NORTH-HOLLAND

TECHNICAL NOTE

EXTENDING PROLOG WITH NONMONOTONIC REASONING*

WEIDONG CHEN

▷ Nonmonotonic reasoning has been developed to capture common sense inferences. This paper considers nonmonotonic reasoning in logic programs with negation, specifically its implementation using Prolog and its integration with Prolog execution. Even within logic programming frameworks, a variety of inferencing methods, model-theoretic semantics, and language features have been proposed for different forms of nonmonotonic reasoning. A challenging problem is to incorporate them into an integrated system. This paper describes an implementation of such a system in Prolog and different reasoning capabilities that are supported. By combining Prolog technology with various mechanisms of nonmonotonic reasoning, the resulting system offers a more realistic testbed for applications of common sense reasoning. ◁

1. INTRODUCTION

In logic programming, a nonmonotonic inference rule, namely, negation as failure, is normally used to establish negative atoms. Extensive research has been conducted on the declarative semantics of logic programs with negation. The connections between logic programming and nonmonotonic reasoning have attracted strong interests [12, 16, 18]. In particular, stable models [12] for normal logic programs correspond closely to extensions in nonmonotonic reasoning.

This paper considers nonmonotonic reasoning in logic programming frameworks, which share with Prolog [15] a core rule-based language. Prolog is by far the most

*Work supported in part by the National Science Foundation under Grants IRI-92-12074 and IRI-93-14897.

Address correspondence to Weidong Chen, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275-0122.

Received June 1995; accepted November 1995.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

0743-1066/96/$15.00

SSDI 0743-1066(95)00151-4

popular logic programming language, whose compiler technology and programming environments are well developed. Within logic programming frameworks, there are various forms of nonmonotonic reasoning, due to the differences in operational semantics, model-theoretic semantics, and language features.

Operationally, the computational model in Prolog is based upon SLD resolution with negation as failure (SLDNF), which is a top-down depth-first strategy. Although it is efficient for a stack-based implementation, SLDNF may not terminate and may repeat the evaluation of identical calls. To guarantee termination and completeness, tabled evaluation has been proposed. It maintains a table of calls and their associated answers. Repeated calls are then solved using answers in the table instead of rules in a program. Redundant answers of a call are eliminated.

Semantically speaking, one way to characterize the meaning of a logic program is by Clark's completion, which often serves as a declarative counterpart for SLDNF. More recently a dominant approach is to specify a single canonical model as the semantics of a logic program, such as the perfect Herbrand model for stratified programs [17] and the well-founded partial model for normal logic programs [25]. Query evaluation becomes finding the set of answers of a query with respect to the canonical model. Stable models [12] are the most popular semantics for nonmonotonic reasoning with logic programs. A logic program may have zero, one, or more stable models. The same query may have different sets of answers in different stable models. The multiplicity of stable models offers several useful forms of nonmonotonic reasoning, such as abduction and skeptical reasoning.

Syntactically, several extensions have been proposed to enhance the expressive power of specifications of nonmonotonic reasoning. They include general logic programs [24] that allow arbitrary first-order formulas in rule bodies, abductive logic programs with integrity constraints [10], Datalog programs with choice [20], extended logic programs with classical negation and disjunction [13], and epistemic specifications [11].

Pragmatically, different operational strategies, model-theoretic semantics, and language features may be appropriate for different applications or even different parts of the same application. The challenge is to incorporate them into a single system so that applications can exploit the advantages of each of them. This paper describes a working implementation of such a system and various reasoning capabilities that are supported, in the hope that it will spur more research into practical implementations and uses of nonmonotonic reasoning for real applications.

Section 2 presents an implementation of the well-founded semantics of normal logic programs using tabled evaluation. The evaluation of a query produces a *residual* program that can be processed for other forms of nonmonotonic reasoning. In particular, Section 3 focuses on the computation of stable models of residual programs of queries and its use for abductive reasoning. Section 4 describes skeptical reasoning with respect to the intersection of all stable models of residual programs of queries. Section 5 deals with general logic programs with arbitrary first-order formulas in rule bodies. Section 6 concludes with a discussion of the current status of the system.

