ii) let M be a node labelled by a clause of the form: 'H ← A_1,...,A_h,..., A_k', and let A_h be the defined atom selected by the U-Selection rule S.
For each clause 'A ← B_1,...,B_s' in Prog such that there exists a most general unifier σ of A and A_h, M has a son-node N labelled by the clause: (H ← A_1,..., A_{h-1}, B_1,..., B_s, A_{h+1},..., A_k) σ.
We will assume that the rule S is a *partial* function and the clauses for which it is not defined, are leaves of the U-tree. Moreover, if the atom selected by S cannot be unfolded because no head can be unified with it, then the corresponding clause is a leaf of the U-tree. ■

DEFINITION 2. Given a tree T we say that the subtree R of T is an *upper portion* of T iff i) if a node N is in R then also every ancestor of N in T is in R, and ii) if a node N is in R then also every brother of N in T is in R. ■

One can easily show that the least Herbrand model of a program Prog∪{C} is equal to the least Herbrand model of Prog∪L, where C is a definition clause and L is the set of leaves of any upper portion of a U-tree for <Prog, C> via any U-selection rule.

Example 2. Let us consider again the Text-Processing Example, where clause 1.1 is considered to be a definition. The unfolding process starting from clause 1.1, and using clauses 1.2,...,1.9, can be represented by the finite upper portion of the U-tree depicted in Figure 1 below. The <ancestor-clause, foldable clause> pairs (that is, the loops of recurrent patterns of atoms) are: <1.1,3>, <1.1,7>, <1.1,13>, <6,11>,<8,12>, <9,14>, <4,15>, and they are depicted by dashed arrows pointing towards the ancestor-clause. The atom selected for unfolding is delchar in the case of clauses 1.1, 6, 8, and 9, while it is delbcommas for clauses 2 and 4. ■

Notations used in figures. i) Solid arrows denote unfolding steps. ii) Boxed clauses are clauses without defined atoms in their bodies. iii) Bold underlinings and/or dotted lines denote the frontier of the *optimal* upper portion of a foldable U-tree. (This notion will be introduced below.) iv) A predicate in round brackets near a clause denotes that the body of that clause contains a variant of the body of the definition for that predicate. v) A dashed arrow from clause x to clause y associated with the predicate p in round brackets denotes *either* the pair <ancestor-clause y, foldable clause x> *or* a folding step which can be performed using x as old clause and the definition of p as bridge clause. vi) Primed numbers are used to indicate the correspondence between clauses in different figures. ■

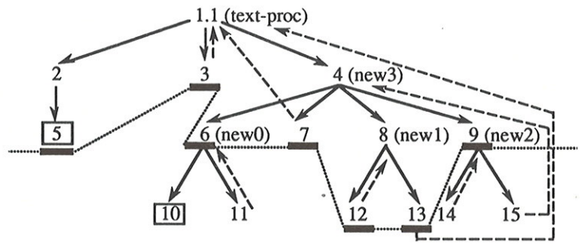


Figure 1. Text-Processing Example. The minimal foldable upper portion of the U-tree for <{clause 1.2,..., clause 1.9}, clause 1.1>.

DEFINITION 3. Let Prog be a program, C a definition clause, and S a U-selection rule. A clause D in a U-tree for <Prog,C> via S is said to be *foldable* iff there is an ancestor-clause A of D in the U-tree such that a *subset of the defined atoms* in the body of D is a variant of the set of *all defined atoms* in the body of A. The clause in the path from the root to D which is the nearest to D satisfying the properties of A, will be called 'the' ancestor-clause of D. ■

DEFINITION 4. Let Prog be a program, C a definition clause, and S a U-selection rule. The *U-tree* for <Prog,C> via S is said to be *foldable* iff it has a *finite* upper portion such that for each leaf-clause L whose body i) has at least one defined atom, and ii) has all defined atoms which unify heads in Prog, we have that L is a foldable clause. That portion will be called *foldable upper portion* of the U-tree. ■
REMARK. The notion of foldability presented here is slightly different from the ones in [Pettorossi-Proietti 89] and in [Proietti-Pettorossi 90]. In the first paper we assume that we are given a set of definitions and we study the problem of constructing the upper portion of a U-tree such that all leaves can be folded using the given definitions as bridge clauses. The requirement of performing those folding steps on the leaves causes the introduction of some extra definitions, namely, the eureka predicates, which indeed allow the desired folding steps, but we are asked to construct some more foldable U-trees, one for each eureka predicate we have introduced. Here, on the contrary, we do *not* assume that a set of definitions is given to us, but we get that set for free once the Foldability Problem (see below) has been solved.
The notion of foldability in [Proietti-Pettorossi 90] is a bit stronger than the one we consider here. In that paper we say that a clause in a U-tree is foldable if the *set* (not a *subset*) of all defined atoms in its body is an instance of the set of the defined atoms in the body of an ancestor clause. As we will see in Sections 3 and 4, this weakening of the foldability notion allows larger classes of programs in which our transformation methodology is successful. ■

We will often consider the *minimal* foldable upper portion of a U-tree, that is, a foldable upper portion which has no foldable upper portions different from itself. The minimal foldable upper portion of a U-tree for <{clause 1.2,...,clause 1.9}, clause 1.1> is depicted in Figure 1.

We will now present a procedure, called *Loop Absorption with Fold on Root*, which realizes the Loop Absorption strategy. Given a foldable U-tree for a program Prog and a clause C, that procedure finds for us the eureka predicates and it derives a program equivalent to Prog∪{C} with improved performances and an optimal structure, as specified in Definition 5.

DEFINITION 5. Given a program Prog, a definition clause C, and a foldable upper portion T1 of a U-tree for <Prog, C>, let us consider the set S of the *upper portions* of T1 such that each element T in S satisfies the following conditions:
i) the leaves of T are either leaves of T1 or ancestor-clauses of foldable clauses of T1, and
ii) all foldable clauses of T1 whose ancestor clause is the *root-clause* C are nodes of T.
An *optimal upper portion* Topt of T1 is an element of S having the minimal number of leaves. ■

Obviously, T1 is an element of S. Condition ii) is based on the observation that better performances can be obtained if the clauses of the final program for evaluating the head predicate of a clause, say C, are derived by performing all possible foldings using the same clause C as bridge clause. Those folding steps are the ones relative to the foldable clauses mentioned in condition ii) above. Indeed, for producing the program clauses for the head predicate of clause C, the following Procedure 1 uses clause C itself as root of the optimal upper portion of a foldable U-tree (and this fact explains the name of that procedure).

PROCEDURE 1. *Loop Absorption with Fold on Root*.
Suppose we are given a program Prog, a definition C, and a U-selection rule S such that there exists a foldable U-tree for <Prog, C> via S. Let DEFdone and DEFdo be sets of definitions, and FinalProg be the new program to be derived.
- Initialize DEFdone to ∅, DEFdo to {C}, and FinalProg to Prog\{C}.
- While there is a definition-clause, say D, in DEFdo perform the following actions:
  1. Construct a minimal foldable upper portion T1 of the U-tree for <Prog, D> via S.
  2. Construct an optimal upper portion, say Topt, of T1.

3. For each foldable leaf-clause P of T1 such that either P itself or its corresponding ancestor clause A_P is a leaf of Topt, define a new eureka predicate newp by introducing a definition-clause D_P whose body consists of the set B of defined atoms in the body of A_P. The set of variables of newp is the minimal set V which allows to fold both A_P and P, using D_P as a bridge clause.
   Add D_P to DEFdo only if it cannot be found a renaming substitution ρ and a clause D1 in DEFdone $\cup$ DEFdo such that the set of variables occurring in the head of D1 is $V\rho$ and the body of D1 is $B\rho$. (Informally speaking, D_P is added to DEFdo only if an equivalent definition is *not* already in DEFdone $\cup$ DEFdo.)
4. Consider the set Ps of all clauses which are leaves of Topt.
5. Simplify the base predicates in the clauses of Ps.
6. Delete from Ps all clauses which are subsumed by other clauses in Ps, and all clauses whose bodies contain defined atoms which do not unify any head in Prog.
7. Fold the clauses in Ps using those in DEFdone $\cup$ DEFdo as bridge clauses.
8. Discard D from DEFdo and add it to DEFdone.
9. Add to FinalProg the clauses in Ps.

• Discard from FinalProg all clauses which are redundant, in the sense that they are not necessary for the evaluation of the required predicates. (This step is in general not algorithmic, but simple abstract interpretation techniques often suffice.) ■

The above procedure always terminates and it derives improved programs with the optimality property presented in Definition 5. The efficiency improvements of FinalProg w.r.t. Prog are as usual, due to the folding steps which allow us to derive the recursive definitions of the eureka predicates. Let us now revisit the Text-Processing example to explain this procedure.

Example 3. Text-Processing revisited.
It is not difficult to see that the derivation of the final program for the Text-Processing Example we have presented in the Introduction, is an application of the Loop Absorption with Fold on Root Procedure described above. In particular, in order to find the clauses for text-proc we construct the minimal foldable upper portion of the U-tree of Figure 1. We consider all clauses which *fold on the root* (3, 7, and 13) and we look for a 'cut nearest to the root' of that U-tree which goes through those clauses, and also through *either* the boxed clauses, *or* the starting *or* the arrival clauses of dashed arrows. In our case it is made out of the clauses 5, 3, 6, 7, 12, 13, and 9. Notice that we cannot use clause 8 instead of 12 and 13 because 13 'folds on the root' and it cannot be discarded from the cut. Analogously the choice of 8 and 13, instead of 12 and 13, is not possible because it is not a cut and it creates redundancy of solutions at run time.
The cut we have found provides the clauses for the predicate text-proc. In fact, it is enough to perform all possible folding steps using the new definitions, that is, the eureka predicates, suggested by the recurrent patterns of the defined atoms in the bodies of the ancestor-clauses occurring in the foldable U-tree. (Recall that those clauses are pointed to by dashed arrows.)
The process of extracting the final program from the minimal upper portion of a foldable U-tree continues by finding the program clauses for the eureka predicates. Let us consider the process for new1, defined by clause 17. (Analogous processes can be followed for the other eurekas.)
We construct the minimal upper portion of a foldable U-tree for new1 (see Figure 2). It can be derived from the tree in Figure 1 by unrolling the loops denoted by the dashed up-arrows. Indeed, it is enough to unfold clause 17 as indicated by the tree below clause 8 of Figure 1. The unfolding proceeds as indicated by the numbering notation which uses the prime signs to relate clauses in different figures.
The clauses which fold on root are 12' and 8'. The cut nearest to the root is made out of the clauses 12', 5', 3', 6', 7', 8', 9' (see our seven clauses for new1 in the Example 1 and Figure 2). Notice that for new3 it is not necessary to derive the corresponding clauses, because no cuts for text-proc, new0,

new1, and new2 goes through a clause which either has new3 as associate predicate or points to such a clause by a dashed arrow.

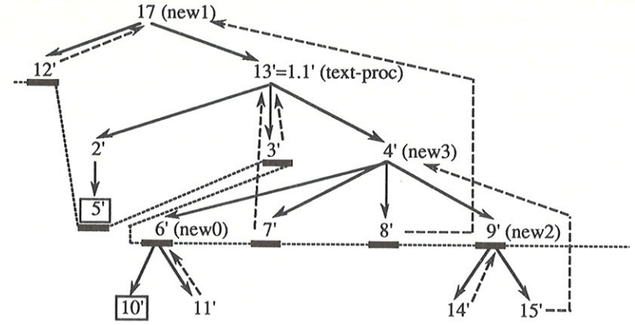


Figure 2. Text-Processing Example. The minimal foldable upper portion of the U-tree for <{clause 1.2,..., clause 1.9}, clause 17>.

Now, in order to evaluate our Loop Absorption with Fold on Root Procedure, let us compare it with another procedure, called *As Soon As Possible Procedure*, for extracting programs from foldable U-trees. Using that procedure the clauses for the initial predicate and for the eurekas are obtained by performing folding steps *as soon as* the definitions of those same predicates can be used as bridge clauses. In Figure 3 below we depicted the U-trees, one for each predicate, which correspond to the extraction of program clauses using the As Soon As Possible Procedure.
Thus, for instance, for text-proc we get 3 clauses corresponding to 5', 3', and 4' folded using new3.

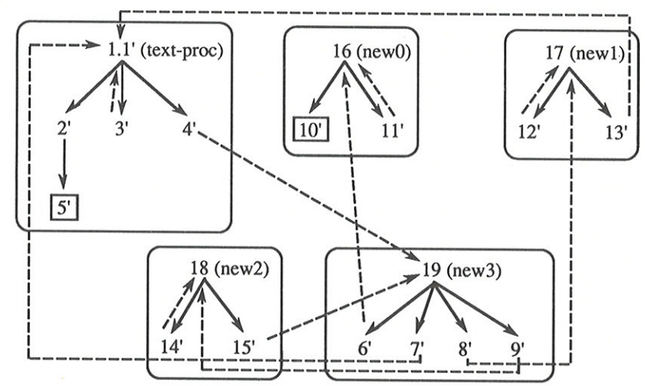


Figure 3. Extracting program clauses via the 'As Soon As Possible' Procedure from the U-tree of Figure 1.

The final program is:
text-proc(C,[],[]).
text-proc(C,[C|T],NT) ← text-proc(C,T,NT).

```
text-proc(C,[D|T],NT) ← C≠D, new3(C,T,D,NT).
new0(C,[]).
new0(C,[C|T]) ← new0(C,T).
new1(C,[C|T],NT) ← new1(C,T,NT).
new1(C,[comma|T],NT) ← C≠comma, text-proc(C,T,NT).
new2(C,[C|T],D,NT) ← new2(C,T,D,NT).
new2(C,[D|T],D,NT) ← C≠D, new3(C,T,D,NT).
new3(C,T,blank,[blank]) ← new0(C,T).
new3(C,T,D,[D|NT]) ← D≠blank, text-proc(C,T,NT).
new3(C,T,blank,[comma|NT]) ← new1(C,T,NT).
new3(C,T,blank,[blank|NT]) ← new2(C,T,D,NT), D≠comma.
```

The efficiency of this program is not as good as the one derived by using our Procedure 1 which 'folds on root'. Experimental results in C-Prolog show that 'Folding on Root' increases efficiency w.r.t. 'As Soon As Possible' of about 7% in time and 43% in space for lists of about 1800 characters.
We do not have yet a theory that gives a full account of such results, but it is obvious that if folding steps are performed too early, then the execution of the final program basically imitates the one of the initial program version. ■

3. THE TRANSFORMATION METHODOLOGY AND THE SOLUTION OF THE FOLDABILITY PROBLEM FOR SOME CLASSES OF PROGRAMS

As we have shown in the previous Sections, our transformation technique requires the construction of foldable U-trees. In this Section we will address the problem of determining classes of programs for which those foldable U-trees can be found.

We formalize the *Foldability Problem* as follows: Given a program Prog and a definition C, is there a U-selection rule S such that the U-tree for <Prog,C> via S is foldable?
THEOREM 6. The Foldability Problem is partially solvable and it is not solvable.
Proof. The Halting Problem for Turing Machines can be reduced to the Foldability Problem. ■

In the remaining part of this Section we introduce the class of non-ascending programs, and we show that foldable U-trees do exist for suitable subclasses of those programs.

DEFINITION 7. Let X be a variable or a constant and t be a term where X occurs. The depth of X in t, denoted by depth(X,t), is defined by structural induction as follows:
- if t=X then depth(X,t) = 0, and for any function symbol f
- if $t=f(t_1,\ldots,t_n)$ then $depth(X,t) = \max\{depth(X,t_i) \mid X \text{ occurs in } t_i \text{ and } i=1, \ldots, n\}+1$.

Given a term t we denote by maxdepth(t) the depth of its deepest variable or constant.
Given a term t, we denote by vars(t) the set of variables occurring in t.
If t and u are linear terms we say that $t \leq u$ iff for each variable X in $vars(t) \cap vars(u)$ we have: $depth(X,t) \leq depth(X,u)$.
Notice that if $t=f(t_1,\ldots,t_n)$ is a linear term and X is a variable, the set $\{depth(X,t_i) \mid X \text{ occurs in } t_i \text{ and } i=1, \ldots, n\}$ is a singleton.
The above definitions of depth, maxdepth, and vars, and of the $\leq$ relation can be extended from terms to atoms by considering the predicate symbols as term constructors. ■

Given two atoms A_1 and A_2 such that $vars(A_1) \cap vars(A_2) \neq \emptyset$, we will write $A_1 \sim A_2$.
THEOREM 8. Let Prog be a program, C a definition, and S a U-selection rule. Let us consider the U-tree T for <Prog, C> via S. Suppose that S is defined for all clauses in T with at least a defined atom in their bodies, and there exist two positive integers H and W such that for each clause D in T:

i) $\max\{maxdepth(A) \mid A \text{ is a defined atom of the body of } D\} \leq H$, and
ii) each sequence $<A_1,A_2,\ldots,A_m>$ of different defined atoms of the body of D such that:
$A_i \sim A_{i+1}$ for i=1,…,m-1, is not longer than W, that is, $m \leq W$.
Then T is a foldable U-tree. ■
In the sequel the sequences of different defined atoms of condition ii) of the above Theorem 8 will be called *variable-chained sequences*.

REMARK. Condition ii) of Theorem 8 cannot be dropped. Indeed, let us consider the case where p(X,Y) ← p(X,Z), q(Z,Y) is in Prog, and C = h(X,Y) ← p(X,Y), r(Y).
Suppose that p, q, and r are defined predicates in Prog. By unfolding the leftmost atom in the body of any clause of the U-tree for <Prog, C> we get a path from the root with the following clauses:
1. h(X,Y) ← p(X,Y), r(Y) (which is clause C),
2. h(X,Y) ← p(X,Y1), q(Y1,Y), r(Y)
3. h(X,Y) ← p(X,Y2), q(Y2,Y1), q(Y1,Y), r(Y), etc.

In this path condition i) of Theorem 8 is satisfied, but condition ii) is not. It is easy to see that no clause of that path is foldable. ■

DEFINITION 9. A term, an atom or a conjunction of atoms is *linear* iff each variable occurs in it at most once. ■
DEFINITION 10. A clause C of a program Prog is said to be a *linear-term clause* (or LT-clause) iff each defined atom in C is linear. If an LT-clause is a definition we will simply say that it is an LT-definition. Prog is said to be a *linear-term program* (or LT-program) iff it is made out of LT-clauses only. A clause C of a program Prog is said to be a *strongly-linear-term clause* (or SLT-clause) iff the conjunction of the defined atoms *in its body* is linear. Prog is said to be a *strongly-linear-term* program (or SLT-program) iff it is made of SLT-clauses only. ■
DEFINITION 11. Let Prog be an LT-program, and C a clause of Prog of the form:
$H \leftarrow A_1,\ldots,A_m,B_1,\ldots,B_n$, where $A_1,\ldots,A_m$ are the defined atoms.
C is said to be *non-ascending* iff $A_i \leq H$ for i=1,…,m.
Prog is said to be non-ascending iff all its clauses are non-ascending.
C is said to be *strongly non-ascending* iff non-ascending and it is an SLT-clause.
Prog is said to be strongly non-ascending iff all its clauses are strongly non-ascending. ■

Example 4. The following clause of the Text-Processing program (see Example 1) is an LT-clause:
delbcommas([blank,comma|T],[comma|T1]) ← delbcommas(T,T1).
The above clause is also a strongly non-ascending clause. The clause:
delchar(C,[C|T],T1) ← delchar(C,T,T1).
is *not* an LT-clause because the variable C occurs twice in its head.
The following clause is an LT (and non-ascending) clause:
text-proc(Char,Text,NewText) ← delchar(Char,Text,D), delbcommas(D,NewText).
It is *not* an SLT-clause, because the variable D occurs twice in the defined atoms of its body. ■

DEFINITION 12. A *Synchronized Descent Rule* (called simply SDR) for an LT-program Prog is a partial function defined as follows. Let D: H ← Body be an LT-clause and $\{A_1,\ldots, A_n\}$ be the set of defined atoms (w.r.t. Prog) in Body. If there exists an index i such that $A_j \leq A_i$ for j=1,…,n then SDR(D,Prog) = A_i. If such an index does not exist then SDR is undefined. ■

We will often refer to an SDR without specifying the corresponding clause D and program Prog, when they can easily be understood from the context. Notice that for any given clause D and program Prog we can have more than one SDR, because the choice of the index i in the above definition is not always unique. For simplicity reasons, unless otherwise specified, we may refer to 'the' SDR instead of 'an' SDR, when the statement is valid for any choice of SDR.

THEOREM 13. Let Prog be a non-ascending program and C an LT-definition, and S an SDR.

