METHODS OF LOGIC IN COMPUTER SCIENCE 1, 61–76 (1994)

Declarative Semantics for Pruning Operators in Logic Programming

Fosca Giannotti

CNUCE–CNR
vis S. Maria 36, 56100 Pisa, Italy

Dino Pedreschi

Dip. di Informatica, Univ. di Pisa
Corso Italia 40, 56125 Pisa, Italy

Carlo Zaniolo

Computer Science Dept.—University of California
Los Angeles, California 90024, U.S.A.

ABSTRACT

Stable models, a concept from autoepistemic logic, have been recently proposed to define the semantics of logic programs with negation. A program may have several stable models, and this multiplicity can be exploited to characterize "don't care" nondeterminism, such as that arising from pruning operators. In the (bottom-up) context of deductive databases, a semantics for the nondeterministic *choice* construct was given by Saccà and Zaniolo. In this paper, we extend their approach to handle the cut-like pruning operators in the top-down evaluation context, inciuding the *one-of* and *commit* operators. The goal is accomplished by means of a series of program transformations. The transformed program is shown to have several stable models, each corresponding to a set of solutions obtained by top-down evaluation augmented with a nondeterministic pruning operator.

1 TOP-DOWN PRUNING OPERATORS AND BOTTOM-UP CHOICE OPERATORS

We can distinguish two main computational approaches to Horn clause logic. One is the top-down, resolution-based style of evaluation of Prolog-like languages. The other is the bottom-up, fixpoint-based style of evaluation of deductive databases and their Datalog-like languages.

Logic languages, both in the Prolog and Datalog families, provide different notions of nondeterminism which can be grouped into two different categories, "don't know" and "don't care" nondeterminism. The former notion is related to the fact that queries to a logic program may have many alternative answers, corresponding to alternative ways of proving the theorem represented by the query. In Prolog-like languages, this is usually accomplished by means of backtracking mechanisms, whereas in Datalog-like languages, "don't know" nondeterminism is usually concealed by the default "all solutions" behavior of the bottom-up evaluation.

"Don't care" nondeterminism is related to the fact that some answers to a query can be deliberately discarded. This form of nondeterminism is accomplished by *pruning operators* in logic programming and by *choice operators* in deductive databases.

The effect of top-down pruning operators is to cut off portions of the search space of the program. Examples of such operators are Prolog's *cut*, the *committed choice* operator of the parallel logic language PARLOG (see Clark and Gregory [1]), and the *one-of* construct proposed by Debray and Warren [2].

The effect of bottom-up choice operators is to arbitrarily extract, from the set of all answers to a query, some subset satisfying a given property. Examples of such operators are the *choice* construct proposed by Krishnamurti and Naqvi [3] for the Logic Data Language (LDL; see Naqvi and Tsur [4]), and the *witness* operator proposed by Abiteboul and Vianu [5]. For instance, the effect of LDL's *choice* is to choose from all the solutions to a query a maximal subset satisfying some functional dependency constraints among program variables. The following is an LDL program which uses the *choice* construct:

$$p(X,Y) \leftarrow q(X,Y),\ \text{choice}((X),(Y)).$$
$$q(a,1).$$
$$q(a,2).$$
$$q(b,1).$$

The query $\leftarrow p(X,Y)$ has two alternative sets of answers with respect to the above program, $\{p(a,1), p(b,1)\}$ and $\{p(a,2), p(b,1)\}$ corresponding to the two maximal subsets of the extension of q which satisfy the functional dependency $X \rightarrow Y$. The evaluation of such a query will return nondeterminstically one of those sets.

From the point of view of declarative semantics, "don't know" nondeterminism corresponds to the existence of a (the) model of the program, that is, a set from which answers can be drawn. On the other hand, there were no declarative characterizations for "don't care" nondeterminism until the recent proposal by Saccà and Zaniolo [6] for bottom-up choice operators. In their proposal, a program with choice constructs is modeled by an equivalent program with negation, which is shown to have several *stable models*. Thus,

"don't care" nondeterminism corresponds to selecting one of these possible models as the one where answers will be drawn. In other words, "don't known" means choosing *from a* model, whereas "don't care" means choosing *a* model.

However, no such declarative characterization is known for pruning operators of top-down languages, such as Prolog. Thus, the goal of this paper is to show how the stable model approach can be extended to deal with the various pruning operators of these languages.

We first consider a pruning construct *!!*, analogous to the *one-of* operator [2], and define a program transformation which eliminates *!!* using a refinement of the technique of Saccà and Zaniolo [6]. It will turn out that the stable model semantics of the transformed program faithfully characterizes the information which can be inferred using leftmost *SLD*-resolution augmented with the pruning mechanism. On this basis, it is possible to model the behavior of several top-down pruning operators like the *committed choice* construct and Prolog's *if-then-else*, up the full *cut*.

2 STABLE MODELS AND CHOICE OPERATORS

Stable model semantics has been introduced to deal with negation in logic programming by Gelfond and Lifschitz [7], with roots in the autoepistemic and default approaches to non-monotonic reasoning. Quoting from Moore [8], stable models correspond to "possible sets of beliefs that a rational agent might consistently hold."

