Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.C.M. Baeten, W.P. Weijland

Semantics for prolog via term rewrite systems

Computer Science/Department of Software Technology Report CS-R8739 August

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Semantics for Prolog via term rewrite systems

J.C.M. Baeten

Dept. of Computer Science, University of Amsterdam,

P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

W.P.Weijland

Dept. of Software Technology, Centre for Math. and Comp. Sci.,

P.O.Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract: We present semantics for logic programs using term rewrite systems. Reading program lines from left to right (so reversing the arrows), considering the result as a rewrite system, immediately gives the usual declarative semantics (the least Herbrand model). Then, we add a priority ordering on the rewrite rules, and obtain a procedural semantics for Prolog with depth-first search rule. This gives us different semantics in the same setting.

Key words and phrases: Prolog, logic programming, term rewrite system, priority rewrite system, depth-first search.
1985 Mathematics subject classification: 68Q45, 68Q50, 68N15.
1987 CR categories: F.4.1, F.4.2, D.3.1, I.2.3, F.3.2.

Note: Partial support received under ESPRIT contract 432, An Integrated Formal Approach to Industrial Software Development (Meteor).
This report will be submitted for publication elsewhere.

1. INTRODUCTION

In this paper, we give a new approach to semantics for logic programming and Prolog. This semantics is based on term rewriting. If we reverse the arrows in a logic program, and consider all closed instances of program clauses, we can, after two minor adaptations, work with the resulting TRS (Term Rewrite System). We prove the following theorems:

1. A ground atom reduces to true in the TRS iff it is an element of the success set of the logic program.

2. A ground atom has no infinite reductions and no normal form true in the TRS iff it is an element of the finite failure set of the logic program.

Thus, there is a clear correspondence between the least Herbrand model and the set of rewrites determined by the TRS, and we get a declarative semantics for logic programs by means of term rewrite systems.

Then, in section 3 we review the theory of Priority Rewrite Systems (PRS), and give a generalization of the theory in [2, 3]. A PRS is a TRS with a partial ordering on the set of rules. W.r.t. this partial ordering, we have a notion of correctness, that tells us when a particular rewrite

Report CS-R8739
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

is allowed; roughly speaking, it is allowed if it is not obstructed by a rewrite of higher priority. We say that a PRS is well-defined if it has a unique sound and complete rewrite set, i.e. a set of rewrites such that a rewrite is an element of it iff it is correct w.r.t. it.
In section 4 we define a notion of correctness for PRS's, that implements the depth-first search rule, such that a PRS with this notion of correctness is always well-defined. Then, we have the following theorems:
1. A ground atom has a correct reduction to true in the PRS iff it is in the success set of the Prolog program, implemented with depth-first search rule.
2. A ground atom has no infinite reduction, and no normal form true in the PRS iff it is in the finite failure set of the Prolog program with depth-first search rule.
Thus, we obtain a procedural semantics for Prolog, using the same formalism as in our earlier semantics. By adopting a reduction strategy on top of the priority ordering, we can implement a particular computation rule. In the last section, we give some indications how one can obtain a declarative semantics for Prolog.
It is the subject of ongoing research to give an implementation in this formalism for control features such as cut.

2. DECLARATIVE SEMANTICS FOR LOGIC PROGRAMS

We give a declarative semantics for logic programs based on term rewrite systems (TRS's). In section 4, we will see that we can use the same approach in order to give a procedural semantics for Prolog with depth-first search rule. First, we give some basic definitions concerning logic programs and term rewrite systems.

2.1 DEFINITION

We have an *alphabet* or *signature* containing (countably many) variables, constants, functions and predicates. Terms, atomic formulas and formulas are defined in the usual way. A term or formula is called **ground** or **closed** iff it contains no variables.
A **clause** is any formula that is the universal closure of a formula of the form $(B_1 \wedge ... \wedge B_n) \rightarrow (A_1 \vee ... \vee A_k)$ ($n+k > 0$, the A_i and B_j are atomic formulas), usually written as

$$A_1, ..., A_k \leftarrow B_1, ..., B_n.$$

A clause is a **program clause** if $k=1$, and a **goal clause** if $k=0$.
A **program** is a finite sequence of program clauses (so the clauses occur in a certain order). We will also assume that the signature of any program does not contain a constant named true, or a predicate named & (if this is not the case, a renaming is necessary).
A **substitution** is a mapping from variables to terms, and a substitution extends in the obvious way to terms and formulas. The result of applying substitution θ to formula A is denoted $A\theta$. A **ground substitution** maps all variables to closed terms.
We define the notions of an SLD-derivation, an SLD-refutation and an SLD-tree as in [5]. Note that an atom has an SLD-refutation iff it is a logical consequence of the program.
The **success set** of a program is the set of all ground atoms that have an SLD-refutation.

