integer programming compilation is a standard relational database management system. Section 5.3 gives more details on how minimal models are stored and used to answer queries. There are several reasons for including a relational database management system in the run-time module. The most obvious one is the efficiency of run-time performance. Answering a query now amounts to performing select, project and join operations, and data retrieval. Such processing can be further facilitated by using indexing and other optimization techniques well studied in the database community. Secondly, most relational database query languages such as SQL support aggregate operations such as COUNT and SUM, and other practical operations such as ORDER-BY and GROUP-BY [Unman 1988]. Thus an expressive language for user querying is supported. Thirdly, a relational database management system in the run-time module readily provides many important and practical facilitates such as concurrency control in a multi-user environment, security and access right management, backup, and recovery.

Just as view management is important in relational databases [Blakeley et al. 1986; Roussopoulos 1991], it is important to support incremental updating of least and minimal models. Thus, in addition to the compilation and run-time modules, there is an update module that takes as inputs added, deleted, and changed clauses of a program whose least and minimal models have been computed. This module reuses the previous computations to generate the new models to reflect the updates. See Gillett [1976] and Hillier and Lieberman [1974] for a detailed description on how linear programming techniques allow reoptimization. Basically, to add, or delete, a constraint from an already optimized linear program, it suffices to iterate to a new optimal solution without starting from scratch.

As far as computation of least and minimal models is concerned, what is required here is for the compilation module to store the optimal linear programming tableaux that correspond to the computed models. However, this task is complicated by the application of Optimization 1 described in Section 4.3. For instance, if \( A \leftarrow B \) is a clause in the program such that there is no clause with \( B \) occurring in the head, or such that \( \neg B \) occurs in the body, then Optimization 1 would throw this clause away. Later, if the fact \( B \leftarrow \) is added, the clause \( A \leftarrow B \) would no longer be available to conclude \( A \). Thus, to provide for updates, Optimization 1 should be modified not to delete constraints outright, but simply to move extraneous constraints to an “auxiliary” area that mixed integer programming solvers do not consider. When updates are performed, clauses in the auxiliary area must be consulted, and applicable clauses must be re-inserted. In our example, when \( B \leftarrow \) is added, we re-insert \( A \leftarrow B \), which in turn may cause other clauses to be re-inserted.

As there remain many issues to be addressed before a full update module can be built, the construction of such a module is beyond the scope of this paper. However, for the interested reader, it is worth pointing out that new development in this direction is reported recently in Gupta et al. [1993].

5.2 Prototype Compiler

So far in this section, we have described the various modules of a query processing system based on mixed integer programming compilation. Because the efficiency of the compilation module is our utmost concern at this stage, we have implemented a prototype compiler to evaluate its performance. Other components mentioned will be implemented and integrated in the future.

The prototype compiler implemented is one based on a single processor, not the multiprocessor version previously discussed. It is written in C running under the UNIX environment. It has roughly 700 lines of C code, implementing Algorithms 1, 2, and Optimization 1. At this stage, inasmuch as the complete preprocessor that translates a disjunctive program to a linear program has not been implemented, the prototype compiler takes as inputs ground clauses in which literals are represented by their indices in a fixed enumeration of the Herbrand base. In other words, our prototype compiler actually performs a portion of the preprocessing described before. Finally, for more straightforward prototyping, the tableaux for the linear programming algorithms are represented in array format. Thus, due to the limited amount of space in our environment, 900 is the maximum number of variables allowed by our prototype. In ongoing research, we plan to change our tableau representation so that the maximum can be raised to accommodate large scale experimentation.

Due to the widespread usage of linear programs, linear programming solvers are easily accessible. Some examples are the MAPLE program running under Sun-3 systems and the LINDO program running under IBM PCs. Both programs run on other machines as well. Nevertheless, we decided to implement our own versions of the simplex method and cutting-plane procedures based on the versions presented in Gillett [1976]. By doing so, we gain flexibility in implementing Algorithm 2. Recall that Algorithm 2 requires adding a constraint in Step 5 and re-optimizes in Step 2 in the next iteration. Because such re-optimization is more efficiently performed by the dual-simplex method than by the simplex method, we also implement the dual-simplex method to speed up the processing of the algorithm. In Section 6, we present some experimental results on the performance of this prototype compiler.

