Higher-order unification [9, 5] (λ-unification) is very useful for metaprogramming, see for instance [4, 7, 8, 13]. However its implementation presents serious practical problems: (a) it is not decidable, and (b) even if two λ-terms are unifiable, in general a single most general unifier does not exist.

In order to avoid the problems related to the full λ-unification, some restrictions have been investigated. The resulting language, L_λ, still presents nice higher order features (modules, for example) and it is likely to support efficient implementations.

The term language of L_λ is the simply typed λ-calculus with equality modulo α, β and η-conversion. Some restrictions on the β-reduction are imposed, and, thanks to this, the unification in L_λ resembles first order unification. Namely the unification problem is decidable and, when unifiers exist, there is *one* most general unifier [11].

Some recent papers [3] have introduced a new approach to the semantics of logic programs, which is richer than the standard one based on ground interpretations. The purpose is to model a notion of observable which captures the computed answer substitutions. The key idea is to choose an appropriate notion of interpretations which allows to define a fixpoint and model theoretic-semantics in the style of Logic Programming. This approach has been proved very useful in many areas which use semantic tools, such as program transformation, abstract interpretations, etc.

The approach has been extended from pure Horn Clause Language to various Prolog constructs such as metaprogramming and higher order constructs in Prolog.

Our goal is to extend the notion of S-semantics, then considering the computed answer substitution as observable, so to obtain a fixpoint S-semantics for L_λ, with the intention, if possible, to extend this approach to the full λ-Prolog.

As extension of Prolog, L_λ presents two important features: universal quantification and implications in the goals and bodies. Universal quantification is simply solved as we will see in the next sections, whereas implications need more attention. The basic idea is the following: given a program P we define an interpretation as a finite tuple of sets

$$I = \langle I_{P_1}, \ldots, I_{P_k} \rangle$$

Each set I_{P_i} consists of pairs $\langle A, \vartheta \rangle$ where A is an atom and ϑ an equational constraint. The indexes P_i's are the programs (contexts) which result from adding to P a particular hierarchical combinations of the clauses which occur (as modules) in the clause bodies.

For example, if P is the following program

$((C_1 \supset q) \supset p.\quad (((C_2 \supset q) \supset r) \supset s) \supset t.$

the clauses used as modules will be C_1, C_2 and $C_3 = ((C_2 \supset q) \supset r)$. Hence any interpretation I of P is indexed as

$$I = \langle I_P, I_{P\cup\{C_1\}}, I_{P\cup\{C_3\}}, I_{P\cup\{C_2,C_3\}} \rangle$$

The necessity to consider more contexts in an interpretation is due to the fact that when we prove a goal $D \supset G$ in a context P, we must prove G in the new context $P \cup \{D\}$ and, after the proof, to return to the old context P.

We define a monotonic and continuous mapping T_P on interpretations. The intended model is the least fixpoint of T_P, and we show that it is equivalent to the operational semantics of L_λ-programs as defined in [11].

Note that our definition of interpretation reflects the notion of module. In particular, given the least fixpoint of T_P, $I^\omega = \langle I_P, I_{P\cup P_1}, I_{P\cup P_2}, \ldots, I_{P\cup P_n} \rangle$, I_P contains all the pairs which are true (according to the S-semantics) in P, $I_{P\cup P_1}$ contains those pairs which are true in P w.r.t. the module P_1 and so on.

This construction extends the S-semantics of [3] in the sense that if we restrict to HCL programs, then every interpretation reduces to a t-uple of one set, and the set in the least fixpoint of T_P is equivalent to the least S-model.

The problem of defining a fixpoint semantics for L_λ was also investigated in [10]. The solution presented there is based on Kripke-like structures. The idea is to associate a module to a "possible word". This is very similar to our idea of splitting an interpretation into components corresponding to the various modules. The main differences are that in [10] the computed answers are not modeled, only the truth of ground atoms is considered. Thus we can say that our approach compares to the one in [10] like the S-semantics compares to the standard least fixpoint semantics of logic programming [3]. Furthermore, in [10] possible worlds are infinite (thus representing all possible modules, even the ones which do not occur in the program).

In [6] was defined a general top down resolution for l_λ, subset of L_λ, and there it is outlined a mixed top down-bottom up approach. Our bottom up approach is more general because l_λ does not permit implications in goals.

The paper is structured in the following way. The next two sections include some notion of S-semantics and λ-calculus, and a brief introduction to L_λ. Section 4 describes the fixpoint operator and its properties. Finally Section 5 shows the equivalence between operational and fixpoint semantics.

2 Preliminaries

2.1 S-semantics

In [3] it has been introduced a new notion of declarative semantics that allo
to capture an observable property of HCL programs which is not reflected
the standard least Herbrand model: the set of the computed answer substi
tions. The S-semantics of a program is defined as least fixpoint of the followi
"immediate consequence operator":

$T_P(I) = \{A \mid \exists\, A' \leftarrow B_1, \ldots, B_p \in P$ and $\exists B'_1, \ldots, B'_p \in I$ s.t.
$\Downarrow(B'_1, \ldots, B'_p), \Downarrow((B'_1, \ldots, B'_p), (B_1, \ldots, B_p, A))$ and
$\exists \vartheta = MGU(\{B_1, \ldots, B_p\}, \{B'_1, \ldots, B'_p\})$ and $A = A'\vartheta\}$

Where $\Downarrow(E_1, \ldots, E_n)$ means that $E_1, \ldots, E_n$ have no variables in common.

2.2 λ-calculus

We describe the simply typed λ-calculus on which L_λ is based; more detail
can be found in [1, 2].

Types: the set of *types* $\mathcal{T}$ contains:
(i) A set of primitive types $\{o, t_1, \ldots, t_n\}$;
(ii) All types $\alpha \to \beta$ where α and β are types.
o is the primitive type representing the sort *Bool*.

Signature: a signature Σ is a finite set of constant symbols: $\Sigma = \{a_1$
$t_1, \ldots, a_n : t_n\}$ such that each symbol a_i has at most one type $t_i \in \mathcal{T}$.

We assume a denumerable set of variables for each type in $\mathcal{T}$; let V be the
class of variable sets.

Terms: given a signature Σ, terms are defined as follows:
(i) the constants and the variables are terms of the corresponding type;
(ii) λ-abstraction: if $x \in V$ is of type α and t is a term of type β then $\lambda x.t$ is
a term of type $\alpha \to \beta$;
(iii) application: if t_1 and t_2 are terms of type $\alpha \to \beta$ and α respectively, then
$(t_1\ t_2)$ is a term of type β.

The free variables of a term t, *freevar(t)*, are defined as usual. $Var(t)$
denote the set of variables of t.

2.3 Logic over terms

Given the above term language, we can define over it a higher order logic
by using the primitive type o which represents the type of predicates. One
particular predicate is *true*, with the intended meaning. We impose that the
types of arguments and the types of variables do not contain o. This brings to

restriction on the quantification, i.e. the language does not allow to quantify
predicate variables.

Let Σ be a signature and consider the following constants not in Σ:
$\wedge : o \to (o \to o)$;
$\forall : (t \to o) \to o$ with $t \in \mathcal{T}$.

Formulas
(i) each term of type o is an atomic Σ-formula (atom); (ii) if B and C
formulas, then $B \wedge C$ and $B \supset C$ are Σ-formulas ; (iii) if $B[y/x]$ is a
$(y : \tau)$ formula, then $\forall x.B$ is a Σ-formula.

A sequent calculus is used to define intuitionistic provability in this lan-
guage. The system is shown in Fig. 1. For more details see [11].

$$\frac{\Sigma;\ B, C, \Delta \longrightarrow E}{\Sigma;\ B \wedge C, \Delta \longrightarrow E}\ \wedge\text{-L} \qquad \frac{\Sigma;\ \Gamma \longrightarrow B \qquad \Sigma;\ \Gamma \longrightarrow C}{\Sigma;\ \Gamma \longrightarrow B \wedge C}\ \wedge\text{-R}$$

$$\frac{\Sigma;\ \Gamma \longrightarrow B \qquad \Sigma;\ C, \Gamma \longrightarrow E}{\Sigma;\ B \supset C, \Gamma \longrightarrow E}\ \supset\text{-L} \qquad \frac{\Sigma;\ B, \Gamma \longrightarrow C}{\Sigma;\ \Gamma \longrightarrow B \supset C}\ \supset\text{-R}$$

$$\frac{\Sigma;\ \lambda norm([x \mapsto t]B), \Gamma \longrightarrow C}{\Sigma;\ \forall_\tau x.\, B, \Gamma \longrightarrow C}\ \forall\text{-L} \qquad \frac{\Sigma \cup \{y{:}\tau\};\ \Gamma \longrightarrow [x \mapsto y]B}{\Sigma;\ \Gamma \longrightarrow \forall_\tau x.\, B}\ \forall\text{-R}$$

$$\frac{\Sigma;\ \Gamma \longrightarrow B \qquad \Sigma;\ B, \Delta \longrightarrow E}{\Sigma;\ \Gamma, \Delta \longrightarrow E}\ \text{cut} \qquad \frac{}{\Sigma;\ \Gamma, B \longrightarrow B}\ \text{initial}$$

Figure 1:Inference rules for intuitionistic provability

In Fig. 1 Σ denotes the signature of the language, B and C denote formulas,
Γ and Δ denotes set of formulas, and x and y denote variables. In λ-calculus
one can simplify a term using the conversion rules presented in [11]. With
$\lambda norm(t)$ we denote the normal form of the term t; namely a term t' that
cannot be simplified further.

3 L_λ-programs

The language L_λ is based on a restriction of the higher-order hereditary Harr formulas, plus a restriction on universally quantified variables [12]. Essentia in L_λ disjunction and existential quantification of goals are not allowed, a there are restrictions on the higher-order unification.

Let A be a syntactical variable that ranges over atoms. Let D and G ra over formulas defined by the following grammar

$$G ::= A \mid G_1 \wedge G_2 \mid D \supset G \mid \forall X.G \quad D ::= A \mid G \supset A \mid \forall X.D$$

In order to simplify the unification in L_λ, restrictions are made on t universally quantified variables which can be instantiated to generic term Such variables are those which are bound by a quantifier occurring at the to level of D in $D \supset G$. Universal quantifiers which occur at the top-level of do not need to be restricted since in the computation their variables will b instantiated only to new constants and not to general terms. For more detail on these restrictions see [11]. Then a goal and a program are a G-formula an a finite collection of D-formulas (i.e. clauses) which satisfy the restriction respectively.