2. TABLED EVALUATION OF THE WELL-FOUNDED SEMANTICS

SLDNF as implemented in Prolog is a top-down depth-first strategy. It is known to be incomplete for deductive query evaluation and nonmonotonic reasoning. Assuming

a leftmost selection rule as in Prolog, a simple left recursive predicate definition can send Prolog into an infinite loop. Tabled evaluation such as OLDT resolution [23] has been developed in order to guarantee termination and completeness for certain classes of logic programs, such as function-free programs.

We support two distinct modes of computation, namely, Prolog execution and tabled evaluation. The latter is based upon SLG resolution [4], which extends OLDT resolution to handle negation and to compute the well-founded semantics of normal logic programs. Three directives are available—`prolog`, `tabled`, and `default`—for users to specify which predicates are executed by Prolog, which predicates are computed using tabled evaluation, and the default mode (`tabled` or `prolog`) of execution. The system default is `prolog`, so that Prolog programs are not affected. Directives for predicates must occur before the corresponding definitions of the predicates. Also the body of each rule of a tabled predicate should be a conjunction of literals (without if-then-else or disjunction).

Example 2.1. The following is a simple program with a tabled predicate:

```
edge(a,b). edge(b,a). edge(c,d).

:- tabled path/2.
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

where the `tabled` directive indicates that all calls to `path/2` are processed using tabled evaluation. Calls to `edge/2` are executed using Prolog by default. A query such as `path(a,N)` will return two answers through backtracking as in Prolog and then terminate.

SLG resolution maintains a table of calls to tabled predicates for detecting repeated calls, where two calls are identical if they are variants of each other. For each call to a tabled predicate, SLG resolution keeps a set of answers and a list of waiting nodes. Repeated calls are solved using answers from the table instead of rules from a program. When a tabled call Q invokes another tabled predicate Q', a waiting node is created that waits for answers from the evaluation of Q'. Waiting nodes are necessary to guarantee completeness because a repeated call may be encountered before all of its answers have been derived.

Let Q be a call to a tabled predicate. The tabled evaluation of Q is carried out as follows:

1. Find all the instances of rules of the tabled predicate by unifying Q with the head of a rule and solving all Prolog predicates that appear before the first occurrence of any tabled predicate in the body of the rule.
2. Process each rule instance obtained in (1).

For `path(a,N)`, the following rule instances will be derived by (1):

```
path(a,b).
path(a,N) :- path(b,N).
```

where calls to `edge/1` that appear before the first occurrence of any tabled predicate are solved.

In (2), let r be an arbitrary rule instance of the form H :- B.

- If B is empty, then r is an answer for Q. If r is a new answer (with respect to variant checking), it is added to the table and is returned to all nodes waiting on Q.
- Otherwise, B starts with a call Q'. If Q' is a Prolog call, r is replaced with the new rule instances obtained by solving Q' in the body of r. These new rule instances are processed recursively. If Q' is a tabled call, r becomes a waiting node for answers of Q'. If Q' is a new call, Q' is processed in the same way as Q following (1) and (2). If Q' is not a new call, all current answers of Q' are returned to r, generating new rule instances for Q that are processed recursively.

The evaluation of `path(a,N)` will lead to a new call `path(b,N)`, whose initial rule instances are

```
path(b,a).
path(b,N) :- path(a,N).
```

By returning answers to waiting nodes, all answers for `path(a,N)` and `path(b,N)` are derived.

When the negation $\backslash+Q$ of a ground call to a tabled predicate is encountered, Q has to be completely evaluated before the success of $\backslash+Q$ can be determined. We have developed an incremental algorithm for detecting tabled calls that are completely evaluated [3].

The basic idea is as follows. The evaluation of tabled predicates as discussed above follows a top-down depth-first strategy, which induces a stack of tabled calls according to the order in which they are created. When Q is added to the table as a new call, its rule instances are processed, which may create new tabled calls and new rule instances that are processed recursively. When the processing of all rule instances of Q finally returns, consider the set of calls from Q to the top of the call stack. If none of them depends upon any call below Q or the negation of any call in the stack, then the set of calls from Q to the top of the stack is completely evaluated. The calls are then popped off the stack and all their waiting nodes are disposed. All nodes that wait on the negation of some of the completed calls are processed. An incremental algorithm for maintenance of dependencies among tabled called is presented in [3].

Example 2.2. Consider the following simple program using `path/2` in Example 2.1:

```
:- tabled nr/2.
nr(N) :- \+path(a,N).
```

The evaluation of `nr(c)` leads to the creation of three new calls to tabled predicates—`nr(c)`, `path(a,c)`, and `path(b,c)`—which are pushed onto the stack in that order.

When the processing of `path(b,c)` and its rule instances returns, it is not completely evaluated because it depends upon a call `path(a,c)` below it in the stack. However, when the processing of `path(a,c)` and its rule instances, which includes the recursive processing of `path(b,c)` and its rule instances, returns, none of the calls from `path(a,c)` to the top of the stack depends upon any call below `path(a,c)`

or the negation of any call in the stack. Thus `path(a,c)` and `path(b,c)` are completely evaluated and are popped off the stack. Because `path(a,c)` does not have any answer, `nr(c)` succeeds.

Two tabled calls may depend upon each other through negation, neither of which can be completely evaluated. To allow the tabled evaluation to proceed, SLG resolution *delays* ground negative literals that are involved in loops, so that the remaining literals in the body of a rule instance can be solved. With delayed literals, a rule instance, in general, is of the form (H, D) :- B, where H is the head, D is a (possibly empty) list of delayed literals, and B is the body. If B is empty, the rule instance is considered an answer (that is, essentially a rule with delayed literals D in the body). The dependency maintenance for detecting completion of tabled calls can also be used for detecting tabled calls that are possibly involved in loops through negation. Notice that delayed literals are not considered in the dependencies among tabled calls.

Example 2.3. Consider the known *win* program with a tabling directive:

```
move(a,b). move(b,a). move(b,c). move(c,d).
:- tabled win/1.
win(X) :- move(X,Y), \+win(Y).
```

The evaluation of `win(X)` creates five tabled calls:

```
win(X),    win(a),    win(b),    win(c),    win(d),
```

which are pushed onto the stack in that order following the top-down and depth-first strategy. Calls `win(c)` and `win(d)` are completed and are popped off the stack. The call `win(b)` is not completed because it depends upon `win(a)` that is below it in the stack. When the processing of `win(a)` and its rule instances returns, `win(a)` and the call `win(b)` on top of it on the stack do not depend upon any call below `win(a)` on the stack. However, they are not completely evaluated because there is a loop through negation among them. At this point, the waiting nodes for tabled calls on the stack are as follows:

```
win(b): (win(b), []) :- \+win(a).
win(a): (win(a), []) :- \+win(b).
win(X): (win(a), []) :- \+win(b).
        (win(b), []) :- \+win(a).
```

By delaying the selected ground negative literals, the dependencies through negation are reset, and the waiting nodes are replaced with the new rule instances

```
win(b): (win(b), [\+win(a)]).
win(a): (win(a), [\+win(b)]).
win(X): (win(a), [\+win(b)]).
        (win(b), [\+win(a)]).
```

all of which are answers. The three calls `win(b)`, `win(a)`, and `win(X)` are then completely evaluated and popped off the stack in that order. The call `win(X)` also has an answer `win(c)` that is not shown here.

Delayed literals may turn out to be true or false, in which case they should be simplified away from answers of tabled calls. That is, delayed literals that are true

should be deleted from answers and answers with delayed literals that are false should be deleted.

We have defined several predicates for query processing under the well-founded semantics. The predicate `slg/1` returns only true answers of a call. An infix system predicate `Call<-Cond` is provided to return every answer through backtracking, where `Cond` is a list of delayed literals (and is [ ] for a true answer). As a counterpart for *findall*, a predicate `slgall` is defined that collects all answers in a list.

Example 2.4. A sample execution of three queries with respect to the program in Example 2.3 is

```
| ?- win(N).
N = c ? ;
no
| ?- slg(win(N)).
N = c ? ;
no
| ?- win(N) <- U.
N = a,
U = [\+win(b)] ? ;
N = b,
U = [\+win(a)] ? ;
N = c,
U = [] ? ;
no
| ?- slgall(win(N), Anss).
Anss = [win(a)<-[\+win(b)], win(b)<-[\+win(a)], win(c)] ? ;
no.
```

The default for evaluating `win(N)` is to return only true answers under the well-founded semantics.

With delayed literals, answers are essentially rules with delayed literals in rule bodies. When an answer with delayed literals is returned to a waiting node, the variable bindings accumulated in the answer head are propagated through unification. However, the delayed literals in the answer body are not propagated, in order to avoid combinatorial explosion [5]. Instead, positive delayed literals are created as place holders for propagating truth values of delayed ground negative literals.

Example 2.5. Consider the program