5.3 Storage and Access of Minimal Models

Using the semantics of the Generalized Closed World Assumption defined by Minker [1982], a positive literal is true if it is true in all minimal models, and a negative literal is true if it is false in all minimal models. Thus it suffices to just store these literals. But there are two reasons why we prefer an explicit storage of all minimal models. First of all, an explicit generation of GCWA(P) may be very demanding in terms of time and space. More importantly, explicit storage of all minimal models facilitates updates to program P, and thus models of P and GCWA(P).

Under the explicit storage approach, an n-ary atom $p(a_1,\ldots,a_n)$ is stored in a relation $p$ whose schema is: $p$(tupleid, field_1,\ldots, field_n). Models of
deductive databases are stored in the following relations: model(number, tupleid), and database(name, modelnumber). Given this way of storing minimal models, updating models can be handled easily. Note that maintaining unique tuple identifiers across multiple relations is in general a costly operation. However, in our environment, when a user wishes to make insertions of deletions, the update would be implemented by our update modules. Physical insertions and updates of tuples are carried out by a controlled update module, at the request of, but not physically by an arbitrary user, therefore maintaining unique identifiers is not a problem.

Positive literals that are true in all minimal models can be retrieved by straightforward SQL queries which primarily involve a division of the relation model by the relation database. As for providing indefinite answers, there are two kinds. The first kind is possible answers. Positive literals that are true in some minimal models can be obtained by a simple selection on the relation model. Another kind of indefinite answer is disjunctive answers. These are disjunctions of atoms true in all minimal models of a database. But to support these answers, the first issue to be addressed is the representation of such answers in relational query interface. In other words, we must decide in what form an answer $A \lor B$ is returned as part of the result of an SQL query. For now, our approach is to use Cartesian product to represent and generate these disjunctions. In future research, we are interested in finding ways of enhancing our support for disjunctive answers.

To provide answering support for negative literals, we rely on the Generalized Closed World Assumption.

Definition 5.3.1. Let $p$ be a predicate symbol. The Generalized Closed World Assumption Set of $p$, denote by GCS($p$), is the set of all atoms with predicate symbol $p$ that are false under the Assumption, that is, $GCS(p) = HB(p) – MM(p)$, where $HB(p)$ is the set of all atoms in the Herbrand base that have predicate symbol $p$, and $MM(p)$ is the set of atoms that have predicate symbol $p$ and are contained in some minimal model.

As indicated, two components must exist to compute GCS($p$). The first one is the set $MM(p)$ which is easy to compute, as discussed. A view, materialized or not, can be defined to facilitate the processing. In supporting GCS($p$), the problem of how exactly to represent $HB(p)$ needs to be addressed. A direct materialization of $HB(p)$ will in most cases be very space consuming. One alternative is to enumerate $HB(p)$ at model storage time, and materialize GCS($p$) directly. Another option is to restrict queries to allowed ones; that is, every variable in a query must occur in a positive literal in the query [Lloyd 1987]. In this case, an allowed query can be answered using $MM(p)$, without directly requiring GCS($p$). Another issue to address is how to specify a negative literal in a relational query. One way is to specify the negative literal using the Generalized Closed World Assumption form of the predicate. For instance, if the query to be asked is $p(X) \land \neg q(X)$, then the transformed query is $p(X) \land GCSq(X)$ where $GCSq$ is the relation or view consisting of elements in the set GCS($q$). In ongoing research, we are studying how to provide more effective support for querying based on GCS($p$).
6. EXPERIMENTAL RESULTS AND COMPARISONS

We have conducted a number of experiments testing the effectiveness and efficiency of our prototype compiler described in Section 5.2. First of all, we have experimented with the programs considered in the literature (e.g., van Gelder [1989]). These include definite, stratified, locally stratified, as well as nonlocally stratified programs. Our prototype compiler handles all those programs correctly, and given the small sizes of those programs, our compiler finishes all computations in less than one second. To further evaluate our approach, we have tested with three "larger" experiments, the results of each of which are reported in detail in this section.