3.1 Constraints

We consider a constraint system whose elements $\vartheta, \sigma, \ldots$ are constituted by equations on λ-terms, the logic conjunction $\wedge$ and the existential quantification $\exists_X$. Namely

(i) if A_1 and A_2 are atoms, then $A_1 = A_2$ is a constraint;
(ii) if t_1, t_2 are λ-terms, then $t_1 = t_2$ is a constraint;
(iii) if ϑ, σ are constraints, then $\vartheta \wedge \sigma$ is a constraint;
(iv) if ϑ is a constraint and $X \in V$, then $\exists_X \vartheta$ is a constraint.

The empty constraint is denoted by ε.

The free variables of a constraint ϑ are defined by

(i) if ϑ is $A_1 = A_2$ then $freevar(\vartheta) = var(A_1) \cup var(A_2)$;
(ii) if ϑ is $t_1 = t_2$ then $freevar(\vartheta) = freevar(t_1) \cup freevar(t_2)$;
(iii) if ϑ is $\vartheta_1 \wedge \vartheta_2$ then $freevar(\vartheta) = freevar(\vartheta_1) \cup freevar(\vartheta_2)$;
(iv) if ϑ is $\exists_X \vartheta'$ then $freevar(\vartheta) = freevar(\vartheta') \setminus \{X\}$.

A mapping v from variables to λ-terms is a (syntactical) solution of ϑ if one of the following cases holds:

(i) ϑ is $A_1 = A_2$ and $A_1 v \equiv A_2 v$;
(ii) ϑ is $t_1 = t_2$ and $t_1 v \equiv t_2 v$;
(iii) ϑ is $\vartheta_1 \wedge \vartheta_2$ and v is a solution of ϑ_1 and ϑ_2;
(iv) ϑ is $\exists_X \vartheta'$ and there exists a λ-term t s.t. v^t_X is a solution of ϑ'.

Where v^t_X is defined as follows:

$$v^t_X(Y) = \begin{cases} v(Y) & \text{if } Y \neq X \\ t & \text{if } Y = X \end{cases}$$

We denote with *false* the constraint without solutions. The entailment relation is defined as usual, namely:

$\vartheta \vdash \sigma$ iff all the (syntactical) solutions of ϑ are solutions of σ.

ϑ and σ are equivalent ($\vartheta \Leftrightarrow \sigma$ iff $\vartheta \vdash \sigma$ and $\sigma \vdash \vartheta$).

We will consider the constraint system modulo the equivalence relation, but for the sake of simplicity we will denote the equivalence classes by their representants.

As usual, a constraint is consistent iff it has a (syntactical) solution, The inconsistent constraint will be denoted by *false*.

For technical reasons we need to introduce the restriction operator $\restriction$ on a constraint ϑ.

Let χ be a set of variables, and let $X_1, \ldots, X_n$ be the free variables of ϑ which do not occur in χ. Define $\vartheta_{\restriction_\chi} = \exists_{X_1}, \ldots, \exists_{X_n} \vartheta$.

4 S-semantics

4.1 Interpretations

Respect to Horn clauses, the main syntactical differences of L_λ are the universal quantifications and the implications in the goals. The universal quantification is simply solved substituting the quantified variable by a "fresh" variable, and executing the resulting goal.

More problems are introduced by the implication, and it influences the syntactical structure of an interpretation. In practice we need to handle the dynamic evolution of the environment of the predicate definitions (program) during the execution of a goal. In fact a goal containing an implication will be processed by evaluating the consequence in the current program enriched with the premise.

In the following we assume that all variables must be distinct among them.

The basic idea is to define an interpretation I for a program P as a tuple sets indexed by the programs which can be relevant for the execution of every possible goal in P. Each set specifies which goals are "true" w.r.t. the corresponding program and w.r.t. I. Concerning implication goals, we will restrict to the ones which occur in the body of "the root program" P. This allows

us to define an interpretation as a finite tuple. In fact, the programs which have to be considered result from the hierarchical combinations of P and the premises of implications. Of course, the resulting interpretation will allow to model only the truth of implications whose premise occurs in the body of the program.

Hence the form of an interpretation I for a program P is $I = \langle I_{P_1}, \ldots, I_{P_k} \rangle$ where $P_1 = P$ and $P_2, \ldots, P_k$ are indexes obtained as follows (Z denotes a fresh variable):

$$\begin{aligned} indexes_D(I, X) : -X &== G \supset A,\ indexes_G(I', G), I = I' \bar{\cup} \{X\} \cup \{X\}; \\ X &== \forall Y.D,\ indexes_D(I, D); \\ X &== A,\ I = \{A\}. \end{aligned}$$

$$\begin{aligned} indexes_G(I, X) : -X &== A,\ I = \emptyset; \\ X &== \forall Y.G,\ indexes_G(I, G); \\ X &== G_1 \wedge G_2,\ indexes_G(I_1, G_1),\ indexes_G(I_2, G_2), \\ & \quad I = I_1 \cup I_2; \\ X &== D \supset G,\ indexes_D(I_1, D),\ indexes_G(I_2, G), \\ & \quad I = I_1 \cup (I'' \bar{\cup} \{D\}). \end{aligned}$$

where $\bar{\cup}$ is defined in the following way (S is a set of sets):

$S \bar{\cup} \{A\} = \{SS\ op\ A) \| SS \in S\}$.

$SS\ op\ A\ =\ if A == \forall Y.A'\ then\ SS\ op\ A'[Y/Z]\ else\ SS \cup \{A\}$.

If the program P is $C_1 \wedge \ldots \wedge C_n$, then, for each clause C_i, we build its set of indexes I_{C_i} using the result of $indexes_D(I_{C_i}, C_i)$. Finally, we build the set of indexes $\{I_{P_1}, \ldots, I_{P_k}\}$ of P as the set $\#(I_{C_1}, \ldots, I_{C_n})$, which includes all the possible combinations of the members of $I_{C_1}, \ldots, I_{C_n}$.

For example consider the following program P:

$\forall X.((((q\ X \supset r\ a) \supset q\ a) \supset c) \supset p\ X).\quad (p\ a \supset p\ a) \supset c.$

the clauses occurring in the program bodies are:

$\{C_1{=}(q\ X \supset r\ a) \supset q\ a,\ C_2{=}q\ X,\ C_3{=}p\ a\}$

hence an interpretation I for P must be indexed in the following way

$$I = \langle I_P, I_{P \cup \{C_1\}}, I_{P \cup \{C_3\}}, I_{P \cup \{C_1, C_2\}} \rangle.$$

Each set I_{P_i} consists of pairs $\langle A, \vartheta \rangle$ where A is an atom and ϑ is a constraint. The intended meaning of $\langle A, \vartheta \rangle \in I_{P_i}$ is "$\leftarrow A$ is refutable in P_i, with computed answer substitution ϑ".

We now generalize this notion to arbitrary goals: we define the statement $I_{P_i} \models \langle G, \vartheta \rangle$ ($\langle G, \vartheta \rangle$ is true in I_{P_i}) with the intended meaning "$\leftarrow G$ is refutable in P_i, with answer substitution ϑ".

Def. 4.1 Truth value in an interpretation

Given a tuple $I = \langle I_{P_1}, \ldots, I_{P_k} \rangle$, the truth of a pair $\langle G, \vartheta \rangle$ in I_{P_i} is defined as follows.

1. $I_{P_i} \models \langle \Box, true \rangle$;
2. $I_{P_i} \models \langle G_1 \wedge G_2, \vartheta \rangle$ *iff* $I_{P_i} \models \langle G_1, \vartheta_1 \rangle$ *and* $I_{P_i} \models \langle G_2, \vartheta_2 \rangle$ *and* ϑ *is* $\vartheta_1 \wedge \vartheta_2 \not\Leftrightarrow false$ *(*)*;
3. $I_{P_i} \models \langle D \supset G, \vartheta \rangle$ *iff* D' *is a renaming of* D *and there exist* j *s.t.* $P \cup \{D'\} = P_j$ *and* $I_{P_j} \models \langle G, \vartheta \rangle$;
4. $I_{P_i} \models \langle \forall X.G, \vartheta \rangle$ *iff* $I_{P_i} \models \langle G[Y/X], \vartheta \rangle$ *and* Y *fresh and* $\vartheta_{\restriction freevar(G[Y/X])} \Leftrightarrow \exists Y \vartheta_{\restriction freevar(G[Y/X])}$ *(**)*;
5. $I_{P_i} \models \langle A, \vartheta \rangle$ *where* A *is an atom iff* $\exists \langle A', \vartheta' \rangle \in I_{P_i}$ *s.t.* ϑ *is* $\vartheta' \wedge (A = A') \not\Leftrightarrow false$.

(*) In (2) the following restriction must be respected.
Let $\chi_1 = freevar(\vartheta_1) \cup freevar(G_1) \setminus [freevar(G_2) \cup freevar(P)]$ *and*
$\chi_2 = freevar(\vartheta_2) \cup freevar(G_2) \setminus [freevar(G_1) \cup freevar(P)]$.
Then $\vartheta_1 \Leftrightarrow \exists_{\chi_2} \vartheta_1$ *and* $\vartheta_2 \Leftrightarrow \exists_{\chi_1} \vartheta_2$ *must hold. Namely* ϑ_1 *must not establish constraints on variables of* ϑ_2 *or* G_2 *which do not occur in* G_1 *or in* P*; symmetrical condition for* ϑ_2.
(**) The condition on case (4) means that $\vartheta_{\restriction freevar(G[Y/X])}$ *does not establish any constraint on* Y.

4.2 Fixpoint operator

Now we define a mapping T_P on interpretations:

$$T_P(\langle I_{P_1}, \ldots, I_{P_k} \rangle) = \langle I'_{P_1}, \ldots, I'_{P_k} \rangle.$$

Each set of the resulting tuple is obtained as follows (where $\bar{X}$ stands for a tuple of variables $X_1, \ldots, X_n$).

$$\begin{aligned} I'_{P_i} = &\{\langle A, \varepsilon \rangle \mid \text{there exists } \forall \bar{X}.A' \in P_i \text{ and } A = A'[\bar{Y}/\bar{X}] \text{ with } \bar{Y} \text{ fresh } \} \cup \\ &\{\langle A, \vartheta \rangle \mid \text{there exists } \forall \bar{X}.(G \supset A') \in P_i \text{ s.t. } I_{P_i} \models \langle G[\bar{Y}/\bar{X}], \vartheta \rangle \text{ and} \\ &\qquad A = A'[\bar{Y}/\bar{X}] \text{ with } \bar{Y} \text{ fresh } \}. \end{aligned}$$

We define the ordering on interpretations in the standard way: let $I = \langle I_{P1}, \ldots, I_{Pk} \rangle$ and $I' = \langle I'_{P1}, \ldots, I'_{Pk} \rangle$ be two interpretations (for the same program), then $I \leq I'$ iff for every index P', $I_{P'} \subseteq I'_{P'}$.

