MAKING PROLOG MORE EXPRESSIVE

J. W. LLOYD AND R. W. TOPOR

▷ This paper introduces extended programs and extended goals for logic programming. A clause in an extended program can have an arbitrary first-order formula as its body. Similarly, an extended goal can have an arbitrary first-order formula as its body. The main results of the paper are the soundness of the negation as failure rule and SLDNF-resolution for extended programs and goals. We show how the increased expressibility of extended programs and goals can be easily implemented in any PROLOG system which has a sound implementation of the negation as failure rule. We also show how these ideas can be used to implement first-order logic as a query language in a deductive database system. An application to integrity constraints in deductive database systems is also given. ◁

1. INTRODUCTION

This paper introduces extended programs and goals for logic programming. A clause in an extended program can have an arbitrary first-order formula as its body. Similarly, an extended goal can have an arbitrary first-order formula as its body. If the body of an extended clause or goal is simply a conjunction of literals, we obtain the special case allowed by current PROLOG systems. We argue that PROLOG systems should allow the increased expressibility of extended programs and goals as a standard feature. The only requirement for implementing such a feature is a sound form of the negation as failure rule.

In Section 2, we prove the soundness of the negation as failure rule and SLDNF-resolution rule for extended programs and goals. These results are proved by first transforming an extended program and goal into a program and goal with the property that the body of the goal and the body of each clause in the program is a conjunction of literals. We then use the fact that the negation as failure rule and

Address correspondence to John W. Lloyd, The University of Melbourne, Department of Computer Science, Parkville, Victoria, 3052, Australia.

52 Vanderbilt Ave., New York, NY 10017 0743-1066/84/$03.00

SLDNF-resolution are known to be sound in this case [2, 8]. This transformation technique can be used to give a straightforward implementation of the extended syntax.

Section 3 contains several applications of extended programs. As well as the direct application to PROLOG systems, we give applications to deductive database systems. In particular, we show how typed first-order formulas can be used to express queries and integrity constraints. This provides a formal justification for the deductive database tools employed in [10].

This paper assumes some knowledge of the theoretical foundations of the negation as failure rule and SLDNF-resolution. Discussions of these matters can be found in [2] or [8]. The terminology and notation of this paper is consistent with [8].

2. SOUNDNESS RESULTS FOR EXTENDED PROGRAMS

In this section, we define extended programs and goals. The main results of the section are the soundness of the negation as failure rule and the soundness of SLDNF-resolution for extended programs and goals. Throughout, we consider first-order formulas involving $\forall$, $\exists$, $\wedge$, $\vee$, $\sim$, and $\leftarrow$. The universal closure of a formula F is denoted by $\forall(F)$.

Definition. An extended program clause is a first-order formula of the form

$$A \leftarrow W$$

where A is an atom and W is a (not necessarily closed) first-order formula. The formula W may be absent. Any variables in A and any free variables in W are assumed to be universally quantified at the front of the program clause.

A is called the *head* of the clause and W is called the *body* of the clause.

Note that, strictly speaking, an extended program clause is not necessarily a clause at all. However, we will find this terminology convenient. Throughout, we make the assumption, as we may, that distinct bound variables have distinct names.

Definition. An *extended program* is a finite set of extended program clauses.

Definition. An *extended goal* is a first-order formula of the form

$$\leftarrow W$$

where W is a (not necessarily closed) first-order formula. Any free variables in W are assumed to be universally quantified at the front of the goal.

PROLOG systems normally only allow program clauses of the form $A \leftarrow W$, where W is a conjunction of literals. Such clauses are called *general program clauses* in [8]. Similarly, a *general goal* is one of the form $\leftarrow W$, where W is a conjunction of literals. The main contribution of this paper is to show how a PROLOG system can be adapted to handle extended programs and extended goals. The latter are particularly useful as a query language since the full first order syntax is available for expressing queries.

Next we define the completion of an extended program P. Throughout, we assume $=$ is a predicate not appearing in P.

Definition. The *definition* of a predicate p appearing in an extended program P is the set of all extended program clauses in P which have p in their head.

Definition. Suppose the definition of an n-ary predicate p in an extended program is

$$A_1 \leftarrow W_1$$
$$\cdots$$
$$A_k \leftarrow W_k$$

Then the *completed definition* of p is the formula

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftrightarrow E_1 \vee \ldots \vee E_k)$$

where E_i is $\exists y_1 \ldots \exists y_d ((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge W_i)$, A_i is $p(t_1, \ldots, t_n)$, $y_1, \ldots, y_d$ are the variables in A_i and the free variables in W_i, and $x_1, \ldots, x_n$ are variables not appearing anywhere in the definition of p.