Suppose that there exists a positive integer W such that, given any clause D in the U-tree T for <Prog, C> via S, we have that each variable-chained sequence of defined atoms of the body of D is not longer than W. Then U-tree T is foldable if S is defined for all clauses in T with at least a defined atom in their bodies. ■

THEOREM 14. Let Prog be a strongly non-ascending program and C be an LT-definition of the form: $h(\ldots) \leftarrow p_1(\ldots), p_2(\ldots)$. Assume that:
i) for each clause D in Prog of the form: $p(\ldots) \leftarrow q_1(\ldots),\ldots, q_n(\ldots),\ldots$, where $q_1,\ldots, q_n$ are the only defined predicates in the body of D, and for each i=1, …, n, we have that:
  i.1) for each argument r and s of q_i and p, respectively, if r has a variable in common with s then r is a subterm of s,
  i.2) distinct arguments of q_i are not subterms of the same argument of p,
ii) there exist two arguments t_1 and t_2 of $p_1(\ldots)$ and $p_2(\ldots)$, respectively, such that:
  ii.1) either t_1 is a subterm of t_2, or t_2 is a subterm of t_1 (the subterm relation is assumed to be reflexive), or $vars(t_1) \cap vars(t_2) = \emptyset$, and
  ii.2) $vars(p_1(\ldots)) \cap vars(p_2(\ldots)) = vars(t_1) \cap vars(t_2)$.
Then there is a foldable U-tree for <Prog, C>. ■
REMARK. The class of pairs <Prog, C> which satisfy the hypotheses of the above theorem extends the one presented in [Proietti-Pettorossi 90]. ■

Example 5. Text-Processing revisited.
Let us consider again the Text-Processing program of Example 1. Let Prog be the set of clauses: {clause 1.2,…,clause 1.9}, and let C be clause 1.1. The pair <Prog, C> does *not* satisfy the hypotheses of Theorem 14 because clauses 1.3, 1.4, and 1.7 are *not* strongly non-ascending. However, they can be transformed into the following strongly non-ascending clauses by using the equality predicate:
1.3* delchar(C,[C1|T],T1) ← C=C1, delchar(C,T,T1).
1.4* delchar(C,[C1|T],[C2|T1]) ← C1=C2, C≠C1, delchar(C,T,T1).
1.7* delbcommas([C|T],[C1|T1]) ← C=C1, C≠blank, delbcommas(T,T1).
Now Prog1={clause 1.2, clause 1.3*, clause 1.4*, clause 1.5, clause 1.6, clause 1.7*, clause 1.8, clause 1.9} together with clause C satisfies the hypotheses of Theorem 14. The reader may verify that we get a foldable upper portion of the U-tree for <Prog1, C> which is equal to that for <Prog, C> shown in Figure 1 except for some equality predicates in the bodies of the clauses. By applying the Loop Absorption with Fold on Root Procedure we get exactly the final program version of Example 1. ■

4. THE GENERALIZATION STRATEGY AND THE TRANSFORMATION TECHNIQUE FOR UNRESTRICTED PROGRAMS

We now introduce one more strategy, called *Generalization strategy*. It is useful to apply that strategy if during program derivation we are not able to obtain a foldable U-tree by performing unfolding steps only, because the program at hand is outside the classes considered in Theorems 13 and 14.
We will see that by performing generalization steps, we get foldable U-trees in all cases, but we cannot guarantee that the derived programs have better performances.

DEFINITION 15. The application of the *Generalization + Equality Introduction Rule* (or *Generalization Rule*, for short) to a clause C of the form: $H \leftarrow A_1, \ldots, A_n$ consists in deriving the new clause of the form: $H \leftarrow GenA_1, \ldots, GenA_n, X_1=t_1, \ldots, X_r=t_r$, where:
$(GenA_1, \ldots, GenA_n)\,\theta = (A_1, \ldots, A_n)$ and $\theta = \{X_1/t_1, \ldots, X_r/t_r\}$. ■
Obviously, when applying the Generalization Rule the least Herbrand model is preserved.

We now allow in the program derivation process unfolding steps and generalization steps. That process can be represented as a tree of clauses, which is still called U-tree. As for unfolding steps, when we perform generalization steps we produce a new son-clause. The equality predicates in the son-clauses should be considered as base predicates and therefore, they will not be unfolded when constructing the U-tree. The notion of foldability of a U-tree is not changed.

Let us now recall that Theorem 8 ensures foldability of a U-tree if in the bodies of its clauses some conditions on the maxdepth of the defined atoms and on the length of their variable-chained sequences are satisfied.
By performing suitable generalization steps, it is possible to transform any clause into an equivalent one for which those conditions are satisfied. Consider, for instance, the following clause C:
  h(X,Y) ← p(t(X,s(Y))), q(t(Y,Z)),r(Z),
where we assume that h, p, q and r are defined predicates.
Clause C has the atom p(t(X,s(Y))) whose maxdepth is 3. In that clause the variable-chained sequence <p(t(X,s(Y))), q(t(Y,Z)), r(Z)> has length 3. Clause C can be transformed into an equivalent one, that is, h(X,Y) ← p(t(X,Y1)), Y1=s(Y), q(t(Y,Z)), r(Z), where 2 is the maximal maxdepth for each defined atom in the body, and 2 is also the maximal length of the variable-chained sequences of the defined atoms in its body. We have the following theorem.

THEOREM 16. Given any program Prog and clause C, there exists a foldable U-tree for <Prog, C>, and it can be constructed using unfolding steps and generalization + equality introduction steps. ■

The procedure below constructs a foldable U-tree for any given program and any clause by performing some generalization + equality introduction steps.

PROCEDURE 2. *Construction of foldable U-trees by generalization + equality introduction.*
Let Prog be a program, C a definition clause, and S a U-selection rule which is defined for all clauses which have at least a defined atom in their bodies. Let H and W be two positive integers, called *vertical* and *horizontal* bound, respectively.
We construct the U-tree for <Prog,C> from clause C by performing unfolding steps via S until we get a foldable U-tree. If during that construction we get a clause D in whose body *either* there exists a defined atom A with maxdepth(A)>H, *or* there exists a variable-chained sequence of defined atoms longer than W, then we perform some generalization + equality introduction steps so that:
- $\max\{maxdepth(A) \mid A$ is a defined atom in the body of $D\} \leq H$, and
- each variable-chained sequence of defined atoms is not longer than W. ■

Notice that generalization steps should be performed with parsimony, because they reduce the number of shared variables among atoms, and if no shared variables occur in the bodies of the foldable clauses of the U-tree, it is not possible to avoid redundant computations of bindings and/or the construction of unnecessary intermediate data structures.

When constructing a foldable U-tree using Procedure 2, the generalization steps are related to the choice of the bounds H and W. One may give various heuristics for establishing suitable values for H and W. In the example below the actual choices will be motivated by the desire of avoiding some generalization steps.

Example 6. Searching a string in a text.
In this example we apply again our transformation methodology, and we show that it can be used for solving a partial evaluation problem.
Let us consider two strings S and T of characters. We say that S occurs in the 'text' string T at position P iff there exist two (possibly empty) strings BeforeS and AfterS such that: i) T is the concatenation of BeforeS, S, and AfterS, and ii) the length of BeforeS is P.
We would like to compute the first occurrence of S in T, that is, the minimum P such that S occurs in T at position P. Here is a logic program, called Firstin, which solves that problem.

```
firstin(T, S, Position) ← fi(T, [], S, 0, Position).
fi(X, Y, [], Acc, s(Position)) ← length(Y, L), plus(L, Position, Acc).
fi([C|RestofText], OldPref, OldSuff, Acc, Position) ←
        newstring(C, OldPref, OldSuff, NewPref, NewSuff),
        fi(RestofText, NewPref, NewSuff, s(Acc), Position).
newstring(C, OldPref, [C|RestOldSuff], NewPref, RestOldSuff) ← append(OldPref, [C], NewPref).
newstring(C, OldPref, [D|RestOldSuff], NewPref, NewSuff) ←
        C≠D, append(OldPref, [C], H), append(V, NewPref, H),
        append(NewPref, R, OldPref), append(R, [D|RestOldSuff], NewSuff).
length([], 0).                          plus(0, N, N).
length([H|T], s(L)) ← length(T, L).     plus(s(M), N, s(P)) ← plus(M, N, P).
append([], L, L).
append([H|T], L, [H|T1]) ← append(T, L, T1).
```

(Thus, the only base predicate in this program is ≠.)

The above program is a slight modification of the one in [Kursawe 88], which tests whether or not a string occurs in a text. In our program the top predicate firstin has the extra argument Position which tells us where the string S occurs in T.
The understanding of the behaviour of the above program is *not* necessary for the application of our transformation techniques. They are syntactically based and improve program performances without relying on any semantic property.
Suppose now that we want to partially evaluate our program with respect to the string S=[a,b]. We will derive the so-called *residual program*, which gives us the position where [a,b] occurs in the text T. This partial evaluation process will be performed by applying our techniques, which will also include some generalization steps. Indeed, we cannot apply the results of Theorem 14, which ensures that no generalization step is necessary, because the given program is not non-ascending (see, for instance, the occurrences of the terms Acc and s(Acc) in the second clause for fi).

We first introduce the new clause 1: firstabin(Text,Position) ← firstin(Text,[a,b],Position), and construct a *foldable* U-tree for <Firstin, clause 1> according to our Procedure 2. Then we apply the Loop Absorption with Fold on Root Procedure which will suggest the eureka predicates and their recursive definitions, and therefore, it will allows us to derive an improved program, as desired.

We choose the selection rule, called LeftS, which selects the leftmost atom in the body of the current clause ('leftmost' has to be considered in the textual order). We also establish suitable values for the vertical and the horizontal bounds. To this regard we notice that after a few unfolding steps starting from clause 1 we get the following two clauses:

```
firstabin([a|R], Pos) ← append([], [a], X), fi(R, X,[b], s(0), Pos).
firstabin([C|R], Pos) ← C≠a, append([], [C], X), append(Y, Z, X), append(Z, V, []),
                        append(V, [a,b], W), fi(R, Z,W, s(0), Pos).
```

where the maximal maxdepth of the defined atoms is 3, and the length of the longest variable-chained sequence of defined atoms is 5. Thus, if we do not want to perform a generalization step before then, we have to assume that the vertical and the horizontal bounds are at least 3 and 5, respectively. We assume that they are exactly 3 and 5, and we will see that those values are in fact adequate for our derivation.

We now construct a foldable U-tree for <Firstin, clause 1>. The relevant parts of the minimal upper portion T1 of that U-tree is depicted in Figure 4, where now (and in Figure 5) a solid arrow between two clauses denotes one or more unfolding or generalization steps, and the numbers stand for the following clauses:

1. firstabin(Text,Position) ← firstin(Text,[a,b],Position).
C2. firstabin([a,b|R],s(0)).

C3. firstabin([a,a,C|R],P) ← a≠b, newstring(C,[a],[b],NP,NS), fi(R,NP,NS,s(s(s(0))),P).
C4. firstabin([a,C,D|R],P) ← C≠b, newstring(D,[],[a,b],NP,NS),fi(R,NP,NS,s(s(X)),P), X=s(0).
C5. firstabin([C,a,D|R],P) ← C≠a, newstring(D,[a],[b],NP,NS), fi(R,NP,NS,s(s(X)),P), X=s(0).
C6. firstabin([C,D,E|R],P) ← C≠a, D≠a, newstring(E,[],[a,b],NP,NS),
fi(R,NP,NS,s(s(X)),P), X=s(0).
C7. firstabin([a,a,C|R],P) ← a≠b, newstring(C,[a],[b],NP,NS), fi(R,NP,NS,s(s(X)),P), X=s(0).
C8. firstabin([a,a,b|R],s(X)) ← a≠b, X=s(0).
C9. firstabin([a,a,a|R],P) ← a≠b, a≠b, fi(R,[a],[b],s(s(X)),P), X=s(0).
C10. firstabin([a,a,C|R],P) ← a≠b, C≠b, fi(R,[],[a,b],s(s(X)),P), X=s(0).
C11. firstabin([a,a,a,C|R],P) ← a≠b, a≠b, newstring(C,[a],[b],NP,NS),
fi(R,NP,NS,s(s(s(X))),P), X=s(0).
C12. firstabin([a,a,a,C|R],P) ← a≠b, a≠b, newstring(C,[a],[b],NP,NS),
fi(R,NP,NS,s(s(X)),P), X=s(Y), Y=s(0).
C13. firstabin([a,a,C,a,D|R],P) ← a≠b, C≠b, newstring(D,[a],[b],NP,NS),
fi(R,NP,NS,s(s(X)),P), X=s(Y), Y=s(Z), Z=s(0).
C14. firstabin([a,a,C,D|R],P) ← a≠b, C≠b, D≠a, fi(R,[],[a,b],s(s(X)),P), X=s(Y), Y=s(0).
C15. firstabin([a,C,a|R],P) ← C≠b, fi(R,[a],[b],s(s(X)),P), X=s(0).
C16. firstabin([a,C,D,E|R],P) ← C≠b, D≠a, newstring(E,[],[a,b],NP,NS),
fi(R,NP,NS,s(s(X)),P), X=s(Y), Y=s(0).
C17. firstabin([a,C,a,b|R],s(P)) ← C≠b, P=s(X), X=s(0).
C18. firstabin([a,C,a,a|R],P) ← C≠b, a≠b, fi(R,[a],[b],s(s(X)),P), X=s(Y), Y=s(0).
C19. firstabin([a,C,a,D,E|R],P) ← C≠b, D≠b, newstring(E,[],[a,b],NP,NS),
fi(R,NP,NS,s(s(X)),P), X=s(Y), Y=s(Z), Z=s(0).

In that figure we also use the label 'gen' to denote single generalization steps which are of interest for our explanations below. The other notations are the usual ones.

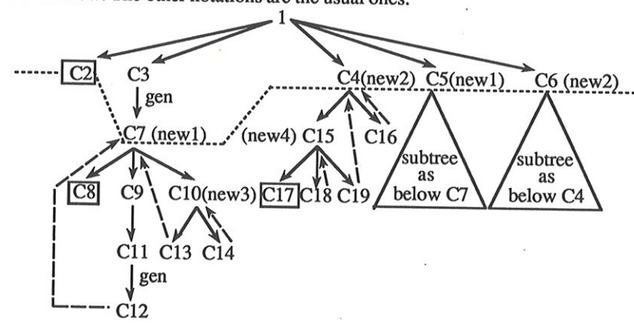


Figure 4. Minimal upper portion T1 of the foldable U-tree for <Firstin, clause 1> via LeftS.

Let us now examine a particular path of that U-tree and show that a generalization step is indeed necessary for obtaining a foldable clause.
That path is generated by unfolding clause 1 thereby obtaining clause C3, where the maxdepth of the atom fi(R,NP,NS,s(s(s(0))),P) is bigger than the vertical bound 3. Thus, we generalize the occurrence of s(0) in that atom, and we get clause C7.
By continuing the unfolding process from C7 we get clause C9, and then clause C11, where the vertical bound 3 is again exceeded. Thus, we generalize C11 and we get clause C12, which determines a loop with clause C7 as ancestor-clause.

To extract the final program version we now apply the Loop Absorption with Fold on Root

Procedure. We consider first the initial definition which is clause 1. We construct the optimal upper portion T1opt of the tree T1 depicted in Figure 4. It is the upper portion of T1 with leaves C2, C7, C4, C5, and C6.

We introduce the eureka predicates new1 and new2 corresponding to the ancestor-clauses C7 and C4:

new1(C,T,A,P) ← newstring(C,[a],[b],NP,NS), fi(T,NP,NS,s(s(A)),P).

new2(C,T,A,P) ← newstring(C,[],[a,b],NP,NS), fi(T,NP,NS,s(s(A)),P).

The current FinalProg includes the following clauses defining the predicate firstabin:

C2. firstabin([a,b|R],s(0)).
C7f. firstabin([a,a,C|R],P) ← new1(C,R,s(0),P).
C4f. firstabin([a,C,D|R],P) ← C≠b, new2(D,R,s(0),P).
C5f. firstabin([C,a,D|R],P) ← C≠a, new1(D,R,s(0),P).
C6f. firstabin([C,D,E|R],P) ← C≠a, D≠a, new2(E,R,s(0),P).

Those clauses are obtained from the leaves of T1opt as follows. C2 is unchanged. C7f and C5f are obtained by first simplifying C7 and C5, whereby getting:

C7s. firstabin([a,a,C|R],P) ← newstring(C,[a],[b],NP,NS), fi(R,NP,NS,s(s(s(0))),P)
C5s. firstabin([C,a,D|R],P) ← C≠a, newstring(D,[a],[b],NP,NS), fi(R,NP,NS,s(s(s(0))),P)

and then by folding clauses C7s and C5s by using the eureka predicates new1. Analogously for C4f and C6f by using the eureka predicate new2.

The application of the Loop Absorption with Fold on Root Procedure continues by inserting the definitions for new1 and new2 in DEFdo. We repeat for those predicates the steps we have performed above for clause 1 (that is, we find a foldable U-tree and its optimal upper portion, we introduce new eureka predicates, and we update the current FinalProg).

Notice, that the search for the foldable U-trees for new1 and new2 is now an easy task because they can be obtained from the foldable U-tree with clause 1 as root. In Figure 5 below we depicted the minimal foldable (and optimal) upper portions TD and TE of the U-trees for <Firstin, clause D1> and <Firstin, clause E1> via LeftS, respectively.

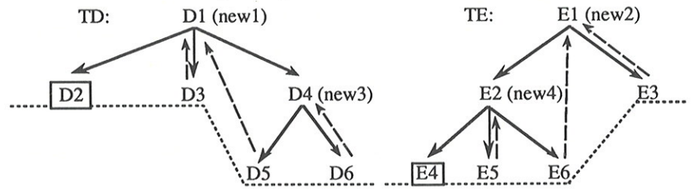


Figure 5. The minimal foldable (and optimal) upper portions TD and TE of the U-trees for <Firstin, clause D1> via LeftS and <Firstin, clause E1> via LeftS, respectively.

The clauses in Figure 5 are:

D1. new1(C,T,A,P) ← newstring(C,[a],[b],NP,NS), fi(T,NP,NS,s(s(A)),P).
D2. new1(b,T,A,s(P)).
D3. new1(a,[C|R],A,P) ← a≠b, newstring(C,[a],[b],NP,NS), fi(R,NP,NS,s(s(X)),P), X=s(A).
D4. new1(C,T,A,P) ← C≠b, fi(T,[],[a,b],s(s(A)),P).
D5. new1(C,[a,D|R],A,P) ← C≠b, newstring(D,[a],[b],NP,NS), fi(R,NP,NS,s(s(X)),P), X=s(Y),Y=s(A).
D6. new1(C,[D|R],A,P) ← C≠b, D≠a, fi(R,[],[a,b],s(s(X)),P), X=s(A).

E1. new2(C,T,A,P) ← newstring(C,[],[a,b],NP,NS), fi(T,NP,NS,s(s(A)),P).
E2. new2(a,T,A,P) ← fi(T,[a],[b],s(s(A)),P).
E3. new2(C,[D|R],A,P) ← C≠a, newstring(D,[],[a,b],NP,NS), fi(R,NP,NS,s(s(X)),P), X=s(A).

E4. new2(a,[b|R],A,s(P)) ← P=s(A).
E5. new2(a,[a|R],A,P) ← a≠b, fi(R,[a],[b],s(s(X)),P), X=s(A).
E6. new2(a,[C,D|R],A,P) ← C≠b, newstring(D,[],[a,b],NP,NS), fi(R,NP,NS,s(s(X)),P), X=s(Y), Y=s(A).

It turns out that for new1 the following new eureka predicate should be introduced:

new3(T,A,P) ← fi(T,[],[a,b],s(s(A)),P) which corresponds to clause D4.

For new2 it is necessary to introduce the eureka predicate:

new4(T,A,P) ← fi(T,[a],[b],s(s(A)),P) which corresponds to clause E2.

The current FinalProg includes the following clauses (derived after simplification and folding from the leaf-clauses of TD and TE) for new1 and new4:

new1(b,T,A,s(A)).
new1(a,[C|R],A,P) ← new1(C,R,s(A),P).
new1(C,[a,D|R],A,P) ← C≠b, new1(D,R,s(s(A)),P).
new1(C,[D|R],A,P) ← C≠b, D≠a, new3(R,s(A),P).
new2(a,[b|R],A,s(s(A))).
new2(a,[a|R],A,P) ← new4(R,s(A),P).
new2(a,[C,D|R],A,P) ← C≠b, new2(D,R,s(s(A)),P).
new2(C,[D|R],A,P) ← C≠a, new2(D,R,s(A),P).