The definition of stable model is based on that of *positive version* P_I of a logic program P with respect to an interpretation I. P_I is obtained from the ground version of program P by deleting:

1. each rule that has a negative literal $\neg B$ in its body, with $B \in I$, and
2. all negative literals in the bodies of the remaining rules.

An interpretation I is a *stable model* of P iff I is the minimal model of P_I. It is worth noting that such a minimal model always exists, as P_I is a negation-free program.

Logic programs may have zero, one, or many stable models. The program $p \leftarrow \neg p$ is the simplest example of a program with no stable models. Negation-free and stratified programs have exactly one stable model, as well as some nonstratified programs [7]. Other programs exhibit a multiplicity of stable models, like the following,

$$p \leftarrow \neg q$$
$$q \leftarrow \neg p$$

which has two stable models: one where p is true and q is false, and the other where p is false and q is true.

The basic intuition of [6] is to exploit such multiplicity as a declarative counterpart of "don't care" nondeterminism. Let us illustrate the approach by an example, while technical details are fully described in [6]. Consider again the LDL program:

$$p(X,Y) \leftarrow q(X,Y),\ \text{choice}((X),(Y)).$$
$$q(a,1).$$
$$q(a,2).$$
$$q(b,1).$$

and transform the clause for predicate p as follows:

$$p(X,Y) \leftarrow q(X,Y),\ \text{chosen}(X,Y).$$
$$\text{chosen}(X,Y) \leftarrow q(X,Y),\ \neg\text{diffchoice}(X,Y).$$
$$\text{diffchoice}(X,Y) \leftarrow \text{chosen}(X,Y1),Y \neq Y1.$$

Notice that this is equivalent to:

$$\text{chosen}(X,Y) \leftarrow q(X,Y),\ \neg(\exists\ Y1.\text{chosen}(X,Y1),Y \neq Y1).$$

which is a logical specification of a functional dependency on the relation q.

The transformed program turns out to have two alternative stable models with the following p-facts:

$$\{p(a,1),\ p(b,1)\} \text{ and } \{p(a,2),\ p(b,1)\}.$$

These correspond to the two possible sets of answers satisfying the functional dependency.

The above transformation can be suitably generalized to consistently replace the *choice* construct with negation, as shown in [6]. Moreover, a constructive definition for stable models is provided by the same authors, with the introduction of a procedure called *backtracking fixpoint*, that nondeterminstically constructs a stable model, if one exists.

It is worth noting that choice and pruning operators are closely related. In fact, the LDL *choice* operator captures the local pruning effect of the Prolog *cut*, that is the fact that only a solution of the goal preceding the *cut* is retained. For instance, a clause like

$$p(X,Y) \leftarrow q(X),!,r(Y).$$

can be rephrased as

$$p(X,Y) \leftarrow q(X),r(Y),\ \text{choice}((),(X)).$$

exploiting a special case of functional dependency to constrain X to be uniquely determined. On the other hand, LDL *choice* does not model the global pruning effect of Prolog *cut*, that is the pruning of the remaining clauses, and the dynamic pruning effect, that is the fact that different calls to the same clause may operate different choices. The following example illustrates the latter issue.

$$q(a).$$
$$q(b).$$
$$q(c).$$
$$p(X) \leftarrow q(X),!.$$
$$r \leftarrow p(a),p(b).$$

With respect to this program, the goal $\leftarrow r$ has a top-down derivation, whereas it cannot be deduced by bottom-up evaluation replacing *cut* with *choice*((), (*X*)) without violating the functional dependency. This observation highlights an intrinsic characteristic of bottom-up choice operators, which perform a single global nondeterministic choice over the extension of a relation, whereas top-down pruning operators may perform different choices at different calls, depending on query instantiations.

3 STABLE MODELS AND PRUNING OPERATORS

This section proposes an approach to the definition of a declarative semantics for top-down pruning operators. The technique is an extension of the one in [6], and as such it is based on stable model semantics and program transformation. The goal is establishing the following result. Assume that P^* is the program obtained via transformation from P, and that M_G is the set of atoms which can be inferred from the evaluation of a query G with respect to P using *SLD*-resolution augmented with some pruning mechanisms. Then M_G is a stable model of P^*. As a consequence, alternative evaluation trees for the same query due to different cuts operated by the pruning mechanism correspond to alternative stable models of the transformed program.

In our approach, the transformation plays a central role and is performed in two steps. First, each predicate in the program is extended with two extra arguments which gather information about the history of the top-down evaluation. In the second step, the pruning operators are replaced with appropriate functional dependency constraints among such histories. The next subsections show the technical details of the proposed construction.

3.1 Pruning and Top-down Evaluation

The Prolog *cut* operator has two distinct effects: a global one that affects a predicate definition as a whole, and a local one that affects the clause where it occurs. The global effect is that a *cut* prunes all program clauses below it. The local effect is that a *cut* prunes all alternative solutions for the conjunction of goals to the left of the cut. Thus, a conjunctive goal followed by a *cut* will produce at most one solution.