Definition. Suppose the predicate p appears in an extended program P, but not in the head of any extended program clause in P. Then the *completed definition* of p is the formula

$$\forall x_1 \ldots \forall x_n \sim p(x_1, \ldots, x_n)$$

Example. Let the definition of p be

$$p(y) \leftarrow q(y) \wedge \forall z (r(y, z) \leftarrow q(z))$$
$$p(f(z)) \leftarrow \sim q(z)$$

Then the completed definition of p is

$$\forall x (p(x) \leftrightarrow (\exists y((x = y) \wedge q(y) \wedge \forall z(r(y, z) \leftarrow q(z)))$$
$$\vee \exists z((x = f(z)) \wedge \sim q(z)))$$

We will also require the usual *equality theory* given by the following axioms:

1. $c \neq d$, for all pairs c, d of distinct constants.
2. $\forall(f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m))$, for all pairs f, g of distinct functions.
3. $\forall(f(x_1, \ldots, x_n) \neq c)$, for each constant c and function f.
4. $\forall(t[x] \neq x)$, for each nonvariable term $t[x]$ containing x.
5. $\forall((x_1 \neq y_1) \vee \ldots \vee (x_n \neq y_n) \rightarrow f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n))$, for each function f.
6. $\forall(x = x)$.
7. $\forall((x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$, for each function f.
8. $\forall((x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \rightarrow (p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n)))$, for each predicate p (including $=$).

Definition. Let P be an extended program. The *completion* of P, denoted comp(P), is the collection of completed definitions for each predicate in P together with the equality theory.

Next we introduce the declarative concept of a correct answer substitution for extended programs and goals. In this definition, if W is a formula and θ is a

substitution for some of the free variables in W, then $W\theta$ is the formula obtained by simultaneously replacing each such variable by its binding in θ. For example, if W is $\forall x \exists y(p(z, f(x)) \leftarrow q(y))$ and θ is $\{z/g(w)\}$, then $W\theta$ is $\forall x \exists y(p(g(w), f(x)) \leftarrow q(y))$. Note that it may be necessary to rename some bound variables in W before applying θ to avoid clashes with the variables in the terms of the bindings of θ.

Definition. Let P be an extended program and G an extended goal $\leftarrow W$. An *answer substitution* is a substitution for free variables in W.

Definition. Let P be an extended program and G an extended goal $\leftarrow W$. A *correct answer substitution* for $\text{comp}(P) \cup \{G\}$ is an answer substitution θ such that $\forall(W\theta)$ is a logical consequence of $\text{comp}(P)$.

This definition, which generalizes the usual definition of correct answer substitution [8], provides the appropriate declarative understanding of the output from an extended program and goal. The next step is to give the definition of the procedural concept of a computed answer substitution. This gives the implementation of the concept of a correct answer substitution. The implementation involves transforming the extended program and goal to a general program and goal, and then using SLDNF-resolution. SLDNF-resolution is just SLD-resolution augmented with the negation as failure rule. Background material on SLDNF-resolution is available in [8, chap. 3].

The first lemma justifies the transformation of an extended goal to a general goal. Suppose P is an extended program and G is an extended goal. Let G have the form $\leftarrow W$, where W has free variables $x_1, \ldots, x_n$. Suppose answer is an n-ary predicate not appearing in P or G. The transformation replaces G by the general goal

$$\leftarrow \text{answer}(x_1, \ldots, x_n)$$

and adds the extended program clause

$$\text{answer}(x_1, \ldots, x_n) \leftarrow W$$

to the extended program P.

Lemma 1. Let P be an extended program, G an extended goal, and θ an answer substitution. Assume G has the form $\leftarrow W$, where W has free variables $x_1, \ldots, x_n$ and answer is an n-ary predicate not appearing in P or G. Then we have
(a) G is a logical consequence of $\text{comp}(P)$ iff $\leftarrow \text{answer}(x_1, \ldots, x_n)$ is a logical consequence of $\text{comp}(P')$, where P' is $P \cup \{\text{answer}(x_1, \ldots, x_n) \leftarrow W\}$.
(b) $\forall(W\theta)$ is a logical consequence of $\text{comp}(P)$ iff $\forall(\text{answer}(x_1, \ldots, x_n)\theta)$ is a logical consequence of $\text{comp}(P')$.

PROOF. Note that in the presence of equality axioms 6, 7, and 8

$$\forall z_1 \ldots \forall z_n \, (\text{answer}(z_1, \ldots, z_n) \leftrightarrow \exists x_1 \ldots \exists x_n((z_1 = x_1) \wedge \ldots \wedge (z_n = x_n) \wedge W)))$$

is logically equivalent to

$$\forall x_1 \ldots \forall x_n \, (\text{answer}(x_1, \ldots, x_n) \leftrightarrow W)$$

Hence we can assume that comp(P') is simply comp(P) together with the latter formula (and an equality axiom 8 for the predicate answer). Both parts of the lemma now follow easily from this. □

The next step is to transform an extended program P into a general program P', called a *general form* of P, by means of the following transformations.