We continue the derivation of FinalProg by finding the clauses for new3 and new4. They are:

new3([a,b|R],A,s(s(s(A)))).
new3([a,a|R],A,P) ← new4(R,s(s(A)),P).
new3([a,C|R],A,P) ← C≠b, new3(R,s(s(A)),P).
new3([C|R],A,P) ← C≠a,new3(R,s(A),P).
new4([b|R],A,s(A)).
new4([a|R],A,P) ← new4(R,s(A),P).
new4([C,a|R],A,P) ← C≠b, new4(R,s(s(A)),P).
new4([C,D|R],A,P) ← C≠b, D≠a, new3(R,s(s(A)),P).

The final program including the above clauses for firstabin, new1, new2, new3, and new4 runs some three times faster than the original version for the goal firstin(Text,[a,b],P). ■

5. FINAL CONSIDERATIONS

We have presented a program transformation methodology and we have applied it to the derivation of some logic programs. We have also shown its relevance in the area of partial evaluation. We have characterized our methodology by introducing the concept of foldable U-tree corresponding to a recursive logic program and by establishing some theorems which ensure the existence of a foldable U-tree for a given initial program.

Before ending the paper we want to say a few words on the problem of applying our transformation techniques to programs written in Prolog, where the order of the atoms and clauses is significant. Often there is not much to care about, and indeed, the reader may easily check that the program derivations we have performed in the previous sections can also be considered as Prolog derivations.

However, it may happen that program termination is not maintained and efficiency is not improved when one applies our techniques to Prolog programs. In order to avoid those difficulties, we may sometimes perform some extra transformation steps as we now indicate.

Case A. If for obtaining the pattern of atoms required for performing a folding step we have to rearrange the order of the atoms, the non-determinism of the derived program may increase, because some predicates could be evaluated before they are sufficiently instantiated. In those cases we can safely perform some transformation steps which can be formalized as applications of the so-called

Goal Inversion Rule, which will be studied in a forthcoming paper. By that rule we can interchange two atoms, if the instantiation of the variables satisfies some suitable conditions. Fortunately those conditions, which may require information on calling modes, can often be derived at compile time via standard techniques based on abstract interpretations or data flow analysis [Bruynooghe et al. 87].
Consider, for instance, a Prolog program with the following clauses (among others):

h(X,Y) :- p(X,Y), q(X). p(X,Y) :- p(X,Z), r(Z,Y).

By unfolding we get the clause D: h(X,Y) :- p(X,Z), r(Z,Y), q(X). Now, in order to perform a folding step we need to move r(Z,Y) either to the left of p(X,Z) or to the right of q(X).
Suppose that, using the information on the calling mode for h, we are able to establish that when we use the clause D one of the following properties holds:

1. p(X,Z) has at least one solution and the evaluation of r(Z,Y) is deterministic independently of the evaluation of p(X,Z);
2. after the evaluation of p(X,Z), r(Z,Y) has at least one solution and either it does not share any variable with q(X) or q(X) is deterministic.

In case 1 we can move r(Z,Y) to the left of p(X,Z). In case 2 we can move r(Z,Y) to the right of q(X). In both cases we affect neither the termination nor the efficiency of the program obtained after moving the atom r(Z,Y) w.r.t. the program before the move.

Case B. The unfolding process may increase the number of generated clauses, as it happens in our examples above, and therefore, at run time each resolution step may need more time when searching for the unifying clause-head. It also allows for the repetition of atoms in different clauses. Those phenomena may limit the improvement of efficiency determined by the use of the eureka predicates, but in those cases we can profitably apply various techniques based on clause fusion [Debray-Warren 88] or clause abstraction [Sterling-Lakhotia 88] or clause indexing (like in the LPA MacProlog optimizing compiler [Clark et al. 87]). Consider, for instance, the following three clauses:

h(X) :- p(X), q(X). q(X) :- r(X). q(X) :- s(X).

By unfolding q(X) we get: h(X) :- p(X), r(X). and h(X) :- p(X), s(X). If during computation r(X) fails, p(X) is evaluated twice. This drawback can be avoided by using clause fusion and obtaining: h(X) :- p(X), (r(X); s(X)), where ';' is the disjunction operator. Obviously, clause fusion is indeed a gain only if ';' is efficiently implemented, and it is not interpreted, as in C-Prolog, like a shorthand for avoiding to write two clauses.

Case C. Once we have derived the final Prolog program by performing unfolding/folding steps, we should make sure that termination is preserved. If it is not preserved, we may modify the order of the clauses of the final program. However, the fact that the transformation rule are correctness preserving ensures only that there exists a 'dynamic rearrangement' (that is, a clause ordering which depends on the computation step) such that if the initial program terminates, so does the final program.
Let us consider the naive rule of keeping at each unfolding step the derived clauses in the same order in which those used for unfolding occur in the given program. It is *not* a safe rule, and we may get a nonterminating Prolog program from a terminating one. Let us consider, for instance, the initial Prolog program made out of the following clause (in this textual order):

h(X,Y) :- p(X), q(Y). p(a). q(b) :- q(b). q(a).

By unfolding q(X) in the first clause we get: h(X,b) :- p(X), q(b) and h(X,a) :- p(X). After a folding step we get the final Prolog program:

h(X,b) :- h(X,b). h(X,a) :- p(X). p(a).

It gets into an endless loop when evaluating the goal h(X,X), while the initial one does not.

6. ACKNOWLEDGEMENTS
We would like to thank the IASI Institute of the National Research Council of Italy and the University of Roma 'Tor Vergata' for providing the necessary facilities and giving the financial support. Many thanks also to the colleagues of the 'Progetto Finalizzato Informatica' who helped us with their stimulating conversations.

7. APPENDIX
The transformation rules which we use for program derivation are the following ones:

- *Definition Rule*. It consists in adding a new clause to the current program version. That clause is considered to be a *definition*, with a new head predicate defined in terms of already existing predicates. Notice that no recursive definitions are allowed, and each new predicate occurs as head of one clause only.
- *Unfolding Rule*. It consists in performing a computation step by applying the SLD-resolution to a chosen clause with respect to a selected atom of its body. The chosen clause is replaced in the current program by *all* those which can be obtained by resolving it with any clause (of the program at hand) whose head is unifiable with the selected atom.
- *Folding Rule*. It consists in replacing an *old clause*: $H \leftarrow A_1,\ldots,A_n,A_{n+1},\ldots,A_r$ by a *new clause*: $H \leftarrow K\sigma,A_{n+1},\ldots,A_r$, using a *bridge clause*: $K \leftarrow B_1,\ldots,B_n$, where σ is a substitution such that: $A_i = B_i\,\sigma$ for all i=1,...,n.
  This rule can be applied only if: (a) unfolding the new clause with respect to the atom $K\sigma$ we obtain again the old clause, and (b) the bridge clause is a definition introduced by a previous application of rule i) and it is different from the old clause.

8. REFERENCES

[Bossi et al. 88] Bossi, A., Cocco, N., and Dulli S.:"A Method for Specializing Logic Programs", Proc. Third Italian Conference on Logic Programming, GULP 88, (Nardi, D., ed.), Roma, 1988, pp. 97-114, (to be published in ACM TOPLAS).
Bruynooghe et al. 87] Bruynooghe, M., Janssens G., Callebaut, A., and Demoen, B.: "Abstract Interpretation: toward the Global Optimization of Prolog Programs", Proc. Symposium on Logic Programming , IEEE Press, 1987, pp. 192-204.
[Burstall-Darlington 77] Burstall, R.M. and Darlington, J.: "A Transformation System for Developing Recursive Programs", JACM, Vol. 24, No. 1, January 1977, pp. 44-67.
[Clark et al. 87] Clark, K., McCabe, F., Johns, N., and Spenser, C.: "LPA MacProlog Reference Manual" Logic Programming Associates, London, 1987.
[Debray-Warren 88] Debray, S.K., and Warren, D.S.: "Automatic Mode Inference for Logic Programs", J. Logic Programming 1988, Vol. 5, pp. 207-229.
[Feather 86] Feather, M.S.: "A Survey and Classification of Some Program Transformation Techniques", TC2 IFIP Working Conference on Program Specification and Transformation, Bad Tölz, Germany, 1986.
[Hogger 81] Hogger, C.J.: "Derivation of Logic Programs", JACM, No. 28, 2, 1981, pp. 372-392.
[Komorowski 82] Komorowski, H.J.: "Partial the Evaluation as Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog", 9th ACM Symp. on Principles of Prog. Lang., Albuquerque, New Mexico, 1982, pp. 255-267.
[Kursawe 88] Kursawe, P.: "Pure Partial Evaluation and Instantiation", in: Partial Evaluation and Mixed Computation, (Bjørner, D., Ershov, A.P., and Jones, N. D., eds.), North Holland, 1988.
[Lloyd 87] Lloyd, J.W.: "Foundations of Logic Programming", Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 2nd edition, 1987.
[Nakagawa 85] Nakagawa, H.: "Prolog Program Transformations and Tree Manipulation Algorithms", J. Logic Programming 1985, 2, pp. 77-91.
[Pettorossi-Proietti 89] Pettorossi, A. and Proietti, M.: "Decidability Results and Characterization of Strategies for the Development of Logic Programs", Proc. 6th Intern. Conf. on Logic Programming, (G. Levi and M. Martelli, eds.), Lisboa (Portugal) 1989, pp. 539-553.
[Proietti-Pettorossi 90] Proietti, M. and Pettorossi, A.: "Synthesis of Eureka Predicates for Developing Logic Programs", to appear in Proc. ESOP 90, Copenhagen, 1990.
[Sterling-Lakhotia 88] Sterling, L. and Lakhotia, A.: "Composing Prolog Meta-Interpreters", Proc. 5th Intern. Conference on Logic Programming, Seattle, WA (USA), 1988.
[Tamaki-Sato 84] Tamaki, H. and Sato, T.: "Unfold/Fold Transformation of Logic Programs", Proc. 2nd International Conference on Logic Programming, Uppsala, 1984.

Fold-Unfold Transformations for Structured Logic Programming

M. Bugliesi E. Lamma P. Mello

ENIDATA S.p.A
Viale Aldo Moro 38,
40121 Bologna

DEIS Università di Bologna
Viale Risorgimento 2,
40136 Bologna

Abstract

In this paper we address the application of optimization techniques based on Partial Evaluation in the framework of logic programs structured as hierachies of separate theories. We define a general well-founded scheme for partial evaluation to be applied to two different classes of logic programming systems, i.e. block-structured and inheritance-based systems. We show how the properties of soundness and completeness of Partial Evaluation in Logic Programming are preserved for the defined transformation scheme.

1 Introduction

Structuring mechanisms have become nowadays a crucial topic in Logic Programming. Several authors have addressed this problem in the literature, on the basis of different approaches: modules and blocks in [GMR88,MPr85], inheritance in [FH86,KG86,Gal86] and viewpoints to support hypothetical reasoning in [GR84,McC88].

In [MNR89] and [BLM89] a general structuring framework is presented, based on an extension of the Contextual Logic Programming paradigm introduced in [MP89]. The underlying idea is that programs can be organized as sets of elementary components, where each component is a set of clauses logically connected in a module (*unit*). The overall meaning of predicate calls depends on the *Context* of units in which they are evaluated. Depending on the different policies used to combine units into context several structuring mechanisms can be devised within this unifying scheme.

Some efforts, [LMN89], have been also devoted to the design of an efficient implementation for contexts and the various mechanisms defined to manipulate them. The approach is based on the definition of an extended Warren Abstract Machine where new instructions and data structures are introduced to deal with units and contexts. An extended compilation scheme is also defined to translate structured programs into sequences of WAM instructions.

In this paper we focus on Partial Evaluation [Kom81], and we show how the compilative approach proposed in [LMN89] can take advantage of this source-to-source program transformation technique.

We introduce a new definition of Partial Evaluation which extends that given in [LS87] to capture the notions of modules and module-composition. Based on this notion, we develop a transformation scheme which is general enough to cope with the different classes of structured logic programming systems defined in [BLM89]. Given a structured program P and a goal G, the transformation produces a new program P' whose units are derived from those in P by specialization with respect to the goal G.

The original structure of the program is preserved and no collapsation of several modules into a more compact configuration occurs. This ensure a high flexibility and generality for the new program. Furthermore it keeps the transformation fully compatible with the compilation technique addressed in [LMN89]

The paper is structured as follows. In section 2, the different classes of structured logic programming that we will considered are described.

In section 3, the theory of partial evaluation for structured logic programming is informally presented together with some simple examples. In section 4, the soundness and completeness conditions for the transformation are stated, and the applicability of the partial evaluation to different structured logic programming system is briefly discussed.

2. Structured Logic Programs

The framework for structuring logic programs we consider is the one presented in [MNR89, BLM89]. A detailed description of the topic is beyond the scope of this paper. For deeper discussion on the practical and semantic issues involved refer to [LMN89] and [BLM89] respectively.

The structuring approach is based on the Contextual Logic Programming paradigm — an extension to logic programming with the concepts of units, context-dependent predicate definitions and variable context of proof [MP89] — . The first concept to be introduced is that of unit. A unit consists of a set of clauses and a unique name to refer it. Units can be composed in contexts, thus allowing for a dynamic configuration of the set of clauses used during the computation. Thus, a computation corresponds to the proof of goals in variable contexts. More in detail, contexts can be represented as lists of units $\lfloor u_N, \ldots, u_i, \ldots, u_1 \rfloor$, that record the history of the formation of the context starting from the empty one (the empty list). Contexts are dynamically built by using the context-extension operator ($\gg$). In particular, given a unit name u_{N+1}, and a goal formula G, the execution of the extension goal $u_{N+1} \gg G$ in the context $C = \lfloor u_N, \ldots, u_i, \ldots, u_1 \rfloor$, causes the proof of G to be executed in a new context $C_1 = \lfloor u_{N+1}, u_N, \ldots, u_i, \ldots, u_1 \rfloor$, obtained by pushing u_{N+1} on top of the previous context C. Let us consider the following

Example 2.1 *Let P be the following structured program:*

$\boldsymbol{unit}(u_1)$:	$\boldsymbol{unit}(u_2)$:	$\boldsymbol{unit}(u_3)$:
$b(1)$.	$a(X) : -b(X)$.	$c(X) : -a(X)$.

The top goal $u_1 \gg u_2 \gg u_3 \gg c(X)$ is proved, with answer substitution $\{X \leftarrow 1\}$. In particular, $c(X)$ is proved in the context $\lfloor u_3, u_2, u_1 \rfloor$, i.e. by "virtually" using the dynamically composed set of clauses:

$c(X) : -a(X)$.
$a(X) : -b(X)$.
$b(1)$.

The computation can be traced in terms of a top-down derivation - i.e. a relation $Ctx \vdash F$ stating that F is derivable in Ctx [MP89] as follows:

$$
\begin{array}{ll}
\lfloor\rfloor & \vdash u_1 \gg u_2 \gg u_3 \gg c(X) \\
\lfloor u_1 \rfloor & \vdash u_2 \gg u_3 \gg c(X) \\
\lfloor u_2, u_1 \rfloor & \vdash u_3 \gg c(X) \\
\lfloor u_3, u_2, u_1 \rfloor & \vdash c(X) \\
\lfloor u_3, u_2, u_1 \rfloor & \vdash a(X) \\
\lfloor u_2, u_1 \rfloor & \vdash b(X) \\
success & : \{X \leftarrow 1\}
\end{array}
$$

Starting from these basic ideas, different design choices and extensions can be introduced. These choices and extensions represent the "guide-line" for classifying and designing different kinds of structured logic programs for which the application of PE will be discussed in section 3.

2.1 Conservative/Evolving Systems

Given a context $C = \lfloor u_N, \ldots, u_i, \ldots, u_1 \rfloor$we introduce the notions of evolving and conservative systems on the account of the context chosen as the environment for solving each predicate call. An evolving system is a logic programming system composed of a set of units where, for each goal (predicate call) g occurring in a unit u_i, the corresponding predicate definition is determined by the whole context C. We will refer to C as the lazy context (LC for short). A conservative system is a logic programming system composed of a set of units where, for each goal (predicate call) g occurring in a unit ui, the corresponding predicate definition is determined only by the subcontext $\lfloor u_i, \ldots, u_1 \rfloor$of the context C. We will refer to this subcontext of C as the eager context (EC for short). In practice, for conservative systems, the answers for predicate calls occurring in a unit U can no longer be modified by new context extensions nested within the extension involving U. By converse, for evolving systems the answers for predicate calls occurring in U can be modified by new context extensions nested within the extension involving U.

Example 2.2 *Let us consider the units defined in 2.1 and the top goal: $u_2 \gg u_1 \gg u_3 \gg c(X)$. An evolving system has the following top-down derivation (see appendix):*

$$
\begin{array}{ll}
\lfloor\rfloor & \vdash u_2 \gg u_1 \gg u_3 \gg c(X) \\
\lfloor u_2 \rfloor & \vdash u_1 \gg u_3 \gg c(X) \\
\lfloor u_1, u_2 \rfloor & \vdash u_3 \gg c(X) \\
\lfloor u_3, u_1, u_2 \rfloor & \vdash c(X) \\
\lfloor u_3, u_1, u_2 \rfloor & \vdash a(X) \\
\lfloor u_3, u_1, u_2 \rfloor & \vdash b(X) \\
success & : \{X \leftarrow 1\}
\end{array}
$$

Here the subgoal of $a(X)$ will be proved in the whole context $\lfloor u_3, u_1, u_2 \rfloor$ (the lazy one). By converse, in a conservative system, for the subgoal of $a(X)$ only the eager context will be considered. Since $\lfloor u_2 \rfloor$ contains no definition for $b(X)$, in this case we get a failure as shown below:

$$
\begin{array}{ll}
\lfloor\ \rfloor & \vdash\ u_2 \gg u_1 \gg u_3 \gg c(X) \\
\lfloor u_2 \rfloor & \vdash\ u_1 \gg u_3 \gg c(X) \\
\lfloor u_1, u_2 \rfloor & \vdash\ u_3 \gg c(X) \\
\lfloor u_3, u_1, u_2 \rfloor & \vdash\ c(X) \\
\lfloor u_1, u_2 \rfloor & \vdash\ a(X) \\
\lfloor u_2 \rfloor & \vdash\ b(X) \\
failure &
\end{array}
$$

The categorization into conservative and evolving systems is not merely an academic issue. Conservative systems are more statical, efficient and safe, but they define a too strict hierarchy of components and cannot support a programming methodology which includes the dynamic building of complex systems [MNR89]. Most proposals for structuring logic programs can be classified in terms of conservative/evolving systems. In particular, proposals such as blocks and modules [GMR88], Meta- Prolog [Bac88], basic Contextual Logic Programming [MP89] are examples of conservative systems, while proposals such as the Miller's one [Mil86], Multi-Prolog [CLM88], N-Prolog [GR84], can be classified as evolving systems.

2.2 Statically/Dynamically Configured Systems

We define as *statically configured* system a logic programming system composed of a set of units where hierarchies among them (i.e. fixed contexts called closures in the following) are established when units are defined. Such hierarchies are automatically forced each time units are used, i.e. are involved in a context extension. In other words, a statically configured system defines, for each unit, a *statically configured environment* whose behaviour does not depend on the dynamic computational environment. We define as *dynamically configured* system a logic programming system composed of a set of units where hierarchies (i.e. contexts) are dynamically established each time units are used, i.e. are involved in a context extension. Note that all the examples previously reported implicitly refer to dynamically configured systems.

Example 2.3 *Let P be the following evolving, statically configured system*

$$
\begin{array}{ll}
\boldsymbol{unit}(m_1, \boldsymbol{closed}(\lfloor\ \rfloor)): & \boldsymbol{unit}(m_2, \boldsymbol{closed}(\lfloor m_1 \rfloor)): \\
p. :- q. & q.
\end{array}
$$

The list of units specified as parameter of the **closed** *structure represents the closure of the closed unit. The top goal:* $m_2 \gg p$ *produces the following top-down derivation:*

$$
\begin{array}{ll}
\lfloor\ \rfloor & \vdash\ m_2 \gg p. \\
\lfloor m_2, m_1 \rfloor & \vdash\ p. \\
\lfloor m_2, m_1 \rfloor & \vdash\ q. \\
success &
\end{array}
$$

In the case of a dynamically configured system the same top goal fails since no definition for the predicate $p/1$ exists in $\lfloor m_2 \rfloor$

A few remarks on statically configured systems:

- They are *more safe* than dynamically configured systems since their behaviour can be statically determined.

- They are more efficient than dynamically configured systems ([LMN89]), since predicate calls can be directly bound to the corresponding predicate definitions when the unit in which they appear is defined. (Notice that this is not possible in dynamically configured systems);
- They are too weak to deal with problems demanding high dynamicity such as artificial intelligence ones.