```
:- default(tabled).
q(X) :- p(X).
p(a).
p(X) :- r.
r :- \+s.
s :- \+r.
```

The evaluation of `q(X)` using SLG resolution returns the following answers (represented as rules) for calls relevant to `q(X)`:

```
q(X):   q(a).
        q(X) :- p(X).
p(X):   p(a).
        p(X) :- r.
r:      r :- \+s.
s:      s :- \+r.
```

Notice that when the answer `r :- \+s` for `r` is returned to the waiting node for `p(X)`, a positive delayed literal `r` is created in `p(X) :- r`, for propagating the truth value of `\+s`. Similarly, a positive delayed literal `p(X)` is created in an answer for `q(X)`.

Had we propagated delayed negative literals when returning answers to waiting nodes, we would have derived the answers

```
q(X):   q(a).
        q(X) :- \+s.
p(X):   p(a).
        p(X) :- \+s.
r:      r :- \+s.
s:      s :- \+r.
```

where the body of each answer is either empty or a conjunction of ground negative literals. However, this propagation may lead to an exponential number of answers in the worst case [5].

The use of positive delayed literals guarantees a polynomial representation of answers of queries that may otherwise have an exponential number of answers with only ground negative literals in answer bodies. Each delayed literal in the body of an answer, such as `p(X)` in `q(X) :- p(X)`, corresponds to the head of some answer of a call, such as `p(X) :- r` of `p(X)`, with delayed literals in its body. Because all variable bindings have been propagated by SLG resolution, we can consider the head of each answer as a proposition, where the heads of two answers of the same call are identical if and only if they are variants of each other. Then the set of answers resulting from query evaluation can be viewed as a *semi-propositional* program. This observation is confirmed by the formal proof of the correctness of SLG resolution in [5], which also showed that SLG resolution preserves all three-valued stable models of normal logic programs.

The *residual program* of a query consists of all answers of the query plus the rules of all head atoms upon which the answers of the query depend, directly or indirectly, through delayed literals in answer bodies. As mentioned above, the residual program of a query can be viewed as a semi-propositional program, which will be processed further for providing other forms of nonmonotonic reasoning.

The idea of conditional answers with ground negative literals in answer bodies has been used in fixpoint semantics of logic programs [1, 8]. The unique feature of SLG resolution is the use of delayed literals in top-down evaluation to handle negative loops and the guarantee of polynomial time data complexity for computing the well-founded semantics by allowing positive delayed literals. Our notion of residual programs is different from that of relevant rules in [7] because we consider dependencies in programs that have been simplified according to the well-founded semantics.

3. ABDUCTIVE REASONING WITH STABLE MODELS

In [14], stable models by Gelfond and Lifschitz [12] are used as a semantic basis for abductive logic programs. In particular, given an abductive logic program $\langle P, A, I\rangle$, where P is a logic program, A is a set of abducible predicates, and I is an integrity constraint. A set Δ of ground abducible atoms is an *abductive explanation* for a query Q in $\langle P, A, I\rangle$ if there is a stable model M of $P \cup \Delta$ such that $M \models I$ and $M \models Q$.

The connection between abduction and negation by failure also has been studied. In [10], Eshghi and Kowalski showed that negation by failure can be accomplished using abduction with integrity constraints. In [21], abducible predicates are simulated using default negation by adding a new predicate np for each abducible predicate p and introducing the rules