(a) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim (V \wedge W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim V \wedge W_{i+1} \wedge \ldots \wedge W_m$
and $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(b) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \forall x_1 \ldots \forall x_n W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim \exists x_1 \ldots \exists x_n \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(c) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim \forall x_1 \ldots \forall x_n W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \exists x_1 \ldots \exists x_n \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(d) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge \ldots \wedge W_m$
and $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(e) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim (V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge \sim V \wedge W_{i+1} \wedge \ldots \wedge W_m$

(f) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \vee W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge \ldots \wedge W_m$
and $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(g) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim (V \vee W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim V \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(h) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(i) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \exists x_1 \ldots \exists x_n W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(j) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim \exists x_1 \ldots \exists x_n W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim p(y_1, \ldots, y_k) \wedge W_{i+1} \wedge \ldots \wedge W_m$
and $p(y_1, \ldots, y_k) \leftarrow \exists x_1 \ldots \exists x_n W$
where $y_1, \ldots, y_k$ are the free variables in $\exists x_1 \ldots \exists x_n W$ and p is a new predicate not already appearing in the program.

Note that from a logical viewpoint, the various transformations for negation could be replaced by a single all-encompassing transformation for negation similar to (j). However, the transformations for negation have been presented as above to try to overcome the limitations of the negation as failure rule. For example, without (h), a subgoal of the form $\sim \sim A$ will delay permanently if A contains any variables. This problem disappears once the subgoal is transformed to A. Similar problems can be overcome by (a), (c), (e), and (g).

We apply transformations (a),...,(j) until no more such transformations are possible. The proposition below shows that this process terminates after a finite number of steps and that the resulting general form of the original extended program is indeed a general program. The general form of an extended program is unique modulo the choice of predicates introduced by transformation (j).

Proposition 1. Let P be an extended program. Then the process of continually applying transformations (a),...,(j) to P terminates after a finite number of steps and results in a general program.

PROOF. The basic idea of the proof is to define a termination function μ from extended programs into the set of all finite multisets of non-negative integers [4]. If M and M' are finite multisets of non-negative integers, then we define $M' < M$ if M' can be obtained from M by replacing one or more elements in M by any finite number of non-negative integers, each of which is smaller than one of the replaced elements. It is shown in [4] that the set of all finite multisets of non-negative integers under $<$ is a well-founded set.

Inductively define the mapping μ as follows:

$\mu(\text{atom}) = 1$

$\mu(V \wedge W) = \mu(V) + \mu(W)$

$\mu(\sim W) = \mu(\exists x W) = \mu(W) + 1$

$\mu(V \leftarrow W) = \mu(V) + \mu(W) + 1$

$\mu(V \vee W) = \mu(V) + \mu(W) + 2$

$\mu(\forall x W) = \mu(W) + 4$

$\mu(\text{extended program } P) = \{\mu(W) | A \leftarrow W \text{ is a clause in } P\}$

It now suffices to remark that if Q' is obtained from an extended program Q by a single transformation (a) or ... or (j), then $\mu(Q') < \mu(Q)$, so the process terminates. Furthermore, the resulting program is a general program since, otherwise, some transformation would be possible. □

Lemma 2. Let P be an extended program and let Q be the extended program which results from a single transformation (a) or ... or (i). Then P and Q are logically equivalent and also comp(P) and comp(Q) are logically equivalent.

PROOF. Straightforward. □

The corresponding result for transformation (j) is more complicated, as the following lemma shows.

Lemma 3. Let P be an extended program and P' a general form of P. If U is a closed formula which is a logical consequence of comp(P') and U only contains predicates which appear in P, then U is a logical consequence of comp(P).

PROOF. It follows from lemma 2 that we only have to prove the lemma for a single application of transformation (j). Suppose that P contains the extended program clause

$$A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$$

and we apply transformation (j) to obtain

$$A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim p(x_1, \ldots, x_n) \wedge W_{i+1} \wedge \ldots \wedge W_m$$

$$p(x_1, \ldots, x_n) \leftarrow W$$

where $x_1, \ldots, x_n$ are the free variables of W and W has the form $\exists y_1 \ldots \exists y_k V$. Let Q be the extended program obtained from P by replacing the clause to which the transformation was applied by these two clauses.

Now comp(Q) contains the formula

$$\forall z_1 \ldots \forall z_n (p(z_1, \ldots, z_n) \leftrightarrow \exists x_1 \ldots \exists x_n ((z_1 = x_1) \wedge \ldots \wedge (z_n = x_n) \wedge W)))$$

As in the proof of lemma 1, we can assume that the latter formula is replaced in comp(Q) by the formula

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftrightarrow W)$$

It follows easily from this that if U is a closed formula which is a logical consequence of comp(Q) and U contains only predicates which appear in P, then U is a logical consequence of comp(P). □

Now we are in a position to define R-computed answer substitutions for extended programs and goals, and to show that R-computed answer substitutions are correct. For this, we require the concept of a safe computation rule [8].