Unless otherwise stated, the computation times of our prototype compiler presented in the following include all computations. More precisely, the figures are the total time taken to: read a (ground) program, set up the simplex tableau, perform Optimization 1 if used, compute all minimal models, find out that no more minimal models exist, and output the results. Moreover, the figures are averages of running exactly the same experiments 10 times using a DEC-5000 workstation.

To facilitate comparison of our framework with existing systems, we include, whenever possible, run-time figures using SICStus-Prolog. As we shall see, because some of the experiments used contain cycles (through negations), Prolog loops forever in those cases. Thus, the following figures presented for Prolog include only the times when Prolog terminates. Note that in many ways comparing our prototype with Prolog is not appropriate and fair. For instance, on one hand, Prolog can handle function symbols, whereas our system cannot. However, our prototype computes all minimal models, and once these models are stored, repeated querying costs little time. Thus, to set up a closer comparison, we make our Prolog programs compute all answers — by using `setof` or `bagof` on all predicates. More specifically, the figures for Prolog are the total time taken to `consult` or to `compile` a program, and to compute all answers of all predicates. Again we reiterate that Prolog figures are included for reference only.

6.1 Effectiveness of Optimization 1

The first experiment is an animal deductive database, listed in the Appendix. It consists of 32 predicates (30 unary and 2 binary), 40 clauses, and 15 constant symbols, giving rise to a Herbrand base of size 900. The dependency graph of this program is rather complex. In addition, this program has 8 minimal models. Table I shows the results of our compiler, executing Optimization 1 from 0 to 4 iterations (i.e., the variable $N$ described in Optimization 1 is varied from 0 to 4).

<table>
<thead>
<tr>
<th>Iterations</th>
<th>$N = 0$</th>
<th>$N = 1$</th>
<th>$N = 2$</th>
<th>$N = 3$</th>
<th>$N = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (secs)</td>
<td>19</td>
<td>2</td>
<td>1</td>
<td>0.85</td>
<td>0.78</td>
</tr>
</tbody>
</table>

4 The UNIX utility program `profile` is used to record computation time.
5 The Prolog predicate `statistics` is used to report the CPU time taken for Prolog computations.
In this particular example, Optimization 1 requires only four iterations to reach a minimal set of constraints and variables. It is clear from Table I that Optimization 1 is effective in purging extraneous constraints and variables.

As for Prolog, it takes 25 seconds to consult the program and to compute the bagof of all predicates. Note that this figure does not include the time taken for computing the “looping” predicates largemouth and smallmouth. Moreover, if compile is used instead of consult, the time taken is over 30 seconds.

6.2 Effectiveness of Algorithm 2

To examine the effectiveness of our iterative algorithm for computing minimal models, we record for each minimal model the time taken and the number of steps used in the simplex and the dual-simplex methods\(^6\). The following table shows the average figures for each model, with Optimization 1 iterated 4 times (i.e., when \(N\) is set to 4).

<table>
<thead>
<tr>
<th>Minimal Model</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>More</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (secs)</td>
<td>0.42</td>
<td>0.04</td>
<td>0.07</td>
<td>0.03</td>
<td>0.07</td>
<td>0.03</td>
<td>0.05</td>
<td>0.03</td>
<td>0.4</td>
</tr>
<tr>
<td>Steps (simplex/dual-simplex)</td>
<td>33</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

It is clear from the table that after the first minimal model is obtained, subsequent minimal models can be computed efficiently. Moreover, as the last column of the table indicates, the fact that no more minimal models exist can be detected rapidly as well. These favorable results are due to the efficient reoptimization facility provided by the simplex and the dual-simplex methods, and our iterative algorithm takes full advantage of these facilities.