The set of interpretations (w.r.t. a given program), with the above ordering, is a complete lattice. The least element is the tuple of empty sets, and the least upper bound of a set of interpretations $\{I_l\}_{l\in\mathcal{L}}$ ($\mathcal{L}$ arbitrary set of indexes) is given by $lub_{l\in\mathcal{L}} = \langle \bigcup_{l\in\mathcal{L}} I_{P_1}, \ldots, \bigcup_{l\in\mathcal{L}} I_{P_n}\rangle$

Lemma 4.2 *$\models$ is monotonic, namely if $I \leq I'$ then for each goal G, $I_P \models G$ implies $I'_P \models G$.*

Theorem 4.3 *The operator T_P is continuous.*

By the monotonicity and by the Knaster-Tarski theorem, the least fixpoint of T_P exists and is unique. By the continuity, it is equal to $\mathcal{T}_P \uparrow \omega$. We define it as the semantics of P. The intended meaning of the least fixpoint tuple $I^w = \langle I^w_{P_1}, \ldots, I^w_{P_k}\rangle$ is that each set $I^w_{P_i}$ contains all the pairs which are deducible by the program P_i.

Example 4.1 *Let P be the following program:*
$r\ a.\quad q\ a.\quad \forall Y\ q\ Y.\quad \forall X\ (q\ X \supset \forall Z.(r\ a \wedge q\ Z)) \supset p\ X.$
The intended semantics of P would be $S = \{r\ a, q\ a, q\ Y, p\ X\}$.

We build the fixpoint of this program.

There is only a clause in the clause bodies: $C = q\ X$, then the possible indexes of an interpretation are only P and $P\cup\{C\}$; we start from the empty tuple $I^0 = \langle\emptyset,\emptyset\rangle$; in the first step we add each fact to all sets obtaining
$I^1 = \langle\{\langle r\ a,\varepsilon\rangle, \langle q\ a,\varepsilon\rangle, \langle q\ X_2,\varepsilon\rangle\}, \{\langle r\ a,\varepsilon\rangle, \langle q\ a,\varepsilon\rangle, \langle q\ X_2,\varepsilon\rangle, \langle q\ X,\varepsilon\rangle\}\rangle$

In the second step we add $p\ X_3$ to I^2_P and $I^2_{P\cup\{C\}}$ if there exists ϑ constraint s.t. $I^1_P \models \langle q\ X_3 \supset \forall Z.(r\ a \wedge q\ Z), \vartheta\rangle$. For the case (3) of Def. 4.1, we must prove $I^1_{P\cup\{C\}} \models \langle \forall Z.(r\ a \wedge q\ Z), \vartheta\rangle$ and it holds iff $I^1_{P\cup\{C\}} \models \langle r\ a \wedge q\ X_4, \vartheta\rangle$ with the right condition on ϑ, and then ϑ is $\{X_4 = X_2\}$. Of consequence we add $\langle p\ X_3, \{X_4 = X_2\}\rangle$ which is equivalent to $\langle p\ X_3, \varepsilon\rangle$.
$I^2 = \langle\{\langle r\ a,\varepsilon\rangle, \langle q\ a,\varepsilon\rangle, \langle q\ X_2,\varepsilon\rangle, \langle p\ X_3,\varepsilon\rangle\},$
$\{\langle r\ a,\varepsilon\rangle, \langle q\ a,\varepsilon\rangle, \langle q\ X,\varepsilon\rangle, \langle q\ X_2,\varepsilon\rangle\langle p\ X_3,\varepsilon\rangle\}\rangle = I^\omega.$

A more complex example is the following:

Example 4.2
$\forall X(((q\ X) \supset (r\ X)) \supset p\ X).\quad \forall X.(s\ X \supset r\ X).$
$\forall X(q\ X \supset s\ X).\quad \forall X(((p\ a \wedge p\ b) \supset (p\ a)) \supset q\ X).$

Let $C_1 = (q\ X)$ and $C_2 = (p\ a \wedge p\ b)$ be the clauses occurring in the clause bodies; the tuple is indexed as $I = \langle I_P, I_{P\cup\{C_1\}}, I_{P\cup\{C_2\}}\rangle$
$I^0 = \langle\emptyset,\emptyset,\emptyset\rangle$
$I^1 = \langle\emptyset, \{\langle q\ X,\varepsilon\rangle\}, \{\langle p\ a,\varepsilon\rangle, \langle p\ b,\varepsilon\rangle\}\rangle$

$I^2 = \langle\{\langle q\ X_2,\varepsilon\rangle\}, \{\langle s\ X_1, \{X_1 = X\}\rangle, \langle q\ X,\varepsilon\rangle, \langle q\ X_2,\varepsilon\rangle\},$
$\{\langle p\ a,\varepsilon\rangle, \langle p\ b,\varepsilon\rangle, \langle q\ X_2,\varepsilon\rangle\}\rangle$
$I^3 = \langle\{\langle q\ X_2,\varepsilon\rangle\}, I^2_{P\cup\{C_1\}} \cup \{\langle r\ X_3, \{X_1 = X, X_3 = X_1\}\rangle\},$
$I^2_{P\cup\{C_2\}} \cup \{\langle s\ X_5, \{X_5 = X_2\}\rangle\}\rangle$
$I^4 = \langle\{\langle q\ X_2,\varepsilon\rangle, \langle p\ X_5,\varepsilon\rangle\}, I^3_{P\cup\{C_1\}} \cup \{\langle p\ X_6,\varepsilon\rangle\},$
$I^3_{P\cup\{C_2\}} \cup \{\langle r\ X_7, \{X_7 = X_4, X_4 = X_2\}\rangle, \langle p\ X_8,\varepsilon\rangle\}\rangle$
The fourth step is the least fixpoint of the program.

Example 4.3 *Let P be the following program:*
$r\ a.$
$\forall X\ (((s\ X \supset (r\ X \wedge s\ b)) \supset p\ b)$

The tuple I is indexed as $I = \langle I_P, I_{P\cup\{s\ X\}}\rangle$
Now we build the least fixpoint of the program.
$I^0 = \langle\emptyset,\emptyset,\emptyset\rangle \quad I^1 = \langle\{\langle r\ a,\varepsilon\rangle\}, \{\langle r\ a,\varepsilon\rangle, \langle s\ X,\varepsilon\rangle\}$
The first step is also the least fixpoint of the program. In fact $(p\ b)$ is not provable because the variable X is instantiated with a and, of consequence, $(s\ b)$ is false.

Now we see the last example.

Example 4.4 *Let P be the following program:*
$\forall Y.(\forall X.p\ X\ Y \supset p\ X\ (f\ X)) \supset q.$
$\forall X.\forall Y.p\ X\ (f\ X).$

In P the atom p is false because we can not unify the atom $p\ X\ Y$ with $p\ X\ (f\ X)$. In fact the least fixpoint includes only $\langle p\ K\ (f\ K), \{X = K\}\rangle$, because when we try to solve the goal $\forall X.p\ X\ Y \supset p\ X\ (f\ X)$, namely we try $\{\langle p\ K\ (f\ K), \{X = K\}\rangle\} \models \langle p\ Z\ Y \supset p\ Z\ (f\ Z), \vartheta\rangle$ we only obtain $\vartheta = \{Z = K, Y = f\ Z\}$ and the condition (***) in case (4) of Definition 4.1 does not hold.

This behaviour is right because $\forall X, p\ X\ Y$ is equivalent to $\exists Y.\forall X.p\ Y$ and it is not implied by $\forall X.\forall Y.p\ X\ (f\ X)$.

We note that the definition of the fixpoint operator is the extension of the S-semantics fixpoint operator T_S for definite programs [3]. In fact, if we restrict the form of goals and clauses to first order Horn clause (without implications and higher order unification), the interpretation structure reduces to a single set and the definition of T_P reduces to T_S.

(R 1) $\dfrac{}{\langle A, \vartheta\rangle \longrightarrow_P \langle G, \vartheta \wedge \vartheta'\rangle}$ if $G \supset A' \in P$, $\vartheta' \equiv A = A'$ and $\vartheta \wedge \vartheta' \neq$ false

(R 2) $\dfrac{}{\langle A, \vartheta\rangle \longrightarrow_P \langle \Box, \vartheta \wedge \vartheta'\rangle}$ if $A' \in P$, $\vartheta' \equiv A = A'$ and $\vartheta \wedge \vartheta' \neq$ false

(R 3) $\dfrac{\langle G_1, \vartheta\rangle \longrightarrow_P \langle G_1', \vartheta \wedge \vartheta'\rangle \mid \langle \Box, \vartheta \wedge \vartheta'\rangle}{\langle G_1 \wedge G_2, \vartheta\rangle \longrightarrow_P \langle G_1' \wedge G_2, \vartheta \wedge \vartheta'\rangle \mid \langle G_2, \vartheta \wedge \vartheta'\rangle}$ (*)

(R 4) $\dfrac{\langle G, \vartheta\rangle \longrightarrow_{P \cup \{D\}} \langle G', \vartheta \wedge \vartheta'\rangle \mid \langle \Box, \vartheta \wedge \vartheta'\rangle}{\langle D \supset G, \vartheta\rangle \longrightarrow_P \langle D \supset G', \vartheta \wedge \vartheta'\rangle \mid \langle \Box, \vartheta \wedge \vartheta'\rangle}$

(R 5) $\dfrac{\langle A, \vartheta\rangle \longrightarrow_{P \cup D'} \langle G, \vartheta \wedge \vartheta'\rangle \mid \langle \Box, \vartheta \wedge \vartheta'\rangle}{\langle A, \vartheta\rangle \longrightarrow_{P \cup \forall X.D} \langle G, \vartheta \wedge \vartheta'\rangle \mid \langle \Box, \vartheta \wedge \vartheta'\rangle}$ (**)

(R 6) $\dfrac{\langle G, \vartheta\rangle \longrightarrow_P \langle G', \vartheta \wedge \vartheta'\rangle \mid \langle \Box, \vartheta \wedge \vartheta'\rangle}{\langle \forall X.G, \vartheta\rangle \longrightarrow_P \langle \forall X.G', \vartheta \wedge \vartheta'\rangle \mid \langle \Box, \vartheta \wedge \vartheta'\rangle}$ (***)

(*) ϑ' does not contain variables in G_2 which do not occur in G_1, ϑ or in P.
(**) $D' = D[Y/X]$ where Y does not occur either in P in A or in ϑ
(***) $\vartheta'_{\lfloor freevar(P,G)} = \exists X.\vartheta'_{\lfloor freevar(P,G)}$

Table 1: Operational semantics of L_λ

5 Correspondence with the operational semantics of L_λ

In this section we show the correspondence of our fixpoint semantics with the operational semantics of L_λ as described by Miller in [11] (Section 5). To this purpose we reformulate the operational semantics in terms of a transition system T, in the style of SOS [15].