Definition. A computation rule R (for SLDNF-resolution) is *safe* if the following conditions are satisfied: (a) R only selects negative literals which are ground. (b) Having selected a ground negative literal $\sim A$ in some general goal, R attempts to finish the construction of a finitely failed SLDNF-tree with root $\leftarrow A$ before continuing with the remainder of the computation.

There are no restrictions at all on the selection of positive literals. This safeness condition is the usual condition that is applied to assure a sound implementation of the negation as failure rule [2, 8]. A safe computation rule can be implemented by delaying selected negative literals until they have become ground.

Definition. Let P be an extended program and G an extended goal $\leftarrow W$. A *general form* of $P \cup \{G\}$ is the general program and goal $P' \cup \{G'\}$, where G' is $\leftarrow$ answer$(x_1, \ldots, x_n)$ and P' is a general form of $P \cup \{\text{answer}(x_1, \ldots, x_n) \leftarrow W\}$.

Definition. Let P be an extended program, G an extended goal, and R a safe computation rule. An *SLDNF-refutation* of $P \cup \{G\}$ via R is an SLDNF-refutation [8] of $P' \cup \{G'\}$ via R, where $P' \cup \{G'\}$ is a general form of $P \cup \{G\}$. An *R-computed answer substitution* for $P \cup \{G\}$ is an R-computed answer substitution [8] for $P' \cup \{G'\}$.

Definition. Let P be an extended program, G an extended goal, and R a safe computation rule. An *SLDNF-tree* for $P \cup \{G\}$ via R is an SLDNF-tree [8] for $P' \cup \{G'\}$ via R, where $P' \cup \{G'\}$ is a general form for $P \cup \{G\}$. This tree is *finitely failed* if the one for $P' \cup \{G'\}$ is finitely failed [8].

Finally, we present the main results.

Theorem 1. (*Soundness of the negation as failure rule for extended programs*) *Let P be an extended program, G an extended goal, and R a safe computation rule. If* $P \cup \{G\}$ *has a finitely failed SLDNF-tree via R, then G is a logical consequence of comp*(P).

PROOF. Note first that the result is known to hold when P is a general program and G is a general goal [2, 8]. Suppose G is the extended goal $\leftarrow W$, where W has free variables $x_1, \ldots, x_n$. Let P'' be $P \cup \{\text{answer}(x_1, \ldots, x_n) \leftarrow W\}$. Suppose $P \cup \{G\}$ has a finitely failed SLDNF-tree via R. By definition, $P' \cup \{G'\}$ has a finitely failed SLDNF-tree via R, where G' is $\leftarrow \text{answer}(x_1, \ldots, x_n)$ and P' is a general form of P''. Thus, G' is a logical consequence of comp(P'). By lemma 3, G' is a logical consequence of comp(P''). Thus, by lemma 1a, G is a logical consequence of comp(P). □

Theorem 2. (Soundness of SLDNF-resolution for extended programs) Let P be an extended program, G an extended goal, and R a safe computation rule. Then every R-computed answer substitution for $P \cup \{G\}$ is a correct answer substitution for comp(P) $\cup \{G\}$.

PROOF. Note first that the result is known to hold when P is a general program and G is a general goal [8]. Suppose G is the extended goal $\leftarrow W$, where W has free variables $x_1, \ldots, x_n$. Let P'' be $P \cup \{\text{answer}(x_1, \ldots, x_n) \leftarrow W\}$ and θ be an R-computed answer substitution for $P \cup \{G\}$. By definition, θ is an R-computed answer substitution for $P' \cup \{G'\}$, where G' is $\leftarrow \text{answer}(x_1, \ldots, x_n)$ and P' is a general form of P''. Hence, θ is a correct answer substitution for comp(P') $\cup \{G'\}$. By lemma 3, $\forall(\text{answer}(x_1, \ldots, x_n)\theta)$ is a logical consequence of comp(P''). Thus, by lemma 1b, $\forall(W\theta)$ is a logical consequence of comp(P). That is, θ is a correct answer substitution for comp(P) $\cup \{G\}$. □

3. APPLICATIONS OF EXTENDED PROGRAMS

In this section, we describe the application of extended programs to general programming problems and to typed deductive database systems.

The first application of extended programs is to allow the solution of many problems to be expressed in a form similar to the specification of the problem. This simplifies the task of proving that the proposed solution satisfies the specification of the problem. The extended program is usually clearer and simpler than a corresponding solution using only a general program.