We will study logic programs containing the symbol *!!*, which denotes the *one-of* construct. This construct retains only the local behavior of Prolog *cut*.

The following definitions introduce *SLD*-resolution with the leftmost selection rule, termed *LD-resolution*, and its modification to handle the *one-of* construct, which is termed *LD!!-resolution*.

Definition 3.1. An *LD-derivation* for a given goal G is a *SLD*-derivation for G in which each resolvent G_i is derived from the previous one G_{i-1} by selecting the leftmost atom in G_{i-1}. Analogously are defined the notions of *LD-refutation* and *LD-tree* for a given goal.

The introduction of the pruning operator *!!* affects the definition of the tree of the derivations, as in certain points all derivations but one are pruned. The following presentation closely follows the style of Hill, Lloyd and Shepherdson [9].

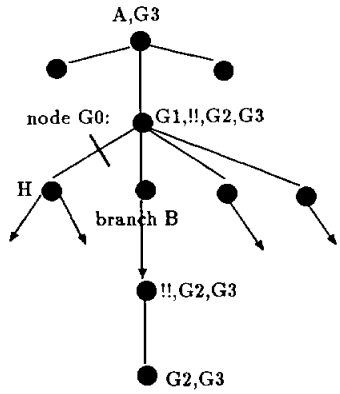

Figure 1. A pruning step.

Definition 3.2. An LD!!-tree *T!!* for a given goal G with respect to a program P that contains *!!* is an LD-tree T for G with respect to P where *!!* is viewed as an atom that always succeeds.

Definition 3.3. Let S be a subtree of an LD!!-tree. We say that the tree S' is obtained from S by a pruning step on S at G_0 if the following conditions are satisfied.

1. G_0 is a node of the form $(G_1,!!,G_2,G_3)$ which is a child of the node (A,G_3), such that there exists a branch B from G_0 to (G_2,G_3) in S;
2. H is a descendant of G_0 not on B;
3. S' is obtained from S by removing the subtree of S rooted at H.

G_0 is referred to as a *pruning node*.

With reference to the above definition, Figure 1 represents a pruning step applied onto an LD!!-tree.

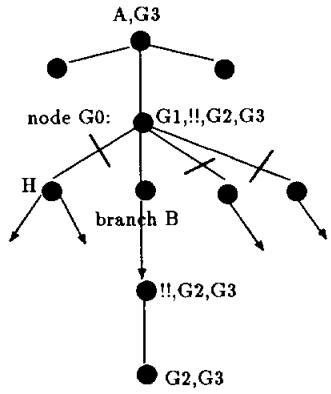

Figure 2. A final tree.

Definition 3.4. Let S be an LD!!-tree and S' a subtree of S. We say that S' is a pruned subtree of S if there exists a sequence of pruning steps which transforms S into S'. A pruned subtree S' of S is final if no further pruning steps can be applied to S'.

With reference to the above definition, Figure 2 represents a pruning sequence applied onto an LD!!-tree, yielding a final LD!!-tree.

It is worth noting that a final subtree S' of an LD!!-tree S satisfies the following property: For each node (B, G) in S with a child node $(G_1, !!, G_2, G)$, there exists at most one derivation leading to the node (G_2, G) in S'.

To illustrate the above points, consider the following program TC, expressing the transitive closure of relation q, modified by using the pruning mechanism:

$$p_0 : p(X,Y) \leftarrow q(X,Y).$$
$$p_1 : p(X,Y) \leftarrow q(X,Z),\ !!,\ p(Z,Y).$$

$$q_0 : q(a,b).$$
$$q_1 : q(a,c).$$
$$q_2 : q(b,d).$$
$$q_3 : q(c,e).$$

Figure 3 represents the LD!!-tree associated with the query $\leftarrow p(X,Y)$, while Figure 4 represents a final LD!!-tree for the same query, obtained by applying a sequence of pruning steps to the tree of Figure 3. Observe that, in the rightmost part of the final tree, only one of the possible bindings is selected among those that solve the q-subgoal preceding $!!$. As a consequence, the fact $p(a,d)$ is the only one that can be inferred by transitivity, while $p(a,e)$ is discarded by the pruning mechanism.

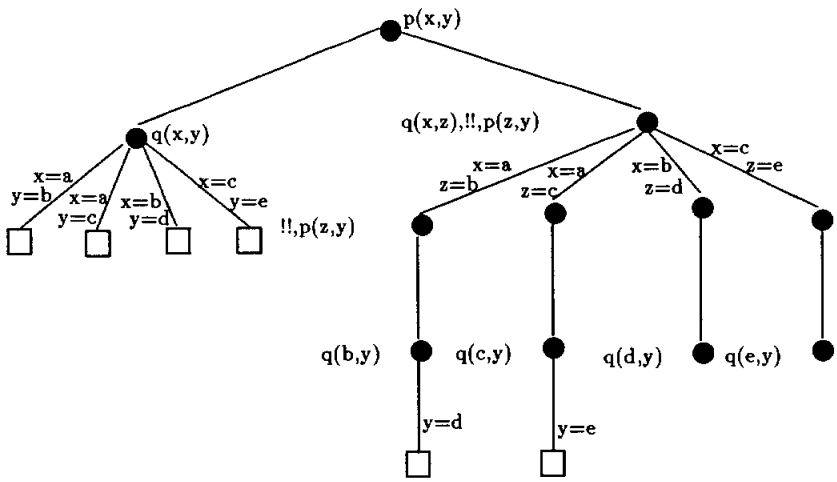

Figure 3. The *LD*!!-tree for the query *p(x,y)*.