2.3 Classifying Different Structuring Mechanisms

It is possible to show that the above classification is general and flexible enough to subsume some of the most useful ways of structuring logic programs introduced in the literature to face software engineering and artificial intelligence problems (see [MNR89,Mel90]). In particular:

1. Systems for hypothetical reasoning and viewpoints can be classified as a particular form of dynamically configured evolving system where hypotheses or viewpoints are collected in units. These systems have to be evolving since old knowledge has to be always updated by newly added knowledge. Similarly, they have to be dynamically configured, so that new hypotheses or viewpoints can be added to the dynamic state of the computation.
2. Systems with blocks and modules can be classified as statically configured conservative systems. The nested structure of blocks is statically represented by the use of closures. When proving a goal occurring in a particular block B, only predicate definitions within B and its external blocks have to be taken into account. This kind of behaviour is different from the one of hypothetical reasoning systems, where a context extension determines the addition of new clauses to the previous context without any prefixed order so that they become indistinguishable.
3. Inheritance- and object-based systems can be classified as statically configured evolving systems. By analogy with inheritance-based systems [WZ88], we can interpret contexts as the explicit representation of a branch in an inheritance tree (for the sake of simplicity we will not consider multiple inheritance). The first unit in the context list is the tip node, while the last is the root. Inheritance-based systems have to be evolving and not conservative since, as stated in [WZ88], in these systems self-reference in a type or class is bound to the object on whose behalf an operation/demonstration is being executed, rather than on the textual module/unit in which the self- reference occurs. Moreover, an inheritance-based system has to be statically configured and not dynamically configured, since traditional inheritance is conceived as a property of the specific object (unit) independently from the dynamic evolution of the computation.

3 Extending Partial Evaluation

Partial Evaluation (PE) has been devoted great attention in the last few years in the logic programming field since Komorowski [Kom81] introduced it in 1981. PE can be conceived as a source-to-source transformation which, given a program P and a goal G, produces a new program P', which is more efficient than P and has the same answer substitutions for

the goal G or its instances. The basic technique for obtaining P' from P is to construct a partial search tree for P and suitably chosen atoms as goals, and then extract the definitions — the *resultants* — associated with the branches of the tree. P' is then obtained from P by replacing the set of clauses in P whose head contains one of the predicate symbols appearing in G with the set of the produced resultants.

As pointed out in [LS87], the main foundational questions about PE concern soundness and completeness. Soundness of the partially evaluated program P' wrt the original program P and the goal G means that each correct (computed) answer for G and P' is a correct (computed) answer for G and P. Completeness is the converse of this.

In standard logic programming, soundness follows from the soundness of SLD-resolution. A further closedness condition must be satisfied by the resulting program in order to ensure completeness [LS87].

The application of these issues to the structuring framework presented in section 2 involves several major extensions to the basic principles of Partial Evaluation.

First of all, we have to consider that the evaluation of the goal occurs in a context and, furthermore, we have to take into account the modular configuration of the program. The standard principles of PE are to be extended to include the notion of context.

Given a program P , a goal G and a context C, our main goal will be the definition of a new program P' which is more efficient than P wrt G in the context C. Furthermore, in order to ensure a high level of flexibility, we will impose that the transformation preserves the original structure of P.

This rises further problems with respect to the case of logic programming. Given the set of resultants corresponding to a partial search tree one has first to determine which unit each resultant should be assigned to. The derived program is the obtained from the original one by replacing (possibly) all the units in the initial context with their specialized versions. These, in turn are obtained by erasing the old definitions for the initial goal and by introducing the new ones according to a suitable *assignment* function.

In the following, we will refer to the partial evaluation of P wrt a goal G and a context C as $PE(P, G, C)$. A formal definition of $PE(P, G, C)$ can be given in terms of the following notion of program representation:

Definition 3.1 (Representation of a set of units) *Let $U(P)$ be the set of units of a program P. We denote by $|u|$ the set of clauses of a unit u. Then the representation of P is given by:*

$$\mathcal{R}(P) = \{\langle d, u\rangle : \ u \in U(P), d \in |u|\}$$

Definition 3.2 (Partial Evaluation of a goal wrt a context) *Let P be a program, C a context, G an atomic goal, and R_T the set of resultants of a search-tree T for P and $\langle G, C\rangle$. Let also $\mathcal{F}_{ass} : \ R_T \mapsto U(P)$ be an assignment function, which assigns resultants to units of P. The partial evaluation of G wrt C is given by:*

$$PE(G, C) = \{\langle d, u\rangle : \ d \in R_T \text{ and } u = \mathcal{F}_{ass}(d)\}$$

Definition 3.3 (Partial Evaluation of P wrt C, G) *Let P be a program, C a context, G an atomic goal. Then:*

$$PE(P, G, C) = P' \ \text{ such that : } \ \mathcal{R}(P') = (\mathcal{R}(P) \setminus \mathcal{D}(C, G)) \cup PE(G, C)$$

where $\mathcal{D}(C, G)$ is the representation of the set of definitions for the goal G in the context C.

Different choices for the assignment function determine different configurations for the transformed program and, correspondingly, different properties for the transformation. Therefore, the choice of the assignment function will be deeply influenced by the properties we expect from the partial evaluation process.

As a matter of fact, we are interested in preserving the equivalence between P and P' with respect to instances of G (as in [LS87]) and *subcontexts* of C, i.e. with respect to all those computations in which (instances of) G are evaluated in sub-contexts of C.

This is actually the only reasonable choice. In fact, any extension to the initial context involved into the unfolding process wouldn't affect, in the transformed program, the behaviour of those calls which have been statically solved. Therefore, any possible extra answer substitution due to the new added definitions wouldn't be returned in the transformed program, whereas that would in the original one.

However, even in subcontexts of the initial one, ensuring the safeness of the transformation is not straightforward. Let's consider the following definition of the assignment function. Let *cl* be a clause in a unit u and let S be the set of resultants which are produced by unfolding. Let's then suppose we assign each resultant in S to u itself. Accordingly, the $PE(P, G, C)$ is the program obtained from P by replacing each clause of the units of C whose head unifies with G by the corresponding set of resultants.

If this naive assignment function is assumed, the transformation is not sound for subcontexts of C. Let us consider the following

Example 3.1 *Let P be an evolving, statically configured program:*

$$\begin{array}{ll} \textbf{\textit{unit}}(m_1, \textbf{\textit{closed}}(\lfloor\ \rfloor)) : & \textbf{\textit{unit}}(m_2, \textbf{\textit{closed}}(\lfloor m_1 \rfloor)) : \\ p :- q. & q. \end{array}$$

Let P' be obtained by partially evaluating P wrt the goal $m_2 \gg p$ which corresponds to the pair $\langle p, \lfloor m_2, m_1 \rfloor\rangle$:

$$\begin{array}{ll} \textbf{\textit{unit}}(m_1, \textbf{\textit{closed}}(\lfloor\ \rfloor)) : & \textbf{\textit{unit}}(m_2, \textbf{\textit{closed}}(\lfloor m_1 \rfloor)) : \\ p. & q. \end{array}$$

From the operational point of view, we get a successful top-down derivation for the goal $u_2 \gg p$ in both P and P'. On the contrary, the goal $m_1 \gg p$ fails in P but succeeds in P'. Therefore we lose soundness wrt the context $\lfloor m_1 \rfloor$ which is a subcontext of the original one $\lfloor m_2, m_1 \rfloor$.

3.1 Configuring the Transformed Program

In order to avoid this kind of behaviour, a different assignment function has to be chosen. Given an initial context C and a goal G, the idea is to first identify, for each resultant R_i, the *minimal* sub-context of C, C', needed to derive R_i and then to assign R_i to the top unit of C'.

A formal definition of an assignment function satisfying the above requirements needs introducing some preliminary notation and definition. For the sake of simplicity, we first consider a restricted class of structured programs in which no dynamic context extension occurs during the evaluation of a query. Let:

- $C' \sqsubseteq C$ denote a partial order over contexts such that $C' \sqsubseteq C$ if C' is an initial sublist of C.

- $\langle G, C\rangle$ denote a *c-atom*, where G is an atomic goal and C a context,
- a *c-goal* be a conjunction of c-atoms
- $\langle(g_1, \ldots, g_n), C\rangle$ a shorthand for $\langle g_1, C\rangle, \ldots, \langle g_n, C\rangle$.

Definition 3.4 (Matching Context Set for $\langle G, C\rangle$) *Let G be an atomic goal. Then the matching context set for $\langle G, C\rangle$ is given by:*

$$\mathcal{M}cs(G, C) = \{C' : C' \sqsubseteq C \bigwedge |top(C')| \text{ contains a definition for } G\}$$

In other words, each context in $\mathcal{M}cs(G, C)$ is an initial sublist of C which contains some definition for G.

Definition 3.5 (Derivability) *Let P be a program, CG be the c-goal*

$$\langle g_1, c_1\rangle, \ldots, \langle g_i, c_i\rangle, \ldots, \langle g_n, c_n\rangle$$

and let $\langle g_i, c_i\rangle$ the selected c-atom. Then we say that the new c-goal CG' is derived from CG and P via the substitution σ, the clause Cl and the context L_C, if the following conditions hold:

$$\begin{array}{ll} CG' & = [\langle g_1, c_1\rangle, .., \langle g_{i-1}, c_{i-1}\rangle, \langle(b_1, \ldots, b_m), C'\rangle, \langle g_{i+1}, c_{i+1}\rangle, \ldots, \langle g_n, c_n\rangle]\sigma \\ \sigma & = mgu(h, g_i) \\ Cl & = h : -b_1, \ldots, b_m \text{ is the selected clause} \\ L_C & \in \mathcal{M}cs(G, c_i) \text{ is the matching context corresponding to } Cl \end{array}$$

$$C' = \begin{cases} L_C & \text{if the system is conservative} \\ c_i & \text{if the system is evolving} \end{cases}$$

We will hereafter denote a derivation step by $CG \vdash_{L_C} CG'$ where, for the sake of semplicity, we omit mentioning the substitution.

Definition 3.6 (C-SLD Derivation) *Let P a program and CG_0 a c-goal. A C-SLD derivation of CG_0 from P consists of a (finite or infinite) sequence of derivation steps:*

$$CG_0 \vdash_{c_1} \ldots \vdash_{c_j} CG_j \ldots$$

Definition 3.7 (c-resultant for $\langle G, C\rangle$) *Let $\langle G_0, C_0\rangle$ denote a c-atom. Given a C-SLD derivation:*

$$\langle G_0, C_0\rangle \vdash_{c1} \ldots \vdash_{cj} \langle g_1, c_k\rangle, \ldots, \langle g_i, c_i\rangle, \ldots, \langle g_n, c_n\rangle$$

then the c-resultant corresponding to the resultant $G_0 : -g_1, \ldots, g_n$, is given by:

$$G_0 : -\langle g_1, c_1\rangle, \ldots, \langle g_i, c_i\rangle, \ldots, \langle g_n, c_n\rangle.$$

Definition 3.8 (Deduction context: $\mathcal{D}c(R)$) *Given a C-SLD derivation:*

$$\langle G_0, C_0\rangle \vdash_{c_1} \ldots \vdash_{c_j} \langle G_j, C_j\rangle$$

we define the deduction context $\mathcal{D}c(R)$ for the corresponding c-resultant $R = G_0 : -\langle G_j, C_j\rangle$ as the maximum element of the set $\{c_j, \ldots, c_1\}$ which is also less than (under $\sqsubseteq$) C_0.

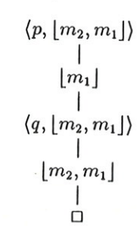
Determining the deduction context corresponds to determining the part of C_0 which has been effectively used to derive R.

Let's then note that in the source program P, $\mathcal{D}c(R)$ is the minimal sub-context of C_0 needed to derive R. I other words, R cannot be derived in any context $C' \sqsubset \mathcal{D}c(R)$ of P. This property must therefore be preserved by the transformation. This is achieved by the following definition of the assignment function.

Definition 3.9 (Assignment function: $\mathcal{F}_{ass}$) *Let T be a finite C-SLD tree. Then the assignment function is defined as follows: for each c-resultant R in T let $\mathcal{D}c(R) = \lfloor u_k, \ldots, u_1\rfloor$, then*

$$\mathcal{F}_{ass}(R) = u_k.$$

Example 3.2 *With reference to example 3.1, we get the following (finite) C-SLD tree, where each branch of the tree is labelled by the context used in the derivation step.*



$R = p$ is the only c-resultant, whose deduction context is $\mathcal{D}c(R) = \lfloor m_2, m_1\rfloor$ and therefore the new clause produced by $PE(P, p, \lfloor m_2, m_1\rfloor)$ is assigned to m_2 and the previous definition for p is dropped from m_1. Accordingly the following program P' would be produced in example 3.1

$$\begin{array}{ll} \textbf{\textit{unit}}(m_1, \textbf{\textit{closed}}(\lfloor\ \rfloor)) : & \textbf{\textit{unit}}(m_2, \textbf{\textit{closed}}(\lfloor m_2\rfloor)) : \\ & p. \\ & q. \end{array}$$

since the whole context $\lfloor m_2, m_1\rfloor$ is needed for deriving the resultant p.

From the definition of the assignment function immediately follows that only units belonging to C_0 are modified by the PE process according to the requirement stated in the previous section.

3.2 Determining the Context for the Residuals

The choice of a well-defined assignment function only partially answers the questions about the soundness and completeness of the transformation. Once the new configuration of the program has been established, there is still the problem of the residuals in the bodies of the resultants created by the unfolding process. Namely, for each resultant of the form $h : -b_1, \ldots, b_n$, it must be ensured that each of the b_is be evaluated in *equivalent* contexts before and after the transformation. As a matter of fact, two contexts C and C' are equivalent if for any goal, the same answer substitutions can be obtained deriving the goal in C and C'.

Example 3.3 *In an conservative dynamically configured system, let P be a program given by the following units:*

$\boldsymbol{unit}(u_1)$:	$\boldsymbol{unit}(u_2)$:	$\boldsymbol{unit}(u_3)$:
$q : -r.$	$r.$	$p : -q.$

Let us consider a $PE(P, p, \lfloor u_3, u_2, u_1 \rfloor)$. *A finite C-SLD tree for* $\langle p, \lfloor u_3, u_2, u_1 \rfloor \rangle$ *is:*

$$\begin{array}{c} \langle p, \lfloor u_3, u_2, u_1 \rfloor \rangle \\ | \\ \lfloor u_3, u_2, u_1 \rfloor \\ | \\ \langle q, \lfloor u_3, u_2, u_1 \rfloor \rangle \\ | \\ \lfloor u_1 \rfloor \\ | \\ \langle r, \lfloor u_1 \rfloor \rangle \end{array}$$

Then the resulting program P' is given by:

$\boldsymbol{unit}(u_1)$:	$\boldsymbol{unit}(u_2)$:	$\boldsymbol{unit}(u_3)$:
$q : -r.$	$r.$	$p : -r.$

The transformation is clearly unsound, since r is evaluated in $\lfloor u_3, u_2, u_1 \rfloor$ *in P' and in* $\lfloor u_1 \rfloor$ *in P. The goal* $u_1 \gg u_2 \gg u_3 \gg p$ *fails in P, while it succeeds in P' since* $\lfloor u_3, u_2, u_1 \rfloor$ *and* $\lfloor u_1 \rfloor$ *are not equivalent.*

Determining the correct context to be associated to each atom in the body of the resultants is straightforward in our scheme and it can be achieved by the use of a suitable application of the folding rule [TS84].

Definition 3.10 (Residuals) *Let* $R = G : -\langle g_i, C_i \rangle$ *be a c-resultant for the c-goal* $\langle G_0, C_0 \rangle$ *. Let i.e.* $C = \mathcal{D}c(R)$. *Then, depending on the relationship between C and* C_i, *the following rules are applied:*

(i) $C_i = C$. *Then*

$$R = G : -g_i \quad \overset{\mathcal{F}_{ass}}{\mapsto} \quad top(\mathcal{D}c(R))$$

(ii) $C_i \sqsubset C$, *i.e.* $C = \lfloor u_N, \ldots, u_j, \ldots, u_1 \rfloor$ *and* $C_i = \lfloor u_j, \ldots, u_1 \rfloor$. *Then:*

$$R = G : -\langle g_i, C_i \rangle \overset{fold}{\Longrightarrow} \left\langle \begin{array}{lll} G : -new & \overset{\mathcal{F}_{ass}}{\mapsto} & top(\mathcal{D}c(R)) \\ new : -g_i & \overset{\mathcal{F}_{ass}}{\mapsto} & u_j = top(C_j) \end{array} \right.$$

The general case of a resultant whose body contains more than one atom can be simply obtained by the previous cases.

Example 3.4 *In example 3.3 condition* **(ii)** *occurred. Accordingly the transformation produces the following set of units*

$\boldsymbol{unit}(u_1)$:	$\boldsymbol{unit}(u_2)$:	$\boldsymbol{unit}(u_3)$:
$q : -r.$	$r.$	$p : -new.$
$new : -r.$		

4 Soundness and Completeness

As it happens for Horn Clause Logic (HCL) [LS87], the transformation scheme described here does not automatically guarantee the completeness and soundness of the transformation. We will show that some *closedness* conditions can be stated, independently of the system category involved, to guarantee soundness and completeness. These *closedness* conditions can be considered an extension of the one given for HCL in [LS87], in order to deal with the notion of context.

More formally, let P be a program, $\langle G, C \rangle$ a c-atom and P' a partial evaluation of P wrt $\langle G, C \rangle$.

Definition 4.1 (Sound- and complete-closedness) *Let P be a program,* $G = g(T)$ *a goal and C a context. We say that:*

- *P is sound-closed wrt* $\langle G, C \rangle$ *iff each occurrence of g is evaluated in a context* $C' \sqsubseteq C$ *or in a context that has no unit in common with C.*
- *P is complete-closed wrt* $\langle G, C \rangle$ *iff each occurrence of g is an instance of* $g(T)$ *and is evaluated in a context* $C' \sqsubseteq C$ *or in a context that has no unit in common with C.*

According to the notion of sound- and complete-closedness, we now introduce the concept of soundness and completeness of PE.

Definition 4.2 (Soundness) *If* $\langle G, C \rangle$ *has a C-SLD refutation with computed answer* θ *in P', then* $\langle G, C \rangle$ *has a C-SLD refutation with computed answer* θ *in P.*

Definition 4.3 (Completeness) *If* $\langle G, C \rangle$ *has a C-SLD refutation with computed answer* θ *in P, then* $\langle G, C \rangle$ *has a C-SLD refutation with computed answer* θ *in P'.*

Proposition 1 (Soundness and completeness of the transformation) *Given a program P, a goal* G_0 *and a context* C_0, *the PE of P wrt* G_0 *in* C_0 *is sound and complete if the transformed program P' is sound-closed and complete-closed wrt* $\langle G_0, C_0 \rangle$.

It is worth noticing that the conditions given here for soundness and completeness are sufficient (and not necessary) conditions that hold for each class of structured system. Less restrictive conditions can be imposed in some particular cases. For example, to ensure soundness the sound-closedness condition can be restated as follows:

Proposition 2 (Soundness Revised) *A program P is sound-closed wrt* $\langle G, C \rangle$ *if each occurrence of g is evaluated in a context C' which does not share units with C or is of the form* $\lfloor \ldots | C'' | \ldots \rfloor$ *where* $C'' \sqsubseteq C$.

Similarly, in conservative systems, the complete-closedness condition can be relaxed as follows:

Proposition 3 (Completeness in Conservative Systems) *A program P is complete-closed wrt* $\langle G, C \rangle$ *iff each occurrence of g is an instance of* $g(T)$ *and is evaluated in a context C' which does not share units with C or is of the form* $\lfloor \ldots | C'' \rfloor$ *where* $C'' \sqsubseteq C$.

Proof*(sketched). In conservative systems, in fact, adding new units on top of a context, does not affect the old predicate definitions.*

From these considerations, some interesting results for closed systems can be derived. In fact, in these systems the sound- closedness condition introduced in proposition 2 is always automatically guaranteed thanks to the semantics of the context extension (see appendix), while the complete-closedness condition holds for conservative closed systems only. For this reason we can state that for closed conservative systems, i.e. for system representing static nesting of blocks, soundness and completeness of the transformation hold as in HCL [LS87], and they can be statically checked.

For closed evolving systems, soundness is automatically guaranteed, while completeness holds only for contexts $C' \sqsubseteq C_0$. where C_0 is the context of the top goal. Therefore, in an inheritance-based system, completeness is guaranteed only if the partial evaluation is performed starting from the leaves of the inheritance tree, and this condition can be statically verified.