The configuration of the transition system are pairs $\langle G, \vartheta\rangle$. The transition relations $\longrightarrow_P$, whose index is a set of D_clauses P (the program), are defined by the rules in Table 1. From the system T we define the observable of a program as follows

Def. 5.1 $\mathcal{O}(P) = \{\langle G, \vartheta_{\lfloor freevar(G,P)}\rangle \mid \langle G, true\rangle \longrightarrow_P^* \langle \Box, \vartheta\rangle\}$.

From T we then derive the so-called "big step semantics", namely a transition system T' whose transition relations $\Longrightarrow_P$ represent the maximal $\longrightarrow_P$-transition chains.

In practice, this amounts to defining the relation $G \Longrightarrow_P \vartheta$ whose intended meaning is $\langle G, true\rangle \longrightarrow_P^* \langle \Box, \vartheta\rangle$.

The rules for $\Longrightarrow_P$ are described in Table 2. They are derived by "grouping

(R 1) $\dfrac{}{A \Longrightarrow_P \vartheta}$ if $A' \in P$ and $\vartheta \equiv A = A'$

(R 2) $\dfrac{G \Longrightarrow_P \vartheta}{A \Longrightarrow_P \vartheta \wedge \vartheta'}$ if $G \supset A' \in P$ and $\vartheta' \equiv A = A'$ and $\vartheta \wedge \vartheta' \not\equiv$ false

(R 3) $\dfrac{G_1 \Longrightarrow_P \vartheta_1, G_2 \Longrightarrow_P \vartheta_2}{G_1 \wedge G_2 \Longrightarrow_P \vartheta_1 \wedge \vartheta_2}$ (*)

(R 4) $\dfrac{G \Longrightarrow_{P \cup D} \vartheta}{D \supset G \Longrightarrow_P \vartheta}$

(R 5) $\dfrac{G[a/X] \Longrightarrow_P \vartheta}{\forall X.G \Longrightarrow_P \vartheta}$ a fresh

(R 6) $\dfrac{G \Longrightarrow_{P \cup D'} \vartheta}{G \Longrightarrow_{P \cup \forall X.D} \vartheta}$ $D' = D[Y/X]$ Y fresh

(*) ϑ_1 does not contain variables in G_2 do not occur in G_1, ϑ_2 or in P; symmetrical condition on ϑ_2.

Table 2: Big Step semantics of L_λ

together" the premises and the conditions which model in T a $\longrightarrow_P$-sequence evolving into $\langle \Box, \vartheta\rangle$.

Note that these rules define an interpreter with a maximal degree of parallelism.

This transition system T' is similar in the structure to the proof system in Section 2.2 of [11], the main difference is that here we model not only provability, but also the computed answer substitution.

The observables can be characterized in terms of $\Longrightarrow_P$:

Proposition. 5.2 *For every program P,*
$\mathcal{O}(P) = \{\langle G, \vartheta_{\lfloor freevar(G,P)}\rangle \mid G \Longrightarrow_P \vartheta\}$

Moreover the system T' has an immediate correspondence with the fixpoint semantics: the application of the transformation T_P corresponds to using the rules top-down (whereas a goal-directed proof uses the rules bottom-up). Hence T' can be considered as the "connecting ring" between the operational semantics of Miller and our fixpoint semantics.

Thus we can state the correspondence between our fixpoint semantics and the operational semantics of [11].

Theorem 5.3 *The least fixpoint $T_P \uparrow \omega$ is equivalent to the observables $\mathcal{O}(P)$. Namely*

(correctness) if $\langle G, \vartheta\rangle \longrightarrow_P^* \langle \Box, \eta'\rangle$ then $\exists k, \eta$ s.t. $I_P^k \models \langle G, \eta\rangle$ and $\eta' = \eta \wedge \vartheta$.

(completeness) if $I^k_P \models \langle G, \eta \rangle$ *and* $\eta \wedge \vartheta \neq$ *false then* $\langle G, \vartheta \rangle \longrightarrow^*_P \langle \Box, \vartheta \wedge \eta \rangle$

6 Conclusion

λProlog [14] is an higher order extension of Prolog. The more useful characteristics of λ-Prolog are higher order unification, polymorphic types and mechanisms for building modules and abstract data types. These features are provided extending the classical first order theory of Horn clauses to the intuitionistic higher order theory of *hereditary Harrop formulas* [12].

In the paper we have given a fixpoint S-semantics, and then a bottom-up semantics, for a logic programming language which implements a particular restriction of the higher order hereditary Harrop formulas: L_λ [11]. L_λ is contained in λ-Prolog in a way that makes easy to implement unification, but it maintains characteristics such as λ-abstraction, function variables and modules.

To handle the modules mechanism, we have defined an interpretation as a tuple of sets indexed by programs that are the union of the program P with any possible hierarchical combination of the clauses appearing in the bodies of P. Each set contains pairs of the form $\langle A, \vartheta \rangle$ where A is an atom and ϑ is an equational constraint.

The fixpoint operator that we have defined in the paper is a suitable extension of the fixpoint operator for the S-semantics of definite programs [3]; in fact, if we restrict an L_λ-program to be a definite program, then the definition of T_P coincides with the definition of T_P given in [3].

Our goal is to extend this result to obtain an S-semantics (still by using a by fixpoint operator) for the whole λ-Prolog, and then to generalize the approach to the higher order hereditary harrop formulas.

References

[1] P. Andrews. An introduction to mathematical logic and type theory. Technical report, Academic Press, 1986.

[2] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation*, 103(1):86–113, 1993.

[4] A. Felty and D. Miller. Specifying theorem provers in a higher order logic programming language. In *it Ninth International Conference on Automated Deduction, Argonne*, volume 310, pages 61–80. Springer Verlag Lecture Notes in Computer Science, 1988.

[5] Gallier and Snyder. Higher-Order Unification Revisited: Complete Sets of Transformations. *Journal of Symbolic Computations*, 1988.

[6] A. Hui Bon Hoa. Intuitionistic Resolution for a Logic Programming Language with Scoping Constructs. In TACS, 1994.

[7] J. Hannan and D. Miller. Uses of higher-order unification for implementing program transformers. In *Fifth International Conference and Symposium on Logic Programming*, pages 942–959. K. Bowen and R. Kowalsky, MIT Press, 1988.

[8] J. Hannan and D. Miller. A meta language for functional programs. *Meta-Programming in Logic Programming*, pages 453–476, 1989.

[9] G. Huet. A Unification Algorithm for Typed lambda-calculus. *Theoretical Computer Science*, 1, 1975.

[10] D. Miller. A logical analysis of modules in logic programming. *Logic Programming*, (6).

[11] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Extensions of Logic Programming*. Peter Schroeder-Heister, Springer Lecture Notes in Artificial Intelligence, 1990.

[12] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[13] G. Nadathur and D. Miller. A logic programming approach to manipulating formulas and programs. In *Fourth Symposium of Logic Programming*, pages 379–388, 1987.

[14] G. Nadathur and D. Miller. An overview of λ-prolog. In *Fifth International Conference on Logic Programming*, pages 810–827. R. Kowalsky and K. Bowen, MIT Press, 1988.

[15] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

Modularity of Term Rewriting Systems Revisited

Jean-Pierre Jouannaud

CNRS and LRI
Bât. 490, Université de Paris Sud
91405 Orsay CEDEX, France
jouannaud@lri.fr

Abstract

A property of term rewriting systems is modular when it is closed under union. Modularity of termination and confluence are key properties of both algebraic and functional languages. Neither property is modular in general, but interesting subcases have been investigated: combined systems (shared symbols must be constructors), and hierarchical systems (defined symbols of the first system may be constructors of the second). Modularity of confluence will be briefly discussed first. We will then show how to capture most of the existing results for modularity of termination in a unified framework, before to explore the relationships with the recent work of Albert Rubio about extension orderings.

Total Correctness of the Goal Replacement Rule Based on the Unfold/Fold Proof Method

Maurizio Proietti

IASI-CNR
Viale Manzoni 30
00185 Roma, Italy
proietti@iasi.rm.cnr.it

Alberto Pettorossi

Electronics Department
University of Rome II
00133 Roma, Italy
adp@iasi.rm.cnr.it

Abstract

When using the program transformation methodology for developing logic programs, it is often necessary to replace in the body of a given clause old goals by new goals. This transformation rule is called goal replacement rule. In order to apply this rule we have first to show that the new goals are equivalent to the old goals w.r.t. a given semantics.

We show that these equivalence proofs can be done within the framework of the transformation methodology itself. Indeed, transformation rules similar to the ones in [24] and [23] can be used to prove equivalences of goals w.r.t. the computed answer substitution semantics of definite and general programs, respectively. These equivalence proofs will be called *unfold/fold proofs*.

We then show that the goal replacement rule is only partially correct w.r.t. the computed answer substitution semantics of both definite and general programs.

Finally, we give some syntactical conditions on the unfold/fold proofs which ensure that the corresponding goal replacements are totally correct w.r.t. the computed answer substitution semantics. These sufficient conditions allow for a powerful application of the transformation methodology by guaranteeing the correctness of the derived programs via syntactically checkable properties.

1 Introduction

The program transformation methodology is a useful technique for the derivation of efficient and correct programs [6]. When following this methodology it is often the case that a given program can be transformed into a more efficient program only if one can apply suitable properties (or lemmas).

These properties may be applied by replacing some old literals by new literals, that is, by applying the so-called *goal replacement rule*.

Examples of properties are: associativity and commutativity of operations, functionality of

This work has been partially supported by 'Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo' of CNR under grant n. 89.00026.69, 'Progetto Speciale Anatra', and by Esprit Compulog II.

predicates, etc. These properties play a crucial role in the derivation of efficient programs. However, during program transformation it may be difficult to know, at any given step, which property should be applied for obtaining the desired efficiency improvements. This problem can be solved in the cases when one can guarantee efficiency improvements via syntactic conditions on the programs, like, for instance, the linearity of their recursive structure or the absence of unnecessary variables [21]. In these cases, in fact, it is the achievement of those syntactic conditions which suggests the applications of suitable goal replacements.