Example. Consider the extended program clause

$$A \leftarrow \forall x_1 \ldots \forall x_n (\exists y_1 \ldots \exists y_k W \leftarrow W_1 \wedge \ldots \wedge W_m)$$

Many useful extended program clauses are of this form. Typically, $W, W_1, \ldots, W_m$ are atoms and the y_i are absent. If $u_1, \ldots, u_s$ are the free variables in the body and $w_1, \ldots, w_d$ are the free variables in $\exists y_1 \ldots \exists y_k W$, then the above extended program clause can be transformed to

$$A \leftarrow \sim p(u_1, \ldots, u_s)$$

$$p(u_1, \ldots, u_s) \leftarrow W_1 \wedge \ldots \wedge W_m \wedge \sim q(w_1, \ldots, w_d)$$

$$q(w_1, \ldots, w_d) \leftarrow W$$

Several examples of this kind of extended program clause were given by Kowalski [7, p. 203, 219]. Clark [2, pp. 299–300] has also discussed this particular transformation and a version of it has been implemented in micro-PROLOG [3].

Example. [7, p. 219] The subset predicate ($\subseteq$) can be defined by the extended program clause

$$x \subseteq y \leftarrow \forall u(u \in y \leftarrow u \in x)$$

A general form of this program clause is

$$x \subseteq y \leftarrow \sim p(x, y)$$

$$p(x, y) \leftarrow \sim (u \in y) \wedge u \in x$$

It is essential that a safe computation rule be used to evaluate (general forms of) extended programs; otherwise, incorrect answers can be computed.

Example. Consider the extended program P

$$p(a) \leftarrow$$

and the extended goal

$$\leftarrow \forall x p(x)$$

whose general form is

$$\leftarrow \text{answer}()$$

where

$$\text{answer}() \leftarrow \sim q()$$

$$q() \leftarrow \sim p(x)$$

A system with an unsafe computation rule will answer "yes". However, $\forall x p(x)$ is not a logical consequence of comp(P).

Note that, with a safe computation rule, goals may delay permanently if they contain nonground negative literals. For example, evaluation of the above extended goal $\leftarrow \forall x p(x)$ will be permanently delayed when the subgoal $\sim p(x)$ is encountered. This behavior is reasonable as the PROLOG system does not know the domain of x in $\forall x p(x)$.

Example. Consider the extended program P

$$p(a, a) \leftarrow$$

$$q(b, y) \leftarrow$$

$$r(a) \leftarrow \forall y(q(x, y) \leftarrow p(x, y))$$

and the extended goal

$$\leftarrow r(a)$$

A system with an unsafe computation rule will answer "no". In fact, $r(a)$ is a logical consequence of comp(P).

The benefits of programming with extended programs and goals are, of course, also available in the nonclausal logic programming systems of [1] and [6]. The main advantage of the approach in this paper is the ease and efficiency of the implementation. On the other hand, the more general approaches of [1] and [6] allow programs to be written in even larger subsets of logic than we allow.

The second application of extended programs and goals is to deductive database systems. A deductive database is an extended program such that all the extended program clauses are typed. Similarly, a query is a typed extended goal. We make the usual restriction that databases and queries do not contain any functions. This restriction is not essential, but it does greatly simplify the proof of the key lemma 4 below. We will prove a soundness result for an implementation of correct answer substitutions for deductive databases.

Every predicate in a deductive database or query must have a type associated with each of its argument positions. Each constant and variable is also typed. We use the notation $\forall x/\tau W$ and $\exists x/\tau W$ to indicate that the bound variable x of the quantifier is of type τ. The concepts of interpretation, model, logical consequence, and so on, are defined in the natural way for typed first-order logic (also called many-sorted first-order logic). Background material on types is contained in [5].

Definition. A *database clause* is a typed function-free first-order formula of the form

$$A \leftarrow W$$

where A is an atom and W is a typed first-order formula. The formula W may be absent. Any variables in A and any free variables in W are assumed to be universally quantified at the front of the clause.

Definition. A *database* is a finite set of database clauses.

Definition. A *query* is a typed function-free first-order formula of the form

$$\leftarrow W$$

where W is a typed first-order formula and any free variables of W are assumed to be universally quantified at the front of the query. If $x_1, \ldots, x_m$ are the free variables of W and $\tau_1, \ldots, \tau_m$ are their types, we write such a query as $x_1/\tau_1, \ldots, x_m/\tau_m : W$.