The **finite failure set** of a program consists of all ground atoms A that have a finitely failed SLD-tree (i.e. a finite SLD-tree with no success branches).

2.2 DEFINITION

In a *term rewrite system* (TRS), there is no difference between predicates and functions (they are all called functions), and so there is also no difference between terms and atomic formulas (they are all called terms). A TRS is a finite sequence of **(rewrite) rules**, pairs of terms <t,s>, usually written as $t \to s$.

The left-hand side term t in such a rule must not be only a variable, but here we do not adopt the usual requirement that any variable occurring in the right-hand side s must also occur in t.

These rules generate a **reduction relation** on terms: this is the closure of the rules under substitution and contexts, and is also denoted $\to$. If $t \to s$ for *closed* terms t,s, we call $t \to s$ a **rewrite**. The semantics of a TRS is its set of rewrites. Further, $\to^*$ is the transitive and reflexive closure of $\to$. A term t is a **normal form** if for no term s we have $t \to s$; a term t' **has a normal form** if there is a normal form t with $t' \to^* t$.

2.3 TRANSLATION

Now with every program we will associate a term rewrite system. The signature of the TRS will contain, besides the elements of the signature of the program, a special constant true, and a special binary predicate (function) &. We will use the following conventions in order to use & as a function of variable arity:

$\&(t)$	is another notation for	t
$\&(t_1, ..., t_{n+1})$	is another notation for	$\&(\&(t_1, ...,t_n), t_{n+1})$.

Now the transformation from a logic program to a TRS will translate a clause $A \leftarrow B_1, ..., B_n$ with $n \geq 1$ into a rule $A \to \&(B_1, ..., B_n)$ and a clause $A \leftarrow$ into a rule $A \to$ true; further, the transformation will preserve the ordering and will finally add two rules

$\&(x, \&(y, z)) \to \&(\&(x, y), z)$

$\&(true, true) \to true$.

If $\mathbb{P}$ is a logic program, call the TRS that results by applying this transformation $\mathbb{P}^*$.

2.4 EXAMPLE

Let $\mathbb{P}$ be the logic program in table 1a. Then the TRS $\mathbb{P}^*$ is displayed in table 1b.

$p \leftarrow q, r(x)$	$p \to \&(q, r(x))$
$p \leftarrow$	$p \to true$
$q \leftarrow r(x)$	$q \to r(x)$
	$\&(x, \&(y, z)) \to \&(\&(x, y), z)$
	$\&(true, true) \to true$

Table 1a. Table 1b.

Now we can formulate the following theorem. In the proof, we use terminology of [5].

2.5 THEOREM

The ground atom G is an element of the success set of the logic program $\mathbb{P}$ iff G has true as a normal form in the TRS $\mathbb{P}^*$.

PROOF: Suppose G is a ground atom. Since G is in the success set of $\mathbb{P}$, there is an SLD-refutation of $\mathbb{P} \cup \{\leftarrow G\}$. Using the obtained answer substitution, we can turn this refutation into an (unrestricted) SLD-refutation in which all successive goals consist of ground atoms. Now each step in the SLD-refutation will correspond to a rewrite step in the TRS $\mathbb{P}^*$, as follows: for each step from a goal $\leftarrow A_1, ..., A_m, ..., A_k$ (consisting of ground atoms) to a resolvent goal $\leftarrow A_1, ..., A_{m-1}, B_1\theta, ..., B_q\theta, A_{m+1}, ..., A_k$, using a rule $A \leftarrow B_1, ..., B_q$ and a ground substitution θ with $A\theta = A_m$, there is a corresponding reduction sequence $\&(A_1, ..., A_m, ..., A_k) \rightarrow \&(A_1, ..., A_{m-1}, \&(B_1\theta, ..., B_q\theta), A_{m+1}, ..., A_k) \rightarrow^* \&(A_1, ..., A_{m-1}, B_1\theta, ..., B_q\theta, A_{m+1}, ..., A_k)$, using the θ-instantiation of rule $A \rightarrow \&(B_1, ..., B_q)$ in the first step, the added rules in the following steps. The empty clause $\square$ corresponds to the term true.