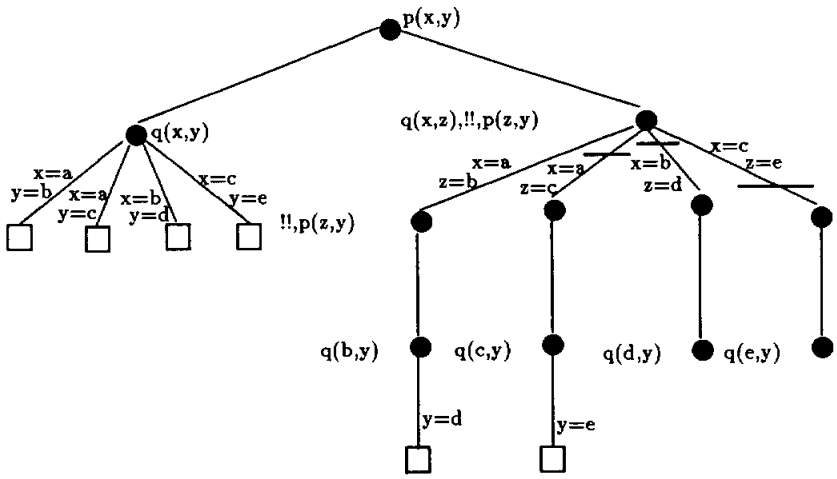

Figure 4. A final *LD*!!-tree for the query *p(x,y)*.

3.2 Augmented Programs

The next step of this construction introduces a program transformation aimed at keeping track of top-down evaluation inside the program itself. For this purpose, each predicate symbol is *augmented* with two extra arguments, *in* and *out*. The former represents the partial history of the evaluation preceding a predicate call; the latter represents the global history from the call to its success node. *In* and *out* are lists of new constant symbols which uniquely identify the program clauses.

The following notation will be adopted: Given an atom $A = p(t_1, \ldots, t_k)$, let $A_{[in,out]}$ denote the atom $p(t_1, \ldots, t_k, in, out)$. Analogously, given a goal $G = A_1, \ldots, A_n$, let $G_{[in,out]}$ denote the goal $A_{1[in,out_1]}, A_{2[out_1,out_2]}, \ldots, A_{n[out_{n-1},out]}$.

Definition 3.5. Given a program P we define its *augmented* version P_{io} as the following collection of clauses:

- If P contains the clause
  $r : A_0 \leftarrow A_1, \ldots, A_n$.
  then P_{io} contains the clause
  $r : A_{0[In,Out]} \leftarrow A_{1[[r|In],Out_1]}, A_{2[Out_1,Out_2]}, \ldots, A_{n[Out_{n-1},Out]}$.
- If P contains the fact clause
  $r : A$
  then P_{io} contains the fact clause
  $r : A_{[In,[r|In]]}$.

We now introduce some lemmas which point out the relationships between the original program and the augmented one. Lemma 3.1 states that a resolution step can be performed with respect to P iff it can be performed with respect to P_{io}. Lemma 3.1 is a straightforward consequence of Definition 3.5.

Lemma 3.1. The following two statements are equivalent:

- The goal G_i is derived from G_{i-1} in an LD-step with respect to program P by rewriting the leftmost atom A with $A_1, \ldots, A_n$ using clause r;
- The goal $G_{i[[r|In],Out]}$ is derived from $G_{i-1[In,Out]}$ in an LD-step with respect to program P_{io} by rewriting the leftmost atom $A_{[In,Out']}$ with $A_{1[[r|In],Out_1]}, \ldots, A_{n[Out_{n-1},Out']}$ using clause r.

Lemma 3.2 states that P and P_{io} are operationally equivalent and highlights the role of *In* and *Out* arguments in keeping track of the history of the refutation. Lemma 3.2 is a direct consequence of Lemma 3.1. Its proof is a routine induction on the length of the refutation for goal G, and is omitted.

Lemma 3.2. Suppose that the goal G has an LD-refutation with respect to program P using clauses $r_1, \ldots, r_n$ with computed answer substitution ϑ. Then the goal $G_{[In,Out]}$ has an LD-refutation with respect to P_{io} using the same clauses $r_1, \ldots, r_n$ with c.a.s. ϑ' such that:

- $\vartheta' = \vartheta$ with respect to the original variables of G,
- $In\vartheta' = In$,
- $Out\vartheta' = [r_n, r_{n-1}, \ldots, r_1|In]$.

Finally, Lemma 3.3 points out that the *In* list of each node in an LD-tree uniquely determines such a node. Lemma 3.3 follows from the observation that a node G in an LD-tree is determined by the sequence of clauses used in the derivation from the root node to G.