6.3 Varying the Number of Constant Symbols and Facts

Recall that the ground version of a deductive database determines directly the size of the corresponding linear program. The number of constant symbols in the database in turn determines the size of the ground version. Thus in the following we examine the effect of increasing the number of constant symbols on the performance of our prototype compiler. This time the experiment is based on the win–move program discussed in van Gelder [1989]. This program contains only one nonground clause:

\[
\text{wins}(X) \leftarrow \text{move}(X, Y) \land \neg \text{wins}(Y),
\]

giving rise to a nonlocally stratified program. It also contains facts on the move predicate. Thus, if \(n\) is the number of constant symbols appearing in

\(^6\) Recall that apart from the simplex and dual-simplex methods, the cutting-plane algorithm is also invoked to compute the minimal models. However, here we only show the numbers of steps executed in the simplex and dual-simplex methods, because these numbers are more representative and illustrative of the time taken to compute each minimal model.

the facts, there are \((n^2 + n)\) elements in the Herbrand base. To ensure a fair comparison as \(n\) increases, we fix the number of randomly generated facts to be \(n\) as well, giving rise to \((n^2 + n)\) ground clauses and constraints. The following table describes the performance results as \(n\) varies.

<table>
<thead>
<tr>
<th>Number of Constants</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Herbrand Base</td>
<td>6</td>
<td>20</td>
<td>72</td>
<td>156</td>
</tr>
<tr>
<td>Time without Optimization 1 (secs)</td>
<td>0.031</td>
<td>0.088</td>
<td>0.794</td>
<td>3.782</td>
</tr>
<tr>
<td>Time with Optimization 1 fully applied (secs)</td>
<td>0.027</td>
<td>0.042</td>
<td>0.116</td>
<td>0.151</td>
</tr>
</tbody>
</table>

As shown by the second row of the table without Optimization 1 to purge extraneous constraints and variables, increasing the number of constant symbols degrades and performance of the prototype quite drastically. But once extraneous constraints are purged, the performance becomes more directly dependent on the actual number of facts in the database.

As for Prolog, the figures are very similar to those obtained by running our prototype with Optimization 1. But because of the “looping” nature of the predicate \(\text{win}\), the figures for Prolog may not be too meaningful, as Prolog does not terminate in some cases. For the same reason, Prolog figures are not included for the experiment mentioned in the following.

The previous experiment indicates that the performance of our prototype changes according to the number of facts in the database. To further examine this phenomenon, we use the \(\text{win–move}\) example with varying number of randomly generated facts, while keeping the number of constant symbols fixed at 8. The table below summarizes the results obtained with Optimization 1.

<table>
<thead>
<tr>
<th>Number of Facts</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (secs)</td>
<td>0.081</td>
<td>0.128</td>
<td>0.203</td>
<td>0.229</td>
<td>0.272</td>
<td>0.378</td>
<td>0.489</td>
<td>0.685</td>
</tr>
</tbody>
</table>

The figures listed in Table IV show that as the number of facts increases, our prototype takes more time to perform all computations. But the increase in facts does not appear to yield computational performance that degrades exponentially.

### 6.4 Effectiveness for Definite Databases

The two programs considered thus far have more than one minimal model. In the following we evaluate the effectiveness of our prototype for definite deductive databases. The definite database chosen is the one used by Bancilhon and Ramakrishnan [1986] to study various methods for computing definite clauses. This is the \(\text{ancestor}\) example with the following two clauses:

\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Z) \land \text{ancestor}(Z, Y) \\
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Y)
\]
It also contains six facts about the parent predicate. The table below shows the results of our compiler, executing Optimization 1 from 1 to 4 iterations.

<table>
<thead>
<tr>
<th>Iterations</th>
<th>$N = 1$</th>
<th>$N = 2$</th>
<th>$N = 3$</th>
<th>$N = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>60</td>
<td>28</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>Variables</td>
<td>70</td>
<td>38</td>
<td>24</td>
<td>19</td>
</tr>
<tr>
<td>Time (secs)</td>
<td>0.101</td>
<td>0.07</td>
<td>0.066</td>
<td>0.062</td>
</tr>
</tbody>
</table>

The figures listed in Table V confirm the effectiveness of Optimization 1 in purging extraneous variables and constraints. Recall from Section 6.1 that for the animal database, our prototype performs many times faster than Prolog. But direct comparison with Prolog can be misleading: first, Prolog cannot compute more than one model; second, for the animal database, the uses of setof and bagof on all predicates penalize Prolog as some computations are repeated many times. Of course, Prolog often performs redundant computations even if neither setof nor bagof is used. These are the reasons why the current ancestor example is a better example for comparing our prototype with Prolog. For the ancestor example is definite in nature and is simple enough to avoid repeated computations in setof and bagof. Nevertheless, Prolog takes around 0.18 seconds for this example.