The reverse direction is equally easy.

2.6 THEOREM

The ground atom G is an element of the finite failure set of the logic program $\mathbb{P}$ iff G has no infinite reductions in $\mathbb{P}^*$, and does not have true as a normal form in $\mathbb{P}^*$.

PROOF: Just like the previous theorem. Note that if we have an SLD-tree that has no success branch and no infinite branch, then by König's lemma there must be a k such that the tree is finitely failed of depth $\leq k$.

3. PRIORITY REWRITE SYSTEMS

In this section we give a review of the theory of Priority Rewrite Systems (PRS's). The notion of a PRS was introduced in BAETEN, BERGSTRA & KLOP [2], and rephrased by the same authors in [3]. Here, we give a more general treatment, in order to define a special purpose PRS in the following section. For more explanations and examples, see [3] or [2].

3.1 DEFINITION

A **Priority Rewrite System** is a triple $\langle \mathbb{R}, <, c \rangle$, where $\mathbb{R}$ is a Term Rewrite System, $<$ is a partial ordering on the rewrite rules, and c is an anti-monotonic mapping on sets of rewrites.

In this paper, we will only consider the linear ordering, i.e. if a PRS has rules $r_1,...,r_n$ (presented in this order), then $r_i < r_j$ iff $i < j$. If $r < s$, we say rule r **has priority over** rule s.

Instead of $c(R)$ we write R^c. Anti-monotonicity means that $R \subseteq S$ implies that $R^c \supseteq S^c$. If a rewrite is in R^c, we say it is **correct** w.r.t. R.

Now, let $\mathbb{P}$ be a PRS. Considered as a TRS, its semantics is the set of all rewrites, i.e. the set of all closed instantiations of its rewrite rules. As a PRS, some of these rewrites are invalidated, are not correct, because of the existence of a rule that takes priority. We clarify our intention with the

following example.

3.2 EXAMPLE

Consider a PRS containing the rules in table 2 below. Rule r_1 has priority over rule r_2. Let a be a constant. Now the rewrite eq(a,a) → false is incorrect, because the term eq(a,a) is also an instantiation of the left-hand side of rule r_1, and the rewrite eq(a,a) → true will take priority.

r_1: eq(x,x) → true
r_2: eq(x,y) → false

Table 2.

To be more precise, we could use the following definition of the mapping c: a rewrite r: p → q is incorrect w.r.t. a set of rewrites R iff there is a rewrite r': p → ... in R with r' < r. It is obvious that this definition makes c into an anti-monotonic mapping.
In references [2] and [3], different notions of correctness were used. In this paper, we will need still another notion, in order to give a procedural semantics for Prolog.

3.4 DEFINITION

Let $\mathbb{P}$ be a PRS, and R a set of rewrites of $\mathbb{P}$.
i. R is **sound** iff $R \subseteq R^c$, i.e. every rewrite in R is correct w.r.t. R;
ii. R is **complete** iff $R \supseteq R^c$, i.e. every rewrite, that is correct w.r.t. R, is in R;
iii. $\mathbb{P}$ is **well-defined** iff there exists a unique sound and complete rewrite set for $\mathbb{P}$. Such a rewrite set is called the **semantics** of $\mathbb{P}$.

3.5 NOTE

It is by no means the case that every PRS is well-defined. In [3], examples are presented of a PRS with no sound and complete rewrite set, and a PRS with two sound and complete rewrite sets. It is the subject of ongoing research to determine classes of PRS's that are well-defined. For example, in [3], it is proven that all strongly terminating PRS's are well-defined (w.r.t. the definition of correctness in [3]).

3.6 LEMMA

Let $\mathbb{P}$ be a PRS, and R a set of rewrites of $\mathbb{P}$.
i. if $R \supseteq S$, and S is sound and complete, then R is complete;
ii. if $R \subseteq S$, and S is sound and complete, then R is sound.
PROOF: i: $R \supseteq S$ implies $R^c \subseteq S^c$. Since S is sound and complete, $S^c = S$. Thus $R \supseteq R^c$, which means that R is complete. The proof of ii. is similar.