Example 4.1 *Let us consider the following program P, describing a statically configured, evolving system (i.e. an inheritance-based system);*

$$\begin{array}{ll} \boldsymbol{unit}(animal, \boldsymbol{closed}([])): & \left[\begin{array}{l} mode(walk). \\ mode(run) : -no\_of\_legs(2). \\ mode(gallop) : -no\_of\_legs(4) \end{array}\right. \\ \\ \boldsymbol{unit}(bird, \boldsymbol{closed}([animal])): & \left[\begin{array}{l} no\_of\_legs(2). \\ mode(fly). \\ covering(feather). \end{array}\right. \\ \\ \boldsymbol{unit}(tweety, \boldsymbol{closed}([bird])): & [...] \end{array}$$

P represents the following taxonomy:

$$\begin{array}{c} animal \\ \uparrow \\ bird \\ \uparrow \\ tweety \end{array}$$

Now let's consider the partial evaluation with respect to the goal bird $\gg$ mode(X). If the partial search tree is exaustively explored, we get the following program:

$$\begin{array}{ll} \boldsymbol{unit}(animal, \boldsymbol{closed}([])): & \left[\begin{array}{l} mode(walk). \end{array}\right. \\ \\ \boldsymbol{unit}(bird, \boldsymbol{closed}([animal])): & \left[\begin{array}{l} mode(run) \\ mode(fly). \\ no\_of\_legs(2) \\ covering(feather). \end{array}\right. \\ \\ \boldsymbol{unit}(tweety, \boldsymbol{closed}([bird])): & [...] \end{array}$$

Notice that the inheritance link used to deduce mode(run) for bird is now statically available.

It is worth noticing that by partially evaluating wrt the top goal animal $\gg$ mode(X), we would get the uncomplete *program:*

$$\begin{array}{ll} \boldsymbol{unit}(animal, \boldsymbol{closed}([])): & \left[\begin{array}{l} mode(walk). \end{array}\right. \\ \\ \boldsymbol{unit}(bird, \boldsymbol{closed}([animal])): & \left[\begin{array}{l} no\_of\_legs(2). \\ mode(fly). \\ covering(feather). \end{array}\right. \\ \\ \boldsymbol{unit}(tweety, \boldsymbol{closed}([bird])): & [...] \end{array}$$

The problem here is that the goal bird $\gg$ mode(X) does not does not satisfy the complete-closedness condition. Therefore, trying to evaluate bird $\gg$ mode(X) in the transformed configuration, we cannot derive $\{X \leftarrow run\}$ which is one of the bindings we get from the original program.

4.1 Dealing with $\gg$ During Partial Evaluation

So far we have considered a restricted class of computations, those in which a context extension can only occur at the top level goal.

As a matter of fact dealing with generalized occurrences of context extension during unfolding involves several extensions to the framework we have being drawing so far. A first, important, remark concerns the definition of derivability, and more specifically, the label context to be associated to a derivation step involving an extension goal.

The most natural solution is to choose $\lfloor \rfloor$ as the label context for the derivation of a goal of the form $u \gg g$ since $\gg$ is a built-in predicate. Still, if this definition is assumed, some resultants may get assigned to units which do not belong to the initial context C_0.

Example 4.2 *Let us consider an conservative, dynamically configured system and the following program :*

$$\begin{array}{ll} \boldsymbol{unit}(u_1): & \boldsymbol{unit}(u_2): \\ p : -u_2 \gg q & q. \end{array}$$

$PE(P, p, \lfloor u_1 \rfloor)$ produces the following C-SLD tree:

$$\begin{array}{c} \langle p, \lfloor u_1 \rfloor \rangle \\ | \\ \lfloor u_1 \rfloor \\ | \\ \langle u_2 \gg q, \lfloor u_1 \rfloor \rangle \\ | \\ \lfloor \rfloor \\ | \\ \langle q, \lfloor u_2, u_1 \rfloor \rangle \\ | \\ \lfloor u_2, u_1 \rfloor \\ | \\ \square \end{array}$$

Accordingly, $\mathcal{F}_{ass}(p) = u_2$. This, in turn, yields uncompleteness for the subcontext $\lfloor u_1 \rfloor$. In fact, the derivation $\lfloor u_1 \rfloor \vdash p$ succeeds in P, while it fails in P'.

To avoid this problem, a reference context is to be associated with each node of the C-SLD tree. The reference context of a node N represents an upper bound (under $\sqsubseteq$) for the label contexts associated with the branches steaming from N.

New derivation rules have to be defined in order to cope with the notion of reference context. As a matter of fact they simply extend the previously defined ones. Each node of a C-SLD tree is now given by a triple $\langle G, C, R_C \rangle$.

Definition 4.4 (Derivability revised) *Let's consider a c-derivation of the form*

$$\langle G_0, C_0, R_{C_0} \rangle \vdash \ldots \vdash \langle G_n, C_n, R_{C_n} \rangle$$

The reference context R_{C_i} is defined inductively as follows:

- $R_{C_0} = C_0$;
- *given a node $N = \langle G, C, R_C \rangle$, for each descendant $\langle G', C', R'_C \rangle$ $R'_C = min\{C, R_C\}$.*

The label context L_C of a branch steaming from a node $N = \langle G, C, R_C \rangle$ is determined as follows:

$$L_C : \begin{cases} = \lfloor\ \rfloor & \text{if } G \text{ is an extension goal} \\ \in \mathcal{M}cs(G, C) & \text{if } G \text{ is a non-extension goal and} L_C \sqsubseteq R_C \\ = R_C & \text{otherwise} \end{cases}$$

Example 4.3 *According to the above definition, the following derivation tree results from the example 4.2*

$$\begin{array}{c} \langle p, \lfloor u_1 \rfloor \lfloor u_1 \rfloor \rangle \\ | \\ \lfloor u_1 \rfloor \\ | \\ \langle u_2 \gg q, \lfloor u_1 \rfloor \lfloor u_1 \rfloor \rangle \\ | \\ \lfloor\ \rfloor \\ | \\ \langle q, \lfloor u_2, u_1 \rfloor \lfloor u_1 \rfloor \rangle \\ | \\ \lfloor u_1 \rfloor \\ | \\ \square \end{array}$$

and , accordingly $\mathcal{F}_{ass}(p) = u_1$.

Finally, the rules for determining the context for the residuals must be extended to deal with $\gg$:

Definition 4.5 (Residuals) *Let $R = G : -\langle g_i, C_i \rangle$ be a c-resultant for the c-goal $\langle G_0, C_0 \rangle$. Let i.e. $C = \mathcal{D}c(R)$. Then, depending on the relationship between C and C_i, the following rules are applied:*

(i) *$C_i \sqsubseteq C$, i.e. $C = \lfloor u_N, \ldots, u_i, \ldots, u_1 \rfloor$ and $C_i = \lfloor u_j, \ldots, u_1 \rfloor$. Then:*

$$R = G : -\langle g_i, C_i \rangle \stackrel{fold}{\Longrightarrow} \left\langle \begin{array}{lll} G : -new & \stackrel{\mathcal{F}_{ass}}{\longmapsto} & top(\mathcal{D}c(R)) \\ new : -g_i & \stackrel{\mathcal{F}_{ass}}{\longmapsto} & u_j = top(C_j) \end{array} \right.$$

(ii) *C_i and C have a common initial sublist, i.e. $C = \lfloor u_n, \ldots, u_j, \ldots, u_1 \rfloor$ and $C_i = \lfloor v_k, \ldots, v_1, u_j, \ldots, u_1 \rfloor$. Then*

$$R = G : -\langle g_i, C_i \rangle \stackrel{fold}{\Longrightarrow} \left\langle \begin{array}{lll} G : -new & \stackrel{\mathcal{F}_{ass}}{\longmapsto} & top(\mathcal{D}c(R)) \\ new : -v_1 \gg \ldots \gg v_k \gg g_i & \stackrel{\mathcal{F}_{ass}}{\longmapsto} & u_j = top(C_j) \end{array} \right.$$

(iii) *C_i and C have no initial common segment. If $C_i = \lfloor v_k, \ldots, v_1 \rfloor$, then the c-resultant will be:*

$$G : -v_1 \gg \ldots \gg v_k \gg g_i \quad \stackrel{\mathcal{F}_{ass}}{\longmapsto} \quad top(\mathcal{D}c(R))$$

5 Conclusions

In this work we presented fold/unfold transformation techniques that can be suitably applied for optimizing different structured logic programming systems that are classified as statically/dynamically configured and evolving/conservative. These transformations preserve the structure of the logic program involved. The partial evaluation techniques here described extend those presented in [LS87] for Horn Clause Logic, to deal with units, contexts and context extension. In particular, we showed that some "closedeness" conditions can be stated, independently of the structured system category involved, to guarantee soundness and completeness of the transformation. It is important to note that for the most static category of structured logic programming system, i.e. the conservative statically configured one, representing traditional, block-based system, the check for soundness and completeness of the transformation is the same as that given in [LS87] for HCL. For statically configured evolving systems, such as inheritance-based ones, the check of closedness conditions is more complex, but still it can be statically performed. For dynamically configured systems, the most dynamic ones, the closedness conditions can hardly be statically checked. For this reasons, abstract interpretation techniques could be conveniently applied in this case.

Acknowledgements

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grants n.890004269. We also would like to thank ENIDATA that partially supported this work and the attendees of the Workshop on Partial Evaluation held in Milan last November for their helpful comments.

References

[Bac88] H. Bacha. Meta-prolog design and implementation. In *Proc. 5th Int. Conf. and Symp. on Logic Programming*, 1988. Seattle, MIT press.

[BLM89] A. Brogi, E. Lamma, and P. Mello. *Structuring Logic Programs: A Unifying Framework and its Declarative and Operational Semantics.* Technical Report, University of Bologna, 1989. Technical Report.

[CLM88] M. Cavalieri, E. Lamma, and P. Mello. An extended prolog machine for dynamic context handling. In *Proc. ECAI88*, 1988. Munich, Pitsman Publishing.

[FH86] K. Fukunaga and S. Hirose. An experience with a prolog-based object-oriented language. In *OOPSLA-86*, 1986. Portland, Oregon.

[Gal86] H. Gallaire. Merging objects and logic programming: relational semantics. In *AAAI '86*, 1986.

[GMR88] L. Giordano, A. Martelli, and G. F. Rossi. Local definitions with static scope rules in logic languages. In *Proceedings of the FGCS Int. Conf.*, 1988. Tokyo.

[GR84] D.M. Gabbay and N. Reyle. N-prolog: an extension of prolog with hypothetical implications. In *Journal of Logic Programming, n. 4*, 1984. pages 319-355.

[KG86] H. Kauffman and A. Grumbach. Multilog: multiple worlds in logic programming. In *Proc. ECAI86*, 1986. North-Holland.

[Kom81] H.J Komorowski. A specification of an abstract Prolog machine and its application to Partial Evaluation. In *Linkoping Studies in Science and Technology Dissertation*, 1981.

[LMN89] E. Lamma, P. Mello, and A. Natali. The design of an abstract machine for efficient implementation of contextx in logic programming. In *Proc. 6th Int. Conf. on. Logic Programming*, Lisbon 1989.

[LS87] J. W. LLoyd and J. C. Shepherdson. *Partial Evaluation in Logic Programming.* Technical Report CS-87-09, Dept. of Comp. Sc. Univ. of Bristol, Dept. of Math., Univ. Walk Bristol, December 1987.

[McC88] L.T. McCarty. Clausal intuitionistic logic. 1. fixed-point semantics. In *Journal of Logic Programming, no.5*, 1988. pages 1-31.

[Mel90] P. Mello. Inheritance as combination of horn clause theory. In *Inheritance Hierarchies in Knowledge Representation*, 1990. Wiley eds, chapter 14.

[Mil86] D. Miller. Theory of modules for logic programming. In *Proc. 1986 Symp. on Logic Programming*, 1986. Salt Lake City (USA).

[MNR89] P. Mello, A. Natali, and C. Ruggieri. Logic porgramming in a software engineering perspective. In *Proc. North American Conf. on L.P.*, Cleveland U.S. 1989. MIT press.

[MP89] L. Monteiro and A. Porto. Contextual logic programming. In *Proc. 6th Int. Conf. on. L.P.*, Lisbon 1989.

[MPr85] MPr. Mprolog language reference manual. In *LOGICWARE Inc.*, 1985. Toronto (C).

[TS84] H. Tamaky and T. Sato. Unfold/fold tranformation of logic programs. In *2th Int. Logic Progr. Conf.*, 1984.

[WZ88] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isnt like. In *in Proc. ECOOP88, and in LNCS, Vol. 322*, 1988. Springer-Verlag.

Appendix: The Operational Semantics

Let:

- A, A' be atomic goal formulae;
- g be an atomic goal or an extension goal;
- G a conjunction of atomic goals;
- ϵ, θ, σ δ be answer subsitutions; ϵ be the empty answer substitution;
- $(\theta\sigma)$ be the composition of the answer substitutions θ and σ;
- $G\theta$ be the application of the substitution θ to the formula G;
- $mgu(A, A')$ be the most general unifier of the atomic formulas A and A';
- the atomic clauses have the conventional body "true", which always holds;
- $U(P)$ be the set $\{u \mid u \text{ is a unit name}\}$;
- $|u| = \{c \mid c \text{ is a clause in u}\}$;
- $C(P) = \{ctx \mid ctx \text{ is a list of unit names}\}$
- $ctx \vdash_\theta g_i$ a top down derivation of g_i from the context ctx with substitution θ

Dynamically Configured Systems

The following set of rules is inspired by those reported in [MP89]

TRUE:

$$\frac{}{\lfloor u_N, \ldots, u_1 \rfloor \vdash_\epsilon true} \quad (1)$$

CONJUNCTION:

$$\frac{\lfloor u_N, \ldots, u_1 \rfloor \vdash_\theta g; \lfloor u_N, \ldots, u_1 \rfloor \vdash_\sigma G\theta}{\lfloor u_N, \ldots, u_1 \rfloor \vdash_{\theta\sigma} (g, G)} \quad (2)$$

ATOMIC GOAL **(Evolving Systems)**:

$$\frac{A' : -G \in |u_i|, \theta = mgu(A, A'); \lfloor u_N, \ldots, u_1 \rfloor \vdash_\sigma G\theta}{\lfloor u_N, \ldots, u_1 \rfloor \vdash_{\theta\sigma} A} \quad (3)$$

CONTEXT EXTENSION:

$$\frac{u \in U(P) \lfloor u, u_n, \ldots, u_1 \rfloor \vdash_\theta G}{\lfloor u_N, \ldots, u_1 \rfloor \vdash_\theta u \gg G} \quad (4)$$

ATOMIC GOAL **(Conservative Systems)**:

$$\frac{A' : -G \in |u_i|, \theta = mgu(A, A'); \lfloor u_i, \ldots, u_1 \rfloor \vdash_\sigma G\theta}{\lfloor u_N, \ldots, u_1 \rfloor \vdash_{\theta\sigma} A} \quad (5)$$

Statically Configured Systems

We introduce the function $closure = U(P) \mapsto C(P)$, defined in terms of the auxiliary function $seq - closure$ as follows:

$$closure(u) = \lfloor u.closure(u') \rfloor \text{ if } u' \text{ is the unit associated with } u.$$

$$closure(u) = \lfloor\ \rfloor \text{ if } u \text{ has no associated unit}$$

The context extension rule has to be modifies to capture the behaviour of closed systems:

CONTEXT EXTENSION:

$$\frac{u \in U(P); ctx = closure(u); \lfloor u|ctx \rfloor \vdash_\theta G}{\lfloor u_N, \ldots, u_1 \rfloor \vdash_\theta u \gg G} \quad (6)$$

In statically configured systems, each context extension results in a context switching to the context identified by the closure of the unit involved in the extension.

SEMANTICA

SLD-Resolution Completeness without Lifting, without Switching

Ron Sigal *
Dipartimento di Matematica
Università di Catania
Viale A. Doria 6
95125 Catania, Italy
alfredo@astrct.infn.it

1 Introduction.

It has often been remarked that the minimal Herbrand model semantics of Horn clause programs, while giving an elegant characterization of the meaning of programs, does not tell the whole story in the context of a query answering system. For example, given the program P consisting of a single clause $p(a)\leftarrow$, the statement $(\forall x)p(x)$ is true in the minimal Herbrand model M_P but not in all models. Therefore the empty substitution is neither a correct nor a computed answer substitution for the query $\leftarrow p(x)$. We could either consider M_P to be the *exact* meaning of P, in which case SLD-resolution is incomplete, or we could consider M_P to be an approximation of the meaning of P. The latter seems to be the more common point of view.

An alternative model-theoretic semantics given in terms of the class of minimal, but not necessarily Herbrand, models of a program is discussed in, for example, [Gelfond 88] and [Przymusinska 89]. While the minimal model semantics coincides, for positive programs, with the semantics of logical consequence, it loses the elegance of having a single canonical model as a point of reference. In [Falaschi 89a] and [Falaschi 89b] the use of term models based on non-ground terms is considered, and it shown that a single minimal model exists which also coincides with logical consequence. Indeed, it is shown that different subsets of the (non-ground) Herbrand base can be used to characterize operational aspects of SLD-resolution, including completeness with respect to answer substitutions. Non-ground term models have also been studied in [Clark 79], [Deransart 87], [Ferrand 86], [Ferrand 87], and [Padawitz 88] .

In this paper we focus on showing that minimal non-ground term models yield a rather straightforward proof of the SLD-resolution completeness theorem. In particular, we dispense with the Lifting Lemma, and, by the use of transfinite induction, the Switching Lemma as well. We present definitions and preliminary results in §2. While we define most of the terms used in the paper, we assume the reader is familiar with answer substitutions and SLD-resolution, which are treated thoroughly in [Lloyd 87]. In §3 we characterize the relationship between correct answer substitutions and minimal non-ground term models, and in §4 we present a proof of the completeness theorem.

2 Preliminaries.

A *signature* is a triple $\Sigma = \langle \mathcal{F}, \mathcal{P}, \mathrm{ar} \rangle$, where

*This paper was written while the author was a Visiting Professor supported by the Italian National Research Council (CNR). Current address: Yale University, Department of Mathematics, Box 2155 Yale Station, New Haven, CT 06520, USA. e-mail: sigal-ron@cs.yale.edu

- $\mathcal{F}$ is a countable (possibly infinite) set of function symbols;
- $\mathcal{P}$ is a countable (possibly infinite) set of predicate symbols;
- $\mathrm{ar} : \mathcal{F} \bigcup \mathcal{P} \to \mathrm{Nat}$, where Nat is the set of natural numbers.

We also assume the availability of a countably infinite set $\mathcal{V}$ of variable symbols, independent of Σ. The set of Σ-*terms*, denoted TM_Σ, consists of

- all $v \in \mathcal{V}$;
- all $f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}$, $\mathrm{ar}(f) = n$, and $t_1, \ldots, t_n \in \mathrm{TM}_\Sigma$.

GT_Σ denotes the set of *ground* Σ-*terms*. Σ-terms $f \in \mathcal{F}$, where $\mathrm{ar}(f) = 0$, are Σ-*constants*. Σ-*atoms*, *Horn* Σ-*clauses*, *Horn* Σ-*programs*, and Σ-*goals* are defined in the usual way.

A Σ-*pre-interpretation* is a pair $\langle \mathcal{D}, \mu \rangle$, where

- the *domain* $\mathcal{D}$ is a set of objects, and
- μ is a map defined on $\mathcal{F}$ such that $\mu(f) : \mathcal{D}^n \to \mathcal{D}$, for $f \in \mathcal{F}$, $\mathrm{ar}(f) = n$,

and a Σ-*interpretation* based on Σ-pre-interpretation $\langle \mathcal{D}, \mu \rangle$ is a pair $\langle \mathcal{D}, \mu \bigcup \mu' \rangle$, where

- μ' is a map defined on $\mathcal{P}$ such that $\mu'(p) \subseteq \mathcal{D}^n$, for $p \in \mathcal{P}$, $\mathrm{ar}(p) = n$.

