PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

LOCAL DEFINITIONS WITH STATIC SCOPE RULES IN LOGIC PROGRAMMING

L.Giordano A.Martelli G.F.Rossi

Dipartimento di Informatica - Universita' di Torino
C.so Svizzera 185 - 10149 TORINO (ITALY)

ABSTRACT

The paper deals with the problem of extending positive Horn clause logic by introducing blocks, that is local definitions of clauses, as a tool for structuring programs. The extension consists in allowing a goal G_i in a clause $G_1 \wedge .. \wedge G_n$ -> A to be not only an atom but also a pair P=>G, where P is a set of clauses and G a goal. Similar proposals have already appeared in the literature, mainly to deal with hypothetical reasoning. An analysis of these proposals shows that clauses have dynamic scope rules, because to derive a goal P=>G in a program P' they derive G in the program P $\cup$ P', thus making all clauses of P accessible to all clauses of P'. Gabbay showed that, in a model theoretic semantics, this corresponds to interpreting both implications -> and => as intuitionistic implication. In this paper we analyse static scope rules, where a goal can refer only to clauses defined in statically surrounding blocks, and we argue that this kind of block is a more natural extension of Horn clauses when used as a programming language. We show it by defining an operational, fixpoint and model theoretic semantics which are extensions of the standard ones, and by proving their equivalence. We show also that static scope rules can be obtained by interpreting -> as classical and => as intuitionistic implication, with respect to Herbrand interpretations. Finally some more concrete operational semantics are presented to show that the proposed construct can be easily implemented.

1 INTRODUCTION

Logic programming is widely recognized as a useful paradigm for solving many classes of problems. However pure Horn clause logic has severe limitations, which have led to numerous proposals of extensions. In particular one of the main hindrances to the use of Horn clauses as a programming language is the lack of constructs, such as modules or blocks, to structure programs.

In this paper we focus on the problem of extending positive Horn clause logic by introducing blocks, that is local declarations of clauses. The concept of *block* is well known in the area of traditional imperative or functional languages, and we argue that it can be usefully introduced in logic programming as well. Several proposals for dealing with local definitions have already appeared in the literature. For instance, Bowen and Kowalski (1982) show how to introduce local definitions at the metalevel, whereas Warren (1984) proposes a modal operator "assume". Gabbay and Reyle (1984,1985) present N_Prolog, an extension of logic programming which allows local definitions and which is designed mainly to deal with hypothetical reasoning. A similar extension is proposed by Miller (1986) and more recently by McCarty (1988). Nait Abdallah (1986) defines ions to deal with local definitions.

This work has been partially supported by MPI 40%

Informally, local definitions can be added to a logic program by allowing a goal G_i in a clause $G_1 \wedge .. \wedge G_n$ -> A to be not only an atom but also a *block*, D=>G, where D is a set of clauses and G a goal. Indeed the set of clauses D corresponds to local procedure declarations in conventional programming languages. The usual operational semantics of local definitions is that a goal D=>G can be solved in a program P if the goal G can be solved in the program P $\cup$ D. For instance, to solve the goal a=>b in the program {a->b}, we solve b in the program {a->b, a}. This block structure is adequate to support *hypothetical reasoning*, i.e. by assuming a and knowing a->b we can deduce b.

As another example, let us consider the following program:

Example-1.

P={r->q,
(((q->p)$\wedge$r)=>p)->s}.

The proof of the goal s in P yields
goal (((q->p)$\wedge$r)=>p) in P
goal p in P'=P$\cup${q->p,r}
goal q in P'
goal r in P'
which succeeds.

From a logical viewpoint, a block D=>G can be considered as an implication, but it is well known that this implication cannot be the classical one. For instance, in classical logic (a->b)->a |- a, whereas a cannot be derived from (a=>b)->a according to the previously given informal operational semantics. Gabbay and Reyle have shown that, instead, by having a unique implication symbol $\supset$ in place of both => and -> in goals and clauses, the interpretation of $\supset$ as the *intuitionistic implication* corresponds to the operational semantics, and have given a model-theoretic semantics based on worlds. Miller has given a fixpoint semantics for the same implication.

Although the semantics of local definitions described above appears to be quite natural, it is by no means the only possible way to deal with them. A block structured language requires the introduction of *scope rules* specifying visibility rules for locally defined clauses and variables. Two alternatives are feasible as usual, namely static scope rules or dynamic ones.

The use of explicit quantifiers allows to give static scope rules to variables. However, scope rules for clauses are the main concern of this paper. It is worth to notice that scope rules for clauses can be more complex than scope rules for procedures in conventional programming languages, because in a logic program a predicate definition is usually given by means of several clauses which can be scattered throughout the program in different blocks.

It is possible to see that the above semantics requires *dynamic* scope rules. In fact, given the goal D=>G to be proved in the program P, after the program P has been enlarged by adding the clauses in D, they are no more distinguishable from other clauses of the program and can be used in the subsequent refutation as global clauses. The added clauses are no more visible as soon as the proof of the goal G terminates (i.e. they are removed from the set of global clauses). Therefore, the set of clauses which can be used to solve a goal G depends on the sequence of goals generated till that moment in the proof containing G. Of course, this set can be determined only dynamically. For instance, the proof of the goal q in Example-1 uses the first clause (r->q) of P and the clause r which is local to the second clause of P.

In this paper we pursue the idea of defining a logic language with *static* scope rules for clause definitions, in which, as in most programming languages, the rules for using a clause are determined by the static nesting of blocks in the program text. In this way to solve an atomic goal which comes from the body of a clause declared in a block, only the clauses declared in that block or in external enclosing blocks can be used. Therefore the set of clauses which can be used in the refutation of a goal depends only on the block structure of the program and can be statically determined. For instance, Example-1 would fail with static scope rules because the clause r, defined locally to the second clause of P, is not visible from the first clause of P. On the contrary, the following example will succeed with static scope rules:
Example-2.
G=s
P={q,
$(((r\wedge q\text{->}p)\wedge r)\text{=>}p)\text{->}s\}$,
the goal G succeeds from the program P, since in this case r is used in the same block where it is defined and q is used from an inner block.

Static scope rules have the well known advantages, to be discussed at the end of the paper, to allow more efficient implementations by allowing compilation of procedure calls. Thus, this kind of block appears to be a suitable extension of Horn clauses used as a programming language. On the other hand, we remark that hypothetical reasoning cannot be carried on in this case. The idea to add the standard programming language concept of block to logic programming languages has already been proposed in (Martelli and Rossi 1986), mainly from an implementation viewpoint. In this paper we define blocks more rigorously.