In this paper we address the problem of proving the correctness of the goal replacements by showing the equivalences between the corresponding pairs of sequences of literals. In particular, we will focus our attention on the unfold/fold proof method introduced in [22]. Our work is related to several other methods which have been proposed in the literature for proving properties of programs using first order logic and computational induction (see, for instance, [1,8,9,11,13,16]).

However, the unfold/fold proof method has the advantage of using the same rules which are used for deriving, by transformation, new programs from old ones. Thus, it can be incorporated into a program transformation system, more easily than other ad-hoc techniques for proving properties of programs.

The idea of the unfold/fold proofs goes back to [15], where it is presented within the framework of functional languages. In the case of logic languages, somewhat related ideas have been described in [4,5].

When applying goal replacements whose equivalences have been shown by unfold/fold proofs (or by any other method), only partial correctness is preserved w.r.t. the computed answer substitution semantics. In this paper we give some sufficient conditions which ensure that, after goal replacements, the derived programs are totally correct. (The notions of partial and total correctness will be introduced below.)

The problem of providing totally correct goal replacement rules has been addressed by several researchers who have considered various semantics for logic programs (see, for instance, [3,10,12,18,19,20,24,25] and [21] for a survey). Our results extend the already available ones to the case of computed answer substitutions for definite and general programs. Moreover, in contrast to most of the previous work, our conditions for total correctness are based on decidable and easily checkable properties of the equivalence proofs.

2 Preliminaries

In what follows we adopt the standard notions used in logic programming [17]. We also introduce the following notations and assumptions.

Given a clause C, we denote its head by hd(C) and its body by bd(C). Given a clause C, we will say that a variable X which occurs in bd(C) is an *existential variable* of C iff X does not occur in hd(C).

We assume that variables of clauses can be renamed, so that two distinct clauses can be assumed not to have variables in common. All operations on sets of clauses, such as union and difference, are defined modulo renaming of variables.

There exists an atom 'fail' which does not unify with the head of any clause. When 'fail' occurs in the body of a clause C we say that C is a *failing clause.*

We also assume that bodies of clauses are *multisets* of literals. When dealing with bodies of clauses, we will use the notions of union, difference, inclusion, etc. in the multiset sense. Notice that the deletion of a duplicate literal in the body of a clause does not preserve the computed answer substitution semantics.

Given a term t, we denote by vars(t) the set of variables occurring in t. Similar notation will be used for variables occurring in literals, goals, and clauses. Given any syntactic construct E, we denote by preds(E) the set of predicate symbols occurring in E. In particular, given the program P, preds(P) denotes the set of predicate symbols occurring in P.

Given a substitution $\sigma = \{X_1/t_1,\ldots,X_n/t_n\}$, dom(σ) denotes $\{X_1,\ldots,X_n\}$ and range(σ) denotes $\{t_1,\ldots,t_n\}$. We assume that in the binding X_i/t_i the term t_i may be equal to X_i. (This assumption is used in Section 4 when introducing SLD- and SLDNF-equivalence of multiset of literals.)

3 Unfold/Fold Transformation Rules

We represent the process of transforming a given program using the unfold/fold rules as the construction of a set of trees of clauses. This representation will be useful for presenting our correctness results and it differs from the one in [24], where the transformation process is represented as a sequence of programs.

An *unfold/fold forest*, also called *UF-forest* for short, for a given program P, is a set of trees, say TF, which is constructed, starting from the empty forest, by applying a (possibly empty) sequence of rules, each of which is either R1 or R2 or R3 or R4 (see below). As formally indicated by these rules, every *positive* literal occurring in the body of the clauses in TF will be marked as *inherited (from a definition)* or *not inherited (from a definition)* (see [23] for a similar notion).

We now list the rules R1, R2, R3, and R4, each of which, given a UF-forest, say E, produces a new UF-forest.

R1. If C is a clause of P which does not label any root node of E, then we add to E a new tree with one node only, labeled by C. All positive literals in bd(C) are not inherited.

R2. *Definition Rule.* By *definition* we derive from P a clause D of the form $newp(X_1,\ldots,X_m) \leftarrow A_1,\ldots,A_n$, such that: i) newp occurs neither in P nor in any root of the trees of E, ii) $A_1,\ldots,A_n$ are *positive* literals, iii) $X_1,\ldots,X_m$ are distinct variables, and iv) $preds(bd(D)) \subseteq preds(P)$.
We add to E a new tree with one node only, labeled by D.
All literals in bd(D) are inherited.

R3. *Unfolding Rule.* Let N be a leaf node of a tree in E labeled by a clause C and let A be a *positive* literal in bd(C) such that $preds(A) \subseteq preds(P)$. By *unfolding* the clause C w.r.t. A we derive the set $\{U_1,\ldots,U_k\}$ of clauses, with $k \geq 0$, as follows. We first consider the set $\{D_1,\ldots,D_k\}$ of clauses in P such that for i=1,...,k, $hd(D_i)$ is unifiable with A via an idempotent most general unifier θ_i. Then, for i=1,...,k, we consider the clause
U_i: $hd(C)\theta_i \leftarrow (bd(C) - \{A\})\theta_i \cup bd(D_i)\theta_i$.
If k=0 then we add one son of N labeled by the clause $hd(C) \leftarrow (bd(C) - \{A\}) \cup \{fail\}$.
The literal 'fail' is not inherited, and each positive literal occurring in $bd(C)-\{A\}$ is inherited iff it is inherited in C.
If k>0 then we add k sons of N labeled by $U_1,\ldots,U_k$, respectively.
For i=1,...,k, in $bd(U_i)$ all positive literals occurring in $bd(D_i)\theta_i$ are not inherited, and each positive literal, say $B\theta_i$, occurring in $(bd(C)-\{A\})\theta_i$ is inherited iff B is inherited in C.

R4. *Folding Rule.* Let N be a leaf node of a tree T in E labeled by a clause C and let D be a clause introduced by applying the definition rule. Let B be a multiset of *positive* literals

contained in bd(C). Suppose that there exists a substitution θ for the variables of bd(D) such that B = bd(D)θ. Let η and ν be the substitutions which are the restrictions of θ to the existential and non-existential variables of bd(D), respectively. Suppose also that η is a variable renaming whose range does not share any variable with (hd(C)) $\cup$ (bd(C) – B) $\cup$ range(ν). By *folding* the clause C *w.r.t.* the multiset B *using* the clause D we derive the clause F = hd(C) $\leftarrow$ (bd(C) – B) $\cup$ {hd(D)θ}.
We add a son N_1 of N labeled by F.
In bd(F) the literal hd(D)θ is not inherited, and each positive literal occurring in bd(C) – B is inherited iff so it is in C.
(The conditions on η and ν ensure that, after folding, we may obtain a variant of the given clause C by unfolding F w.r.t. hd(D)θ using the clause D.)

In the case where a UF-forest is made out of one tree only, we will also call it a *UF-tree*, and we will not distinguish between a singleton UF-forest and the corresponding UF-tree.

Given a program P and a UF-forest TF for P, the set of all clauses which label the roots of the trees in TF is denoted by Roots(TF). The set of all clauses which are not failing and label the leaves of the trees in TF is denoted by Leaves(TF). The set Roots(TF) – P is denoted by Defs(TF). The set Defs(TF) is made out of the clauses which are introduced by the definition rule.

From a UF-forest TF for P, we derive a new program TransfP = (P – Roots(TF)) $\cup$ Leaves(TF). TF will also be called a *UF-forest from* P *to* TransfP.

Example 1. Let us consider the following program P:

p(a) $\leftarrow$
p(X) $\leftarrow$ q(X), r(a)

q(a) $\leftarrow$
q(X) $\leftarrow$ q(X), r(X)
r(b) $\leftarrow$

A UF-forest TF for P is depicted in Fig. 1, where 'u' and 'f' stand for unfolding and folding, respectively, and we have underlined the unfolded atoms (these conventions will be used throughout the paper). During the construction of TF we have introduced, by using the definition rule, the clause: newp(X) $\leftarrow$ q(X), r(a). From TF we derive the new program TransfP:

p(a) $\leftarrow$
p(X) $\leftarrow$ newp(X)
newp(X) $\leftarrow$ newp(X), r(X)

q(a) $\leftarrow$
q(X) $\leftarrow$ q(X), r(X)
r(b) $\leftarrow$. ■

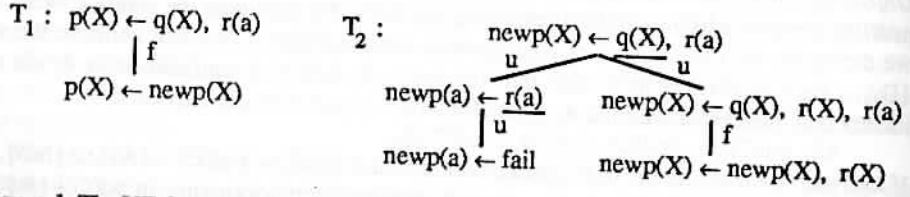

Figure 1. The UF-forest TF for program P of Example 1. TF is made out of the trees T_1 and T_2. Every occurrence of r(a) in T_2 is inherited.

It can be shown that the unfold/fold transformations determined by rules R1, R2, R3, and R4, are *partially correct* w.r.t. the set of answer substitutions computed by SLD-resolution (or SLDNF-resolution), in the sense that, if there exists a UF-forest from a program P to a program TransfP then for the predicates occurring in P the program TransfP computes a

subset of the answer substitutions which are computed by P.

In order to deal with the *total correctness* of those transformations, that is, the equality of sets of answer substitutions computed by P and TransfP, we now consider the following notion.

DEFINITION 1. Let P be a program and let TF be a UF-forest from P to a program TransfP. Suppose that for each tree T in TF which contains a node produced by an application of the folding rule, at least one of the following two conditions holds: i) T has a root node labeled by a clause belonging to P, ii) the sons of the root of T are obtained by an application of the unfolding rule. Then TF is said to be a *progressive* UF-forest for P (or from P to TransfP).

THEOREM 2. [14] (*Total correctness of the unfold/fold transformations w.r.t. SLD–resolution*) Suppose that there exists a progressive UF-forest from a definite program P to a definite program TransfP.
Then for every goal $\leftarrow$ G with preds(G) $\subseteq$ preds(P) we have that P $\cup$ {$\leftarrow$ G} has an SLD-refutation with computed answer θ iff TransfP $\cup$ {$\leftarrow$ G} does.

Finite failures may not be preserved by unfold/fold transformations satisfying the hypotheses of Theorem 2. Indeed, with reference to Example 1, we have that P $\cup$ {$\leftarrow$ p(b)} finitely fails, while TransfP $\cup$ {$\leftarrow$ p(b)} does not. In order to study the preservation of finite failures we introduce the following notion of *fairly progressive* UF-forest.