Let $\langle \mathcal{D}, \mu \rangle$ be a Σ-pre-interpretation. A *variable assignment* into $\mathcal{D}$ is any map $\alpha: \mathcal{V} \to \mathcal{D}$, and the Σ-*term assignment* $\alpha_{\Sigma,\mu}: \mathrm{TM}_\Sigma \to \mathcal{D}$ is the extension of α defined

$$\alpha_{\Sigma,\mu}(t) = \begin{cases} \alpha(x) & \text{if } t \text{ is } x \in \mathcal{V} \\ \mu(f)(\alpha_{\Sigma,\mu}(t_1), \ldots, \alpha_{\Sigma,\mu}(t_n)) & \text{if } t \text{ is } f(t_1, \ldots, t_n). \end{cases}$$

Now, let $\mathrm{I} = \langle \mathcal{D}, \mu \rangle$ be a Σ-interpretation, α a variable assignment into $\mathcal{D}$, $p(t_1, \ldots, t_n)$ a Σ-atom, $A \leftarrow A_1, \ldots, A_n$, where $n \geq 0$, a Σ-clause, and P a Horn Σ-program. The 3-ary *satisfaction* relation $\models$ is defined

$$\begin{array}{lll}
\mathrm{I}, \alpha \models p(t_1, \ldots, t_n) & \text{iff} & \langle \alpha_{\Sigma,\mu}(t_1), \ldots, \alpha_{\Sigma,\mu}(t_n) \rangle \in \mu(p) \\
\mathrm{I}, \alpha \models \sim p(t_1, \ldots, t_n) & \text{iff} & \langle \alpha_{\Sigma,\mu}(t_1), \ldots, \alpha_{\Sigma,\mu}(t_n) \rangle \notin \mu(p) \\
\mathrm{I}, \alpha \models A \leftarrow A_1, \ldots, A_n & \text{iff} & \mathrm{I}, \alpha \models \sim A_i \text{ for some } i = 1, \ldots, n \quad \text{or} \quad \mathrm{I}, \alpha \models A \\
\mathrm{I}, \alpha \models \mathrm{P} & \text{iff} & \mathrm{I}, \alpha \models C \text{ for all clauses } C \text{ in P}
\end{array}$$

and the binary satisfaction relation $\models$ is defined

$$\begin{array}{lll}
\mathrm{I} \models p(t_1, \ldots, t_n) & \text{iff} & \mathrm{I}, \alpha \models p(t_1, \ldots, t_n) \text{ for all assignments } \alpha: \mathcal{V} \to \mathcal{D} \\
\mathrm{I} \models \sim p(t_1, \ldots, t_n) & \text{iff} & \mathrm{I}, \alpha \models \sim p(t_1, \ldots, t_n) \text{ for all assignments } \alpha: \mathcal{V} \to \mathcal{D} \\
\mathrm{I} \models A \leftarrow A_1, \ldots, A_n & \text{iff} & \mathrm{I}, \alpha \models A \leftarrow A_1, \ldots, A_n \text{ for all assignments } \alpha : \mathcal{V} \to \mathcal{D} \\
\mathrm{I} \models \mathrm{P} & \text{iff} & \mathrm{I} \models C \quad \text{for all clauses } C \text{ in P.}
\end{array}$$

If I is a Σ-interpretation, A is an atom, and $\mathrm{I} \models A$, we will say A is *true* in I. If P is a Σ-program and $\mathrm{I} \models \mathrm{P}$, then I is a Σ-*model* of P. If $\mathrm{M} \models A$ for all Σ-models M of P, then A is a *logical consequence* of P, denoted $\mathrm{P} \models A$.

We will be interested in interpretations whose domains are sets of terms. The GT_Σ-*pre-interpretation* is the Σ-pre-interpretation $\langle \mathrm{GT}_\Sigma, \mu^g_\Sigma \rangle$ such that μ^g_Σ satisfies

$$\mu^g_\Sigma(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n) \quad \text{for all } f \in \mathcal{F}, \mathrm{ar}(f) = n, \text{ and } t_i \in \mathrm{GT}_\Sigma,\ i = 1, \ldots, n,$$

and the TM_Σ-*pre-interpretation* is the Σ-pre-interpretation $\langle \mathrm{TM}_\Sigma, \mu_\Sigma \rangle$ such that μ_Σ satisfies

$$\mu_\Sigma(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n) \quad \text{for all } f \in \mathcal{F}, \mathrm{ar}(f) = n, \text{ and } t_i \in \mathrm{TM}_\Sigma,\ i = 1, \ldots, n.$$

A GT_Σ-*interpretation* is any Σ-interpretation based on the GT_Σ-pre-interpretation, and a TM_Σ-*interpretation* is any Σ-interpretation based on the TM_Σ-pre-interpretation. The former are generally called *Herbrand interpretations*, and we will call the latter *generic Herbrand interpretations*.

A Σ-*substitution* is a variable assignment $\theta: \mathcal{V} \to \mathrm{TM}_\Sigma$, which, like any variable assignment, can be extended to the Σ-term assignment $\theta_{\Sigma,\mu_\Sigma}: \mathrm{TM}_\Sigma \to \mathrm{TM}_\Sigma$, where μ_Σ comes from the TM-pre-interpretation. We will write $t\theta$ to denote $\theta_{\Sigma,\mu_\Sigma}(t)$. (That is, informally, that $t\theta$ works the way one expects.) We will also use the shorthand notation

$$\begin{array}{lll}
p(t_1, \ldots, t_n)\theta & \text{abbreviates} & p(t_1\theta, \ldots, t_n\theta) \\
(A \leftarrow A_1, \ldots, A_n)\theta & \text{abbreviates} & A \leftarrow A_1\theta, \ldots, A_n\theta
\end{array}$$

where $p(t_1, \ldots, t_n)$ is an atom and $A \leftarrow A_1, \ldots, A_n$ is a clause. If θ is a Σ-substitution and E is a Σ-term, Σ-atom, or Σ-clause, then $E\theta$ is an *instance* of E. The *domain* of a Σ-substitution θ, denoted $\mathrm{dom}(\theta)$, is a subset $X \subseteq \mathcal{V}$ such that $\theta(x) \neq x$ for $x \in X$ and $\theta(x) = x$ for $x \in \mathcal{V} \setminus X$. The *identity substitution*, denoted ε, is the variable assignment such that $\mathrm{dom}(\varepsilon) = \emptyset$. A substitution $\rho: \mathcal{V} \to \mathcal{V}$ is a *renaming substitution*, and $E\rho$ is a *variant* of E, where E is a Σ-term, Σ-atom, or Σ-clause. For Σ-substitution θ and $X \subseteq \mathcal{V}$, the Σ-substitution $\theta|_X$ is

$$\theta|_X(x) = \begin{cases} \theta(x) & \text{if } x \in X \\ x & \text{otherwise.} \end{cases}$$

Composition of Σ-substitutions θ and γ, denoted $\theta \circ \gamma$, is defined by

$$\theta \circ \gamma(x) = \theta_{\Sigma,\mu_\Sigma}(\gamma(x)) \text{ for all } x \in \mathcal{V},$$

and will be written $x\gamma\theta$. A Σ-substitution θ is *idempotent* if $\theta = \theta \circ \theta$. We will sometimes make implicit use of the equality

$$(\theta \circ \gamma)_{\Sigma,\mu_\Sigma}(t) = \theta_{\Sigma,\mu_\Sigma}(\gamma_{\Sigma,\mu_\Sigma}(t) \text{ for all } t \in \mathrm{TM}_\Sigma$$

(easily proven by induction on t) by writing $t\gamma\theta$ to denote, ambiguously, either side of the equality.

If P is a Σ-program and G is a Σ-goal $\leftarrow A_1, \ldots, A_n$, the Σ-substitution θ is an *answer substitution* for $\mathrm{P} \bigcup \{\mathrm{G}\}$ if $\mathrm{dom}(\theta) \subseteq \mathrm{var}(\mathrm{G})$, where $\mathrm{var}(\mathrm{G})$ denotes the set of variables which occur in G. θ is a *correct answer substitution* for $\mathrm{P} \bigcup \{\mathrm{G}\}$ if $\mathrm{P} \models A_i\theta$, $i = 1, \ldots, n$. An R-*computed answer substitution* for $\mathrm{P} \bigcup \{\mathrm{G}\}$ is one returned by an SLD-refutation of $\mathrm{P} \bigcup \{\mathrm{G}\}$ via computation rule R. We refer the reader to [Lloyd 87] for the precise definition.

The following proposition is a straightforward consequence of the definition of $\models$.

Proposition 1 *Let* $\mathrm{I} = \langle \mathrm{TM}_\Sigma, \mu \rangle$ *be a* Σ-*interpretation,* $p(t_1, \ldots, t_n)$ *a* Σ-*atom, and* $\theta: \mathcal{V} \to \mathrm{TM}_\Sigma$ *a* Σ-*substitution. Then*

$$\mathrm{I} \models p(t_1, \ldots, t_n) \quad \textit{implies} \quad \mathrm{I} \models p(t_1\theta, \ldots, t_n\theta).$$

A TM_Σ-interpretation $\langle \mathrm{TM}_\Sigma, \mu \rangle$ is Σ-*closed* if, for all $p \in \mathcal{P}$,

$$\langle t_1, \ldots, t_n \rangle \in \mu(p) \quad \text{implies} \quad \langle t_1\theta, \ldots, t_n\theta \rangle \in \mu(p), \text{ for all } \theta: \mathcal{V} \to \mathrm{TM}_\Sigma.$$

For Σ-closed TM_Σ-interpretations (and GT_Σ-interpretations) the following easily proved propositions hold.

Proposition 2 *Let* $\mathrm{I} = \langle \mathrm{TM}_\Sigma, \mu \rangle$ *be a* Σ-*closed* TM_Σ-*interpretation (a* GT_Σ-*interpretation),* $p(t_1, \ldots, t_n)$ *a* Σ-*atom, and* $\theta: \mathcal{V} \to \mathrm{TM}_\Sigma$ (*resp.,* $\theta: \mathcal{V} \to \mathrm{GT}_\Sigma$) *a variable assignment. Then*

$$\mathrm{I}, \theta \models p(t_1, \ldots, t_n) \quad \textit{if and only if} \quad \mathrm{I} \models p(t_1\theta, \ldots, t_n\theta).$$

Proposition 3 *Let* $\mathrm{I} = \langle \mathrm{TM}_\Sigma, \mu \rangle$ *be a* Σ-*closed* TM_Σ-*interpretation and* $p(t_1, \ldots, t_n)$ *a* Σ-*atom. Then*

1. $\mathrm{I} \models p(t_1, \ldots, t_n)$ *if and only if* *for all* $\theta: \mathcal{V} \to \mathrm{TM}_\Sigma$, $\mathrm{I} \models p(t_1\theta, \ldots, t_n\theta)$.
2. $\mathrm{I} \models {\sim}p(t_1, \ldots, t_n)$ *if and only if* *for all* $\theta: \mathcal{V} \to \mathrm{TM}_\Sigma$, $\mathrm{I} \not\models p(t_1\theta, \ldots, t_n\theta)$.

It is customary to identify Herbrand interpretations with subsets of the *Herbrand* Σ-*base*

$$\mathrm{B}_\Sigma = \{A \mid A \text{ is a ground } \Sigma\text{-atom}\},$$

and we can make a similar identification for generic Herbrand interpretations. The *generic Herbrand* Σ-*base*, denoted $\mathrm{B}^{\mathrm{v}}_\Sigma$, is

$$\{A \mid A \text{ is a } \Sigma\text{-atom}\}.$$

A subset $S \subseteq \mathrm{B}^{\mathrm{v}}_\Sigma$ is Σ-*closed* if

$$p(t_1, \ldots, t_n) \in S \quad \text{implies} \quad p(t_1\theta, \ldots, t_n\theta) \in S, \text{ for all } \theta: \mathcal{V} \to \mathrm{TM}_\Sigma.$$

Let I be a TM_Σ-interpretation. The subset $\hat{\mathrm{I}} \subseteq \mathrm{B}^{\mathrm{v}}_\Sigma$ *associated with* I is given by

$$\hat{\mathrm{I}} = \{A \in \mathrm{B}^{\mathrm{v}}_\Sigma \mid \mathrm{I} \models A\}.$$

On the other hand, given a subset $S \subseteq \mathrm{B}^{\mathrm{v}}_\Sigma$, the TM_Σ-interpretation I_S *associated with* S is $\langle \mathrm{TM}_\Sigma, \mu \rangle$, where

$$\mu(p) = \{\langle t_1, \ldots, t_n \rangle \mid t_1, \ldots, t_n \text{ are } \Sigma\text{-terms and } p(t_1, \ldots, t_n) \in S\}$$

for all $p \in \mathcal{P}$. Now we have the following simple propositions, whose proofs are omitted.

Proposition 4 *Let* I *be a* Σ-*closed* TM_Σ-*interpretation and* S *a* Σ-*closed subset of* $\mathrm{B}^{\mathrm{v}}_\Sigma$. *Then*

1. $\hat{\mathrm{I}}$ *is* Σ-*closed;*
2. I_S *is* Σ-*closed;*

3. $\mathrm{I} = \mathrm{I}_{\hat{\mathrm{I}}}$;
4. $S = \widehat{\mathrm{I}_S}$.

Henceforth all TM_Σ-interpretations will be assumed to be Σ-closed. By Propositions 4.3 and 4.4 there is a one-one correspondence between Σ-closed TM_Σ-interpretations and Σ-closed subsets of $\mathrm{B}^{\mathrm{v}}_\Sigma$, so we can use, as convenient, either characterization of generic Herbrand interpretations.

Proposition 5 *Let* I *be a* Σ-*closed* TM_Σ-*interpretation and* S *a* Σ-*closed subset of* $\mathrm{B}^{\mathrm{v}}_\Sigma$. *Then*

1. $\mathrm{I} \models p(t_1, \ldots, t_n)$ *if and only if* $p(t_1, \ldots, t_n) \in \hat{\mathrm{I}}$;
2. $p(t_1, \ldots, t_n) \in S$ *if and only if* $\mathrm{I}_S \models p(t_1, \ldots, t_n)$.

We will also have recourse to one other kind of term-based interpretation. Let $\Sigma = \langle \mathcal{F}, \mathcal{P}, \mathrm{ar} \rangle$ be a signature, and let $\mathcal{C}$ be a set of constants such that $\mathcal{F} \bigcap \mathcal{C} = \emptyset$. We will denote by $\Sigma_\mathcal{C}$ the signature $\langle \mathcal{F} \bigcup \mathcal{C}, \mathcal{P}, \mathrm{ar} \bigcup \mathrm{ar}_\mathcal{C} \rangle$, where $\mathrm{ar}_\mathcal{C}$ indicates that the elements of $\mathcal{C}$ are constants. A $(\mathrm{TM}_\Sigma, \mathcal{C})$-*interpretation* is any $\Sigma_\mathcal{C}$-interpretation $\langle \mathrm{TM}_\Sigma, \mu \bigcup \mu_\mathcal{C} \rangle$ such that $\langle \mathrm{TM}_\Sigma, \mu \rangle$ is a (Σ-closed) TM_Σ-interpretation and $\mu_\mathcal{C}(c) \in \mathrm{TM}_\Sigma$, for all $c \in \mathcal{C}$. Intuitively, the elements of $\mathcal{C}$ are to be Skolem constants, interpreted in TM_Σ. A *Skolem assignment* for Σ is a one-one variable assignment $\sigma: \mathcal{V} \to \mathcal{C} \bigcup \mathcal{V}$ such that $\mathcal{F} \bigcap \mathcal{C} = \emptyset$ and such that $\sigma(x) \in \mathcal{V}$ implies $\sigma(x) = x$. If t is a Σ-term and σ a Skolem assignment $\sigma: \mathcal{V} \to \mathcal{C} \bigcup \mathcal{V}$, then $t\sigma$ denotes $\sigma_{\Sigma_\mathcal{C}, \mu_{\Sigma_\mathcal{C}}}(t)$.

3 Minimal generic Herbrand models.

In this section we show (1) that every program has a minimal generic Herbrand model and (2) that this model characterizes correct answer substitutions for the program. First we show that the model intersection property holds for generic Herbrand models of a program, which implies the first result. The second result, Theorem 10 below, is a non-ground analog to the theorem ([Lloyd 87], Theorem 6.2, p. 37) that the minimal Herbrand model of a program P is the set of ground atoms which are logical consequences of P. The proof of Theorem 10 is similar to that given in [Lloyd 87] (and originally in [van Emden 76]), but is more involved because negation of non-ground atoms entails the use of Skolem constants, which enlarges the language and changes the class of Herbrand models. Lemma 7, the straightforward proof of which is omitted, justifies the use of Skolem constants in our quantifier-free languages. Lemma 8 shows that for a set $\mathcal{C}$ of Skolem constants not occurring in signature Σ we can, in some cases, interpret the elements of $\mathcal{C}$ as non-ground Σ-terms (variables, in particular) and so exchange Herbrand $\Sigma_\mathcal{C}$ models for generic Herbrand Σ-models. Lemma 9 shows that Skolem constants behave essentially as variables.

Theorem 6 (Model intersection property) *Let* P *be a* Σ-*program,* $\{\mathrm{M}_i \mid i \in \Phi\}$ *a non-empty family of* TM_Σ-*models of* P, *and* $S = \bigcap\{\hat{\mathrm{M}}_i \mid i \in \Phi\}$. *Then* I_S *is a* TM_Σ-*model of* P.

Proof. It is clear that I_S is a TM_Σ-interpretation, so we need to show that it is a model of P. Let $A \leftarrow A_1, \ldots, A_n$, $n \geq 0$, be a clause in P, and let $\theta: \mathcal{V} \to \mathrm{TM}_\Sigma$ be any assignment. Then

$$\begin{aligned} \mathrm{M}_i \models A \leftarrow A_1, \ldots, A_n \quad \text{for all } i \in \Phi \quad &\Rightarrow \quad \mathrm{M}_i, \theta \models A \leftarrow A_1, \ldots, A_n \quad \text{for all } i \in \Phi \\ &\Rightarrow \quad \mathrm{M}_i, \theta \models A \quad \text{or} \quad \mathrm{M}_i, \theta \models {\sim}A_j \quad \text{for some } j = 1, \ldots, n. \end{aligned}$$

If $\mathrm{M}_i, \theta \models {\sim}A_j$, for some $i \in \Phi$, $j = 1, \ldots, n$, then

	$M_i, \theta \models \sim A_j$	
$\Rightarrow$	$M_i, \theta \not\models A_j$	by definition of $\models$
$\Rightarrow$	$M_i \not\models A_j\theta$	by Proposition 2
$\Rightarrow$	$A_j\theta \notin \hat{M}_i$	by Proposition 5.1
$\Rightarrow$	$A_j\theta \notin S$	by definition of S
$\Rightarrow$	$I_S \not\models A_j\theta$	by Proposition 5.2
$\Rightarrow$	$I_S, \theta \not\models A_j$	by Proposition 2
$\Rightarrow$	$I_S, \theta \models \sim A_j$	by definition of $\models$
$\Rightarrow$	$I_S, \theta \models A \leftarrow A_1, \ldots, A_n$	by definition of $\models$.

Otherwise $M_i, \theta \models A$ for all $i \in \Phi$, and so

	$M_i \models A\theta$	for all $i \in \Phi$, by Proposition 2
$\Rightarrow$	$A\theta \in \hat{M}_i$	for all $i \in \Phi$, by Proposition 5.1
$\Rightarrow$	$A\theta \in S$	by definition of S
$\Rightarrow$	$I_S \models A\theta$	by Proposition 5.2
$\Rightarrow$	$I_S, \theta \models A$	by Proposition 2
$\Rightarrow$	$I_S, \theta \models A \leftarrow A_1, \ldots, A_n$	by definition of $\models$.

In either case we have $I_S, \theta \models A \leftarrow A_1, \ldots, A_n$, and since θ was arbitrary, we have $I_S \models A \leftarrow A_1, \ldots, A_n$. Thus I_S satisfies each clause in P, i.e., $I_S \models P$. □

Since B^u_Σ is a model of any Σ-program P, the set of TM_Σ-models of P is always non-empty, and so the *minimal generic Herbrand Σ-model*

$$M^u_{\Sigma,P} = I_S, \text{ where } S = \bigcap\{\hat{M} \mid M \text{ is a } TM_\Sigma\text{-model of P}\},$$

of P always exists. Before we prove the theorem which gives the relationship between $M^u_{\Sigma,P}$ and correct answer substitutions for P, we need three lemmas.