In the next section we define more precisely the language with statically scoped blocks by means of its *operational semantics*. We show there that a unique implication connective is not sufficient, as in the case of dynamic scope rules. Thus we must introduce two different implications with different meanings: one in definite clauses ($G_1\wedge..\wedge G_n$->A) and the other one in goals (D=>G).

In section 3 we define the *fixpoint semantics* of the language as an extension of the standard fixpoint semantics of Horn clauses. Instead of defining the semantics of a program P as a subset of the Herbrand base B(P), we define it as a function from $2^{B(P)}$ to $2^{B(P)}$. The operational semantics is shown to be sound and complete with respect to this fixpoint semantics.

An interesting problem is to see if this kind of language allows also a *model-theoretic semantics*. Indeed in section 4 we show that a very simple semantics can be given as an extension of standard semantics by defining satisfiability of formulas with respect to interpretations (subsets of B(P)). Satisfiability is defined as in the standard case (so -> is the classical implication) with the addition of the definition for D=>G, which is satisfiable in an interpretation I if and only if D implies G in all the interpretations I' which contain I. Thus we do not need a semantics with worlds as in the case of N_Prolog, whose model-theoretic semantics is that of intuitionistic logic. This kind of semantics possesses properties analogous to those of the classical case, such as a minimal model which can be shown to be equal to the least fixpoint.

It is easy to see that the model-theoretic semantics of => given in section 4 is different in general from the intuitionistic semantics. However we show in section 5 that the two semantics coincide for the language of the paper with respect to Herbrand interpretations, and therefore we can consider -> and => as the classical and intuitionistic implication respectively.

In sections 6 and 7 we introduce more concrete operational semantics for the language, an we give some hints on the possible efficient implementations of this language, in the style of conventional programming languages.

2 THE LANGUAGE AND ITS OPERATIONAL SEMANTICS

In this section we define a logical language which extends positive Horn clause logic by introducing blocks, that is local declarations of clauses. To describe the syntax and the semantics of this language we shall use the notation of (Miller 1986).

Let A, G and D be metalinguistic variables which represent, respectively, atomic formulas, goals and definite clauses and let T be a propositional constant (true). The syntax of the language is the following:

$$G := T \mid A \mid G_1\wedge G_2 \mid \exists x G \mid D\text{=>}G$$
$$D := G\text{->}A \mid D_1\wedge D_2 \mid \forall x D .$$

A *program* is defined as a set of closed definite clauses.

Notice that what we call clauses are actually not standard clauses, since they can be composed of a conjunction of clauses and the left-hand part of a clause G->A is allowed to contain implications. Notice also that a clause of the form T->A will be simply written as A.

As an example, let us consider the well known logic program which implements the quicksort algorithm. First we present the usual Prolog implementation and then the corresponding implementation in the language we have defined so far. We use a Prolog-like notation in which "," is used in place of "∧" to represent a conjunction of goals and a program is a sequence of clauses separated by ".".

Example-3.

```
split(H,[A|X],[A|Y],Z) <-
          order(A,H),split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]) <-
          order(H,A),split(H,X,Y,Z).
split(_,[ ],[ ],[ ]).

quicksort([H|T],S) <-
          split(H,T,A,B),
          quicksort(A,A1),
          quicksort(B,B1),
          append(A1,[H|B1],S).
quicksort([ ],[ ]).
```

Since predicate "split" is used only by "quicksort", we can move its definition inside the body of "quicksort" in an inner block declaration as follows:

```
∀H,T,S quicksort([H|T],S) <-
  ∃A,B,A1,B1
      ( (∀A,X,Y,Z split([A|X],[A|Y],Z) <-
                    order(A,H), split(X,Y,Z)).
         ∀A,X,Y,Z split([A|X],Y,[A|Z]) <-
                    order(H,A), split(X,Y,Z)).
         split([ ],[ ],[ ]) ) => split(T,A,B) ),
       quicksort(A,A1),
       quicksort(B,B1),
       append(A1,[H|B1],S) ).
quicksort([ ],[ ]).
```

Variables can be explicitly quantified allowing local variables to be distinguished from global ones. For instance, the scope of variable H is the whole clause defining "quicksort"; thus H can be used in the body of the "split" procedure as a global variable and must not be specified as a parameter of the "split" procedure itself (the procedure has now one parameter less than the previous above definition). Variables can be quantified for each clause and goal. For instance, we have chosen to quantify variables used in the body of "quicksort" inside the body itself using an existential quantifier, whereas variables which are local to the definition of "split" are universally quantified for each clause of the procedure.

Given a program P and a closed goal G, we want now to define the meaning of G being *operationally derivable* from P, that is P |- G. To cope with the situation in which a goal is an implication (i.e. a block), we have to consider *lists of programs* $P_1|..|P_n$ instead of simply programs. The list is used to record the ordering relation among sets of clauses (i.e. programs) which are introduced during the computation of a goal. In a list $P_1|..|P_n$, P_1 is the initial program while each P_i, for i>1, is the conjunction of clauses contained in the D_i of a block declaration D_i=>G_i. As higher is the index i as deeper is the nesting of the block D_i=>G_i. Thus the list $P_1|..|P_n$ represents the static nesting of blocks in a program P, at some step of a possible derivation of the given goal G from P. On the contrary, with dynamic scope rules the union of clauses in P and D_i is considered for each goal D_i=>G_i, so that local definitions are no more distinguishable from global ones.

In order to avoid problems with variable renaming and substitutions we follow the approach of (Miller 1986) replacing universally quantified variables in a program with all their possible substitutions (an operational semantics with substitutions is given in section 6). Moreover, conjunctions of clauses are replaced by the corresponding set of clauses. The program which is obtained from P in such a way is denoted by [P]. [P] can be defined recursively as the smallest set of formulas such that:

(i) P⊑[P];
(ii) if $D_1 \wedge D_2 \in [P]$ then $D_1 \in [P]$ and $D_2 \in [P]$;
(iii) if $\forall x D \in [P]$ then $[x/t]D \in [P]$ for all closed term t.

We define the *derivability* of a closed goal G from a not empty list of programs $P_1|..|P_n$ by induction on the structure of G, with the following rules:

(1) $P_1|..|P_n$ |- T;
(2) if A is a closed atomic formula, $P_1|..|P_n$ |- A iff, for some i, there is a formula G->A∈ $[P_i]$ and $P_1|..|P_i$ |- G;
(3) $P_1|..|P_n$ |- $G_1 \wedge G_2$ iff $P_1|..|P_n$ |- G_1 and $P_1|..|P_n$ |- G_2;
(4) $P_1|..|P_n$ |- ∃xG iff there is some closed term t such that $P_1|..|P_n$ |- [x/t]G;
(5) $P_1|..|P_n$ |- D=>G iff $P_1|..|P_n|D$ |- G.