DEFINITION 3. A UF-forest TF is said to be *fairly progressive* (or simply, *fair*) iff for each tree T in TF and for each node N in T produced by an application of the folding rule w.r.t. a multiset, say B_N, of positive literals, at least one of the following two conditions holds: i) T has a root node labeled by a clause belonging to P, ii) no literal in B_N is inherited.

THEOREM 4. [23] (*Total correctness of the unfold/fold transformations w.r.t. SLDNF-resolution*) Suppose that there exists a fair UF-forest from a general program P to a general program TransfP. Then for every goal $\leftarrow$ G with preds(G) $\subseteq$ preds(P) we have that:
1) P $\cup$ {$\leftarrow$ G} has an SLDNF-refutation with computed answer θ iff TransfP $\cup$ {$\leftarrow$ G} does, and
2) P $\cup$ {$\leftarrow$ G} has a finitely failed SLDNF-derivation iff TransfP $\cup$ {$\leftarrow$ G} does.

Notice that Seki [23] proved the above Theorem 4 for stratified programs only. However, its extension to arbitrary general programs is quite straightforward.

4 The Goal Replacement Rule

A *replacement law* is a triple G $\Rightarrow_V$ H, where G and H are (finite) multisets of *positive* literals and V is a set of variables. The introduction in a replacement law of the parameter set V is due to the fact that when the literals of Gθ in a given clause C are replaced by those of Hθ, for some substitution θ, it is necessary to preserve the computed answer substitutions and the finite failures w.r.t. the set of variables of the context where Gθ occurs. This context is hd(C) $\leftarrow$ (bd(C) – Gθ) (see the goal replacement rule R5 below).

We assume that the variables of any replacement law can be consistently renamed using a renaming substitution. Thus, any replacement law can be assumed to share no variables with

any other replacement law or program clause.

Given a *definite* program P, two multisets G and H of positive literals, and a set V of variables, we say that G and H are *SLD-equivalent* w.r.t. V and P iff the following condition holds:

- *if* $P \cup \{\leftarrow G\}$ has an SLD-refutation with computed answer θ *then* $P \cup \{\leftarrow H\}$ has an SLD-refutation with computed answer η such that $V\theta = V\eta$, and
- *symmetrically*, by interchanging (G,θ) with (H,η).

Given a *general* program P, two multisets G and H of positive literals, and a set V of variables, we say that G and H are *SLDNF-equivalent* w.r.t. V and P iff the following two conditions hold (see [7] for similar requirements):

1) for every substitution σ with $dom(\sigma)=V$,
   - *if* $P \cup \{\leftarrow G\sigma\}$ has an SLDNF-refutation with computed answer θ *then* $P \cup \{\leftarrow H\sigma\}$ has an SLDNF-refutation with computed answer η such that $V\sigma\theta = V\sigma\eta$, and
   - *symmetrically*, by interchanging (G,θ) with (H,η), and
2) for every substitution σ with $dom(\sigma)=V$, $P \cup \{\leftarrow G\sigma\}$ has a finitely failed SLDNF-tree iff $P \cup \{\leftarrow H\sigma\}$ does.

A replacement law $G \Rightarrow_V H$ is said to be *SLD-sound* w.r.t. a definite program P iff G and H are *SLD-equivalent* w.r.t. V and P.

A replacement law $G \Rightarrow_V H$ is said to be *SLDNF-sound* w.r.t. a general program P iff G and H are *SLDNF-equivalent* w.r.t. V and P.

An example of SLD-sound replacement law is given in Example 3 below.

We now generalize the notion of UF-forest introduced in Section 3, by allowing the use of the goal replacement rule. We do so by introducing the notion of *UFR-forest* for a given program P. This forest is a set of trees of clauses constructed by using the rules R1, R2, R3, and R4, defined in Section 3, and also the goal replacement rule R5, defined as follows.

R5. *Goal Replacement Rule.* Let N be a leaf node of a tree T in the current UFR-forest E constructed for a given program P. Let C be the clause labeling N, and $G \Rightarrow_V H$ a replacement law such that $preds(G) \cup preds(H) \subseteq preds(P)$. We consider two cases:

i) there exists a substitution θ such that $G\theta \subseteq bd(C)$ and $vars(hd(C) \leftarrow (bd(C) - G\theta)) = V$.
In this case, by *replacement* of $G\theta$ in C using $G \Rightarrow_V H$, we derive the clause
R: $hd(C) \leftarrow (bd(C) - G\theta) \cup H\theta$.
In bd(R) every positive literals belonging to $H\theta$ is inherited iff *at least* one positive literal of $G\theta$ in C is inherited. Any positive literal occurring in $bd(C) - G\theta$ is inherited iff so it is in C.

ii) G and H are made out of a single atom, $vars(G) = vars(H)$, and there exists a substitution θ such that $\neg G\theta \in bd(C)$ and $vars(hd(C) \leftarrow (bd(C) - \{\neg G\theta\})) = V$. In this case, by *replacement* of $\neg G\theta$ in C using $G \Rightarrow_V H$, we derive the clause
R: $hd(C) \leftarrow (bd(C) - \{\neg G\theta\}) \cup \{\neg H\theta\}$.
Any positive literal occurring in $bd(C) - \{\neg G\theta\}$ is inherited iff so it is in C.

In both cases i) and ii) we add a son N_1 of N labeled by R.

We extend in the obvious way the notions of Roots, Leaves, Defs, progressive UF-forest, fair UF-forest, and derivation of a new program from a UF-forest to the case of UFR-forests.

In particular, in the Definitions 1 and 3 we consider the notion of UFR-forest, instead of UF-forest, and the inheritance marking is assumed to be determined by the rules R1 through R5, instead of R1 through R4, only.

We have the following partial correctness results for the definition, unfolding, folding, and goal replacement transformation rules w.r.t. SLD and SLDNF semantics. Our results extend the ones in: i) [24,25] which refer to the least Herbrand model semantics of definite programs, ii) [14] and [2,23] which do not consider the goal replacement rule, and iii) [12] where a weaker form of folding is assumed.

THEOREM 5. (*Partial correctness of transformations of definite programs w.r.t. SLD-resolution*) Suppose that there exists a progressive UFR-forest from a definite program P to a definite program TransfP. Suppose also that each replacement law used for constructing TF is SLD-sound w.r.t. P. Then for every goal $\leftarrow G$ with $preds(G) \subseteq preds(P)$ we have that:
$P \cup \{\leftarrow G\}$ has an SLD-refutation with computed answer θ if $TransfP \cup \{\leftarrow G\}$ does.

THEOREM 6. (*Partial correctness of transformations of general programs w.r.t. SLDNF-resolution*) Let P be a general program and let TF be a fair UFR-forest from P to a program TransfP. Suppose that each replacement law used for constructing TF is SLDNF-sound w.r.t. P. Then for every goal $\leftarrow G$ with $preds(G) \subseteq preds(P)$ we have that:
1) $P \cup \{\leftarrow G\}$ has an SLDNF-refutation with computed answer θ if $TransfP \cup \{\leftarrow G\}$ does, and
2) $P \cup \{\leftarrow G\}$ has a finitely failed SLDNF-derivation if $TransfP \cup \{\leftarrow G\}$ does.

The proof of Theorem 5 is by induction on the length of the SLD-refutations, and analogously, the one of Theorem 6 is by induction on the length of the SLDNF-derivations.

Notice that, when allowing goal replacements, our rules may be not totally correct, that is, in the conclusions of Theorems 5 and 6 the if's cannot be replaced by iff's. The following Example 2 shows that the goal replacement rule is not totally correct.

Moreover, when applying partially correct goal replacement rules to general programs, nothing is said about the preservation of floundering of the derived programs w.r.t. the initial programs.

Example 2. Let us consider the following program P:

$p \leftarrow q$
$q \leftarrow$

We have that the replacement law $q \Rightarrow_{\{\}} p$ is SLD-sound (and SLDNF-sound, as well) w.r.t. P. On the other hand, by the UFR-forest constructed by applying exactly once the goal replacement rule to clause $p \leftarrow q$, we derive the following program TransfP:

$p \leftarrow p$
$q \leftarrow$

We have that $P \cup \{\leftarrow p\}$ has an SLD-refutation, while $TransfP \cup \{\leftarrow p\}$ does not. ■

In Section 6 we will provide some conditions which ensure total correctness when the goal replacement rule is included in the set of transformation rules.

5 The Unfold/Fold Proof Method

We now address the problem of proving soundness of a replacement law by
unfold/fold transformation rules presented in the previous section. The results
this section show that the unfold/fold proof method considered in [22] for the case of
Herbrand model semantics of definite programs, can be extended, with minor chang
case of computed answer substitutions of definite and general programs. The un
proof method for proving the soundness of a replacement law can be described as follo

DEFINITION 7. Let P be a general (or definite) program and $G \Rightarrow_V H$ a replacement
us consider the clauses:

D_G: $newp(X_1,\ldots,X_n) \leftarrow G$
D_H: $newp(X_1,\ldots,X_n) \leftarrow H$

where newp is a fresh predicate symbol and $\{X_1,\ldots,X_n\} = V$.
Suppose that there exist two UF-trees T_G and T_H for P such that:
- the root of T_G is labeled by D_G, the root of T_H is labeled by D_H, and
- $Leaves(T_G) = Leaves(T_H)$.

We also assume that D_G and D_H are introduced by applying the definition rule.

If T_G and T_H are progressive then we say that the replacement law $G \Rightarrow_V H$ is *provable*
progressive unfold/fold w.r.t. P. We also say that the two UF-trees T_G and T_H consti
progressive unfold/fold proof of $G \Rightarrow_V H$.

If T_G and T_H are fair then we say that the replacement law $G \Rightarrow_V H$ is *provable by*
unfold/fold w.r.t. P. We also say that the two UF-trees T_G and T_H constitute a
unfold/fold proof of $G \Rightarrow_V H$.

In the above definition we have considered the soundeness proof of one replacement
only. The extension to the case where we have a *set* of replacement laws whose soundn
should be *simultaneously* proved by the unfold/fold method, can be done by considerin
instead of a pair of (progressive or fair) UF-trees, a pair of UF-forests whose roots are label
by the clauses whose bodies are the left hand sides and the right hand sides, respectively, o
the replacement laws in the given set.

THEOREM 8. (*Soundness of the unfold/fold proof method w.r.t. SLD-resolution*) Let P be a
definite program. If a replacement law is provable by progressive unfold/fold w.r.t. P then it
is SLD-sound.