Lemma 7 *Let $\Sigma = \langle \mathcal{F}, \mathcal{P}, \mathrm{ar}\rangle$ be a signature, P a Σ-program, A a Σ-atom, C a set of constants such that $C \bigcap \mathcal{F} = \emptyset$, and $\kappa: C \bigcup \mathcal{V}$ a Skolem assignment such that $A\kappa$ is ground. Then*

$$M \models A \quad \textit{for all } \Sigma\textit{-models of } P \quad \textit{if and only if} \quad P \bigcup \{\sim A\kappa\} \quad \textit{has no } \Sigma_C\textit{-models.}$$

Lemma 8 *Let $\Sigma = \langle \mathcal{F}, \mathcal{P}, \mathrm{ar}\rangle$ be a signature, P a Σ-program, and A a ground Σ_C-atom $p(t_1, \ldots, t_n)$, where C is some set of constants such that $C \bigcap \mathcal{F} = \emptyset$. Then $P \bigcup \{\sim A\}$ has a (TM_Σ, C)-model if and only if it has a GT_{Σ_C}-model.*

Proof.
($\Leftarrow$) For Σ_C-atom A we will write $[A]_{\Sigma_C}$ to denote $\{A\theta \mid \theta: \mathcal{V} \rightarrow GT_{\Sigma_C}\}$. Now, let

- M be a GT_{Σ_C}-model of $P \bigcup \{\sim A\}$;
- $S = \{A \in B^u_\Sigma \mid [A]_{\Sigma_C} \subseteq \hat{M}\}$;

- $I_S = \langle TM_\Sigma, \mu_1\rangle$ be the TM_Σ-interpretation associated with S;
- $M' = \langle TM_\Sigma, \mu\rangle$, where $\mu = \mu_1 \bigcup \mu_2$ and $\mu_2: C \rightarrow \mathcal{V}$ interprets each $c \in C$ as some variable x_c.

It is clear that M' is a (TM_Σ, C)-interpretation, and so we must show that it is a model of $P \bigcup \{\sim A\}$.

First we show that $M' \models \sim p(t_1, \ldots, t_n)$. Let $A_\mu = p(\varepsilon_{\Sigma_C,\mu}(t_1), \ldots, \varepsilon_{\Sigma_C,\mu}(t_n))$. It is easy to see that t_i is an instance of $\varepsilon_{\Sigma_C,\mu}(t_i)$, $i = 1, \ldots, n$, and that $\varepsilon_{\Sigma_C,\mu}(t_i)$ is a Σ-term, $i = 1, \ldots, n$. We have

	$M \models \sim A$	by assumption
$\Rightarrow$	$M \not\models A$	
$\Rightarrow$	$A \notin \hat{M}$	by Proposition 5.1
$\Rightarrow$	$[A_\mu]_{\Sigma_C} \not\subseteq \hat{M}$	since A is an instance of A_μ
$\Rightarrow$	$A_\mu \notin S$	by definition of S
$\Rightarrow$	$I_S \not\models A_\mu$	by Proposition 5.2
$\Rightarrow$	$I_S, \varepsilon \not\models A_\mu$	by Proposition 2 (note that I_S is a TM_Σ-interpretation and A_μ is a Σ-atom)
$\Rightarrow$	$\langle \varepsilon_{\Sigma,\mu_1}(\varepsilon_{\Sigma_C,\mu}(t_1)), \ldots, \varepsilon_{\Sigma,\mu_1}(\varepsilon_{\Sigma_C,\mu}(t_n))\rangle \notin \mu_1(p)$	by definition of $\models$ and A_μ
$\Rightarrow$	$\langle \varepsilon_{\Sigma_C,\mu}(t_1), \ldots, \varepsilon_{\Sigma_C,\mu}(t_n)\rangle \notin \mu_1(p)$	since $\varepsilon_{\Sigma_C,\mu}(t_i)$ is a Σ-term, $i = 1, \ldots, n$
$\Rightarrow$	$\langle \varepsilon_{\Sigma_C,\mu}(t_1), \ldots, \varepsilon_{\Sigma_C,\mu}(t_n)\rangle \notin \mu(p)$	by definition of M'
$\Rightarrow$	$\langle \theta_{\Sigma_C,\mu}(t_1), \ldots, \theta_{\Sigma_C,\mu}(t_n)\rangle \notin \mu(p)$	for all $\theta: \mathcal{V} \rightarrow TM_\Sigma$, since t_i, $i = 1, \ldots, n$, is ground
$\Rightarrow$	$M', \theta \models \sim p(t_1, \ldots, t_n)$	for all $\theta: \mathcal{V} \rightarrow TM_\Sigma$
$\Rightarrow$	$M' \models \sim p(t_1, \ldots, t_n)$	by definition of $\models$.

Now let C be a clause $B \leftarrow B_1, \ldots, B_n$ in P. We must show that $M', \theta \models C$ for any $\theta: \mathcal{V} \rightarrow TM_\Sigma$. If $M', \theta \models \sim B_i$ for some i we are done, so assume otherwise. Let $\gamma: \mathcal{V} \rightarrow GT_{\Sigma_C}$ be any substitution. Then

	$M', \theta \models B_i, \ \ i = 1, \ldots, n$	by assumption
$\Rightarrow$	$I_S, \theta \models B_i, \ \ i = 1, \ldots, n$	since C is a Σ-clause
$\Rightarrow$	$I_S \models B_i\theta, \ \ i = 1, \ldots, n$	by Proposition 2, since I_S is a TM_Σ-interpretation
$\Rightarrow$	$B_i\theta \in S, \ \ i = 1, \ldots, n$	by Proposition 5.2
$\Rightarrow$	$[B_i\theta]_{\Sigma_C} \subseteq \hat{M}, \ \ i = 1, \ldots, n$	by definition of S
$\Rightarrow$	$B_i\theta\gamma \in \hat{M}, \ \ i = 1, \ldots, n$	by definition of $[B_i\theta]_{\Sigma_C}$
$\Rightarrow$	$M \models B_i\theta\gamma, \ \ i = 1, \ldots, n$	by Proposition 5.1
$\Rightarrow$	$M, \theta\gamma \models B_i, \ \ i = 1, \ldots, n$	by Proposition 2
$\Rightarrow$	$M, \theta\gamma \models B$	since $M, \theta\gamma \models B \leftarrow B_1, \ldots, B_n$
$\Rightarrow$	$B\theta\gamma \in \hat{M}$	by Propositions 2 and 5.1

$\Rightarrow$ $[B\theta]_{\Sigma_C} \subseteq \hat{M}$ since γ was arbitrary
$\Rightarrow$ $B\theta \in S$ by definition of S
$\Rightarrow$ $I_S, \theta \models B$ by Propositions 5.2 and 2
$\Rightarrow$ $M', \theta \models B$ since B is a Σ-atom
$\Rightarrow$ $M', \theta \models B \leftarrow B_1, \ldots, B_n$ by definition of $\models$.

($\Rightarrow$) Let $M = \langle TM_\Sigma, \mu \rangle$ be a (TM_Σ, C)-model of $P \cup \{\sim A\}$, and let $M' = \langle GT_{\Sigma_C}, \mu' \rangle$ be the GT_{Σ_C}-interpretation such that

- $\mu'(p) = \{\langle t_1, \ldots, t_n \rangle \mid t_1, \ldots, t_n \in GT_{\Sigma_C}$ and $M \models p(t_1, \ldots, t_n)\}$ for all $p \in \mathcal{P}$.

Let $q(u_1, \ldots, u_m)$ be any ground Σ_C-atom such that $M' \models q(u_1, \ldots, u_m)$. Then

$M' \models q(u_1, \ldots, u_m)$
$\Rightarrow$ $M', \theta \models q(u_1, \ldots, u_m)$ for any $\theta: \mathcal{V} \to GT_{\Sigma_C}$
$\Rightarrow$ $\langle \theta_{\Sigma_C,\mu'}(u_1), \ldots, \theta_{\Sigma_C,\mu'}(u_m) \rangle \in \mu'(q)$ by definition of $\models$
$\Rightarrow$ $\langle u_1, \ldots, u_m \rangle \in \mu'(q)$ since u_i is a ground Σ_C-term, $i = 1, \ldots, n$
$\Rightarrow$ $M \models q(u_1, \ldots, u_m)$ by definition of μ',

i.e.,

$$M' \models B \quad \text{implies} \quad M \models B, \text{ for all ground } \Sigma_C\text{-atoms } B, \tag{1}$$

and so, in particular,

$M \models \sim A \Rightarrow M \not\models A$
$\Rightarrow$ $M' \not\models A$ by (1)
$\Rightarrow$ $M' \models \sim A$ since A is ground.

Now assume $q(u_1, \ldots, u_m)$ is any (not necessarily ground) Σ_C-atom such that $M \models q(u_1, \ldots, u_m)$, and let $\theta: \mathcal{V} \to GT_{\Sigma_C}$ be any assignment. Then

$M \models q(u_1, \ldots, u_m)$
$\Rightarrow$ $M \models q(u_1\theta, \ldots, u_m\theta)$ by Proposition 1
$\Rightarrow$ $\langle u_1\theta, \ldots, u_m\theta \rangle \in \mu'(q)$ by definition of μ'
$\Rightarrow$ $\langle \theta_{\Sigma_C,\mu'}(u_1), \ldots, \theta_{\Sigma_C,\mu'}(u_m) \rangle \in \mu'(q)$ since M' is a GT_{Σ_C}-interpretation
$\Rightarrow$ $M', \theta \models q(u_1, \ldots, u_m)$ by definition of $\models$
$\Rightarrow$ $M' \models q(u_1, \ldots, u_m)$ since θ was arbitrary,

i.e.,

$$M \models B \quad \text{implies} \quad M' \models B, \text{ for all } \Sigma_C\text{-atoms } B, \tag{2}$$

and thus $M' \models B$ for any clause $B \leftarrow$ in P.

Now let $A \leftarrow A_1, \ldots, A_n$, $n > 0$, be a clause in P, and let $\theta: \mathcal{V} \to GT_{\Sigma_C}$ be any assignment. We will show that $M', \theta \models A \leftarrow A_1, \ldots, A_n$, and therefore, since $\theta: \mathcal{V} \to GT_{\Sigma_C}$ is arbitrary, $M' \models A \leftarrow A_1, \ldots, A_n$. If there is an A_i, $i = 1, \ldots, n$, such that $M', \theta \models \sim A_i$ we are done, so assume otherwise. Then

$M', \theta \models A_i$, $i = 1, \ldots, n$
$\Rightarrow$ $M' \models A_i\theta$, $i = 1, \ldots, n$ by Proposition 2
$\Rightarrow$ $M \models A_i\theta$, $i = 1, \ldots, n$ by (1), since $A_i\theta$ is a ground Σ_C-atom, $i = 1, \ldots, n$
$\Rightarrow$ $M \models A\theta$ since $M \models A \leftarrow A_1, \ldots, A_n$
$\Rightarrow$ $M' \models A\theta$ by (2)
$\Rightarrow$ $M', \theta \models A$ by Proposition 2
$\Rightarrow$ $M', \theta \models A \leftarrow A_1, \ldots, A_n$ by definition of $\models$,

and the proof is complete. □

Lemma 9 *Let $\Sigma = \langle \mathcal{F}, \mathcal{P}, \mathrm{ar} \rangle$ be a signature, P a Σ-program, A a Σ-atom, C a set of constants such that $C \cap \mathcal{F} = \emptyset$, $\kappa: \mathcal{V} \to C \cup \mathcal{V}$ a Skolem assignment such that C is contained in the range of κ. Then $A\kappa$ is true in all (TM_Σ, C)-models of P if and only if A is true in all TM_Σ-models of P.*

Proof.
($\Rightarrow$) Assume that $A\kappa$ is true in all (TM_Σ, C)-models of P, let $M = \langle TM_\Sigma, \mu \rangle$ be any TM_Σ-model of P, and let $\theta: \mathcal{V} \to TM_\Sigma$ be any variable assignment. We need to show that $M, \theta \models A$. Let $M_\theta = \langle TM_\Sigma, \mu_\theta \rangle$ be the (TM_Σ, C)-model of P such that

- $\mu_\theta(c) = (\kappa^{-1}(c))\theta$, for $c \in C$;
- $\mu_\theta(f) = \mu(f)$, for $f \in \mathcal{F}$;
- $\mu_\theta(p) = \mu(p)$, for $p \in \mathcal{P}$.

It is clear that

$$M_\theta, \theta \models A\kappa \quad \text{if and only if} \quad M, \theta \models A,$$

and $M_\theta, \theta \models A\kappa$ by assumption, so we have $M, \theta \models A$. But θ was arbitrary, so we have $M \models A$.
($\Leftarrow$) Assume that A is true in all TM_Σ-models of P, and let $M = \langle TM_\Sigma, \mu \rangle$ be any (TM_Σ, C)-model of P. We need to show that $M \models A\kappa$. Let $M' = \langle TM_\Sigma, \mu' \rangle$ be the TM_Σ-model of P such that $\mu' = \mu|_{\mathcal{F} \cup \mathcal{P}}$, let $\theta: \mathcal{V} \to TM_\Sigma$ be any Σ-substitution, and let $\theta': \mathcal{V} \to TM_\Sigma$ be the Σ-substitution

$$\theta'(x) = \begin{cases} \mu(\kappa(x)) & \text{if } \kappa(x) \in C \\ \theta(x) & \text{otherwise.} \end{cases}$$

Then it is clear that

$$M', \theta' \models A \quad \text{if and only if} \quad M, \theta \models A\kappa,$$

and $M', \theta' \models A$ by assumption, so we have $M, \theta \models A\kappa$. But θ was arbitrary, and so we have $M \models A\kappa$, and the proof is complete. □

Now we can prove the main theorem of this section.

Theorem 10 *Let $\Sigma = \langle \mathcal{F}, \mathcal{P}, \mathrm{ar} \rangle$ be a signature, P a Σ-program, and A a Σ-atom. Then Σ-substitution θ is a correct answer substitution for $P \cup \{\leftarrow A\}$ if and only if $M^v_{\Sigma,P} \models A\theta$.*

Proof. Let $\mathrm{var}(A\theta) = \{x_1, \ldots, x_n\}$, let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be a set of constant symbols such that $\mathcal{C} \bigcap \mathcal{F} = \emptyset$, and let κ be a Skolem assignment such that $\kappa(x_i) = c_i$, $i = 1, \ldots, n$. Then

$\mathrm{P} \models A\theta$

iff $\mathrm{M} \models A\theta$ for all Σ-models M of P

iff $\mathrm{P} \bigcup \{\sim A\theta\kappa\}$ has no $\Sigma_{\mathcal{C}}$-models, by Lemma 7

iff $\mathrm{P} \bigcup \{\sim A\theta\kappa\}$ has no $\mathrm{GT}_{\Sigma_{\mathcal{C}}}$-models, since $\mathrm{P} \bigcup \{\sim A\theta\kappa\}$ is a set of $\Sigma_{\mathcal{C}}$-clauses

iff $\mathrm{P} \bigcup \{\sim A\theta\kappa\}$ has no $(\mathrm{TM}_\Sigma, \mathcal{C})$-models, by Lemma 0

iff $A\theta\kappa$ is true in all $(\mathrm{TM}_\Sigma, \mathcal{C})$-models of P, since $A\theta\kappa$ is a ground $\Sigma_{\mathcal{C}}$-atom

iff $A\theta$ is true in all TM_Σ-models of P, by Lemma 0

iff $\mathrm{M}^{\mathrm{v}}_{\Sigma,\mathrm{P}} \models A\theta$. □

Corollary 11 *Let* P *be a* Σ*-program and* G *a* Σ*-goal* $\leftarrow A_1, \ldots, A_n$*. Then* θ *is a correct answer substitution for* $\mathrm{P} \bigcup \{\mathrm{G}\}$ *if and only if* $\mathrm{M}^{\mathrm{v}}_\Sigma \models A_i\theta$, $i = 1, \ldots, n$.

4 SLD-completeness.

In this section we give another proof of the completeness of SLD-resolution with respect to Horn clauses. In [Lloyd 87] the constructive fixpoint characterization of minimal Herbrand models is a key part of the completeness proof, and we adopt the same strategy here with respect to minimal generic Herbrand models.

Let $\mathrm{CP}_\Sigma = \{S \subseteq \mathrm{B}^{\mathrm{v}}_\Sigma \mid S \text{ is } \Sigma\text{-closed}\}$. It is easy to see that CP_Σ is a complete lattice. For each Σ-program P we define the transformation function $\mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}} \colon \mathrm{CP}_\Sigma \to \mathrm{CP}_\Sigma$

$$\mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}(\mathrm{I}) = \{A \in \mathrm{B}^{\mathrm{v}}_\Sigma \mid \begin{array}{l} 1).\ B \leftarrow B_1, \ldots, B_n \text{ is a variant of a clause in P;} \\ 2).\ A = B\theta \text{ for some } \theta \colon \mathcal{V} \to \mathrm{TM}_\Sigma; \\ 3).\ B_1\theta, \ldots, B_n\theta \in \mathrm{I}\ \}. \end{array}$$

It is clear that $\mathrm{I} \in \mathrm{CP}_\Sigma$ implies $\mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}(\mathrm{I}) \in \mathrm{CP}_\Sigma$. The ordinal powers $\mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}{\uparrow}\alpha$ of $\mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}$ are defined in the usual way, and the proof of the following theorem is virtually identical to that given in [Lloyd 87] for its ground Herbrand interpretation counterpart.

Theorem 12 *Let* P *be a* Σ*-program. Then* $\widehat{\mathrm{M}^{\mathrm{v}}_{\Sigma,\mathrm{P}}} = \mathrm{lfp}(\mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}) = \mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}{\uparrow}\omega$.

We now turn to the completeness theorem. We avoid the use of the Switching Lemma ([Lloyd 87], pp. 50–51) in proving completeness for an arbitrary computation rule by means of induction on a transfinite ordinal measure assigned to (goal, correct answer substitution) pairs. Let P be a Σ-program, A a Σ-atom, and θ a correct answer substitution for $\mathrm{P} \bigcup \{\leftarrow A\}$. We define

$$\Omega_{\Sigma,\mathrm{P}}(\leftarrow A, \theta) = \omega^{n_0}, \quad \text{where } n_0 = \min_n A\theta \in \mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}{\uparrow}n.$$

Now let Π_n be the set of permutations of the sequence $1, \ldots, n$, and let $\leftarrow A_1, \ldots, A_n$ be a Σ-goal G and θ a correct answer substitution for $\mathrm{P} \bigcup \{\mathrm{G}\}$. Then we define

$$\Omega_{\Sigma,\mathrm{P}}(\mathrm{G}, \theta) = \max_{\pi \in \Pi_n} \sum_{i=1}^{n} \Omega_{\Sigma,\mathrm{P}}(\leftarrow A_{\pi(i)}, \theta).$$

That is, $\Omega_{\Sigma,\mathrm{P}}(\mathrm{G}, \theta)$ is the largest ordinal that can be obtained by summing over $\Omega_{\Sigma,\mathrm{P}}(\leftarrow A_i, \theta)$, $i = 1, \ldots, n$.

Theorem 13 (Completeness of SLD-resolution) *Let* P *be a* Σ*-program,* G *a* Σ*-goal* $\leftarrow A_1, \ldots, A_n$*, and* R *a computation rule. For every correct answer substitution* θ *for* $\mathrm{P} \bigcup \{\mathrm{G}\}$ *there exists an* R*-computed answer substitution* σ *for* $\mathrm{P} \bigcup \{\mathrm{G}\}$ *and a substitution* γ *such that*[1] $\theta = (\sigma\gamma)|_{\mathrm{var}(\mathrm{G})}$.

Proof. We assume that $\mathrm{P} \bigcup \{\mathrm{G}\}$ has a correct answer substitution θ (otherwise the theorem is vacuously true) and argue by transfinite induction on $\Omega_{\Sigma,\mathrm{P}}(\mathrm{G}, \theta)$. By Corollary 0 and Theorem 12 we have $A_1\theta, \ldots, A_n\theta \in \mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}{\uparrow}\omega$. Let A_k be the atom in G chosen by R, and let $\Omega_{\Sigma,\mathrm{P}}(\leftarrow A_k, \theta) = \omega^{n_0}$. Note that $n_0 > 0$ since $\mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}{\uparrow}0 = \emptyset$. $A_k\theta$ owes its presence in $\mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}{\uparrow}n_0$ to

(a) a variant $B_0 \leftarrow B_1, \ldots, B_m$ of a clause in P, with $m \geq 0$;

(b) atoms $B'_1, \ldots, B'_m \in \mathrm{T}^{\mathrm{v}}_{\Sigma,\mathrm{P}}{\uparrow}n_0 - 1$;

(c) a substitution τ such that $B_i\tau = B'_i$, $i = 1, \ldots, m$, and $B_0\tau = A_k\theta$.

We can assume without loss of generality that $\mathrm{dom}(\tau) \subseteq \bigcup_{i=0}^{m} \mathrm{var}(B_i)$ and that $B_0 \leftarrow B_1, \ldots, B_m$ and $B'_1, \ldots, B'_m$ are chosen so that $\mathrm{var}(\mathrm{G}) \bigcap \bigcup_{i=0}^{m} \mathrm{var}(B_i) = \emptyset$ and that $\mathrm{var}(\mathrm{G}) \bigcap \bigcup_{i=1}^{m} \mathrm{var}(B'_i) = \emptyset$, and therefore

$$B'_i\theta = B'_i, \ i = 1, \ldots, m \quad \text{and} \quad A_j\tau = A_j, \ j = 1, \ldots, n. \tag{3}$$

We also assume, for now, that θ is idempotent.