Consider first rule (2): when a clause G->A in P_i is used to refute an atomic goal A, then the clauses in $P_{i+1},..,P_n$ cannot be used any more to prove G. This is because the blocks corresponding to $P_{i+1},..,P_n$ do not contain the block from which G is called and therefore are not visible from G. As we can see from rule (5), when the goal is a block D=>G, then the set of local declarations D is added to the list of programs as the tail element and G is proved from the resulting list of programs, so that the clauses in D can be used only to refute goals which come from D itself or from G.

We define a *derivation* of G from a not empty list of programs $P_1|..|P_n$ as a finite sequence of pairs (W_1,G_1),....,(W_m,G_m), where $W_1=P_1|..|P_n$, G_1=G, G_m=T, and for i=1,..,m, W_i |- G_i. The derivability of G_i from W_i can be obtained from the members of this sequence which follow (W_i,G_i), using the above rules (how many members of the sequence must be considered depends on the rule which is applied).

Example-4. Let us consider the program P of Example-2. The following is a derivation of the goal G=s from P:

G_1=s, $W_1=P_1$={T->q, (((r∧q->p)∧(T->r))=>p)->s}
by rule (2): G_2=((r∧q->p)∧(T->r))=>p, $W_2=P_1$
by rule (5): G_3=p, $W_3=P_1|P_2$, P_2=(r∧q->p)∧(T->r)
by rule (2): G_4=r∧q, $W_4=P_1|P_2$
by rule (3): G_5=q, $W_5=P_1|P_2$
by rule (2): G_6=T, $W_6=P_1$
by rule (3) applied to (G_4,W_4): G_7=r, $W_7=P_1|P_2$
by rule (2): G_8=T, $W_8=P_1|P_2$.

If the first clause of P is replaced by r->q, then the derivation of G=s from P is no more feasible. In fact, in this case we have: G_6=r, $W_6=P_1$, and P_1 does not contain the definition of r, which, on the contrary, is defined in the inner block P_2 and therefore is not visible at this point.

In the next sections we'll present the fixpoint and model-theoretic semantics for our language and we'll prove the equivalence between the operational semantics and the fixpoint semantics and between the fixpoint and the model-theoretic semantics. All these semantics are defined by extending the corresponding standard semantics given for positive Horn clause logic (see (Apt and van Emden 1982) and (Lloyd 1984)).

3 FIXPOINT SEMANTICS

Given a program P, let U(P) be the Herbrand Universe for P, that is the set of all ground terms that can be formed out of the constant and functional symbols occurring in P and let B(P) be the Herbrand base for P, that is the set of all ground atoms which can be formed by using predicates of P and terms in U(P). An *Herbrand interpretation* for P is a subset of B(P). The set of all Herbrand interpretations for P (the power set of B(P)) is a complete lattice under inclusion, with B(P) as the top element and $\varnothing$ as the bottom element.

Let I and X be Herbrand interpretations for P. We define a mapping $T_{P,I}$ from the lattice of Herbrand interpretations to itself as follows:

$$T_{P,I}(X) = I \cup \{ A \in B(P) : \text{there is a } G\text{->}A \in [P] \text{ and } X \} G \},$$

where $\}$ is the weak relation of satisfiability between Herbrand interpretations and closed goals and is defined as follows:

- $X \} T$
- $X \} A$ iff $A \in X$
- $X \} G_1 \wedge G_2$ iff $X \} G_1$ and $X \} G_2$
- $X \} \exists xG$ iff $X \} [x/t]G$ for some $t \in U(P)$
- $X \} D \text{=>} G$ iff $T^{\omega}_{D,X}(\varnothing) \} G$.

It can be proved (see (Giordano et al. 1988)) that $T_{P,I}$ is *monotone* and *continuous*, and therefore that it has a least fixpoint $\mathrm{lpf}(T_{P,I}) = \cup_{k=0}^{\omega} T^{k}_{P,I}(\varnothing)$ ($T^{\omega}_{P,I}(\varnothing)$ for short). The semantics of a program P, in this case, is a mapping from an interpretation X (a subset of B(P)) to another interpretation, namely the least fixpoint of $T_{P,I}$, $T^{\omega}_{P,I}(\varnothing)$.

The set I in $T_{P,I}$ is intended to convey all the necessary informations about the enclosing environment of the program P. Such informations are required since in our language a program P can be a subcomponent of a larger program P', that is P can occur in a block inside P'. Since a Herbrand interpretation consists of a set of ground atomic formulas on the Herbrand Universe, we can consider such an interpretation as an environment which associates with each predicate symbol in the program a denotation, that is the set of tuples of terms for which the predicate is true. Hence, there is an immediate parallel with standard programming languages, whose denotational semantics is defined as a mapping between environments.

We shall prove that $T^{\omega}_{P,\varnothing}(\varnothing)$ is the set of all ground atomic formulas operationally derivable from P, and, more generally, using the relation of weak satisfiability, that

$$T^{\omega}_{P,\varnothing}(\varnothing) \} G \text{ iff } P \mid\!- G,$$

namely, that the fixpoint semantics is equivalent to the operational semantics. To prove this equivalence, we prove separately soundness and completeness of the operational semantics with respect to the fixpoint semantics.

Theorem 1 (Soundness). Let P be a program and G be a closed goal.

$$P \mid\!- G \Rightarrow T^{\omega}_{P,\varnothing}(\varnothing) \} G.$$

Proof. If $P \mid\!- G$ then there is a derivation $(W_1,G_1),\ldots,(W_n,G_n)$, where $W_1=P$, $G_1=G$ and $G_n=T$. We prove by induction on $i<n$ that, if $W_{n-i}=P_1|..|P_m$, for some m, then $T^{\omega}_{Pm,Ym-1}(\varnothing) \} G_{n-i}$, where $Y_0=\varnothing$ and $Y_j=T^{\omega}_{Pj,Yj-1}(\varnothing)$. Thus for $i=n-1$ we have that $T^{\omega}_{P,\varnothing}(\varnothing) \} G$.