3.7 DEFINITION

Let $\mathbb{P}$ be a PRS, and R a set of rewrites of $\mathbb{P}$. We put $\mathfrak{R}_{\mathbb{P}}(R) = (R^c)^c$. If there is no confusion possible, we write $\mathfrak{R}$ instead of $\mathfrak{R}_{\mathbb{P}}$. Since c is anti-monotonic, it follows that $\mathfrak{R}$ is a monotonic mapping. In general, however, $\mathfrak{R}$ need not be continuous. Since $\mathfrak{R}$ is monotonic, we have a least fixed point $\mathrm{lfp}(\mathfrak{R})$ and a greatest fixed point $\mathrm{gfp}(\mathfrak{R})$.

Also, we can define the sets $\mathfrak{R}\uparrow\alpha$ and $\mathfrak{R}\downarrow\alpha$ for every ordinal α as in e.g. LLOYD [5]. We have:

1. $\mathfrak{R}\uparrow 0 = \emptyset$ and $\mathfrak{R}\downarrow 0 = \emptyset^c$
2. $\mathfrak{R}\uparrow(\alpha+1) = \mathfrak{R}(\mathfrak{R}\uparrow\alpha)$ and $\mathfrak{R}\downarrow(\alpha+1) = \mathfrak{R}(\mathfrak{R}\downarrow\alpha)$
3. $\mathfrak{R}\uparrow\lambda = \cup_{\alpha<\lambda}\, \mathfrak{R}\uparrow\alpha$ and $\mathfrak{R}\downarrow\lambda = \cap_{\alpha<\lambda}\, \mathfrak{R}\downarrow\alpha$, if λ is a limit ordinal.

3.8 LEMMA

Let $\mathbb{P}$ be a PRS, and let $n \in \mathbb{N}$. Then:

i. $\mathfrak{R}\uparrow n$ is sound and $\mathfrak{R}\downarrow n$ is complete

ii. $(\mathfrak{R}\uparrow n)^c = \mathfrak{R}\downarrow n$

iii. $(\mathfrak{R}\downarrow n)^c = \mathfrak{R}\uparrow(n+1)$.

PROOF: By induction on n. For $n=0$, we have (i) since $\emptyset \subseteq \emptyset^c$, which implies $\emptyset^c \supseteq \emptyset^{cc}$, and (ii) and (iii) follow by definition. The induction step is just as easy.

3.9 LEMMA (Stabilization Lemma, see also [2] or [3])

Let $\mathbb{P}$ be a PRS. If for some n,m $\mathfrak{R}\uparrow n = \mathfrak{R}\downarrow m$, then $\mathbb{P}$ is well-defined.

PROOF: Immediate from 3.6 and 3.8. $\mathfrak{R}\uparrow n$ is the least and greatest fixed point of the mapping $\mathfrak{R}$.

4. PROCEDURAL SEMANTICS FOR PROLOG

Now we will define a notion of correctness for priority rewrite systems, such that adding this priority to a term rewrite system as in section 2 will give us a procedural semantics for Prolog using a depth-first search rule (for terminology, see LLOYD [5]). We use the word Prolog for logic programs implemented with the depth-first search rule. Initially, we do not consider a certain computation rule (one could say that we discuss concurrent Prolog). Later on, we will see that the adoption of a selection rule amounts to the adoption of a reduction strategy in the corresponding rewrite systems.

4.1 INTUITION

Let $\mathbb{P}$ be a term rewrite system with the linear rule ordering (see 3.1). We will say that a rule instance $r: p \to q$ is incorrect, if there is a rule $r' < r$ and a rewrite $r': p \to \ldots$, such that Prolog will never consider the rewrite $r: p \to q$. Since Prolog will consider the rules in order, first the rewrite $r': p \to \ldots$ will be carried out, but later, by backtracking, $r: p \to q$ may also be considered. There are two cases in which $r: p \to q$ will never be considered:

1. $r': p \to \ldots$ results in a normal form true;
2. $r': p \to \ldots$ gives rise to an infinite reduction.

A second circumstance in which the rewrite $r: p \to q$ is incorrect, is when there is also a rewrite r:

$p \to q'$ with a different q' (so this only happens when the rule r introduces a variable), such that the substitution made in q will never be considered. We give examples of both types of incorrectness.

4.2 EXAMPLES

1. Consider the following Prolog program $\mathbb{P}$ in table 3 (written as a TRS according to the definitions of section 2, with the two added rules for & left out):

r_1: $p \to q$
r_2: $p \to true$
r_3: $q \to q$

Table 3.