Now, by item (b) above, Corollary 0, and Theorem 12, ε is a correct answer substitution for $\mathrm{P} \bigcup \{\leftarrow B'_1, \ldots, B'_m\}$, and so θ is a correct answer substitution for

$$\mathrm{P} \bigcup \{\leftarrow A_1, \ldots, A_{k-1}, B'_1, \ldots, B'_m, A_{k+1}, \ldots, A_n\}.$$

Moreover, since

$$(\leftarrow A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)\tau = \leftarrow A_1, \ldots, A_{k-1}, B'_1, \ldots, B'_m, A_{k+1}, \ldots, A_n,$$

$\tau\theta$ is a correct answer substitution for $\mathrm{P} \bigcup \{\mathrm{G}_1\}$, where G_1 denotes

$$\leftarrow A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n.$$

Now,

$$\begin{array}{rcll} A_k\tau\theta & = & A_k\theta & \text{by (3)} \\ & = & (A_k\theta)\theta & \text{by idempotence} \\ & = & (B_0\tau)\theta & \text{by (c),} \end{array}$$

so that $\tau\theta$ is a unifier of A_k and B_0. Let μ be an mgu of A_k and B_0. Then there is a substitution δ such that

$$\mu\delta = \tau\theta. \tag{4}$$

Therefore $\mu\delta$ is a correct answer substitution for $\mathrm{P} \bigcup \{\mathrm{G}_1\}$, and δ is a correct answer substitution for $\mathrm{P} \bigcup \{\mathrm{G}_1\mu\}$. Now,

[1] We point out here that there seems to be a slight flaw in the statement of the theorem given in [Lloyd 87], where the concluding equation is $\theta = \sigma\gamma$. If there are $x \in \mathrm{dom}(\sigma)$, $y \in \mathrm{var}(x\sigma)$ such that $y \in \mathrm{dom}(\gamma)$ and $y \notin \mathrm{var}(\mathrm{G})$, then $\mathrm{dom}(\sigma\gamma) \not\subseteq \mathrm{var}(\mathrm{G})$. For example, if program P consists of the single clause $p(f(x)) \leftarrow$, then $\theta = \{y/f(a)\}$ is a correct answer substitution for $\mathrm{P} \bigcup \{\leftarrow p(y)\}$, and $\sigma = \{y/f(x_1)\}$ is a computed answer substitution (assuming that x is renamed to x_1). Let $\gamma = \{x_1/a\}$. Then $p(f(y))\sigma\gamma = p(f(a)) = p(f(y))\theta$, but $\mathrm{dom}(\sigma\gamma) = \{y, x_1\} \not\subseteq \mathrm{var}(\leftarrow p(y))$.

$$\begin{aligned}
\Omega_{\Sigma,P}(\leftarrow B_1,\ldots,B_m,\tau\theta) &= \Omega_{\Sigma,P}(\leftarrow B_1,\ldots,B_m,\tau) && \text{by (3), since } B_i\tau = B_i',\ i=1,\ldots,m\\
&\preceq m\cdot\omega^{n_0-1} && \text{by (b)}\\
&\prec \omega^{n_0}\\
&= \Omega_{\Sigma,P}(\leftarrow A_k,\theta) && \text{by assumption,}\\
&= \Omega_{\Sigma,P}(\leftarrow A_k,\tau\theta) && \text{by (3),}
\end{aligned}$$

i.e.,

$$\Omega_{\Sigma,P}(\leftarrow B_1,\ldots,B_m,\tau\theta) \prec \Omega_{\Sigma,P}(\leftarrow A_k,\tau\theta) \tag{5}$$

so that

$$\begin{aligned}
\Omega_{\Sigma,P}(G_1\mu,\delta) &= \Omega_{\Sigma,P}(G_1,\mu\delta)\\
&= \Omega_{\Sigma,P}(G_1,\tau\theta) && \text{by (4)}\\
&\prec \Omega_{\Sigma,P}(G,\tau\theta) && \text{by (5)}\\
&\prec \Omega_{\Sigma,P}(G,\theta) && \text{by (3)}
\end{aligned}$$

and we can appeal to the induction hypothesis to conclude that there is an R-computed answer substitution σ' for $P\bigcup\{G_1\mu\}$ and a substitution γ such that

$$\delta = (\sigma'\gamma)|_{\mathrm{var}(G_1\mu)}. \tag{6}$$

The definition of a computed answer substitution shows that it is of the form $(\theta_1\circ\cdots\circ\theta_r)|_{\mathrm{var}(G')}$, where G' is the input goal and $\theta_1,\ldots,\theta_r$ is the sequence of mgu's created during the resolution process. Therefore, σ' is $\sigma''|_{\mathrm{var}(G_1\mu)}$ for some σ'', and since A_k is the atom in G chosen by R and μ is the mgu of A_k and the head of clause $B_0\leftarrow B_1,\ldots,B_m$, it follows that $(\mu\sigma'')_{\mathrm{var}(G)}$ is an R-computed answer substitution for $P\bigcup\{G\}$. Before we can finish we need to prove the following fact:

$$(\mu\circ(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)})|_{\mathrm{var}(G)} = (\mu\sigma''\gamma)|_{\mathrm{var}(G)}. \tag{7}$$

For any substitution ζ, let $\mathrm{var}(\zeta) = \mathrm{dom}(\zeta)\bigcup\bigcup_{x\in\mathrm{dom}(\zeta)}\mathrm{var}(x\zeta)$. Examination of the usual unification algorithms shows that if terms t, u are unifiable, there is an mgu ζ such that $\mathrm{var}(\zeta)\subseteq \mathrm{var}(t)\bigcup\mathrm{var}(u)$. Thus, if $\chi|_{\mathrm{var}(G')}$ is a computed answer substitution for some goal G', the variables in $\mathrm{var}(\chi)$ come from either G' or from the variants of program clauses chosen during the refutation, and since the variants are arbitrary, we can assume $\mathrm{var}(\chi)\subseteq \mathrm{var}(G')\bigcup X$ for any (sufficiently large) X we like. In particular, we assume

$$\mathrm{var}(\sigma'')\subseteq \mathrm{var}(G_1\mu)\bigcup X, \tag{8}$$

where X is such that

$$(\mathrm{var}(B_0)\setminus\mathrm{var}(G_1\mu))\bigcap X = \emptyset \tag{9}$$

and

$$(\mathrm{var}(G)\setminus\mathrm{var}(G_1\mu))\bigcap X = \emptyset. \tag{10}$$

Since $\mathrm{var}(\mu)\subseteq \mathrm{var}(G)\bigcup\mathrm{var}(B_0)$, it follows from (9) and (10) that

$$(\mathrm{var}(\mu)\setminus\mathrm{var}(G_1\mu))\bigcap X = \emptyset. \tag{11}$$

Also, we can assume without loss of generality that γ is such that

$$\mathrm{dom}(\gamma)\subseteq \mathrm{var}(\sigma')\bigcup\mathrm{var}(G_1\mu)\subseteq \mathrm{var}(\sigma'')\bigcup\mathrm{var}(G_1\mu)\subseteq \mathrm{var}(G_1\mu)\bigcup X. \tag{12}$$

Now, if $x\in(\mathrm{var}(G)\setminus\mathrm{dom}(\mu))\bigcap\mathrm{var}(G_1\mu)$, then, by definition of σ', we have $x\sigma' = x\sigma''$, and so

$$\begin{aligned}
x(\mu\circ(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)})|_{\mathrm{var}(G)} &= x(\mu\circ(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)})\\
&= x(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)}\\
&= x(\sigma'\gamma)\\
&= x(\sigma''\gamma) = x(\mu\sigma''\gamma) = x(\mu\sigma''\gamma)|_{\mathrm{var}(G)}.
\end{aligned}$$

If $x\in(\mathrm{var}(G)\setminus\mathrm{dom}(\mu))\setminus\mathrm{var}(G_1\mu)$ then, by (10), we have $x\notin\mathrm{var}(G_1\mu)\bigcup X$, which implies, by (8) and (12) respectively, that $x\notin\mathrm{dom}(\sigma'')$ and $x\notin\mathrm{dom}(\gamma)$, and we have

$$x(\mu\circ(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)})|_{\mathrm{var}(G)} = x(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)} = x = x(\sigma''\gamma) = x(\mu\sigma''\gamma)|_{\mathrm{var}(G)}.$$

Now assume $x\in\mathrm{var}(G)\bigcap\mathrm{dom}(\mu)$, and let $y\in\mathrm{var}(x\mu)$. By (11) either $y\in\mathrm{var}(G_1\mu)$, in which case

$$y(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)} = y(\sigma'\gamma) = y(\sigma''\gamma),$$

or $y\notin\mathrm{var}(G_1\mu)\bigcup X$, so that $y\notin\mathrm{dom}(\sigma'')$ and $y\notin\mathrm{dom}(\gamma)$, and again we have

$$y(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)} = y = y(\sigma''\gamma).$$

Since y was any variable in $x\mu$ we now have

$$x(\mu\circ(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)})|_{\mathrm{var}(G)} = x(\mu\circ(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)}) = x(\mu\sigma''\gamma) = x(\mu\sigma''\gamma)|_{\mathrm{var}(G)},$$

and since we have covered every $x\in\mathrm{var}(G)$, (7) is proved.

So, if we set $\sigma = (\mu\sigma'')|_{\mathrm{var}(G)}$, then we have R-computed answer substitution σ and substitution γ such that

$$\begin{aligned}
(\sigma\gamma)|_{\mathrm{var}(G)} &= ((\mu\sigma'')|_{\mathrm{var}(G)}\circ\gamma)|_{\mathrm{var}(G)} && \text{by definition of } \sigma\\
&= (\mu\sigma''\gamma)|_{\mathrm{var}(G)}\\
&= (\mu\circ(\sigma'\gamma)|_{\mathrm{var}(G_1\mu)})|_{\mathrm{var}(G)} && \text{by (7)}\\
&= (\mu\delta)|_{\mathrm{var}(G)} && \text{by (6)}\\
&= (\tau\theta)|_{\mathrm{var}(G)} && \text{by (4)}\\
&= \theta && \text{since } \mathrm{dom}(\tau)\bigcap\mathrm{var}(G) = \emptyset \text{ and } \mathrm{dom}(\theta)\subseteq\mathrm{var}(G),
\end{aligned}$$

concluding the proof for idempotent θ.

Finally, if θ is not idempotent, we can take the idempotent correct answer substitution $\theta' = (\theta\rho)|_{\mathrm{var}(G)}$, where ρ is a one-one renaming substitution that renames the variables which occur in terms in the range of θ to variables not in the domain of θ, apply the argument above to get $\theta' = (\sigma\gamma')|_{\mathrm{var}(G)}$, and set $\gamma = \gamma'\rho^{-1}$, so that

$$\begin{aligned}
\theta &= (\theta\rho\rho^{-1})|_{\mathrm{var}(G)}\\
&= ((\theta\rho)|_{\mathrm{var}(G)}\circ\rho^{-1})|_{\mathrm{var}(G)}\\
&= (\theta'\rho^{-1})|_{\mathrm{var}(G)}\\
&= ((\sigma\gamma')|_{\mathrm{var}(G)}\circ\rho^{-1})|_{\mathrm{var}(G)}\\
&= (\sigma\gamma'\rho^{-1})|_{\mathrm{var}(G)}\\
&= (\sigma\gamma)|_{\mathrm{var}(G)},
\end{aligned}$$

concluding the proof for arbitrary θ. □

References

[Clark 79] Clark, K.L., "Predicate logic as a computational formalism," Research report 79/59, Dept. of Computing, Imperial College, 1979.

[Deransart 87] Deransart, P., and G. Ferrand, "Programmation en logique avec negation: presentation formelle," Rapport de recherche 87/3, Laboratoire d'Informatique, Département de Mathématiques et d'Informatique, Université d'Orléans, France, 1987.

[Falaschi 89a] Falaschi, M., Levi, G., Martelli, M., Palamidessi, C., "Declarative modelling of the operational behavior of logic languages", *Theoretical Computer Science,* 70 (1989).

[Falaschi 89b] Falaschi, M., Levi, G., Martelli, M., Palamidessi, C., "A model-theoretic reconstruction of the operational semantics of logic programs," Dipartimento di Informatica tech. rep. TR-32/89, Univ. Pisa.

[Ferrand 86] Ferrand, G., "A reconstruction of logic programming with negation," Rapport de recherche 86/5, Laboratoire d'Informatique, Département de Mathématiques et d'Informatique, Université d'Orléans, France, 1986.

[Ferrand 87] Ferrand, G., "Error diagnosis in logic programming, an adaption of E. Y. Shapiro's method," *Journal of Logic Programming,* 4 (1987), 177–198.

[Gelfond 88] Gelfond, M., Przymusinska, H., Przymusinska, T., "Minimal model semantics vs. negation as failure: A comparison of semantics," *Proc. ACM SIGART Int. Symp. on Methodologies for Intelligent Systems,* Torino, Italy, 1988, 335–343.

[Lloyd 87] Lloyd, J. W., *Foundations of Logic Programming,* 2nd ed., Springer-Verlag, Berlin, 1987.

[Padawitz 88] Padawitz, P, *Computing in Horn Clause Theories,* Springer-Verlag, Berlin, 1988.

[Przymusinska 89] Przymusinska, T., "On the declarative and procedural semantics of logic programs," *Journal of Automated Reasoning,* 5 (1989), 167–206.

[van Emden 76] "The semantics of predicate logic as a programming language," *Journal of the ACM,* 23 (1976), 733–742.

A contribution to the automated treatment of membership theories *

E. G. Omodeo, F. Parlamento
Dipartimento di Matematica e Informatica, Università di Udine
Via Zanon 6, 33100 Udine, Italy.

A. Policriti
Computer Science Department
Courant Institute of Mathematical Sciences, New York University
251 Mercer St., 10012 New York, New York; and
Dipartimento di Matematica e Informatica, Università di Udine

Abstract. The class of all sentences of the form $\exists x_1 \cdots \exists x_n \forall x\, p$, where the unquantified matrix p involves only $\in$ and $=$ (along with first-order variables and propositional connectives), is proved complete, hence decidable, in various theories of membership. Among these Zermelo-Fraenkel theory, where a finite bound can be calculated *a priori* for the minimal rank of sets —if any— satisfying $\forall x\, p$. Novel methods ensue for deciding multilevel syllogistic augmented with the singleton operator. Furthermore, an explicit procedure is supplied for decomposing any given $\forall x\, p$ as an irredundant disjunction $\bigvee_{t \in \mathcal{T}_p} \Phi_t$. The decomposition method can be tuned to different membership theories, including some that are too weak to make the class of $\exists_n \forall$-sentences complete. The latter theories admit a non-refutability test, although not a satisfiability test, applicable to $\forall x\, p$-formulae.

Key phrases: satisfiability problem, set theory, decision test, multilevel syllogistic.

1 Introduction

A declarative programming system —and, more generally, any automated theorem-prover— promises to be extremely reliable. Hopefully, at the same time, it will offer support to high level dictions and respond quickly to easy goals. These three requirements are, however, hard to conciliate with one another. Can a system entirely based on resolution meet them?

*This work has been partially supported by the AXL project of ENI and ENIDATA.

The resolution principle, thanks to its simplicity (hence robustness) has gained increasing popularity through the years. To date, however, no general heuristics have been devised that can effectively dominate the combinatorial explosion taking place during the search for a refutation.

When high level dictions come into play (think, e.g., of the `bagof` and `setof` constructs of `Prolog`, or of lazy variants of these), the situation becomes even worse. Such dictions, in fact, increase the charge loaded on the inference engine by thick families of clauses, without the alleviation of sophisticated data-structures or algorithms. They do ease programming —at the price, unfortunately, of a dramatic loss of efficiency.

Many have felt bitterly disappointed with resolution for this reason, and have advocated specialized theorem-proving techniques and domain-dependent heuristics. Among these, Bledsoe [Ble77] says:

> *The author was one of the researchers working on resolution type systems who "made the switch". It was in trying to prove a rather simple theorem in set theory by paramodulation and resolution, where the program was experiencing a great deal of difficulty, that we became convinced that we were on the wrong track.*[1]

Early in the history of automated deduction an indication came on how to move towards the fulfillment of the various requirements (robustness, speed, expressiveness): the semantics of a few very special constructs, so pervasive in mathematics that they had become central in logical investigations too, was to be built into the basic inferential machinery of theorem-provers. Robinson [Rob67] wrote:

> *[...] we are sorely handicapped by having to treat equality as simply one more binary relation symbol, with a special collection of extra axioms, in each problem, being required to make it behave as it should. This, it seems to me, is the next large problem which must be solved before we get off the present plateau.*
>
> *Beyond that, and much more difficult still, is the problem of handling the* membership *relation in an efficient way, so that theorem-proving problems involving set-theoretic notions can be treated. The only available way at present is to use a finite set of axioms for set theory, say, those given in Gödel [...], and to formulate all our other theorem-proving problems* within *that theory. While in principle this approach is capable of handling any mathematical problem which can be formulated in any meaningful way that is known at present, it is of purely academic and theoretical interest unless the efficiency of the basic first-order theorem-proving procedures is improved by further orders of magnitude.*
>
> *As to whether such improvements are possible, one can only say that nothing is known at the present time which conclusively establishes that they are not.*

Progress along this guide has been slow but steady. While our ability to master equality is still developing (cf., e.g., [DLS89]), a significant amount of preparatory work has been done concerning membership and other basic set-theoretic constructs (see [Pas78, Bro78, CFO89] among others). Even the approach regarded with initial pessimism by Robinson, namely treating set theory as just an ordinary first-order theory, has been sifted to the bottom [BLM*86].

[1] The 'rather simple theorem' was $\mathcal{P}(A \cap B) = \mathcal{P}(A) \cap \mathcal{P}(B)$, with $\mathcal{P}$ representing the powerset operation.

It must be noted, however, that in spite of the efforts sustained by many reseachers through the last decade and more, we still know very little about the interactions between membership and quantification.

We add the remark that, unlike equality, membership does not possess a uniquely determined behaviour. Actually, it can be constrained logically in various manners, conflicting with one another. However, the flexibility in the semantics of membership, appealing for some applications —see [Acz88]— has not attracted much attention till recently, outside logic.

In this paper a fresh restart is attempted to the study of membership. Although the class of quantified set-theoretic formulae that we investigate is rather narrow, it spans the whole multi-level syllogistic extended with the singleton operator [FOS80], and a little more. To make the results more general and useful, various possible interpretations of membership have been taken into account: the ordinary one (extensional, acyclic and well-founded), a membership that is not well-founded, one that is not extensional, and one that is neither extensional nor well-founded.

The class of formulae taken under study here, was shown complete by Gogol in 1976 (cf. [Gog78]). Gogol's result states that one of the two opposite sentences

$$\exists x_1 \cdots \exists x_n \forall x\, p\,, \qquad \forall x_1 \cdots \forall x_n \exists x\, \neg p$$

is a logical consequence of ZFC (Zermelo-Fraenkel set theory including the axioms of regularity and choice) when p is an unquantified formula all of whose symbols are drawn from $x, x_1, \ldots, x_n, =, \in, \neg, \&$.[2] It follows in a standard way that one can decide whether or not $\forall x\, p$ is *satisfiable* (i.e. $\exists x_1 \cdots \exists x_n \forall x\, p$ is provable).

Exploiting finitary proof-theoretic arguments, instead of the model-theoretic methods employed by Gogol, we have discovered that his completeness result does not depend in the least on strong (hence debatable) assumptions about sets, such as those embodied by ZFC. As a matter of fact, the same result applies to the non-standard theories of memberships mentioned above.

We have succeeded in setting the ground for totally explicit satisfiability decision tests applicable to $\forall x\, p$-formulae.

One of our methods presupposes only the extensionality and regularity axioms, together with axioms stating the existence of an empty set and the closure of the class of all sets with respect to the operations `with` and `less` (addition and removal of an element)[3].

Of course, one could exploit resolution for an alternative implementation of this satisfiability test. In order to ease an implementation of this kind, we have laid down a clausal formulation of the theory (Sections 2, 3). This could be done also for the other theories, antithetic to ZF, but then one would have to cope with heavily encumbering sets of clauses. On the other hand, with our direct satisfiability test —see next paragraph— these other theories are easier to deal with.

[2] Throughout, this will be the tacit assumption about p (with or without subscripts), and $x, x_1, \ldots, x_n$ will stand for distinct variables ranging over all sets.

[3] It is interesting to note that $\emptyset$, `with` and `less`, $=$ and $\in$, are among the basic constructs of the programming language SETL too, cf. [DF89].