```
p(X) :- \+np(X).
np(X) :- \+p(X).
```

For each ground abducible atom, say `p(a)`, the preceding pair of rules has two stable models, one in which `p(a)` is true and the other in which `\+p(a)` is true. The integrity constraint remains as a condition for selecting stable models. Thus every abductive logic program can be transformed into a logic program with integrity constraints.

It is known that stable models do not satisfy the principle of relevance [7]. Given a program P, adding a new rule of the form `p :- \+p` to a program may eliminate all stable models of the program. Our approach to abduction based upon stable models consists of two steps. First, given a query Q and an integrity constraint I, both Q and I are evaluated using SLG resolution and their residual programs are derived. Second, a backtracking algorithm is applied to compute stable models of the union of the residual programs of Q and I that satisfy I [6].

The decision to compute stable models of residual programs instead of those of the original entire program is based upon practical considerations, because there may not be a single complete program when each logic program module is loaded dynamically. Nevertheless, a user can compute stable models of an entire program P by adding a new predicate, say `m`, and a rule of the form `m :-` $p(\vec{X})$ for each predicate p in P, where $p(\vec{X})$ is the most general form of an atom of p. Then assuming that floundering does not occur, stable models of the residual program for `m` correspond to stable models of the entire program P, with the interpretation for `m` omitted.

Let P be a normal logic program and let Q be a query. Let P_Q be the residual program of Q. Recall that all negative literals occurring in P_Q are ground and all variable bindings have already been propagated by SLG resolution. Stable models of P_Q are computed as follows:

- Identify the set $\mathcal{N}$ of all ground negative literals occurring in P_Q.
- For each ground negative literal $\backslash+ B \in \mathcal{N}$, make a choice of either `true` or `false` for B and simplify the program P_Q as follows:

  —Delete all rules that have B in the body if B is assumed to be `false`.
  —Delete all rules that have $\backslash+ B$ in the body if B is assumed to be `true`.
  —Delete all occurences of $\backslash+ B$ if B is assumed to be `false`.

  If inconsistency occurs, backtrack to the most recent choice point.

Notice that if B is assumed to be `true`, this assumption cannot be used to delete occurrences of B in rule bodies. When simplifying the residual program P_Q based upon the choice of a truth value for B, the truth values of some remaining ground negative literals in $\mathcal{N}$ may be determined, which can be used to avoid unnecessary choice points. The details of the backtracking algorithm and its correctness proof are in [6].

A predicate `stall(Q,Anss,SM)` is defined that collects all answers of a query `Q` in a stable model `SM` of the residual program of `Q`. Unlike `slgall/1`, `stall(Q,Anss, SM)` may succeed multiple times if there is more than one stable model of the residual program of `Q`.

If there is any integrity constraint, it can be used to select stable models that satisfy the constraint. A predicate `stselect(Q,Cond,Anss,SM)` has been defined that computes only those stable models of the union of the residual programs of `Q` and `Cond`, where `Cond` is true. `Cond` is a conjunction (represented as a list) of *ground* literals. In `stselect`, the residual program consists of all answers that are relevant to `Q` or `Cond`.

Example 3.1. Consider the following program that selects an arbitrary student from each class:

```
take(sean, ai). take(jenny, ai).
take(brad, db). take(jenny, db).
:- tabled ch/2, df/2, chj/0.
ch(S,C) :- take(S,C), \+df(S, C).
df(S, C) :- ch(S1,C), \+S=S1.
chj :- ch(jenny, _).
```

The following query chooses only those stable models in which Jenny is selected:

```
| ?- stselect(ch(S, C), [chj], Anss, SM).
Anss = [ch(brad, db), ch(jenny, ai)],
SM = [chj, ch(brad, db), ch(jenny, ai),
      \+df(brad, db), \+df(jenny, ai), df(jenny, db), df(sean, ai)] ? ;
Anss = [ch(jenny, ai), ch(jenny, db)],
SM = [chj, ch(jenny, ai), ch(jenny, db),
      df(brad, db), \+df(jenny, ai), \+df(jenny, db), df(sean, ai)]?;
Anss = [ch(jenny, db), ch(sean, ai)],
SM = [chj, ch(jenny, db), ch(sean, ai),
      df(brad, db), df(jenny, ai), \+df(jenny, db), \+df(sean, ai)] ? ;
no.
```

Notice that `chj/0` must be tabled in this case. Otherwise, the default of calling a tabled predicate from a Prolog predicate is to compute true answers with respect to the well-founded semantics.

The condition in `stselect` can be used to implement global integrity constraints. In particular, each constraint may be of the form

$$::- L_1, \ldots, L_n.$$

where $n > 0$ and each $L_i (1 \leq i \leq n)$ is a literal. Integrity constraints are automatically transformed into rules of a special tabled predicate `inconsistent`, and

`\+inconsistent` is automatically included as part of a condition for `stselect`. (If modular logic programming is supported, the effects of integrity constraints should be localized to predicates in a module so that unrelated predicates in other modules are not affected.)

Semantically our approach does not suffer the anomalies of semantics such as STABLE** in [7], even though it is goal-oriented. The reason is that we treat each stable model as a complete scenario relevant to a query and to its associated integrity constraint. Operationally we retain for abductive reasoning the advantages of goal-oriented evaluation, handling of nonground programs, and, more importantly, guaranteed termination for function-free programs. A unique feature is the separation of the derivation of residual programs from the backtracking computation of their stable models. This is in contrast with other goal-oriented procedures for abductive logic programming [9, 10, 22], which interleave backward chaining and consistency checking and does not guarantee termination. Even if tabling techniques are used, the interleaving computation prevents the full sharing of answers because identical subgoals may have different assumptions associated with them.

4. SKEPTICAL REASONING THROUGH NEGATION AS FAILURE

Whereas abductive reasoning involves selecting one stable model that satisfies certain integrity constraints, skeptical reasoning focuses on deriving answers of a query that hold in all stable models. Again we focus on stable models of residual programs of queries.

Our approach to skeptical reasoning is as follows. Let P be a normal logic program and let Q be a ground atom. First, the residual program of Q with respect to P is derived using SLG resolution, which will be denoted by P_Q. Second, Q is true in all stable models of P_Q if and only if both of the following statements hold:

(a) There is at least one stable model of P_Q in which Q is true.
(b) There is no stable model of P_Q in which $\backslash{+}Q$ is true.

The conditions (a) and (b) can be easily implemented using the predicate `stselect` and negation as failure in Prolog.

If Q is not ground, then Q may have multiple answers with different head atoms. The technique can be extended to nonground queries by treating each distinct answer head as a distinct proposition, *provided that no two distinct answer heads are unifiable with each other* (see Example 4.2). A predicate `stintall(Q,Anss)` has been defined that computes all answers of a query that are true in (the intersection of) all stable models of the residual program of `Q`.

Example 4.1. Consider the following scenario, where there are two guns, one of which is loaded, and `foe` is already killed:

```
:- default(tabled).
loaded(1) :- \+loaded(2).
loaded(2) :- \+loaded(1).
trigger(1). trigger(2).
killed(doe) :- loaded(X), trigger(X).
killed(foe).
```

It should be apparent that `killed(doe)` holds if the trigger is pulled for both guns:

```
| ?- stintall(killed(X), Anss).
Anss = [killed(foe), killed(doe)] ? ;
no.
```

However, if a query has two answers under the well-founded semantics whose head atoms are unifiable but not variants of each other, `stintall/2` may not work.

Example 4.2. Consider the simple program

```
:- default(tabled).
p :- \+q.
q :- \+p.
r(f(X, b)) :- p.
r(f(a, Y)) :- q.
```

A sample script of query evaluation is

```
| ?- stintall(r(f(a, b)), Anss).
Anss = [r(f(a, b))] ? ;
no
| ?- stintall(r(f(X, Y)), Anss).
Anss = [] ? ;
no.
```

In other words, `stintall/2` is able to conclude that `r(f(a,b))` as a ground query is true in all stable models, but could not find any answer for `r(f(X,Y))` that is true in all stable models. The reason is that for the nonground query `r(f(X,Y))`, there are two answers with distinct but unifiable head atoms, namely, `r(f(X,b))` and `r(f(a,Y))`. They are treated separately by the current implementation.

5. HANDLING FIRST-ORDER FORMULAS

General logic programs allow arbitrary first-order formulas in rule bodies. Their semantics is characterized by Van Gelder's alternating fixpoint logic [24]. It has been shown [2] that a straightforward translation of general logic programs into normal logic programs does not preserve the semantics of alternating fixpoint logic. However, every general logic program can be translated into an equivalent one that contains two kinds of rules, normal rules and *universal rules* whose bodies are universally quantified disjunctions of literals.

We have implemented query evaluation with respect to the alternating fixpoint logic. A special syntax is introduced for universal rules, of the form

$$H \text{<--} L_1; \cdots; L_n,$$

where H is an atom, and each $L_i (1 \leq i \leq n, n > 0)$ is a literal. Let $\vec{X}$ be all the variables that occur in both the head and the body and let $\vec{Y}$ be all the variables that occur only in the body. Then the universal rule is viewed as the

formula

$$\forall \vec{X}.(H \leftarrow (\forall \vec{Y}.(L_1 \vee \cdots \vee L_n))).$$

Predicates with universal rules must be tabled.

It has been shown [2] that all universal rules can be reduced to the form

$$H \text{<-} \forall \vec{Y}.(\neg A \vee B),$$

where H is an atom and A and B are atoms that may be missing. In predicate calculus the universal disjunction in the body is equivalent to

$$\neg \exists \vec{Y}.(A \wedge \neg B).$$

For *safe* evaluation, we require the following conditions:

- All calls to tabled predicates with universal rules must be ground, which means that a universal disjunction does not have any free variables when it is evaluated.
- All variables in $\vec{Y}$ should be bound to ground terms after A succeeds, which means that $\neg B$ is ground when it is evaluated.

To solve a universal disjunction of the form $\forall \vec{Y}.(\neg A \vee B)$, we first evaluate A completely and let $A_1, \ldots, A_n$ be all the answers of A represented as instances of A. Then the universal disjunction is reduced to a conjunction $\forall(B\theta_1) \wedge \cdots \wedge \forall(B\theta_n)$, where each θ_i $(1 \leq i \leq n)$ is a most general unifier of A and A_i. If the requirements for safe evaluation of universal disjunctions are satisfied, every $B\theta_i$ $(1 \leq i \leq n)$ is a ground atom, and the universal disjunction is reduced to a conjunction of ground atoms.

The implementation of the alternating fixpoint logic is an extension of that of SLG resolution. Modifications include the interactions between universal disjunctions and the issues of dependency maintenance and delayed literals. Formal details can be found in [2].

Example 5.1. Consider the following general logic program, which defines *founded* nodes in a directed graph:

```
:- tabled fnode/1, founded/1.
edge(a,b). edge(b,a). edge(b,c).
node(a). node(b). node(c).

fnode(N) :- node(N), founded(N).
founded(x) <-- \+ edge(X,Y);founded(Y).,
```

where the rule for `founded/1` is basically the formula

$$\forall X.(\text{founded}(X) \leftarrow \forall Y.(\text{edge}(X,Y) \rightarrow \text{founded}(Y))).$$

A sample query evaluation is

```
| ?- slgall(fnode(N),Anss).
Anss = [fnode(c)] ? ;
no.
```

Direct use of universal quantifiers in rule bodies proves convenient in many situations, even when the translation of a general logic program into a normal logic program preserves the alternating fixpoint semantics. An interesting feature is that the handling of universal rules is combined with the computation of stable models satisfying certain constraints to solve useful problems, although a formal notion of stable models for general logic programs is not yet defined. One possibility for the notion of stable models of general logic programs is to use the fixpoints of the *stability transformation* $\widetilde{S}_P$ on sets of ground negative literals of P. A systematic study of the stable model semantics for general logic programs is an interesting issue, but is beyond the scope of this paper.

Example 5.2. A proper coloring is such that no two vertices connected by an arc have the same color. Assuming a four-coloring problem, the following program assures that each vertex gets at most one color, and then uses `stselect` to select those stable models that represent proper colorings:

```
color(green). color(red). color(yellow). color(orange).
:- tabled color/2, unique_color/2, incon/0.
color(V,C) :- vertex(V), color(C), unique_color(V,C).
unique_color(V,C) <-- \+color(D); C=D; \+color(V,D).
incon :- arc(X,Y), color(X,C), color(Y,C).
qcolor(Coloring) :- stselect(color(_V,_C),[\+incon],Coloring,_SM).
```

The rule `incon` represents an inconsistency, whose negation is used for the selection of proper colorings.

6. CURRENT STATUS OF THE SYSTEM

All the functionalities discussed in this paper have been implemented using a meta interpreter on top of Prolog, with all `prolog` predicates executed in Prolog without loss of any efficiency.[1] Meta interpretation and the lack of mutable data structures in Prolog cause significant run time overhead for tabled evaluation. A prototype system that stores tables in C and uses a Prolog-C interface has been developed, which also avoids meta interpretation. Preliminary results indicate performance improvement of an order of magnitude over the meta interpreter implementation [19]. The prototype system has been extended to compute stable models of residual programs of queries.

A distinctive feature of our system is that it takes advantage of the mature Prolog technology and programming environments for nonmonotonic reasoning. Combined with integrated support for various mechanisms of nonmonotonic reasoning, the resulting system offers a more realistic testbed for applications of common sense reasoning.

The author thanks anonymous referees for their careful reading of the paper and helpful comments and suggestions and for bringing reference 7 to the author's attention. Example 2.1 is due to Michael Gelfond.

[1] The system is available over the World Wide Web at http://www.seas.smu.edu/~wchen/slg.htm.

REFERENCES

1. Bry, F., Logic Programming as Constructivism: A Formalization and its Application to Databases, in: *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ACM Press, New York, 1989, pp. 34–50.
2. Chen, W., Query Evaluation in Deductive Databases with Alternating Fixpoint Semantics, *ACM Trans. Database Systems* 20(3):239–287 (1995).
3. Chen, W., Swift, T., and Warren, D. S., Efficient Top-Down Computation of Queries under the Well Founded Semantics, *J. Logic Programming* 24(3):161–199 (1995).
4. Chen, W. and Warren, D. S., Query Evaluation under the Well-Founded Semantics, in: *The Twelfth ACM Symposium on Principles of Database Systems*, 1993.
5. Chen, W. and Warren, D. S., Tabled Evaluation with Delaying for General Logic Programs, *J. ACM* (1996). To appear.
6. Chen, W. and Warren, D. S., Computation of Stable Models and its Integration with Logical Query Processing, *IEEE Trans. Knowledge and Data Eng.* (1996). To appear.
7. Dix, J. and Mueller, M., The Stable Semantics and its Variants: A Comparison of Recent Approaches, in: L. Dreschler-Fischer and B. Nebel (eds.), *Proceedings of the 18th German Annual Conference on Artificial Intelligence (KI '94), Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, 1994, Vol. 861, pp. 82–93.
8. Dung, P. M. and Kanchanasut, K., A Fixpoint Approach to Declarative Semantics of Logic Programs, in: *North American Conference on Logic Programming*, The MIT Press, Cambridge, MA, 1989, pp. 605–625.
9. Dung, P. M., Negation as Hypotheses: An Abductive Foundation for Logic Programming, in: *International Conference on Logic Programming*, June 1991.
10. Eshghi, K. and Kowalski, R. A., Abduction Compared with Negation by Failure, in: *International Conference on Logic Programming*, 1989, pp. 235–254.
11. Gelfond, M., Logic Programming and Reasoning with Incomplete Information, *Ann. Math. Artificial Intelligence* 12:89–116 (1994).
12. Gelfond, M. and Lifschitz, V., The Stable Model Semantics for Logic Programming, in: R. A. Kowalski and K. A. Bowen (eds.), *Joint International Conference and Symposium on Logic Programming*, 1988, pp. 1070–1080.
13. Gelfond, M. and Lifschitz, V., Classical Negation in Logic Programs and Disjunctive Databases, *New Generation Computing* 9:365–385 (1991).
14. Kakas, A. C., Kowalski, R. A., and Toni, F., Abductive Logic Programming, *J. Logic Comput.* 2(6):719–770 (1993).
15. Lloyd, J. W., *Foundations of Logic Programming*, 2nd ed., Springer-Verlag, New York, 1987.
16. Marek, W. and Truszczynski, M., Autoepistemic Logic, *J. ACM* 38(3):588–619 (1991).
17. Przymusinski, T. C., On the Declarative and Procedural Semantics of Logic Programs, *J. Automated Reasoning* 5:167–205 (1989).
18. Przymusinski, T. C., Three-Valued Nonmonotonic Formalisms and Semantics of Logic Programs, *Artificial Intelligence* 49:309–343 (1991).
19. Ramesh, R. and Chen, W., A Portable Method of Integrating SLG Resolution into Prolog Systems, in: *International Logic Programming Symposium*, November 1994.
20. Saccà, D. and Zaniolo, C., Stable Models and Non-Determinism for Logic Programs with Negation, in: *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1990, pp. 205–217.
21. Satoh, K. and Iwayama, N., Computing Abduction by Using the TMS, in: *International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1991.

22. Satoh, K. and Iwayama, N., A Query Evaluation Method for Abductive Logic Programming, in: *Joint International Conference and Symposium on Logic Programming*, 1992.
23. Tamaki, H. and Sato, T., OLD Resolution with Tabulation, in: *International Conference on Logic Programming*, 1986, pp. 84–98.
24. Van Gelder, A., The Alternating Fixpoint of Logic Programs with Negation, *J. Computer System Sci.* 47:185–221 (1993).
25. Van Gelder, A., Ross, K. A., and Schlipf, J. S., The Well-Founded Semantics for General Logic Programs, *J. ACM* 38(3):620–650 (1991).