In this section we have presented the experimental results of three sets and experiments. They show that Algorithm 2 computes all minimal models efficiently and Optimization 1 purges extraneous constraints and variables effectively. Given these encouraging results, more elaborate and larger scale experimentation will be carried out in future work.

6.5 Comparison with Related Work

Reading Hailperin [1976], Nerode suggested in the early 1980s to his former student R. Jeroslow, that, as Jeroslow was at the time the only expert in both logic and Operations Research, Jeroslow should develop the Boole initiative of logical inference by solving systems of real or integral linear inequalities. Boole's idea predates linear programming by a hundred years. Jeroslow developed this for propositional and predicate logic in a series of working papers at the time, and then in Blair et al. [1988], Jeroslow [1988], and Jeroslow and Wang [1990]. J Hooker and others [Hooker 1988; Hooker and Fedjki 1990] have continued this line of work with more refined results. Hooker's recent work, for example, is a branch-and-cut algorithm that combines branch-and-bound with cutting-planes.

Our work reorients the Boole-Jeroslow-Hooker approach, concentrating attention on what kind of compiler should be developed. As our inspiration, we take efficiency in handling definite and disjunctive deductive databases, where mixed integer programming deduction takes on a new flavor due to such new things as minimal models and the Generalized Closed World Assumptions. In this respect our work is analogous to that of Kowalski. He took Robinson's resolution method for pure logic out of the context of pure logic. He used Colmerauer's system as an example and produced the paradigm of declarative programming and Prolog, which is quite different in flavor from...
pure resolution theorem proving and suggests whole new areas of logical and computational and implementation problems. In concentrating on developing a compiler that does deduction at compilation rather than run-time and is incremental, we obtain evident efficiencies. Our objective, as indicated in Section 5, is to build a complete compiler-based incremental query processing system that takes full advantage of decades of Operations Research and relational database technology.

Linear programming techniques were also introduced into logic programming by Jaffar and Lassez [1987] in their constraint logic programming system CLP(R). In constraint logic programs, clauses take the form:

$$A \leftarrow G | B_1 \land \cdots \land B_n,$$

where $A$ is an atom, $B_i$s are literals, and $G$ is a “guard.” Intuitively, the guard is a condition whose truth needs to be determined over the domain of the real numbers, and this may be done using linear programming methods.

We quote Michael Maher [1991] who stated in private conversation precisely the difference between CLP and our approach. He said that “CLP evolves from conventional logic programming, replacing unification with constraint satisfaction tests, which in the case of CLP(R) involves linear programming methods,” whereas our system “replaces resolution by linear programming methods.” Thus, in Maher’s words, “the two are somewhat orthogonal uses of linear programming methods in a logic programming context.” In addition, Maher points out that there is also a fundamental difference in how the two systems execute; CLP(R) uses a top-down execution scheme, whereas ours uses a bottom-up scheme.

In the deductive database literature, two well-known techniques for recursive query evaluation are semi-naive evaluation and magic sets [Bancilhon et al. 1986; Bancilhon and Ramakrishnan 1986; Beeri and Ramakrishnan 1987]. Both these techniques compute based on given queries, whereas our method tries to obtain all logical consequences. Furthermore, it is not clear how these methods can be used to support the minimal model (or the stable model) semantics. There are also proposals on how to process disjunctive databases reported in Fernandez and Minker [1991], Liu and Sunderraman [1990]; and Yahya and Henschen [1985]. The one most closely related to our research is that of Fernandez and Minker [1991]. They have an algorithm for computing minimal models. It is unclear that their algorithm is efficient, and we know of no implementation.