Lemma 3.3. Let S be the LD-tree for the goal $G_{0[[],Out]}$ with respect to P_{io}. Then, each node in S has the form $G_{[[r_1,\ldots,r_k],Out]}$, and the value $[r_1, \ldots, r_k]$ is unique in S.

With reference to the program *TC* of Section 3.1, the following augmented version TC_{io} is obtained:

$$p_0 : p(X,Y,In,Out) \leftarrow q(X,Y,[p_0|In],Out).$$
$$p_1 : p(X,Y,In,Out) \leftarrow q(X,Z,[p_1|In],Out_1),!!, p(Z,Y,Out_1,Out).$$

$$q_0 : q(a,b,In,[q_0|In]).$$
$$q_1 : q(a,c,In,[q_1|In]).$$
$$q_2 : q(b,d,In,[q_2|In]).$$
$$q_3 : q(c,e,In,[q_3|In]).$$

The LD!!-trees of Figures 3 and 4 can be now turned into trees for TC_{io}, as shown in Figure 5. Following the rightmost derivation in the final LD!!-tree of Figure 5, the query $\leftarrow p(X,Y,[],Out)$ succeeds with answer: $X = a$, $Y = d$, $Out = [q_2,p_0,q_0,p_1]$, where *Out* represents the sequence of clauses used in the derivation (in reverse order).

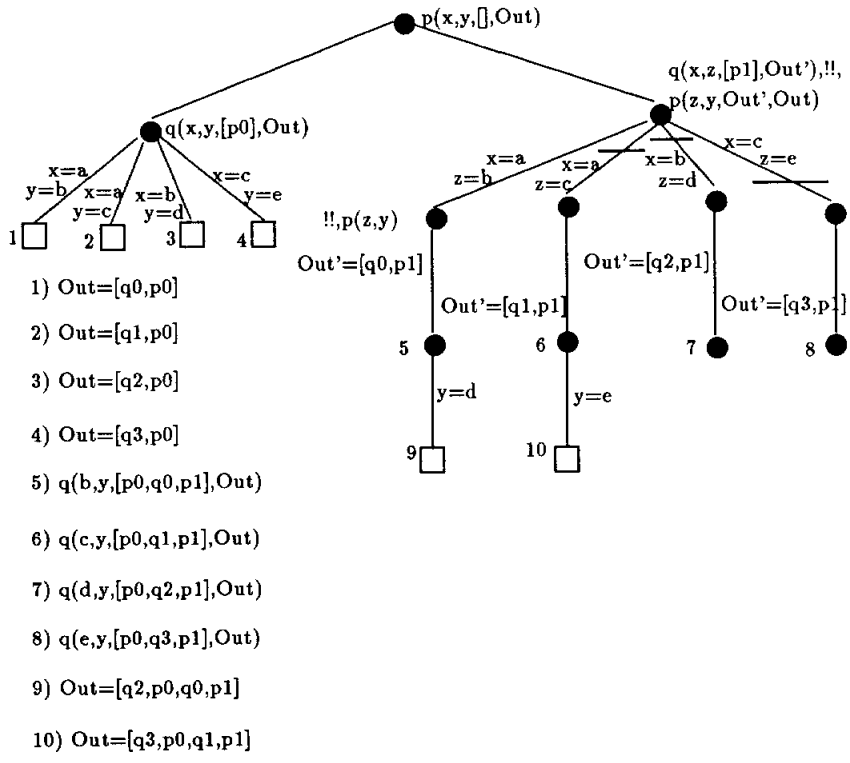

Figure 5. A final LD!!-tree for the query $p(x,y,[],Out)$ with respect to TC_{io}.

3.3 Removing the Pruning Operator

This section is aimed at introducing a declarative counterpart for programs with the *!!* construct. This is accomplished by a further transformation step from P_{io} which relies on the use of negation. The rationale behind the transformation is to replace the pruning operator with a relation that satisfies an appropriate functional dependency constraint among the possible histories of the program.

Definition 3.6. Given programs P and P_{io} as in Definition 3.5, we define the *cut-free* version P^* as the following collection of clauses:

1. If P_{io} contains the clause
   $r : A_{[In,Out]} \leftarrow G_{[[r|In],Out']}$, !!, $H_{[Out',Out]}$.
   then P^* contains the clauses:
   $r : A_{[In,Out]} \leftarrow G_{[[r|In],Out']}$, $chosen_r([r|In],Out')$, $H_{[Out',Out]}$
   $chosen_r([r|In],Out) \leftarrow G_{[[r|In],Out]}$, $\neg diffchoice_r([r|In],Out)$
   $diffchoice_r(In,Out) \leftarrow chosen_r(In,Out')$, $Out' \neq Out$;
2. If P_{io} contains the fact clause

$r : A_{[In,[r|In]]}$
then P^*
contains the clauses:
$r : A_{[In,[r|In]]} \leftarrow call_r(In)$
$call_r(In) \leftarrow \neg uncall_r(In)$
$uncall_r(In) \leftarrow \neg call_r(In)$

The idea behind this definition is that under the *stable model* interpretation of P^* the following properties hold.