Example. Consider a supplier-part-job database, whose predicates have types associated with them as follows:

supplier (snum, sname, city)

local_supplier (snum)

major_supplier (snum)

part (pnum, pname, colour, weight)

job (jnum, jname, city)

spj (snum, pnum, jnum, quantity)

In a typical state, the database may contain the following clauses:

supplier(S1, Smith, Carlton) ←

supplier(S2, Jones, Sydney) ←

supplier(S3, Jones, Perth) ←

local_supplier(S1) ←

local_supplier(s) ← supplier(s,_,Melbourne)

major_supplier(s) $\leftarrow \forall$j/jnum $\exists$q/quantity (spj(s,_,j,q) $\wedge$ q $\geq$ 100)

part(P1, Screw, White, 10) $\leftarrow$

part(P2, Nut, Black, 20) $\leftarrow$

job(J1, Build, Melbourne) $\leftarrow$

job(J2, Repair, Sydney) $\leftarrow$

spj(S1, P1, J1, 100) $\leftarrow$

spj(S2, P2, J3, 200) $\leftarrow$

In these database clauses and in subsequent queries and integrity constraints, each underscore ("_") represents a unique variable existentially quantified immediately before the atom containing it. Some possible queries that may be asked of this database are the following:

(1) Find all suppliers who supply the same part to all jobs in Perth:

$$\text{s/snum: } \exists \text{p/pnum } \forall \text{j/jnum } (\text{spj(s,p,j,\_)} \leftarrow \text{job(j,\_,Perth)})$$

(2) Find all parts and jobs supplied by all suppliers who supply some red part:

p/pnum, j/jnum:

$$\forall \text{s/snum } (\text{spj(s,p,j,\_)} \leftarrow \exists \text{p1/pnum}(\text{spj(s,p1,\_,\_)} \wedge \text{part(p1,\_,Red,\_)}))$$

(3) Find all major suppliers who supply every part or job supplied by S1:

$$\text{s/snum: major\_supplier(s)} \wedge$$

$$\forall \text{p/pnum } \forall \text{j/jnum } (\text{spj(s,p,\_,\_)} \vee \text{spj(s,\_,j,\_)} \leftarrow \text{spj(S1,p,j,\_)})$$

Definition. Let $Q \equiv x_1/\tau_1, \ldots, x_m/\tau_m : W$ be a query. An *answer substitution* is a substitution for some or all of the variables $x_1, \ldots, x_m$.

It is understood that the substitution is correctly typed in that each variable is bound to a term of the same type as the variable.

We will require the concept of the completion of a database. This definition is similar to the definition of the completion of an extended program. However, instead of a single equality symbol $=$, there is an equality symbol $=_\tau$ for each type τ. The completed definition of each predicate is then made in an analogous way to the type-free case.

Example. Let the definition of p be

$p(x) \leftarrow q(x, y)$

$p(b) \leftarrow$

where x has type τ and y has type σ. Then the completed definition for p is

$$\forall z/\tau (p(z) \leftrightarrow (\exists x/\tau \exists y/\sigma ((z =_\tau x) \wedge q(x, y)) \vee (z =_\tau b)))$$

The *equality theory* for a database is as follows:

1. $c \neq_\tau d$, for every pair c, d of distinct constants of type τ and for every type τ.
2. $\forall x/\tau \ (x =_\tau x)$, for every type τ.

3. $\forall((x_1 =_{\tau_1} y_1) \wedge \ldots \wedge (x_n =_{\tau_n} y_n) \rightarrow (p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n)))$, for every predicate p (including every $=_\tau$), where $x_1, \ldots, x_n$, $y_1, \ldots, y_n$ are appropriately typed for p and $\forall$ is the typed universal closure.
4. $\forall x/\tau\ ((x =_\tau a_1) \vee \ldots \vee (x =_\tau a_k))$, where $a_1, \ldots, a_k$ are all the constants of type τ, for every type τ.

The axioms 4 are the *domain closure axioms*. These axioms play a crucial role in Lemma 4 below.

Definition. Let D be a database. The *completion* of D, denoted comp(D), is the collection of completed definitions for each predicate in D together with the equality theory.

Definition. Let D be a database and Q a query $\leftarrow W$. A *correct answer substitution* for comp(D) $\cup \{Q\}$ is an answer substitution θ such that $\forall(W\theta)$ is a logical consequence of comp(D).

Now that we have the appropriate declarative concept, we show how it can be implemented. For this, we need to use a standard transformation which takes any typed first-order formula into a corresponding (type-free) first-order formula [5].

Definition. Let W be a typed first-order formula. For each type τ, we associate a unary predicate also denoted by τ. Then the *type-free form* W^* of W is the first-order formula obtained from W by applying the following transformations to subformulas of W of the form $\forall x/\tau V$ and $\exists x/\tau V$:

(a) Replace $\forall x/\tau V$ by $\forall x(V \leftarrow \tau(x))$.
(b) Replace $\exists x/\tau V$ by $\exists x(V \wedge \tau(x))$.

Example. Let W be the database clause

$$p(x) \leftarrow \exists y/\sigma q(x, y)$$

where x has type τ. Then W^* is

$$p(x) \leftarrow \exists y(q(x, y) \wedge \sigma(y)) \wedge \tau(x)$$

If Q is the query

$$\leftarrow \forall x/\tau q(x, y)$$

then Q^* is

$$\leftarrow \forall x(q(x, y) \leftarrow \tau(x)) \wedge \sigma(y)$$

We will also require the following *type theory* Φ [5]:
$\tau(a)$, for each constant a of type τ and for each type τ.
Now we are in a position to give the definition of a computed answer substitution.