PROOF. It is based on Theorem 2 and on the fact that $P \cup \{\leftarrow G\}$ has an SLD-refutation with
computed answer substitution θ iff $P \cup \{newp(X_1,\ldots,X_n) \leftarrow G\} \cup \{\leftarrow newp(X_1,\ldots,X_n)\}$
has an SLD-refutation with a computed answer which is the restriction of θ to $X_1,\ldots,X_n$.

Example 3. (*Ancestors*) The following program, call it Ancestors, defines the ancestor relation
as the transitive closure of the parent relation.

ancestor(X,Y) ← parent(X,Y)
ancestor(X,Z) ← ancestor(X,Y), parent(Y,Z).

Let us consider the replacement law

R: ancestor(X,Y), parent(Y,Z) $\Rightarrow_{\{X,Z\}}$ parent(X,W), ancestor(W,Z)

and the clauses:

newp(X,Z) ← ancestor(X,Y), parent(Y,Z)
newp(X,Z) ← parent(X,W), ancestor(W,Z).

that there exist two progressive UF-trees T_G and T_H for Ancestor such that:
root of T_G is labeled by D_G, the root of T_H is labeled by D_H, and
$Leaves(T_G) = Leaves(T_H)$.
Theorem 8, the replacement law R is SLD-sound w.r.t. the program Ancestor.

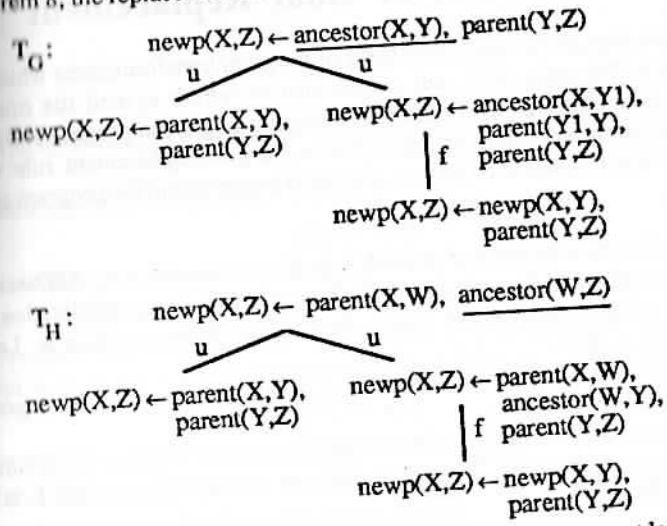

Figure 2. A progressive unfold/fold proof of the replacement law R. ■

The SLDNF-soundness of a replacement law is not ensured by the fact that it is provable
by progressive unfold/fold, as the following example shows.

Example 4. Let us consider the following program:

p ← q, r
q ← q
r ← fail

Consider the replacement law $q, r \Rightarrow_{\{\}} q$. It is not SLDNF-sound w.r.t. the given program.
On the other hand, Fig. 3 shows two progressive UF-trees T_G and T_H such that:
- the root of T_G is labeled by new ← q, r, the root of T_H is labeled by new ← q, and
- $Leaves(T_G) = Leaves(T_H)$. ■

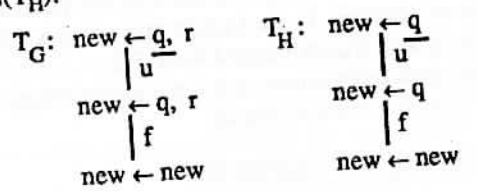

Figure 3. A non fair unfold/fold proof of $q, r \Rightarrow_{\{\}} q$.

In order to ensure SLDNF-soundness, we have to consider fair unfold/fold proofs, as stated by the following result, whose proof is similar to the one of Theorem 8.

THEOREM 9. (*Soundness of the unfold/fold proof method w.r.t. SLDNF-resolution*) Let P be a general program. If a replacement law is provable by fair unfold/fold w.r.t. P then it is SLDNF-sound w.r.t. P

6 Total Correctness of Goal Replacement

In this section we discuss the total correctness of program transformations which use the goal replacement rule. We start with two simple results which extend the ones of [22] by considering the computed answer substitution semantics of general programs. By a *reversible* goal replacement step we mean an application of a goal replacement step which uses a replacement law which is sound both w.r.t. the initial program and the program generated after the replacement.

THEOREM 10. (*Total correctness of reversible goal replacements w.r.t. SLD-resolution*) Let P be a definite program and C a clause in P. Suppose that by an application of the goal replacement rule using a replacement law L, from C we derive a clause R. Let PR be the program $(P - \{C\}) \cup \{R\}$.
If the replacement law L is SLD-sound *both* w.r.t. P and PR, then for every goal $\leftarrow G$ with $preds(G) \subseteq preds(P)$ we have that:
$P \cup \{\leftarrow G\}$ has an SLD-refutation with computed answer θ iff $PR \cup \{\leftarrow G\}$ does.
PROOF. Simple consequence of Theorem 5, because the replacement law L is SLD-sound both w.r.t. P and PR.

THEOREM 11. (*Total correctness of reversible goal replacements w.r.t. SLDNF-resolution*) Let P be a general program and C a clause in P. Suppose that by an application of the goal replacement rule using a replacement law L, from C we derive a clause R. Let PR be the program $(P - \{C\}) \cup \{R\}$.
If the replacement law L is SLDNF-sound *both* w.r.t. P and PR, then for every goal $\leftarrow G$ with $preds(G) \subseteq preds(P)$ we have that:
1) $P \cup \{\leftarrow G\}$ has an SLDNF-refutation with computed answer θ iff $PR \cup \{\leftarrow G\}$ does, and
2) $P \cup \{\leftarrow G\}$ has a finitely failed SLDNF-derivation iff PR does.
PROOF. By Theorem 6 and the fact that the replacement law L is SLDNF-sound both w.r.t. P and PR.

The simple results stated by Theorem 10 and 11 have limited applicability in practice, because for each application of the goal replacement rule two equivalence proofs are required. For instance, in the case of Theorem 11 we have to prove that a replacement law is SLDNF-sound w.r.t. P and that it is also SLDNF-sound w.r.t. PR.

Moreover, a reversible goal replacement step is *not* totally correct when used together with the folding rule, as the following example shows.

Example 5. Let us consider the following program P:

$p(a) \leftarrow$ $\qquad$ $q(X) \leftarrow r(X)$
$p(X) \leftarrow q(X)$ $\qquad$ $r(X) \leftarrow$

and the UFR-forest TF depicted in Fig. 4, where we used the replacement law: $r(X) \Rightarrow_{\{X\}} q(X)$ (see the arc labeled by 'r'), and the only inherited atom is the underlined $q(X)$.

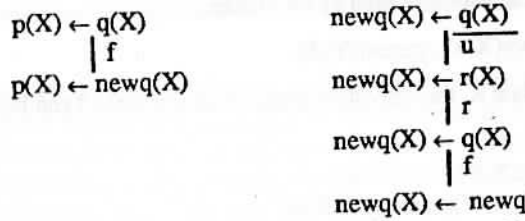

Figure 4. The UFR-forest TF representing a transformation which is not totally correct.

The corresponding transformed program TransfP is:

$p(a) \leftarrow$ $\qquad$ $q(X) \leftarrow r(X)$
$p(X) \leftarrow newq(X)$ $\qquad$ $r(X) \leftarrow$
$newq(X) \leftarrow newq(X)$

The goal replacement is reversible, because the replacement law $r(X) \Rightarrow_{\{X\}} q(X)$ is SLDNF-sound w.r.t. the program *before* the replacement, that is:

$p(a) \leftarrow$ $\qquad$ $q(X) \leftarrow r(X)$
$p(X) \leftarrow newq(X)$ $\qquad$ $r(X) \leftarrow$
$newq(X) \leftarrow r(X)$

and also w.r.t. the program *after* the replacement, that is:

$p(a) \leftarrow$ $\qquad$ $q(X) \leftarrow r(X)$
$p(X) \leftarrow newq(X)$ $\qquad$ $r(X) \leftarrow$
$newq(X) \leftarrow q(X)$.

However, for $G = p(b)$, $P \cup \{\leftarrow G\}$ has an SLDNF-refutation, while $TransfP \cup \{\leftarrow G\}$ does not have an SLDNF-refutation. ■

We now provide some sufficient conditions which ensure that: i) the proof of soundness of a replacement law w.r.t. the program *after* a goal replacement step is not needed, and ii) the goal replacement rule is totally correct, even if it is used together with the folding rule. We first need the following notion.

DEFINITION 12. Let P be a definite program and $G \Rightarrow_V H$ a replacement law which is provable by progressive unfold/fold w.r.t. P, being T_G and T_H the UF-trees of the corresponding proof.
We say that $G \Rightarrow_V H$ is *progressive non-ascending provable w.r.t.* P iff for each path p_G from the root of T_G to a leaf labeled by a clause C different from a failing clause, there exists a path p_H from the root of T_H to a leaf labeled by C such that p_G is not shorter than p_H.

Example 6. (*Ancestors revisited*) Let us consider again the replacement law R:

$ancestor(X,Y), parent(Y,Z) \Rightarrow_{\{X,Z\}} parent(X,W), ancestor(W,Z)$

of Example 3. R is progressive non-ascending provable, because (see Fig 2) for the clause:

C1. newp(X,Z) ← parent(X,Y), parent(Y,Z)

the path from the root of T_G to the leaf labeled by C1 and the path from the root of T_H to the leaf of T_H labeled by the same clause C1 have equal length. The same holds for the paths from the roots of T_G and T_H to the leaves labeled by the clause:

C2. newp(X,Z) ← newp(X,Y), parent(Y,Z).

By using the replacement law R, the Ancestors program of Example 3 can be transformed into the following one:

ancestor(X,Y) ← parent(X,Y)
ancestor(X,Z) ← parent(X,W), ancestor(W,Z).

The total correctness of this transformation is ensured by the following Theorem 13. ■

THEOREM 13. (*Total correctness of non-ascending goal replacement w.r.t. SLD-resolution*) Let P be a definite program and let TF be a progressive UFR-forest from P to a definite program TransfP. Suppose that each replacement law used for the application of the goal replacement rule in TF is progressive non-ascending provable w.r.t. P.
Then for every goal ← G with preds(G) ⊆ preds(P) we have that P ∪ {← G} has an SLD-refutation with computed answer θ iff TransfP ∪ {← G} does.
PROOF. By induction on the length of the SLD-refutations.

Unfortunately, the progressive non-ascending property of Definition 12 does not ensure total correctness w.r.t. SLDNF-resolution, as the following example shows.