• If $i=0$ the thesis holds trivially.
• We assume that the thesis holds for $i<k$ and prove it for $i=k$ considering all possible cases for G (double induction).
- If $G_{n-k}=T$, the thesis obviously holds.
- If $G_{n-k}=A$ then, for some $j \le m$, there is a $G\text{->}A \in [P_j]$ and, for some $h<k$, $G_{n-h}=G$ and $W_{n-h}=P_1|..|P_j$. By inductive hypothesis $T^{\omega}_{Pj,Yj-1}(\varnothing) \} G$. So, by definition of $T_{Pj,Yj-1}$, $A \in T_{Pj,Yj-1}(T^{\omega}_{Pj,Yj-1}(\varnothing)) = T^{\omega}_{Pj,Yj-1}(\varnothing)$. Moreover, for all j, $Y_j \subseteq T^{\omega}_{Pj,Yj-1}(\varnothing)=Y_{j+1}$ and therefore, since $j \le m$, $T^{\omega}_{Pj,Yj-1}(\varnothing)) \subseteq T^{\omega}_{Pm,Ym-1}(\varnothing)$. Thus $A \in T^{\omega}_{Pm,Ym-1}(\varnothing)$.
- If $G_{n-k}=G_1 \wedge G_2$ then there are two non-negative integers $h,j<k$ such that $G_{n-h}=G_1$, $G_{n-j}=G_2$ and $W_{n-h}=W_{n-j}=W_{n-k}$. By inductive hypothesis, $T^{\omega}_{Pm,Ym-1}(\varnothing) \} G_1$ and $T^{\omega}_{Pm,Ym-1}(\varnothing) \} G_2$. Thus $T^{\omega}_{Pm,Ym-1}(\varnothing) \} G_1 \wedge G_2$.
- If $G_{n-k}=\exists xG'$ we proceed as in the previous case.
- If $G_{n-k}=D\text{=>}G'$ then, for some $j<k$, $G_{n-j}=G'$ and $W_{n-j}=P_1|..|P_m|D$. By inductive hypothesis, $T^{\omega}_{D,Ym}(\varnothing) \} G'$, that is $Y_m \} D\text{=>}G'$. Thus, by definition of Y_m, $T^{\omega}_{Pm,Ym-1}(\varnothing) \} D\text{=>}G'$. □

To prove the completeness of the operational semantics with respect to the fixpoint semantics, we shall make use of the following lemmas. I^* denotes the set $\{T\text{->}A: A \in I\}$, where I is a subset of B(P).

Lemma 1. Let $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots$ be a sequence of Herbrand interpretations. If G is a goal and $\cup_{i=0}^{\omega} I_i \} G$ then there exists $k \ge 0$ such that $I_k \} G$.

Lemma 2. Let D_1 and D_2 be programs, G a closed goal and W a list of programs (possibly empty). If $D_1 \subseteq D_2$ then

$$D_1|W \mid\!- G \Rightarrow D_2|W \mid\!- G.$$

Lemma 3. Let P be a program, G a closed goal, I a subset of B(P) and W a list of programs (possibly empty).

$$I^* \cup P|W \mid\!- G \Rightarrow I^*|P|W \mid\!- G.$$

Lemma 4. Let P be a program, G a closed goal. Then

$$\{T\text{->}A : P \mid\!- A\} \mid\!- G \Rightarrow P \mid\!- G.$$

Lemma 1 can be proved by induction on the structure of G. Lemmas 2, 3 and 4 can be proved by double induction on the length of the derivation and on the structure of the goal G (Giordano et al. 1988).

Theorem 2 (Completeness). Let P be a program, G a closed goal, I a subset of B(P).

$$T^{\omega}_{P,I}(\varnothing) \} G \Rightarrow P \cup I^* \mid\!- G.$$

Proof. We only sketch at the proof of the theorem. We proceed by induction on the highest number n of levels of nesting of => in P and G.
• If n=0 then there are no occurrences of => neither in P nor in G. It can be proved that, for every $k \ge 0$,

$$T^{k}_{P,I}(\varnothing) \} G \Rightarrow P \cup I^* \mid\!- G \qquad (*)$$

by double induction on k and on the structure of G. Since, by Lemma 1, if $T^{\omega}_{P,I}(\emptyset) \} G$ then there is a $k \geq 0$ such that $T^{k}_{P,I}(\emptyset) \} G$, we conclude by (*) that $P \cup I^{*} \vdash G$.

• Now we consider the case n>0. We assume, by inductive hypothesis, that the thesis holds for at most n-1 levels of nesting of => in P and G. It can be proved (see (Giordano et al. 1988)) that, given an Herbrand interpretation I and a program P with at most n levels of nesting of =>,

$$T^{\omega}_{P,I}(\emptyset) \subseteq \{A : P \cup I^{*} \vdash A\}.$$

By making use of this inclusion we can prove the thesis for n>0 by induction on the structure of G.

- If G=T, it is obvious.
- If G=A, then $T^{\omega}_{P,I}(\emptyset) \} A \Rightarrow A \in T^{\omega}_{P,I}(\emptyset) \subseteq \{A : P \cup I^{*} \vdash A\} \Rightarrow P \cup I^{*} \vdash A$.
- If $G=G_1 \wedge G_2$, then $T^{\omega}_{P,I}(\emptyset) \} G_1 \wedge G_2$
  - $\Rightarrow T^{\omega}_{P,I}(\emptyset) \} G_1$ and $T^{\omega}_{P,I}(\emptyset) \} G_2$
  - $\Rightarrow P \cup I^{*} \vdash G_1$ and $P \cup I^{*} \vdash G_2$ (by inductive hypothesis)
  - $\Rightarrow P \cup I^{*} \vdash G_1 \wedge G_2$.
- If G=∃xG', we proceed as in the previous case.
- If G=D=>G', then $T^{\omega}_{P,I}(\emptyset) \}$ D=>G'
  - $\Rightarrow T^{\omega}_{D,X}(\emptyset) \} G'$, where $X=T^{\omega}_{P,I}(\emptyset)$
  - $\Rightarrow D \cup X^{*} \vdash G'$, where $X=T^{\omega}_{P,I}(\emptyset)$ (by inductive hypothesis, since D and G' can contain at most n−1 levels of nesting of =>).
  - $\Rightarrow X^{*}|D \vdash G'$, where $X=T^{\omega}_{P,I}(\emptyset)$ (by Lemma 3)
  - $\Rightarrow X^{*} \vdash$ D=>G', where $X=T^{\omega}_{P,I}(\emptyset)$
  - $\Rightarrow (T^{\omega}_{P,I}(\emptyset))^{*} \vdash$ D=>G'
  - $\Rightarrow$ {T->A : $P \cup I^{*} \vdash A$} |- D=>G' (by Lemma 2, since $T^{\omega}_{P,I}(\emptyset) \subseteq \{A : P \cup I^{*} \vdash A\}$)
  - $\Rightarrow P \cup I^{*} \vdash$ D=>G' (by Lemma 4). □

Finally, from Theorems 1 and 2, for $I=\emptyset$, we have

$$T^{\omega}_{P,\emptyset}(\emptyset) \} G \text{ iff } P \vdash G$$

that is the operational semantics is sound and complete with respect to the fixpoint semantics.