- The extension of the relation $chosen_r$ is a set of pairs of histories satisfying the functional dependency between the first and the second argument.
- The extension of the relation $call_r$ is an arbitrary set of histories.

The driving intuition is that the models of such a program precisely correspond to final LD!!-trees with respect to the original program, where pruning has been performed. The functional dependency on the $chosen_r$ relation guarantees that in each choice point (i.e., pruning node) at most one choice is operated. More precisely, for each history reaching clause r, there is at most one continuation of such a history that verifies the part of the body which precedes the pruning operator. The situation is depicted in Figure 6.

On the other hand, $call_r$ captures the possibility that fact clauses with arbitrary incoming histories may be reached during evaluation.

With reference to the program TC of Section 3.1 and its augmented version TC_{io} of Section 3.2, the following cut-free version TC^* is obtained:

$p_0 : p(X,Y,In,Out) \leftarrow q(X,Y,[p_0|In],Out).$
$p_1 : p(X,Y,In,Out) \leftarrow q(X,Z,[p_1|In],Out'),chosen_{p_1}([p_1|In],Out'),p(Z,Y,Out',Out).$

$chosen_{p_1}([p_1|In],Out) \leftarrow q(X,Z,[p_1|In],Out), \neg diffchoice_{p_1}([p_1|In],Out).$
$diffchoice_{p_1}(In,Out) \leftarrow chosen_{p_1}(In,Out'),Out' \neq Out.$

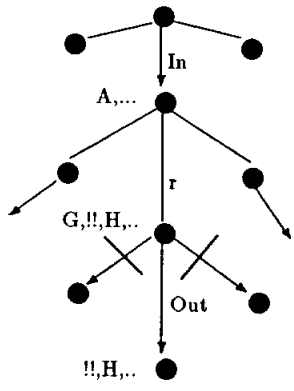

Figure 6. The functional dependency $[r|In] \rightarrow$ Out.

$$q_0 : q(a,b,In,[q_0|In]) \leftarrow call_{q_0}(In).$$
$$q_1 : q(a,c,In,[q_1|In]) \leftarrow call_{q_1}(In).$$
$$q_2 : q(b,d,In,[q_2|In]) \leftarrow call_{q_2}(In).$$
$$q_3 : q(c,e,In,[q_3|In]) \leftarrow call_{q_3}(In).$$

The definition of $call_{q_i}$ is as in Definition 3.6.2.

3.4 Stable Model Semantics for LD!!-Resolution

The next step in our construction is to define how to gather a set of facts from a final LD!!-tree such that this set is a stable model of P^*. Informally, this corresponds to collecting the facts associated with refutations in the final LD!!-tree for a given query.

Definition 3.7. Let G be a goal with respect to program P with associated final LD!!-tree S. Consider the LD!!-tree S_{io} associated to the goal $G_{[],Out]}$ with respect to P_{io}. Define M_G to be a set of atoms associated to S_{io} in the following way. For all refutations $< G_0, G_1, \ldots, G_{n-1}, \Box >$ in S_{io} with $G_0 = G$ and associated computed answer substitution ϑ:

1. $A\vartheta \in M_G$ for all atoms A in G_i, $i = 0, \ldots, n-1$;
2. if G_i is a pruning node of the kind $H_{[[r|In],Out']},!!,H'_{[Out',Out]}$, where r is the clause used to resolve G_i, then $chosen_r([r|In], Out')\vartheta \in M_G$ and $diffchoice_r([r|In],Out'')\vartheta \in M_G$ for all histories $Out'' \neq Out'\vartheta$;
3. if G_{i+1} is obtained from $G_{i[In,Out]}$ using the fact clause r, then $call_r(In)\vartheta \in M_G$.

Finally $uncall_r(In) \in M_G$ iff $call_r(In) \notin M_G$.

With reference to the final LD!!-tree of Figure 5 for the goal

$$G = \leftarrow p(X,Y)$$

from program TC, the set M_G is constructed as follows.

- The following facts are included in M_G by Definition 3.7.1, that is, by observing the successful derivations in the tree:

$p(a,b,[],[q_0,p_0])$, $q(a,b,[p_0],[q_0,p_0])$,
$p(a,c,[],[q_1,p_0])$, $q(a,c,[p_0],[q_1,p_0])$,
$p(b,d,[],[q_2,p_0])$, $q(b,d,[p_0],[q_2,p_0])$,
$p(c,e,[],[q_3,p_0])$, $q(c,e,[p_0],[q_3,p_0])$,
$p(a,d,[],[q_2,p_0,q_0,p_1])$,
$q(a,b,[p_1],[q_0,p_1])$,
$p(b,d,[q_0,p_1],[q_2,p_0,q_0,p_1])$,
$q(b,d,[p_0,q_0,p_1],[q_2,p_0,q_0,p_1])$.