Example 7. Let us consider the following program P:

p ← q, r r ← ¬ s
q ← q s ←

The replacement law R: q, r $\Rightarrow_{\{\}}$ p is progressive non-ascending provable w.r.t. P, as shown in Fig. 5 below. By goal replacement using R, we get the following program TransfP:

p ← p r ← ¬ s
q ← q s ←

We have that P ∪ {← p} finitely fails, while TransfP ∪ {← p} does not. ■

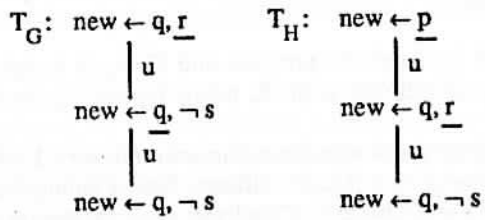

Figure 5. Trees showing that q, r $\Rightarrow_{\{\}}$ p is progressive non-ascending provable.

In order to provide a sufficient condition for the total correctness of our transformation rules w.r.t. SLDNF-resolution, we now introduce the notions of *normal UF-tree* and *proof tree.*

A UF-tree T is said to be *normal* iff no node in T produced by an application of the folding rule has a descendant node produced by an application of the unfolding rule.

It can easily be shown that for every UF-tree T for a program P we can construct a normal UF-tree T1 for P such that the root of T has the same label as T1 and Leaves(T) = Leaves(T1). Thus, without loss of generality, our unfold/fold proof method can be restricted to use normal UF-trees only.

Let us consider a path p of a UF-tree generated by performing h (≥0) unfolding steps starting from a clause C_1 and let $C_1,\ldots,C_{h+1}$ be the sequence of clauses labeling the nodes of p. The *proof tree* corresponding to the path p is the tree T_{h+1} labeled by literals, which is the last tree of the sequence $T_1,\ldots,T_{h+1}$ constructed as follows.

1) If C_1 is the clause $H \leftarrow A_1,\ldots,A_k$ then T_1 is a tree whose root is labeled by H and whose k leaves are the k sons of the root labeled by $A_1,\ldots,A_k$, respectively.
2) For i=1,...,h, from T_i we get the tree T_{i+1} as follows. Let the clause C_{i+1} be derived by unfolding the clause C_i: $K \leftarrow B_1,\ldots,B_{j-1},B_j,B_{j+1},\ldots,B_m$, w.r.t. B_j.
   2.1) If B_j is unifiable with the head of a clause in the given program P via an idempotent mgu θ and C_{i+1} is the clause $(K \leftarrow B_1,\ldots,B_{j-1},D_1,\ldots,D_n,B_{j+1},\ldots,B_m)\,\theta$ then
   - *if* n=0 *then* we add one son of B_j labeled by 'true' *else* we add n sons of B_j labeled by $D_1,\ldots,D_n$, and
   - we apply the substitution θ to all literals in the derived tree.
   2.2) If B_j is unifiable with no head of a clause in the program P then we add one son of B_j labeled by 'fail'.

   We have that, for i=1,...,h+1, in the tree T_i the literals K and $B_1,\ldots,B_m$ which label the root and the leaves of T_i, are the head and the body, respectively, of the clause C_i (apart from possible occurrences of 'true').

Let T be a normal UF-tree and L be a leaf-node of T. Let us consider the path p from the root of T to the leaf-node L. Let p1 be a path of T which is obtained from p by erasing all final nodes generated by folding steps, and let T1 be the proof tree corresponding to p1. The *minimal* (or *maximal*) *unfolding length* of p is the length of the shortest (or longest, respectively) path of T1.

DEFINITION 14. Let P be a general program and G $\Rightarrow_V$ H a replacement law which is provable by *fair* unfold/fold w.r.t. P, being T_G and T_H the UF-trees of the corresponding proof. We say that G$\Rightarrow_V$H is *fair non-ascending provable w.r.t.* P iff i) T_G and T_H are normal UF-trees, and ii) for each path p_G from the root of T_G to a leaf labeled by a clause C and for each path p_H from the root of T_H to a leaf labeled by C, we have that the minimal unfolding length of p_G is not shorter than the maximal unfolding length of p_H.

Example 8. With reference to Example 7, in Fig. 6 we show the proof trees corresponding to the paths of the two UF-trees T_G and T_H of Fig. 5. From Fig. 6 it is easy to see that the minimal unfolding length of the only path (from the root to the leaf) of the UF-tree T_G, which is 2 (see the left proof tree of Fig. 6), is *shorter* than the maximal unfolding length of the only path (from the root to the leaf) of the UF-tree T_H, which is 3 (see the right proof tree of Fig. 6). Thus, we cannot conclude that the replacement law q, r $\Rightarrow_{\{\}}$ p is fair non-ascending provable w.r.t. P. ■

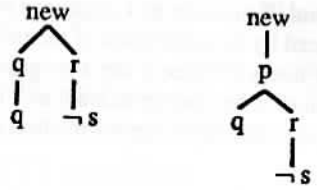

Figure 6. Proof trees corresponding to the unfold/fold proof of $q, r \Rightarrow_{\{\}} p$.

THEOREM 15. (*Total correctness of non-ascending goal replacement w.r.t. SLDNF-resolution*) Let P be a general program and let TF be a fair UFR-forest from P to a program TransfP.

Suppose that each replacement law used for the application of the goal replacement rule in TF is fair non-ascending provable w.r.t. P.

Then for every goal $\leftarrow G$ with $preds(G) \subseteq preds(P)$ we have that:

1) $P \cup \{\leftarrow G\}$ has an SLDNF-refutation with computed answer θ iff $TransfP \cup \{\leftarrow G\}$ does, and
2) $P \cup \{\leftarrow G\}$ has a finitely failed SLDNF-derivation iff $TransfP \cup \{\leftarrow G\}$ does.

7 Conclusions

Our main contribution consists in extending the results obtained by [18,24,25] to the case of computed answer substitution and finite failure semantics both for definite and general programs. Our results are significant for the practical utilization of the program transformation methodology, because the conditions we state for showing the total correctness of the goal replacements are syntactically checkable (unlike, for instance, those of [3,4,7,12,19, 20]), and usually they require very little computational efforts. Indeed, simple computations on the proof trees are enough for checking those conditions.

Moreover, the unfold/fold proof method can easily be incorporated into any transformation system for program development, because it uses the same transformation rules. Thus, our proof method does not require the invocation of ad-hoc Theorem Provers whose integration with program transformation systems may be difficult or computationally expensive.

Acknowledgements

We want also to thanks the referee and our colleagues of the Compulog II Project for many fruitful comments and discussions.

References

[1] A. Bossi and N. Cocco. Verifying Correctness of Logic Programs. In *Proc. TAPSOFT '89*, Vol.2, LNCS n.352, Springer-Verlag, 1989, 96–110.

[2] A. Bossi and N. Cocco. Basic Transformation Operations which Preserve Computed Answer Substitutions of Logic Programs. *J. Logic Programming*, 16, 1993, 47–87.

[3] A. Bossi, N. Cocco, and S. Etalle. Transforming Normal Programs by Replacement. In A. Pettorossi, editor, *Proc. 3rd Int. Workshop on Meta-Programming in Logic* (Meta '92), Uppsala, Sweden, LNCS n. 649, Springer-Verlag, 1992, 265–279.

[4] A. Bossi, N. Cocco, and S. Etalle. On Safe Folding. In *Proc. PLILP '92*, Leuven, Belgium, LNCS n. 631, Springer-Verlag, 1992, 172–186.

[5] D. Boulanger and M. Bruynooghe. Deriving Unfold/Fold Transformations of Logic Programs Using Extended OLDT-based Abstract Interpretation. *J. Symbolic Computation*, Vol. 15, 1993, 495–521.

[6] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *JACM*, Vol. 24, No. 1, January 1977, 44–67.

[7] J. Cook and J.P. Gallagher. A Transformation System for Definite Logic Programs based on Termination Analysis. In *Proc. Lopstr '94*, Pisa, Italy, 1994.

[8] K.L. Clark and S.-Å. Tärnlund. A First Order Theory of Data and Programs. In *Proceedings Information Processing 77*, North Holland, 1977, 939–944.

[9] P. Deransart. Proof Methods of Declarative Properties of Logic Programs. In *Proc. TAPSOFT '89*, Vol.2, LNCS n.352, Springer-Verlag, 1989, 207–226.

[10] L. Fribourg. Equivalence-Preserving Transformations of Inductive Properties of Prolog Programs. In *Proceedings Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988, 893–908.

[11] M. Gabbrielli, G. Levi, and M.C. Meo. Observational Equivalences for Logic Programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, 1992, 131–145.

[12] P.A. Gardner and J.C. Shepherdson. Unfold/Fold Transformations of Logic Programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, MIT Press, 1991, 565–583.

[13] T. Kanamori and H. Seki. Verification of Prolog Programs Using an Extension of Execution. In *Proceedings of the Third International Conference on Logic Programming*, LNCS n. 225, Springer-Verlag, 1986, 475–489.

[14] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation. *Theor. Comp. Sci.* 75, 1990, 139–156.

[15] L. Kott. The McCarthy's Recursion Induction Principle: 'Oldy' but Goody'. *Calcolo*, 19, 1, 1982, 59–59.

[16] J.M. Lever. Proving Program Properties by means of SLS-Resolution. In *Proc. of the Eighth Int. Conference on Logic Programming*, MIT Press, 1991, 614–628.

[17] J.W. Lloyd. Foundations of Logic Programming. Springer-Verlag, 1987.

[18] M.J. Maher. Correctness of a Logic Program Transformation System. *IBM Research Report* RC 13496, T.J. Watson Research Center, 1987.

[19] M.J. Maher. Reasoning About Stable Models (and other Unstable Semantics). *IBM Research Report*, T.J. Watson Research Center, 1990.

[20] M.J. Maher. A Transformation System for Deductive Database Modules with Perfect Model Semantics. *Theoretical Computer Science* 110, 1993, 377–403.

[21] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *IASI-CNR Technical Report* n.369, Roma (Italy), 1993.

[22] M. Proietti and A. Pettorossi. Synthesis of Programs from Unfold/Fold Proofs. In Y. Deville, editor, *Proceedings of Lopstr '93*, Louvain-la-Neuve, Belgium, 7–9 July, 1993, Workshops in Computing , Springer-Verlag, 1994, 141–158.

[23] H. Seki. Unfold/Fold Transformation of Stratified Programs. *Theoretical Computer Science* 86, 1991, 107–139.

[24] H. Tamaki and T. Sato. Unfold/Fold Transformation of Logic Programs. In *Proc. 2nd Int. Conf. Logic Programming*, Uppsala, Sweden, 1984, 243-251.

[25] H. Tamaki and T. Sato. A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation. *Ibaraki University Tech. Report* 86-4, Japan, 1986.