As a particular case, for G=A

$$A \in T^{\omega}_{P,\emptyset}(\emptyset) \text{ iff } P \vdash A,$$

that is

$$T^{\omega}_{P,\emptyset}(\emptyset)=\{A : P \vdash A\}.$$

As an example, let us consider once again the program P of Example-2. We want to determine whether G=s can be derived from P, that is whether $s \in T^{\omega}_{P,\emptyset}(\emptyset)$

Example-5.

$$T^{0}_{P,\emptyset}(\emptyset)=\{q\}$$
$$T^{1}_{P,\emptyset}(\emptyset)=T_{P,\emptyset}(\{q\})=\{q,s\}$$
$$T^{2}_{P,\emptyset}(\emptyset)=T_{P,\emptyset}(\{q,s\})=\{q,s\}=\mathrm{lfp}(T_{P,\emptyset})$$

Notice that to determine whether s belongs to $T_{P,\emptyset}(X)$ it is necessary to determine whether $T^{\omega}_{D,X}(\emptyset) \} p$ with D=(q∧r->p)∧(T->r) (fifth rule of the definition of $\}$). For $X=\emptyset$ we have $\mathrm{lfp}(T_{D,X})=\{r\}$ so that p is not satisfied in $T^{0}_{P,\emptyset}(\emptyset)$. On the contrary, for X={q} we have $\mathrm{lfp}(T_{D,X})=\{q,r,p\}$ so that p is satisfied in $T^{1}_{P,\emptyset}(\emptyset)$ and $s \in T^{1}_{P,\emptyset}(\emptyset)$. If we replace the first clause of P with the clause r->q, the goal G=s is no more satisfiable by P, since in this case $T^{\omega}_{P,\emptyset}(\emptyset)=\emptyset$.

4 MODEL-THEORETIC SEMANTICS

Let α be a closed formula, that is a goal or a definite clause. Given an Herbrand interpretation I for α, we define I|=α (I *satisfies* α) by induction on the structure of α as follows:

- I|=T
- I|=A iff A∈I
- $I \models G_1 \wedge G_2$ iff $I \models G_1$ and $I \models G_2$
- I|=∃xG iff I|=[x/t]G for some t∈U(α)
- I|=D=>G iff, for all I', (I⊆I' and I'|=D) ⇒ I'|=G
- I|=G->A iff I|=G ⇒ I|=A
- I|=∀xD iff I|=[x/t]D for all t∈U(α)
- $I \models D_1 \wedge D_2$ iff $I \models D_1$ and $I \models D_2$.

Let P be a program and I an Herbrand interpretation for P. I satisfies P (I|=P), that is I is a *model* for P, if I satisfies all clauses in P. We denote with M(P) the set of all Herbrand models of P. A closed goal formula G is a *logical consequence* of P (P|=G) iff for all interpretations I of P, I|=P ⇒ I|=G. Notice that, from the definition of [P] given in section 2, it follows that I|=P iff for all G->A∈[P], I|=G->A.

It needs to be noticed that the two different implications -> and => have been given different semantics. The implication -> is the classical one, while the implication => has a semantics similar, under some respects, to that of the implication of intuitionistic logic. Our model-theoretic semantics is, nevertheless, simpler than Kripke semantics for intuitionistic logic (see section 5), since an Herbrand interpretation is defined to be a subset of the Herbrand base as for classical logic; we don't need to introduce the notion of worlds as in Kripke interpretations. As a result, we shall see that, for every program P there exists a least Herbrand model of P and this gives the possibility to prove the equivalence between model-theoretic and fixpoint semantics in the same way as it is done for Horn clause logic in (Apt and van Emden 1982).

To prove the equivalence between the two semantics, we establish some lemmas first.

Lemma 5. Let P be a program, G a closed goal and I_1 and I_2 two Herbrand interpretations for P. If $I_1 \cap I_2 \models G$, then $I_1 \models G$ and $I_2 \models G$.

Proof. Obvious, by induction on the structure of G.

Lemma 6 (Model intersection property). Let P be a program and I_1 and I_2 two Herbrand interpretations for P. If I_1 and I_2 are models of P, then $I_1 \cap I_2$ is a model of P.

Proof. It can be proved, by induction on D, that, for every clause D in P, $I_1 \models D$ and $I_2 \models D \Rightarrow I_1 \cap I_2 \models D$ (see (Giordano et al. 1988)).

As a consequence of Lemma 6 we have that the intersection ∩M(P) of all Herbrand models of P is a model of P, namely the *least Herbrand model* of P.

Lemma 7. ∩M(P) = {A : P|=A}.

Proof. P|=A
- ⇒ for all I, I|=P ⇒ I|=A
- ⇒ for all I, I|=P ⇒ A∈I.

Since M(P)={I : I|=P}, we have that for all Herbrand interpretations I, I∈M(P) ⇒ A∈I.
But ∩M(P)∈M(P), thus A∈∩M(P). □

We now prove soundness and completeness of the fixpoint semantics with respect to the model theoretic semantics.

Theorem 3 (Soundness and completeness). Let P be a program, G a closed goal and I a subset of B(P) (remember that $I^* = \{T \text{->} A,\ A \in I\}$).

$$T^{\omega}_{P,I}(\emptyset) \vdash G \text{ iff } \cap M(P \cup I^*) \models G.$$

Proof. By induction on the highest number n of levels of nesting of => in P and G.

• If n=0 then there are no occurrences of => neither in P nor in G. Let I and X be two Herbrand interpretations. We shall show, that

(1) $X \vdash G$ iff $X \models G$ and

(2) $T^{\omega}_{P,I}(\emptyset) = \cap M(P \cup I^*)$.

From these relations the thesis can be immediately derived.
(1) can be easily proved by induction on the structure of G.
If G=T, it is obvious.
If G=A, $X \vdash A$ iff $A \in X$ iff $X \models A$.
If $G = G_1 \wedge G_2$, $X \vdash G_1 \wedge G_2$
iff $X \vdash G_1$ and $X \vdash G_2$
iff $X \models G_1$ and $X \models G_2$ (by inductive hypothesis)
iff $X \models G_1 \wedge G_2$.
If $G = \exists x G'$, the proof is similar.
The case G=D=>G' does not occur, since n=0.

To prove (2) it is sufficient to show that, given two Herbrand interpretations I and X,

(3) $I \subseteq X$ and $X \in M(P)$ iff $T_{P,I}(X) \models X$.