- The following facts are included in M_G by Definition 3.7.2:

$chosen_{p_1}([\,p_1],[q_0,p_1])$,
$diffchoice_{p_1}([\,p_1],[q_1,p_1])$,
$diffchoice_{p_1}([\,p_1],[q_2,p_1])$,
$diffchoice_{p_1}([\,p_1],[q_3,p_1])$.

- The following facts are included in M_G by Definition 7.3:

$call_{q_i}([\,p_0])$, $i = 0, \ldots, 3$,
$call_{q_0}([\,p_1])$,
$call_{q_2}([\,p_0,q_0,p_1])$.

Moreover, $uncall_{qi}(in)$ is included in M_G if $call_{qi}(in)$ is not in the above list.

We are now ready to characterize the correspondence between final LD!!-trees with respect to P and stable models of P^*. This is accomplished by the two following results that conclude the construction of the previous subsections.

Theorem 3.1. Given a program P and a goal G with respect to P, M_G is a stable model of P^*.

Proof. To establish the theorem we show that the least fix point of the positive version P_{M_G} of P^* with respect to M_G coincides with M_G. P_{M_G} is defined as follows. All the ground instances of the original clauses of P_{io} are in P_{M_G}. The fact $call_r(in)$ is in P_{M_G} iff $uncall_r(in)$ is not in M_G, that is iff $call_r(in)$ is in M_G. Analogously, the fact $uncall_r(in)$ is in P_{M_G} iff $uncall_r(in)$ is in M_G. Finally, the rule instances $chosen_r([r|in],out) \leftarrow G_{[[r|in],out]}$ are in P_{M_G} iff the fact $diffchoice_r([r|in],out)$ is not in M_G, while the rule instances $diffchoice_r(in,out) \leftarrow chosen_r(in,out')$ are in M_G iff $out' \neq out$. The proof is composed by the two following steps.

1. $fix(P_{M_G}) \subseteq M_G$ (by induction on the fixpoint iterations.)
   Base case. The only facts in the program P_{M_G} are the $call_r$ and $uncall_r$ ones, and hence they belong to M_G by construction of P_{M_G}.
   Inductive case. Let A be an atom inferred at the n^{th} fixpoint iteration. The following cases arise.

   (a) A is inferred using the clause
   $r : A_{[in,[r|in]]} \leftarrow call_r(in)$
   corresponding to a fact rule in P, with $call_r(in)$ in the previous fix point iteration. Hence, by induction hypothesis $call_r(in) \in M_G$. By Definition 3.7.3, the fact clause r has been used in a refutation from the goal G with *entry* history *in*. Hence, $A_{[in,[r|in]]}$ occurs in a refutation, and by Definition 3.7.1 it belongs to M_G.

   (b) A is inferred using the clause
   $r : A_{[In,Out]} \leftarrow G_{[[r|In],Out']}, chosen_r([r|In],Out'), H_{[Out',Out]}$,
   corresponding to a *!!*-rule in P, whose body belongs to the previous fixpoint iteration. Hence, by induction hypothesis each atom in the body belongs to M_G.

By Definition 3.7.2 the body corresponds to a pruning node in a refutation from G and it has been generated using the clause r with entry history in. Hence, $A_{[In,Out]}$ occurs in a refutation and by Definition 3.7.1 belongs to M_G.

(c) A is inferred using a clause
$chosen_r(in,out) \leftarrow G_{[in,out]}$
with $G_{[in,out]}$ in the previous fixpoint iteration. Hence, each atom of $G_{[in,out]}$ belongs to M_G and then $G_{[in,out]}$ occurs in a refutation. Since $G_{[in,out]}$ is the portion of the body of rule r preceding $!!$, by Definition 3.7.2 $chosen_r(in,out) \in M_G$.

(d) A is inferred using a clause
$diffchoice_r(in,out) \leftarrow chosen_r(in,out')$,
with $out \neq out'$. Then, $chosen_r(in,out') \in M_G$ and, by Definition 3.7.2, $diffchoice_r(in,out) \in M_G$.

2. $fix(P_{M_G}) \supseteq M_G$ (by contradiction)
Suppose $A \in M_G$ and $A \notin fix(P_{M_G})$. Then clearly A is not an instance of a fact rule in P_{io}, otherwise A would be inferred in the second fixpoint iteration, according to Definition 3.6.2. Hence A is an instance of the head of a rule as in Definition 3.6.1, and each (instance of) atom in the body of such a rule also belongs to M_G. Clearly, at least one such atom does not belong to $fix(P_{M_G})$, otherwise A would also belong to the fix-point. This construction can be repeated indefinitely, eventually generating a contradiction, since every refutation is finite. □

Theorem 3.1 provides a notion of soundness of LD!!-resolution with respect to the stable model semantics of P^*. In fact, Theorem 3.1 establishes a strong notion of soundness, in that it states not only that the information deduced using LD!!-resolution is justified by the presence of a suitable stable model, but also that such a model completely characterizes the computation.

On the other hand, many alternative stable models of P^* exist and it is natural to wonder whether all such models can be *computed* using LD!!-resolution. The simple result below (Theorem 3.2) states that for each fact which is true in an arbitrary stable model, a suitable LD!!-tree exists, which allows one to deduce such a fact. Thus Theorem 3.2 provides a (weak) notion of completeness of LD!!-resolution with respect to the stable model semantics of P^*.