In the standard procedural semantics with depth-first search rule, we find that the goal p gives no answer, because Prolog will get stuck in the infinite reduction sequence $q \to q \to q \to ...$ Thus, we must have that the reduction $p \to true$ is incorrect w.r.t. the set of reductions $\{p \to q, q \to q\}$.

2. Consider the following Prolog program $\mathbb{Q}$ in table 4.

$q(f(x)) \to q(f(x))$
$q(a) \to true$
$p \to q(x)$

Table 4.

In the standard procedural semantics with depth-first search rule, we find that the goal p gives no answer, because Prolog will unify $q(x)$ and $q(f(x))$, and start an infinite reduction. Thus, we must have that the reduction $p \to q(a) \to true$ is incorrect. Since the rewrite $q(a) \to true$ is obviously correct, we will ensure that the rewrite $p \to q(a)$ is incorrect w.r.t. the set $\{p \to q(f(a)), q(f(a)) \to q(f(a))\}$.

Next we give some definitions, that are preparatory to the definition of our notion of correctness.

4.3 DEFINITION

Let σ: $s_1 \to^{r_1} s_2 \to^{r_2} s_3 \to ...$ and ρ: $t_1 \to^{r_1'} t_2 \to^{r_2'} t_3 \to ...$ be two reduction sequences (finite or infinite). The ordering on rules induces a lexicographical ordering on reduction sequences, so $\rho < \sigma$ if $r_1' < r_1$, or $r_1' = r_1$ and either $r_2' < r_2$ or ρ has length 1, or ....

We call a (finite) reduction sequence **successful**, if it ends in the normal form $true$.

We call ρ an **obstruction** of σ, notation $\sigma \blacktriangleright \rho$, if $s_1 \equiv t_1$, $\rho < \sigma$ and ρ is either successful or infinite. See figure 1 (on the following page).

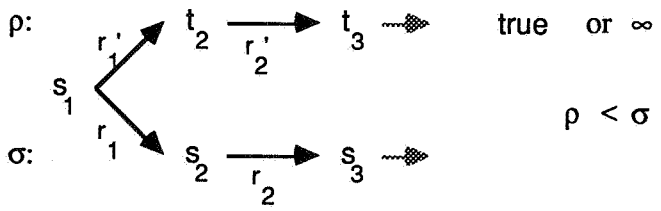

Fig. 1. $\sigma \blacktriangleright \rho$.

Now we can give the formal definition of correctness.

4.4 DEFINITION

Let $\mathbb{P}$ be a term rewrite system, with the linear rule ordering.
Let σ be a reduction sequence, and let R be a set of rewrites.
Then σ is **incorrect** w.r.t. R, if there is a reduction sequence ρ in R such that $\sigma \blacktriangleright \rho$.
Now let r: $p \to q$ be a rewrite (a closed instantiation of rule r). This rewrite is **incorrect** w.r.t. R, if every reduction sequence starting with this rewrite is incorrect w.r.t. R.
Further, if r: $p \to q$ is incorrect, then every application in context of this rewrite is also incorrect.
A rewrite is **correct** w.r.t. R, if it is not incorrect w.r.t. R.
The reader can check for himself that this definition indeed leads to the right results in the examples in 4.2.
It is obvious that this gives us an anti-monotonic mapping c, and so we can use the results of section 3. We will call this notion of correctness **DFS correctness** (correctness pertaining to the depth-first search rule).
Now a procedural semantics for Prolog will be given by the sound and complete rewrite set of the resulting PRS. Therefore, we first have to establish that such PRS's are well-defined. This we do next.

4.5 THEOREM

Let $\mathbb{P}$ be a PRS with the DFS notion of correctness. Then $\mathbb{P}$ is well-defined.
In fact, $\mathfrak{R}\uparrow 1 = \mathfrak{R}\downarrow 1$ (in the notation of section 3).
PROOF: We will prove this in a number of claims.