7. FUTURE WORK AND CONCLUSIONS

7.1 Future Work

Our ongoing research can be classified into two categories. The first one is to complete the implementation and integration of the various modules described in Section 5. Here the major research direction pertains to the run-time module. There are many open problems to be addressed such as: what are the best data structures for storing and accessing the models? What are the best algorithms for implementing the consequences of the General-
ized Close World Assumption of programs? Can the analogy between management of multiple models in our framework, and view management in relational databases be used to improve performance by importing view management technology? [Blakeley et al. 1986; Roussopoulos 1991; Sellis et al. 1990]

Another important question pertains to the compilation module. The following table summarizes the percentages of time taken for the two major components of our compiler. Even though the figures listed are obtained for the animal experiment described in Section 6.2, the percentages are typical and representative.

<table>
<thead>
<tr>
<th>Components</th>
<th>Percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixed integer programming solvers</td>
<td>50</td>
</tr>
<tr>
<td>Preprocessing (Optimization 1)</td>
<td>35</td>
</tr>
<tr>
<td>Other overhead</td>
<td>15</td>
</tr>
</tbody>
</table>

As shown, speeding up the constraint solvers is of great interest to us. In particular, other than the general simplex methods, it would be interesting to study linear programming techniques, such as the $\lambda$-test [Li and Yes 1990], for solving linear programs that contain primarily 1, 0, and -1 as the coefficients of variables. It would also be beneficial to examine whether some of the techniques proposed in Hooker and Fedjki [1990] and Jeroslow and Wang [1990] can be used. Moreover, minimizing with respect to $\text{real}(P)$ is essentially a network problem. If all the ground clauses in $P$ are represented in conjunctive normal form, and a “reason maintenance system” network such as the ones in Charniak et al. [1987] is constructed, then the problem can be solved by simple chaining and propagation on a network. It would be interesting to evaluate the performance of this enhancement. Finally, we are interested in using fast algorithms such as the one described in Karmarkar [1988] and possibly hardware support for solving linear programs.

7.2 Conclusions

Current logic programming and deductive database systems fall short of the ideal in many respects:

(1) there is little support for disjunctive information (the same is true for negative information, but we do not address that in this article);
(2) the relatively poor run-time performance of these languages makes them unattractive to many potential industrial/commercial users;
(3) languages like Prolog contain significant nondeclarative components: for example, the order in which clauses are written and the order in which literals appear in the body of a clause significantly alters the behavior of Prolog and may in fact cause Prolog not to terminate even in situations where the program's semantics is decidable;
(4) the relatively weak query language supported by Prolog does not contain
many important features desired by users (such as aggregate information);

(5) incremental updates are not often well supported.

In this article, we believe we have taken a small step towards addressing each of these problems, and have presented an overall scheme for solving them. We suggest that the deductive component of deductive databases should be incorporated not at run-time, but at compile-time. This is a point that has been made previously, in a different setting, by many others (e.g., Ullman [1989] and the LDL group). This raises the questions: "How should we perform deduction at compile-time? What paradigm for implementing logic deduction most effectively moves the work to compile-time? Is it resolution theorem proving, magic sets, linear programming, or something else?"

We cannot give a definite answer.

From the point of view of integrating the best of current software technology, our approach has certain advantages. First, linear programming algorithms of great efficiency are both developed and implemented in linear programming packages, and can be used by our paradigm for computing minimal models of deductive databases. We do not have to reinvent the wheel. Second, we can store and access minimal models of disjunctive databases using traditional relational database technology. We do not have to reinvent this wheel either. This reduction to tools from two standard successful technologies makes us believe that our approach may help promote large scale usage of deductive databases among those who know and already use one or both of these existing technologies.

Computation of minimal models is known to be hard in the worst cases [Cadoli and Lenzerini 1990]. Minimal models are nevertheless important to compute because they are used to define the meaning of disjunctive databases, and are the heart of nonmonotonic models of negation in logic programming. It is perfectly possible that in practice, those worst cases rarely occur, and that on the average, computing minimal models is computationally feasible. After all, this is the situation for linear programming itself according to Smale's theorem [Blum et al. 1989]; and we have reduced our computational problem to mixed integer programming. The question of average performance will only be answered when a full-fledged implementation is developed and applied to databases currently used in practice. Finally, it is premature to speculate on which approach to computing minimal models (resolution-based theorem proving, mixed integer programming, magic sets) will prove most efficient in the long run. But the mixed integer programming approach presented here does open up many directions for implementation and further research.