Theorem 3.2. Given a program P and a stable model M of P^*, for each $A \in M$ there exists a goal G with respect to P such that $A \in M_G$.

Proof. The theorem is easily established by the following observations.

Consider an atom $A_{[in,[r_k, \ldots, r_1|in]]} \in M_{P_{io}}$, that is, the minimal model of the positive program P_{io} (notice that $M_{P_{io}}$ is the unique stable model of P_{io}). By Lemma 3.2 we have that A has an LD-refutation which uses clauses $r_1, \ldots, r_k$.

Consider next the relationship between $M_{P_{io}}$ and an arbitrary stable model M of P^*. As a consequence of Definition 3.6, M corresponds to (the atoms in) a subset of the refutations which are represented in $M_{P_{io}}$ (actually, it corresponds to a maximal subset of $M_{P_{io}}$ which satisfies the given functional dependency among histories [6].) This implies that

$A_{[in,[r_k,\ \ldots\ ,\ r_l|in]]}$ has an LD-refutation in P_{io}, and hence A has an LD-refutation in P by the above observation. We can choose such a refutation as the LD!!-refutation for the goal $\rightarrow A$, and hence $A \in M_{\rightarrow A}$. □

4 OTHER PRUNING OPERATORS

In this final section we briefly outline how the proposed framework can be adapted to deal with other pruning mechanisms. A notable example is the *committed choice* operator of concurrent logic languages like PARLOG [1], also adopted by Hill, Lloyd and Shepherdson [9] in a sequential language.

Informally, the behavior of *committed choice* at clause level is the same as that of the *one-of* operator, whereas it has also the global effect of pruning all the other matching clauses.

The treatment of the *committed choice* derives directly from that of the *one-of* operator. To cope with the latter, in a clause:

$$r : A_{[In,Out]} \rightarrow G_{[[r|In],Out']},!!,H_{[Out',Out]}$$

the functional dependency $[r|In] \rightarrow Out'$ was adopted. To cope with *committed choice*, denoted by "|", in a clause

$$r : A_{[In,Out]} \leftarrow G_{[[r|In],Out']}|H_{[Out',Out]}$$

the functional dependency $In \rightarrow Out'$ is sufficient. This expresses the fact that for each history reaching the *parent* of clause r there is at most one continuation of such history which verifies the part of the body preceding the *committed choice*. This situation is depicted in Figure 7.

It is natural to extend the approach to deal with Prolog *if-then-else* and (red) *cut*, since these mechanisms can be rephrased in terms of the *one-of* operator and negation. For

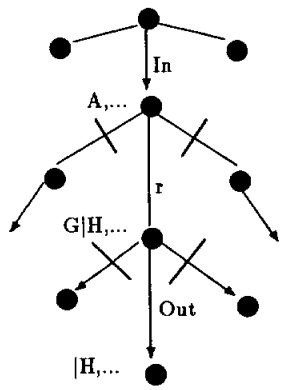

Figure 7. The functional dependency *In* → *Out*.

instance, an *if-then-else* goal: $(A \rightarrow B; C)$ can be replaced by a new goal *ite* defined by the two clauses

$$ite \leftarrow A,!!,B$$
$$ite \leftarrow \neg A,C.$$

The treatment of *cut* requires a bit more drudgery, in as much as negation must be used to represent the textual clause ordering in the program. Appealing, but still in progress, is the investigation of how these principles may apply to the form of nondeterminism present in production systems, which is the bottom-up equivalent of that of *committed choice* languages.

REFERENCES

[1] K.L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 1, 1986, pp. 1–49.

[2] S.K. Debray and D.S. Warren, "Towards Banishing the Cut from Prolog," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, 1990, pp. 335–349.

[3] R. Krishnamurthy and S.A. Naqvi, "Non-Deterministic Choice in Datalog," *Proceedings of the Third International Conference on Data and Knowledge Bases*, 1988, pp. 416–424.

[4] S.A. Naqvi and S. Tsur, *A Logical Data Language for Data and Knowledge Bases*, Computer Science Press, New York, 1989.

[5] S. Abiteboul and V. Vianu, "Fixpoint Extensions of First-Order Logic and Datalog-like Languages," *Proceedings of the Fourth Symposium on Logic in Computer Science (LICS)*, 1989, pp. 71–89.

[6] D. Sacca and C. Zaniolo, "Stable Models and Non-Determinism in Logic Programs with Negation," *Proceedings of the Symposium on Principles of Database Systems: PODS '90*, 1990, pp. 205–217.

[7] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming," *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, 1988, pp. 1070–1080.

[8] R. Moore, "Semantical Considerations in Nonmonotonic Logic," *Artificial Intelligence*, Vol. 25, No. 1, 1985, pp. 75–94.

[9] P.M. Hill, J.W. Lloyd, and J.C. Sheperdson, *Properties of a Pruning Operator*, Technical Report, TR-89-18, Computer Science Department, University of Bristol, Bristol, UK, 1989.