CLAIM 1: Let R be a set of rewrites, and t a closed term. If t has an infinite or successful reduction in R, then t also has an infinite or successful reduction in $R \cap R^c$.
PROOF: Suppose σ: $t \to \to$ ... is an infinite or successful reduction in R, which is not a reduction in R^c. Let $p \to q$ be the first rewrite in σ which is not in R^c. Then $p \to q \to$ ... is incorrect w.r.t. R, so there is an obstruction $p \to q' \to$ ... in R. If the rewrite $p \to q'$ is still not correct w.r.t. R, there is another obstruction $p \to q'' \to$ ... in R. After finitely many steps, we must obtain a reduction sequence $p \to q^* \to$ ... in R of which the first rewrite is correct w.r.t. R (since there are only finitely many rules). Thus, we obtain an infinite or successful reduction sequence σ^*: $t \to \to p \to q^* \to$ ... in R, which is also in R^c for at least one more step (see fig. 2, on the following page).

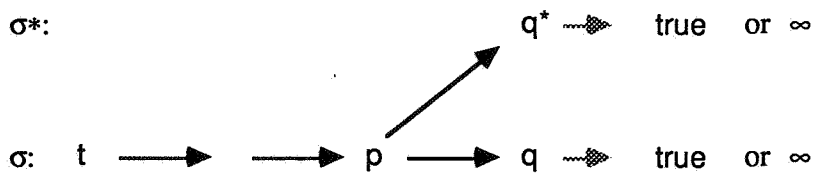

Fig. 2.

Continuing in this fashion, we obtain reduction sequences, of which larger and larger initial segments remain unchanged. It follows that we can write down a reduction in $R \cap R^c$, and the proof is finished.

CLAIM 2: Let R be a set of rewrites. Then $R^{cc} \subseteq R^c$.
PROOF: If a rewrite is not correct w.r.t. R ($\notin R^c$), then there is an obstruction in R. By the previous claim, there is also an iobstruction in R^c, and so the rewrite is also not correct w.r.t. R^c ($\notin R^{cc}$).

CLAIM 3: $\Re\uparrow 1 = \Re\downarrow 1$.
PROOF: Let R be the set of *all* rewrites. By claim 2 we have $R^{cc} \subseteq R^c$. On the other hand, R is complete, so R^c is sound, and thus by definition $R^c \subseteq R^{cc}$. Then $\Re\uparrow 1 = R^c = R^{cc} = \Re\downarrow 1$.

4.6 DEFINITIONS
We define the success set and finite failure set of a Prolog program in the sequel.
1. We define the notion of a **SLD-tree** just like in [5]. For each computation rule, a goal has a different SLD-tree. The edges in the tree are labeled by the name of the rule used. The ordering of the rules induces an ordering on the edges, and in this paper, all SLD-trees will be ordered trees. An example, for the program in 4.2.1 for goal p, is given in fig. 3a.
2. A **DFS-tree** (SLD-tree using the depth-first search rule) of a goal is obtained from the SLD-tree by leaving out all subtrees that have an infinite obstruction in the tree (i.e. from the root there is an infinite branch starting with a higher rule). The DFS-tree of the tree in fig. 3a is given in fig. 3b.

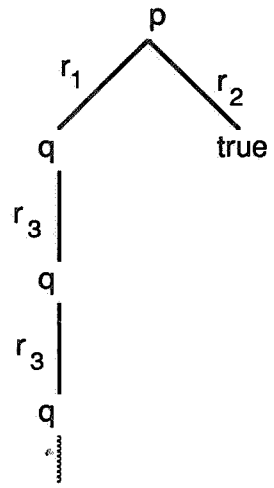

Fig. 3a.

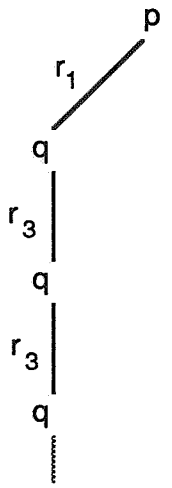

Fig. 3b.

3. A ground atom A is in the **success set** of a Prolog program $\mathbb{P}$, if the goal A has a DFS-tree with success branch (a branch ending in □).
4. A ground atom A is in the **finite failure set** of a Prolog program $\mathbb{P}$, if the goal A has a finite DFS-tree with no success branch.