ACKNOWLEDGMENTS

We are grateful to Allen Brown, Jorge Lobo, Michael Maher, Surya Mantha, Wiktor Marek, Sidney Marshall, Luis Moniz Pereira, Bill Pugh, Jeff Remmel, John Schlipf, Timos Sellis, Toshiro Wakayama, David S. Warren, Alex Yakh-
nis, and Carlo Zaniolo. Carl Smith provided the Smale reference [Blum et al. 1989]. In particular, we are grateful to Michael Maher for this assistance in contrasting our work with constraint logic programming. Last, but not least, we thank anonymous referees for comments and questions that greatly improved the readability of this manuscript.

APPENDIX: ANIMAL DEDUCTIVE DATABASE

\[
\begin{align*}
\text{swim}(X) & : - \text{isfish}(X). \\
\text{swim}(X) & : - \text{ismammal}(X), \text{livesinsea}(X). \\
\text{swim}(X) & : - \text{isbird}(X), \text{haswebbedfeet}(X). \\
\text{livesinsea}(X) & : - \text{isfish}(X), \text{not abfish}(X). \\
\text{livesinsea}(X) & : - \text{iswhale}(X). \\
\text{flys}(X) & : - \text{isbird}(X), \text{not abbird}(X). \\
\text{flys}(X) & : - \text{ismammal}(X), \text{haswings}(X). \\
\text{haswings}(X) & : - \text{hasfeathers}(X). \\
\text{haswebbedfeet}(X) & : - \text{isduck}(X). \\
\text{hasfeathers}(X) & : - \text{isbird}(X). \\
\text{haseggs}(X) & : - \text{isfish}(X). \\
\text{haseggs}(X) & : - \text{isbird}(X). \\
\text{livesonland}(X) & : - \text{ismammal}(X), \text{not abmammal}(X). \\
\text{livesonland}(X) & : - \text{isbird}(X). \\
\text{carnivore}(X) & : - \text{hasfangs}(X), \text{ismammal}(X). \\
\text{carnivore}(X) & : - \text{isfish}(X), \text{largemouth}(X). \\
\text{herbivore}(X) & : - \text{isfish}(X), \text{smallmouth}(X). \\
\text{herbivore}(X) & : - \text{hasmolars}(X), \text{ismammal}(X). \\
\text{hasfangs}(X) & : - \text{iscat}(X). \\
\text{hasmolars}(X) & : - \text{iscow}(X). \\
\text{hasfins}(X) & : - \text{livesinsea}(X), \text{not abseacreature}(X). \\
\text{largemouth}(X) & : - \text{iswhale}(X). \\
\text{smallmouth}(X) & : - \text{iscat}(X). \\
\text{largemouth}(X) & : - \text{not smallmouth}(X). \\
\text{smallmouth}(X) & : - \text{not largemouth}(X). \\
\text{eats}(X,Y) & : - \text{livestother}(X,Y), \text{carnivore}(X), \text{herbivore}(Y). \\
\text{eats}(X,Y) & : - \text{iscat}(X), \text{isbird}(Y). \\
\text{livestother}(X,Y) & : - \text{flys}(X), \text{flys}(Y). \\
\text{livestother}(X,Y) & : - \text{livesonland}(X), \text{livesonland}(Y). \\
\text{livestother}(X,Y) & : - \text{livesinsea}(X), \text{livesinsea}(Y). \\
\text{iswhale}(\text{mobydick}). \\
\text{iscat}(\text{garfield}). \\
\text{isbird}(\text{tweety}). \\
\text{isduck}(\text{donald}). \\
\text{isplatypus}(\text{pogo}). \\
\text{abbird}(\text{tweety}). \\
\text{abmammal}(X) & : - \text{isplatypus}(X). \\
\text{ismammal}(X) & : - \text{isplatypus}(X). \\
\text{abseacreature}(X) & : - \text{iswhale}(X).
\end{align*}
\]
REFERENCES


KOHIN, W. 1991. Declarative control architecture. CACM 34, 8, 64–79.


1, 2, 141–160.
ZANIOLI, C. 1988. Design and implementation of a logic-based language for data-intensive

Received August 1991; accepted June 1994