In fact, $T^{\omega}_{P,I}(\emptyset) =$
$= \cap\{X : T_{P,I}(X) \subseteq X\}$
$= \cap\{X : X \in M(P) \text{ and } I \subseteq X\}$, by (3),
$= \cap\{X : X \in M(P) \text{ and } X \models I^*\}$
$= \cap\{X : X \models P \cup I^*\}$
$= \cap M(P \cup I^*)$.

We shall prove that (3) holds for n=0.
From left to right. Let us assume that $I \subseteq X$ and $X \in M(P)$. We want to show that $T_{P,I}(X) \subseteq X$. If $A \in T_{P,I}(X)$ then either $A \in I$ and then $A \in X$, or there is a $G\text{->}A \in [P]$ such that $X \vdash G$. G does not contain occurrences of =>, therefore $X \vdash G$ implies $X \models G$. Since, in addition, $X \in M(P)$, that is for all $G\text{->}A \in [P]$ $X \models G$ implies $X \models A$, we have that $X \models A$. Thus $A \in X$.
From right to left. Let us assume that $T^{\omega}_{P,I}(X) \subseteq X$. We want to prove that $I \subseteq X$ and $X \in M(P)$.
$T_{P,I}(X) \subseteq X$
$\Rightarrow$ for all A, $A \in T_{P,I}(X) \Rightarrow A \in X$
$\Rightarrow$ for all A, ($A \in I$ or there is $G\text{->}A \in [P]$ such that $X \vdash G$) $\Rightarrow A \in X$.
Thus for all A in I, $A \in I$ implies $A \in X$, that is $I \subseteq X$, and if there exists a $G\text{->}A \in [P]$ such that $X \vdash G$ then $A \in X$. Since $X \vdash G$ implies $X \models G$, we have that, for all $G\text{->}A \in [P]$, if $X \models G$ then $A \in X$, that is $X \in M(P)$. Thus we have proved (3).

• If n>0 we assume, by inductive hypothesis, that the thesis holds for at most n-1 levels of nesting of => in P and G. We prove the thesis for n levels of nesting, as in the case n=0, by proving (1) and (2). Again, (1) is proved by induction on the structure of G.
If G=T, G=A, $G = G_1 \wedge G_2$ or $G = \exists x G'$ we proceed as in the case n=0.
If G=D=>G', $X \vdash D \text{=>} G'$
iff $T^{\omega}_{D,X}(\emptyset) \vdash G'$
iff $\cap M(D \cup X^*) \models G'$ (by inductive hypothesis, since D and G' can contain at most n-1 levels of nesting of ⇒).
iff for all I, $I \in M(D \cup X^*) \Rightarrow I \models G'$
iff for all I, $I \models D \cup X^* \Rightarrow I \models G'$
iff for all I, $X \subseteq I$ and $I \models D \Rightarrow I \models G'$
iff $X \models D \text{=>} G'$.

(2) is proved as for n=0 by showing that (3) holds. In doing this, we use the fact that each formula in P contains at most n levels of nesting of => and that (1) holds for n. □

From Theorem 3, for $I = \emptyset$, we have

$$T^{\omega}_{P,\emptyset}(\emptyset) \vdash G \text{ iff } \cap M(P) \models G.$$

Since $\cap M(P) \models G$
iff for all I, $I \in M(P) \Rightarrow I \models G$
(by lemma 5 and since $\cap M(P) \in M(P)$)
iff for all I, $I \models P \Rightarrow I \models G$
iff $P \models G$,
the following relation holds

$$T^{\omega}_{P,\emptyset}(\emptyset) \vdash G \text{ iff } P \models G,$$

that is the fixpoint semantics is sound and complete with respect to the model-theoretic semantics.

5 AN ALTERNATIVE MODEL-THEORETIC SEMANTICS BASED ON KRIPKE INTERPRETATIONS

In the last section we have defined the model-theoretic semantics of our language by employing Herbrand interpretations defined as subsets of the Herbrand base. We shall now define another model-theoretic semantics for the language, by making use of *Kripke interpretations*. Again we restrict ourselves to consider interpretations defined on the Herbrand universe. The satisfiability relation is defined in the same way as in positive intuitionistic logic, with an extension due to the presence of the two different kinds of implication.

Let α be a closed formula, that is a goal or a definite clause. A Kripke interpretation M for α is a triple $\langle W, \subseteq, I_0 \rangle$, where $W \subseteq 2^{B(\alpha)}$ is a partially ordered set of *worlds* and $I_0 \in W$ is a world of M such that $I_0 \subseteq I$, for any world I of M (i.e. I_0 is the *least world*). We define the satisfiability relation between an interpretation M and a formula α in a given world I of M by induction on the structure of α, as follows:

- $M \models_I T$
- $M \models_I A$ iff $A \in I$
- $M \models_I G_1 \wedge G_2$ iff $M \models_I G_1$ and $M \models_I G_2$
- $M \models_I \exists x G$ iff $M \models_I [x/t]G$ for some $t \in U(\alpha)$
- $M \models_I D \text{=>} G$ iff, for each world I' of M,
  ($I \subseteq I'$ and $M \models_{I'} D$) $\Rightarrow M \models_{I'} G$
- $M \models_I G \text{->} A$ iff $M \models_I G \Rightarrow M \models_I A$
- $M \models_I \forall x D$ iff $M \models_I [x/t]D$ for all $t \in U(\alpha)$
- $M \models_I D_1 \wedge D_2$ iff $M \models_I D_1$ and $M \models_I D_2$

An interpretation $M = \langle W, \subseteq, I_0 \rangle$ satisfies a formula α iff $M \models_{I_0} \alpha$. Let P be a program and M a Kripke interpretation for P. M satisfies P, if M satisfies all the clauses in P. Let G be a closed goal formula. G is a logical consequence of P ($P \models' G$) iff for all Kripke interpretations M for P, $M \models_{I_0} P \Rightarrow M \models_{I_0} G$ (we use an apex to distinguish between logical consequence in the two different model-theoretic semantics).

Notice that if we restrict the language to the propositional case and to have a unique implication symbol (used both in goal and in clause) with the semantics of =>, this semantics is the same (with a change of notation) as that presented in (Gabbay 1985), which is the semantics of intuitionistic logic.

On the other hand, if we restrict the language by eliminating block goals and so the implication =>, we have clearly a semantics for classical logic (only the least interpretation in a world is used). We can therefore consider => to be the intuitionistic implication, while -> is the classical one.