4.7 THEOREM
The ground atom A is an element of the success set of the Prolog program $\mathbb{P}$ iff A has a successful correct reduction in the PRS $\mathbb{P}^*$.
PROOF: $\Rightarrow$: Suppose A is in the success set of the Prolog program $\mathbb{P}$, so there is a DFS-tree with success branch. Using the translation of 2.3, we get a reduction sequence

$$\alpha: A \rightarrow A_1 \rightarrow \ldots \rightarrow \text{true}.$$

If α is incorrect, then there is an obstruction α_1. If α_1 is infinite, then α cannot be in the DFS-tree. Therefore, α_1 is finite and successful. If α_1 is also incorrect, then there is a finite α_2 with $\alpha_1 \blacktriangleright \alpha_2$. In this way, we obtain a sequence $\alpha \blacktriangleright \alpha_1 \blacktriangleright \alpha_2 \ldots$ of successful reductions of A. If this sequence is infinite, then the transitivity of the obstruction relation yields an infinite obstruction for α, and so α is not in the DFS-tree. Thus, the sequence $\alpha \blacktriangleright \alpha_1 \blacktriangleright \alpha_2 \ldots \blacktriangleright \alpha_k$ is finite, and α_k is a correct reduction from A to true.
$\Leftarrow$: Straightforward, by definition of DFS-tree.

4.8 THEOREM
The ground atom A is an element of the finite failure set of the Prolog program $\mathbb{P}$ iff A has no infinite or successful correct reductions in $\mathbb{P}^*$.
PROOF: Similar to the proof of 2.6 and 4.7.

4.9 Thus, we see that the adoption of a priority ordering on a term rewriting system gives us a procedural semantics for Prolog with depth-first search rule. This is interesting since no explicit semantics could be found in the literature, only semantics directly in terms of SLD-trees with the search procedure defined on them.
The priority mechanism seems powerful enough to deal in addition with extra features like cut, or the use of negation in Prolog programs. Details of such extensions are not clear to us at the moment, however.

4.10 So far, we have not discussed the effect of a *computation rule*, i.e. we were dealing with a form of concurrent Prolog, in which each element of a sequence of goals is reduced independently. In the present setting, adoption of a computation rule just amounts to the adoption of a *reduction strategy* on the corresponding PRS. Thus, adoption of the standard computation rule (that always selects the left-most atom) means in terms of rewriting that we only consider reductions in which in each step the left-most redex is contracted.

5. DECLARATIVE SEMANTICS FOR PROLOG

In the previous sections, we found a procedural semantics for Prolog with depth-first search rule, using priority rewrite systems. Moreover, we proved this procedural semantics to be equivalent to the semantics of depth-first search trees.

In this section, we give some indications on how to obtain a declarative equivalent for the DFS-semantics for Prolog. In the case of logic programs, we know that the declarative semantics simply consists of its set of Horn clauses. This declarative semantics is equivalent to the procedural one, in the sense that for any logic program $\mathbb{P}$ we have that an atom A is in the succes set of $\mathbb{P}$ if and only if A is a logical consequence of $\mathbb{P}$. But what to say about the DFS succes set of $\mathbb{P}$?

In order to find an answer to this question, consider the following definitions.

5.1 DEFINITION

In the following we will write substitutions (see [5]) as *equational formulas*. To give an example: suppose we have $\theta \equiv \{x/a\}$ then we can translate this substitution to $\theta(x) \equiv \{x=a\}$. The *most general unifier* of atoms A and B, will be denoted by $mgu(A,B)$. Writing $mgu(A,B)$ as an equational formula we obtain:

for all atoms A and B: $\models \forall (mgu(A,B) \rightarrow (A \leftrightarrow B))$.

5.2 EXAMPLE

Let $A \equiv p(x,f(z))$ and $B \equiv p(a,f(g(y)))$ then clearly, $mgu(A,B) \equiv \{x/a,z/g(y)\}$. Writing this substitution as an equational formula we obtain: $mgu(A,B) \equiv \{x=a,z=g(y)\}$ and clearly we find

$$\models \forall [x=a \wedge z=g(y) \rightarrow (p(x,f(z)) \leftrightarrow p(a,f(g(y))))].$$

5.3 TRANSLATIONS OF PROGRAMS

Next, we will translate a given program $\mathbb{P}$ into a set of Horn clauses $\mathbb{L}(\mathbb{P})$, which is equivalent to the original program except that it is restricted to the depth-first search rule. This can be done as follows.

Let $\mathbb{P}$ be $A \leftarrow B_1, ..., B_q$

$C \leftarrow D_1, ..., D_r$.

With the depth-first search rule, the upper clause $A \leftarrow B_1, ..., B_q$ has highest priority. So we may write: $\forall (B_1 \wedge ... \wedge B_q \rightarrow A) \in \mathbb{L}(\mathbb{P})$.