Definition. Let D be a database, Q a query, and R a safe computation rule. Let D^* and Q^* be the type-free forms of D and Q. An *R-computed answer substitution* for $D \cup \{Q\}$ is an R-computed answer substitution for $D^* \cup \Phi \cup \{Q^*\}$.

In other words, to answer a query Q to a database D, we first transform D and Q to their type-free forms and then apply the techniques of Section 2 to the extended goal Q^* and extended program $D^* \cup \Phi$. Note that, due to the presence of the type predicates, every computed answer is a ground substitution for all the free variables in the body of the query. The next theorem shows that this implementation is sound.

Lemma 4. Let D be a database and W a closed typed function-free first-order formula. Let D^* and W^* be the type-free forms of D and W. If W^* is a logical consequence of comp$(D^* \cup \Phi)$, then W is a logical consequence of comp(D).

PROOF. We outline the proof. Let M be a model for comp(D). We use a standard method [5] to construct a model M^* for comp$(D^* \cup \Phi)$.

For each type τ, the model M has a domain C_τ. Let the domain for M^* be $\cup_\tau C_\tau$. In M^*, we assign to each constant the same element of the domain as in M. Similarly, in M^*, each predicate is assigned the same relation as in M. In M^*, $=$ is assigned the relation $\{(c, d)$: there exists a type τ such that (c, d) is in the relation assigned to $=_\tau$ in $M\}$. This completes the definition of M^*.

Using Lemma 43A of [5], it can now be checked that M^* is a model for comp$(D^* \cup \Phi)$. The domain closure axioms are used to show that M^* is a model for the only-if halves of the completed definitions of the type predicates. Hence M^* is a model for W^* and, using Lemma 43A of [5] again, we obtain that M is a model for W. Thus W is a logical consequence of comp(D). □

Theorem 3. Let D be a database, Q a query, and R a safe computation rule. Then every R-computed answer substitution for $D \cup \{Q\}$ is a correct answer substitution for comp$(D) \cup \{Q\}$.

PROOF. Let θ be an R-computed answer substitution for $D \cup \{Q\}$, where Q is $\leftarrow W$ and W has free variables $x_1, \ldots, x_n$. By theorem 2, $(W^* \wedge \tau_1(x_1) \wedge \ldots \wedge \tau_n(x_n))\theta$ is a logical consequence of comp$(D^* \cup \Phi)$. Thus $(W\theta)^*$ is a logical consequence of comp$(D^* \cup \Phi)$. By Lemma 4, $W\theta$ is a logical consequence of comp(D). That is, θ is a correct answer substitution for comp$(D) \cup \{Q\}$. □

This result provides the basis for the implementation of deductive database systems using PROLOG systems. Techniques for implementing such systems are described in [10].

Example. We cannot omit the domain closure axioms from the definition of comp(D). Let D be the database

$$p(a) \leftarrow$$

and Q be the query $\leftarrow \forall x/\tau p(x)$. Then the identity substitution is a computed answer substitution, but $\forall x/\tau p(x)$ is not a logical consequence of comp(D) if the domain closure axiom $\forall x/\tau\ (x = a)$ is omitted from comp(D).

As we have pointed out above, an extended goal may delay permanently because of a nonground negative literal. However, an important property of the implementation for deductive databases is that no selected negative literal will ever delay permanently. As the next proposition shows, there is always an atom of the form $\tau(x)$ available to ground a variable x.

Proposition 2. Let D be a database and Q a query. Then no selected negative literal will ever delay permanently during the evaluation of Q.

PROOF. We outline the proof. Let us say a free variable x in the body of an extended program clause is *safe* if either x appears in the head of the clause or there is a top level conjunct in the body of the form $\tau(x)$.

Let P be $D^* \cup \Phi \cup \{\text{answer}(x_1, \ldots, x_n) \leftarrow W^* \wedge \tau_1(x_1) \wedge \ldots \wedge \tau_n(x_n)\}$, where Q is $\leftarrow W$ and W has free variables $x_1, \ldots, x_n$. We claim that P', the general form of P, has the property that each free variable in the body of each clause of P' is safe. To prove this, note that P certainly has this property. Furthermore, each transformation (a) to (j) above preserves this property and hence P' has this property too. (Note that transformation (i) reduces a subgoal $\exists x \sim (W \leftarrow \tau(x))$ to $\sim (W \leftarrow \tau(x))$, but an application of transformation (e) restores the property).

Now, in the evaluation of $\leftarrow \text{answer}(x_1, \ldots, x_n)$, every goal of the SLDNF-refutation, except the first, has the property that each variable in the goal appears in a subgoal of the form $\tau(x)$, by the above safeness property. The proposition follows. □

Note that, by an appropriate reordering of literals in the body of each clause in P' (viz., putting atoms of the form $\tau(x)$ first), we can ensure that the standard PROLOG left to right computation rule is safe.