This model-theoretic semantics is not equivalent to that of the previous section in the general case. In fact, for example,

$$a \Rightarrow b \models b \quad \text{and} \quad a \Rightarrow b \not\models' b.$$

In fact, every interpretation I that satisfies a=>b must satisfy b too, because, if not, there is an interpretation $I'=I\cup\{a\}$ reachable from I which satisfies a but not b (against the definition of satisfiability for an implication goal). On the other hand, there are Kripke interpretations which satisfy a=>b in their initial world but do not satisfy b, such as for instance the interpretation $M=\langle\{I_1,I_0\},\subseteq,I_0\rangle$, where $I_0=\emptyset$ and $I_1=\{b\}$. Nevertheless, if we restrict ourselves to determine whether a program is a logical consequence of a goal in our language, the two semantics are equivalent. In fact it can be proved that, given a program P and a closed goal G,

$$P \models' G \text{ iff } P \models G \qquad (**).$$

The previous example, since a=>b is a goal, does not satisfy the restriction and so the equivalence doesn't hold.

To prove (**) we state the following lemmas.

Lemma 8. Let α be a formula (that is a goal or a definite clause), I a subset of $B(\alpha)$ and M the Kripke interpretation $\langle W,\subseteq,I\rangle$, where $W=\{I' : I'\in 2^{B(\alpha)} \text{ and } I\subseteq I'\}$. Then

$$I \models \alpha \text{ iff } M \models_I \alpha.$$

Proof. By induction on the structure of the formula α.

Lemma 9. Let D be a definite clause, G a closed goal (we assume that G contains only non-logical symbols that occur in D), I_0 and I subsets of B(D) and M the Kripke interpretation $\langle W,\subseteq,I_0\rangle$, with $W\subseteq 2^{B(D)}$ and $I\in W$. Then

$$(1)\quad I \models G \Rightarrow M \models_I G.$$
$$(2)\quad I \not\models D \Rightarrow M \not\models_I D.$$

Proof. By induction on the number of levels of nesting of => in D and G.

Since a program is a set of closed definite clauses, if $I\not\models P$ then there is a clause D such that $I\not\models D$ and, by Lemma 9, we have that $M\not\models_I D$. Thus $M\not\models_I P$. Therefore, relation (2) holds for any program P, not only for any definite clauses.

Theorem 4. Let P be a program and G be a goal.

$$P \models' G \text{ iff } P \models G.$$

Proof. From left to right. By hypothesis, for all interpretations M, $M \models_{I_0} P \Rightarrow M \models_{I_0} G$. We want to prove that for all the interpretations I, $I \models P \Rightarrow I \models G$. Let I be an interpretation such that $I\models P$. By Lemma 8 then $M\models_I P$, where M is the interpretation $\langle W,\subseteq,I\rangle$ and $W=\{I' : I'\in 2^{B(P)} \text{ and } I\subseteq I'\}$. Thus, by hypothesis, $M\models_I G$ and, again by Lemma 8, $I\models G$.
From right to left. By hypothesis, for all interpretations I, $I\models P \Rightarrow I\models G$. We want to prove that, for all interpretations M, $M \models_{I_0} P \Rightarrow M \models_{I_0} G$. Given an interpretation $M=\langle W,\subseteq,I_0\rangle$, there are two possible cases: either $I_0\models G$ or $I_0\not\models P$. If $I_0\models G$ then, by Lemma 9, $M\models_{I_0} G$; if $I_0\not\models P$ then, by Lemma 9, $M\not\models_{I_0} P$. Thus we have that either $M\models_{I_0} G$ or $M\not\models_{I_0} P$, namely $M \models_{I_0} P \Rightarrow M \models_{I_0} G$. □

6 AN OPERATIONAL SEMANTICS WITH SUBSTITUTIONS

The definition of the operational semantics given in section 2 is very simple since, given a program P, it introduces, such as in (Miller 1986), the set [P] of all ground instances of the clauses in P and doesn't involve the notions of substitution, unification and variable renaming. We shall now present a less abstract operational semantics, which is clearly equivalent to the previous one and is defined using substitutions, unification and variable renaming.

Let $P_1|..|P_n$ be a not empty list of programs, let G be a closed goal and let θ be a substitution. We define *derivability* of G from the list $P_1|..|P_n$, with substitution θ, by induction on the structure of G, in the following way.

(1) $P_1|..|P_n \vdash T$ with substitution I (identity substitution);
(2) if A is a closed atomic formula, $P_1|..|P_n \vdash A$ with substitution θ iff
for some $i\leq n$ there is a formula $\forall x_1..\forall x_k G\text{->}B\in P_i$ such that $\mu=mgu(A,B')$ and $P_1\mu|..|P_i\mu \vdash G'\mu$ with substitution ϕ and $\theta=\mu\phi$, where $G'\text{->}B'$ is the clause obtained renaming the universally quantified variables $x_1,..,x_k$;
(3) $P_1|..|P_n \vdash G_1\wedge G_2$ with substitution θ iff $P_1|..|P_n \vdash G_1$ with substitution ϕ and $P_1\phi|..|P_n\phi \vdash G_2\phi$ with substitution μ and $\theta=\phi\mu$;
(4) $P_1|..|P_n \vdash \exists xG$ with substitution θ iff $P_1|..|P_n \vdash G'$ with substitution θ, where G' is obtained from G by renaming x;
(5) $P_1|..|P_n \vdash D\Rightarrow G$ with substitution θ iff $P_1|..|P_n|D \vdash G$ with substitution θ.

Notice that free variables can occur both in a goal and in programs of the list $P_1|..|P_n$. For this reason in rules (2) and (3) we apply substitutions not only to the goals, but also to the programs in the list. Since the initial program P_1 is a set of closed clauses and the initial goal is a closed goal formula, they do not contain free variables. Free variables can be introduced in a goal by renaming the existentially quantified variables associated to the goal itself (rule (4)); free variables can be introduced in the list of programs by rule (5) whenever there is some free variable occurring in D, in the block goal D=>G. Existential variables are renamed once, as soon as the existential quantifier is dropped out by rule (4), whereas universal variables of a clause are renamed every time the clause is selected to resolve an atomic goal (rule (2)); the free variables which can occur in that clause are not renamed. Rule (3) is defined in such a way to preserve the sharing of variables between G_1, G_2 and the programs in the list and to prevent from an improper use of the free variables in the programs.

For example, the goal $G=p(a)\wedge p(b)$ must not be operationally derivable from the list of programs L= {q(a),q(b)} | {q(x)->p(x)}, since the variable x in the list is free and not universally quantified. Instead, the goals p(a) and p(b) are individually operationally derivable from that list.

7 TOWARDS A CONCRETE IMPLEMENTATION

Static features of programming languages allow efficient implementations, especially as far as compilation is concerned. In order to show this in the case of the blocks of this paper, we outline here a more concrete operational semantics, where substitutions are not actually carried out but only kept in the form of variable bindings.