The second clause $C \leftarrow D_1, ..., D_r$ cannot simply be added to $\mathbb{L}(\mathbb{P})$, since it will only be used if either the first clause has failed (by backtracking) or if the current goal did not match A. This can be expressed by the following sentences:

$$\forall (mgu(A, C) \wedge \neg B_1 \wedge ... \wedge \neg B_q \wedge D_1 \wedge ... \wedge D_r \rightarrow C) \text{ and}$$

$$\forall (\neg mgu(A, C) \wedge D_1 \wedge ... \wedge D_r \rightarrow C) \in \mathbb{L}(\mathbb{P}).$$

Note that $\neg$ is used to express negation in the sense of finite failure. This procedure can simply be extended to a general procedure for programs with more then only two clauses.

5.4 TURNING $\mathbb{L}(\mathbb{P})$ INTO A GENERAL PROGRAM

Note that $\mathbb{L}(\mathbb{P})$ can easily be translated to a general program, i.e.: a program with negated literals in

its clauses. Taking the program $\mathbb{P}$ from 5.3 again, we see that $\mathbb{L}(\mathbb{P})$ is equivalent to

$\mathbb{P}'$: $A \leftarrow B_1, ..., B_q$

$A \leftarrow \neg B_1, ..., \neg B_q, D_1, ..., D_r$

$C \leftarrow D_1, ..., D_r, \neg mgu(A, C)$.

As we see, we will permit equations in the bodies of the clauses.

5.6 SEMANTICS OF GENERAL PROGRAMS

In FITTING [4] a general theory for fixed point semantics of general programs is presented, using three valued logic. Fitting uses sets of *signed* formulas $T\phi$ or $F\phi$ to indicate whether an atom can be proved true, false, or cannot (yet) be proved equal to either of the two. Of course such a set need not be consistent (for example if it contains both $T\phi$ and $F\phi$), but if it is, it can be seen as a *(partial)* model.

Next, in the same way as is done in APT & VAN EMDEN [1], a monotonic mapping $\Phi_{\mathbb{P}}$ is defined on partial models as follows:

$TA \in \Phi_{\mathbb{P}}(X) \Leftrightarrow$ there is a ground instance $A \leftarrow B_1, ..., B_q, \neg C_1, ..., \neg C_r$ of a clause from $\mathbb{P}$ such that $\{TB_1, ..., TB_q, FC_1, ..., FC_r\} \subseteq X$;

$FA \in \Phi_{\mathbb{P}}(X) \Leftrightarrow$ for every ground instance $A \leftarrow B_1, ..., B_q, \neg C_1, ..., \neg C_r$ of a clause from $\mathbb{P}$ we have $\{TB_1, ..., TB_q, FC_1, ..., FC_r\} \not\subseteq X$.

Now it can be proved that for any program $\mathbb{P}$, $\Phi_{\mathbb{P}}$ has a least fixed point which precisely consists of the succes set of $\mathbb{P}$ and the complement of the finite failure set ([4], proposition 7.3). Moreover, any *general* program has a semantics via $\Phi_{\mathbb{P}}$.

5.7 So, the general program $\mathbb{P}'$ in 5.5, has a clear semantics (see also LLOYD [5]). This semantics could be proved equivalent to the semantics we gave in 3.9 and 4.5. It goes beyond the subject matter of this paper, to try to develop such a proof here.

REFERENCES

[1] K.R.APT & M.H.VAN EMDEN, *Contributions to the Theory of Logic Programming*, J. ACM 29 (3), pp. 841 - 862, 1982.

[2] J.C.M.BAETEN, J.A.BERGSTRA & J.W.KLOP, *Priority rewrite systems*, report CS-R8407, Centre for Math. & Comp. Sci., Amsterdam 1984.

[3] J.C.M.BAETEN, J.A.BERGSTRA & J.W.KLOP, *Term rewrite systems with priorities*, in: Proc. 2nd Conf. on Rewriting Techniques and Applications, Bordeaux 1987, Springer LNCS 256, pp. 83 - 94, 1987.

[4] M.FITTING, *A Kripke semantics for logic programs*, Journal of Logic Programming 4, pp. 295 - 312, 1985.

[5] J.W.LLOYD, *Foundations of logic programming*, Springer 1984.

[6] T.C.PRZYMUSINSKI, *On the declarative and procedural semantics of stratified deductive databases*, Dept. of Math. Sci., Univ. of Texas, El Paso.