The third application of extended programs and goals is to the enforcement of integrity constraints in deductive databases.

Definition. An *integrity constraint* is a closed typed function-free first-order formula.

Any variables other than underscores not explicitly quantified in a constraint are taken to be universally quantified at the front of the constraint.

Example. Some integrity constraints that may be imposed on the above database are the following:

(1) Supplier S1 supplies every job in quantities of at least 100:

$$q \geq 100 \leftarrow \text{spj}(\text{S1}, \_, \_, q)$$

(2) Supplier S2 supplies every job in Sydney:

$$\text{spj}(\text{S2}, \_, j, \_) \leftarrow \text{job}(j, \_, \text{Sydney})$$

(3) Supplier S3 only supplies jobs in Adelaide or Perth:

$$\text{job}(j, \_, \text{Adelaide}) \vee \text{job}(j, \_, \text{Perth}) \leftarrow \text{spj}(\text{S3}, \_, j, \_)$$

Definition. [9] A database D *satisfies* an integrity constraint C if comp(D) is consistent and C is a logical consequence of comp(D). Otherwise, the database *violates* the constraint.

Intuitively, an integrity constraint should be an invariant of the database. The standard method of determining whether a database satisfies or violates an integrity constraint C is by evaluating the query $\leftarrow C$. The following two theorems show that this method is sound.

Definition. Let D be a database, Q a query, and R a safe computation rule. Let D^* and Q^* be the type-free forms of D and Q. An *SLDNF-refutation* of $D \cup \{Q\}$ via R is an SLDNF-refutation of $D^* \cup \Phi \cup \{Q^*\}$ via R. A *finitely failed SLDNF-tree* via R for $D \cup \{Q\}$ is a finitely failed SLDNF-tree via R for $D^* \cup \Phi \cup \{Q^*\}$.

Theorem 4. Let D be a database, C an integrity constraint, and R a safe computation rule. Suppose that comp(D) is consistent. If there exists an SLDNF-refutation of $D \cup \{\leftarrow C\}$ *via R, then D satisfies C.*

PROOF. The theorem follows immediately from theorem 2 and lemma 4. □

Theorem 5. Let D be a database, C an integrity constraint, and R a safe computation rule. If $D \cup \{\leftarrow C\}$ *has a finitely failed SLDNF-tree via R, then D violates C.*

PROOF. The theorem follows immediately from theorem 1 and lemma 4. □

Techniques for restricting the number of constraints to be checked after each database change and for reducing the time required for each constraint checked are described in [10].

4. CONCLUSIONS

We have introduced extended programs and goals, proved the soundness of the negation as failure rule and SLDNF-resolution for extended programs and goals, shown that the use of extended programs and goals simplifies and clarifies some programming problems, and applied extended programs and goals to database clauses, queries, and integrity constraints in deductive databases.

Extended programs and goals are evaluated by transformation into general programs and goals which can be evaluated using any PROLOG system that implements a sound form of the negation as failure rule. Deductive database systems using typed extended programs and goals can be soundly implemented even with the standard PROLOG left to right computation rule.

We conclude that PROLOG systems should implement a sound form of the negation as failure rule and allow the increased expressibility of extended programs and goals as a standard feature. Such systems would reduce the need for programmers to use nonlogical features of current PROLOG systems, and would provide a step towards the goal of "programming in logic".

REFERENCES

1. Bowen, K. A., Programming with Full First-Order Logic, *Machine Intelligence* 10:421–440 (1982).
2. Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum Press, New York, 1978, pp. 293–322.
3. Clark, K. L. and McCabe, F. G., *micro-PROLOG: Programming in Logic*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
4. Dershowitz, N. and Manna, Z., Proving Termination with Multiset Orderings, *Comm. ACM* 22:465–476 (Aug 1979).

5. Enderton, H. B., *A Mathematical Introduction to Logic*, Academic press, New York, 1972.
6. Haridi, S. and Sahlin, D., Evaluation of Logic Programs based on Natural Deduction, TRITA-CS-8305 B, Royal Institute of Technology, Sweden, 1983.
7. Kowalski, R. A., *Logic for Problem Solving*, North Holland, New York, 1979.
8. Lloyd, J. W., Foundations of Logic Programming, Technical Report 82/7 (revised March 1984), Department of Computer Science, University of Melbourne.
9. Reiter, R., A Logical Reconstruction of Relational Database Theory, in: M. L. Brodie et al. (eds.), *Perspectives in Conceptual Modelling*, Springer-Verlag, 1983.
10. Topor, R. W., Keddis, T., and Wright, D., Deductive Database Tools, Technical Report 84/7, Department of Computer Science, University of Melbourne.