According to the formalism introduced in (Martelli and Rossi 1986) and based on that used by the unification algorithm in (Martelli and Montanari 1982), a substitution can be represented by a *system of equations* in solved form, i.e. a system of the form $\{x_1=t_1, x_2=t_2, \ldots, x_n=t_n\}$, where the x_i's are distinct variables and the t_i's are terms.

Analogously to conventional programming languages, in this semantics programs are considered as *code (source terms)* and are never modified during the computation, whereas the terms in a system of equations are considered as *data (constructed terms)*. The renaming of variables in a source term is described by means of an *environment* where variable identifiers of the source term are associated with new variable names. For instance, to rename the clause $q(x)\wedge r(y) \text{->} p(x,y)$ we define an environment $(x/x_1, y/y_1)$ and a system $\{x_1=\varnothing, y_1=\varnothing\}$ to represent the fact that initially the two variables are not bound. Subsequent steps of the computation can change the bindings in the system, whereas the environment will not change. Thus we see that the system of equations plays the role of the *store* in conventional (imperative) languages.

According to this semantics, the meaning of a goal in standard Horn clause logic depends on a program P, an environment Env and a system of equations *Sys*. For instance, if the goal is an atom A, its semantics is $<P,Env,Sys> \mid\!\text{-}\; A$ with solution *Sys*' iff
there is a clause $G\text{->}B \in P$ with variables $x_1,\ldots,x_n$ such that
Sys_1=unify$(A,Env,B,(x_1/x_1',\ldots,x_n/x_n'), Sys\cup\{x_1'=\varnothing,\ldots,x_n'=\varnothing\})$
and $<P,(x_1/x_1',\ldots,x_n/x_n'),Sys_1> \mid\!\text{-} G$ with solution *Sys*', where $x_1',\ldots,x_n'$ are new variables.

"unify(T_1,E_1,T_2,E_2,S)" unifies the two source terms T_1 and T_2 in the environments E_1 and E_2 respectively with respect to the system *S*, and returns, if the unification succeeds, the modified system.

In the case of our extended language, both the program and the environment must be modified to take into account the block structure. Thus Env becomes a list of environments and P becomes a list of pairs (P_i,Env_i), where Env_i is the environment where the variables of P_i are bounded.

The operational semantics is now:

(1) $<P,Env,Sys> \mid\!\text{-}\; T$ with solution *Sys*;
(2) if A is a closed atomic formula,
$<(P_1,E_1)|..|(P_n,E_n),Env,Sys> \mid\!\text{-}\; A$ with solution *Sys*' iff for some i there is a formula $\forall x_1,\ldots,\forall x_k\, G\text{->}B \in P_i$ such that Sys_1= unify$(A,Env,B,E_i|(x_1/x_1',\ldots,x_k/x_k'), Sys\cup\{x_1'=\varnothing,\ldots, x_k'=\varnothing\})$ and $<(P_1,E_1)|..|(P_i,E_i),E_i|(x_1/x_1',\ldots,x_k/x_k'),Sys_1> \mid\!\text{-}$ G with solution *Sys*', where $x_1',\ldots,x_k'$ are new variables;
(3) $<P,Env,Sys> \mid\!\text{-}\; G_1\wedge G_2$ with solution *Sys*' iff
$<P,Env,Sys> \mid\!\text{-}\; G_1$ with solution Sys_1 and
$<P,Env,Sys_1> \mid\!\text{-} G_2$ with solution *Sys*';
(4) $<P,Env,Sys> \mid\!\text{-}\; \exists xG$ with solution *Sys*' iff
$<P,Env|(x/x'),Sys\cup\{x'=\varnothing\}> \mid\!\text{-}\; G$ with solution *Sys*';
(5) $<P,Env,Sys> \mid\!\text{-}\; D\text{=>}G$ with solution *Sys*' iff
$<P|(D,Env),Env,Sys> \mid\!\text{-}\; G$ with solution *Sys*'.

Notice the different treatment of variables and clauses in the above semantics. Variables are introduced by quantifiers; clauses by blocks. There are two different lists for for variables and clauses, the environment list and the program list respectively. The environment list is extended with a new block either when there is an existentially quantified goal (rule (4)) or when a universally quantified clause is selected as input clause for resolution (rule (2)). On the contrary, a new block is added to the list of programs only whenever a goal (D=>G) is encountered.

This semantics is very close to actual implementations and can be used as a guide for writing an interpreter or a compiler. In particular, the system of equations can be easily implemented in a heap-like structure as a graph, where each node corresponds to an equation, and the environments can be put, as in usual block-structured languages, on a stack of activation records. As pointed out in (Martelli and Rossi 1986), the extensions required to handle blocks with respect to implementations of standard Prolog can very naturally be achieved, by applying well known techniques for accessing non local variables such as static pointer or desplay. Further run time structures are of course needed to implement the nondeterminism by means of backtracking. Starting from such an implementation, it is then possible to apply various optimizations which are well-known in the literature dealing with implementation of Prolog-like languages.

REFERENCES

Apt K.R., van Emden M.H., "Contributions to the theory of Logic Programming", *J. ACM*, 29,1982, 841-862.

Bowen K.A., Kowalski R.A., "Amalgamating Language and Metalanguage in Logic Programming", in *Logic Programming*, (Clark and Tarlund, eds.), Academic Press, 1982, 153-172.

Giordano L., Martelli A., Rossi G.F., "Local definitions with static scope rules in Logic Languages", Internal Report, University of Turin, May 1988.

Gabbay D.M., Reyle N., "N_Prolog: An Extension of Prolog with Hypothetical Implications.I.", *Journal of Logic Programming*, no.4 1984, 319-355.

Gabbay D.M., "N_Prolog: An Extension of Prolog with Hypothetical Implications.II. Logical foundations, and negation as failure", *Journal of Logic Programming*, no.4, 1985, 251-283.

Lloyd J.W., *Foundations of Logic Programming*, Springer-Verlag, 1984.

Martelli A., Montanari U., "An Efficient Unification Algorithm", *ACM TOPLAS*, 4, 2, April 1982.

Martelli A., Rossi G.F., "On the semantics of Logic Programming Languages", in *Proc. of the Third Int. Conf. on Logic Programming*, LNCS, vol.225, 1986, 327-334.

McCarty L.T., "Clausal Intuitionistic Logic. 1. Fixed-Point semantics" *Journal of Logic Programming*, no.4, 1985, 251-283.

Miller D.A., "A Theory of Modules for logic Programming", *IEEE Symp. on Logic Programming*, Sept. 1986, 106-114.

Nait Abdallah M.A., "Ions and local definitions in Logic Programming", in *STACS 86*, LNCS, vol.210, 1986, 60-72.

Warren D.S., "Database updates in Prolog", *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1984, 244-253.