SYNTHESIS OF EUREKA PREDICATES FOR DEVELOPING LOGIC PROGRAMS

Maurizio Proietti
IASI-CNR
Viale Manzoni 30
I-00185 Roma (Italy)

Alberto Pettorossi
Department of Electronics
University of Roma Tor Vergata
I-00173 Roma (Italy)

ABSTRACT

We consider the problem of inventing new predicates when developing logic programs by transformation. Those predicates, often called eureka predicates, improve program efficiency by eliminating redundant computations and avoiding multiple visits of data structures. It can be shown that no general method exists for inventing the required eureka predicates for a given initial program. We introduce here two strategies, the *Loop Absorption Strategy* and the *Generalization Strategy*, which in many cases determine the new predicates to be defined during program transformation. We study the properties of those strategies and we present some classes of programs in which they are successful.

1. INTRODUCTION AND A PRELIMINARY EXAMPLE

The program transformation methodology is a valuable technique for deriving correct and efficient programs. It has been introduced for functional languages [Burstall-Darlington 77, Darlington 81, Feather 86], but it can also be applied in the case of logic programs (see, for instance, [Azibi 87, Bossi et al. 88, Bruynooghe et al. 89, Debray 88, Hogger 81, Kawamura-Kanamori 88, Nakagawa 85, Tamaki-Sato 84]).

The basic idea of that methodology is the *separation of concerns*, in the sense that the efficiency requirement is separated from the correctness requirement. The latter one is taken into account by allowing only transformation rules which are correctness preserving. (The programmer is left with the problem of termination, which is not captured by the notion of partial correctness.)

The efficiency requirement is taken into account by considering heuristic techniques. This is indeed the best approach which can be adopted, because of the undecidability results we will present in the paper. The programmer is asked to derive from the initial program a more efficient version by inventing suitable auxiliary predicates, the so-called *eureka predicates*. These inventions are hard steps to be made during the program derivation process, but several strategies can be adopted. Two of them are the *Loop Absorption Strategy* and the *Generalization Strategy*, introduced below. They are powerful techniques which are successful in a large number of cases.

The transformation rules which we will use for program derivation are the following ones (they are named after the corresponding rules which are used for the transformation of functional programs):

- *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 (and not in terms of itself). Each new predicate should occur as head of one definition 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 current program) whose head is

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of the CNR and by the MPI 40%, Italy.

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 $i=1,\ldots,n$. This rule can be applied only if: i) unfolding the new clause with respect to the atom $K\sigma$ we obtain again the old clause, and ii) the bridge clause is a definition (that is, a clause introduced in a previous application of the Definition Rule) and it is different from the old clause. ■

Our rules are a bit simpler than the ones presented in [Tamaki-Sato 84]. However, they allow the same transformations, and it is easy to see that they preserve the least Herbrand model semantics of the predicates occurring in the initial program version (that is, the set of ground facts which are derivable from the initial program is equal to the set of ground facts derivable in the transformed program).

The above rules will be used for deriving more efficient programs from the given initial versions, according to the steps of the transformation methodology we now indicate.

The Transformation Methodology.

i) Given an initial program version the program-transformer (a person or an automatic system) considers a clause and selects in its body some atoms which share one or more variables. Indeed, that sharing may be the cause of inefficiencies which we would like to avoid, like for instance, repeated visits of data, or constructions of unnecessary structures, or redundant computations, etc.

ii) The program-transformer defines a clause C whose body is made out of the selected atoms.
Notice that if clause C differs from a clause already existing in the program because of the head predicate only, the introduction of that clause C can be avoided, and the existing clause can be considered as a definition.

iii) The program-transformer looks for a recursive definition of the head predicate of the clause defined at Step ii) by applying the unfolding and folding rules. This step of the methodology is motivated by the need of removing the cause of inefficiency 'at each level of recursion'. (The examples given in the paper will clarify this idea.)
It is often the case that such recursive definition can only be derived via the introduction of eureka predicates (which in turn should be recursively defined). If the process of unfolding clause C generates clauses with patterns of atoms which are 'recurrent', the suitable eureka predicates can be determined by applying the Loop Absorption Strategy. (The notions occurring here will be formally defined later, and a precise description of this step will be given in Section 4 when defining the Loop Absorption Procedure.)

iv) Sometimes the required clauses with recurrent patterns of atoms may not be found by unfoldings only. In those cases the program-transformer may apply the Generalization Strategy, which will be presented in Section 3. ■

More details about the transformation methodology will also be given when presenting the various examples. Here is the first one.

Example 1. Maximal-Up-Segments.

Let us consider the problem of computing the maximal length of the up-segments of a given list L. Let L be a list of integers $[a_1,\ldots,a_n]$. We say that S is an up-segment of L if it is a sublist of L made out of consecutive elements $a_h,\ldots,a_k$ such that:

i) $a_h \leq a_{h+1} \leq \ldots \leq a_k$, and

ii) $a_{h-1} > a_h$ and $a_k > a_{k+1}$. We assume that $a_0=+\infty$ and $a_{n+1}=-\infty$.

Here is a logic program which solves that problem by computing: i) the list LL of all up-segments of L, and ii) the maximal length of the elements of LL.

1. max-ups(L,Max) ← ups(L,LL), maxlength(LL,Max).
2. ups([],[]).

3. ups([H],[[H]]).
4. ups([H,H1|T1],[[H]|Y]) ← H>H1, ups([H1|T1],Y).
5. ups([H,H1|T1],[[H|Y1]|Y]) ← H≤H1, ups([H1|T1],[Y1|Y]).
6. maxlength([],0).
7. maxlength([H|T],M) ← maxlength(T,M1), length(H,N), max(N,M1,M).

We assume the obvious meaning for the base predicates >, ≤, max, and length.

We would like to improve the above program by avoiding the construction of the intermediate list LL of clause 1. We may achieve that objective by applying our proposed program transformation methodology. We consider clause 1 where the atoms ups and maxlength share the variable LL and we look for a recursive definition of max-ups(L,Max). We do not define a new head predicate, because of the remark made at Step ii) of our methodology.

We unfold clause 1 and the clauses derived from it until we obtain a set of clauses such that in the body of each of them either i) there are no calls of ups and maxlength, or ii) there is a pattern of calls of ups and maxlength which is an instance of the pattern of calls occurring in the body of an *ancestor-clause*. (In Section 2 we will formalize the unfolding process as a *tree* of clauses and the notion of ancestor-clause will be the one based on the tree structure). The clauses of type ii) will be called *foldable clauses*, because as we will see later, they will be used as 'old clauses' for performing folding steps. By unfolding clause 1 we get the following semantically equivalent set of clauses:

8. max-ups([],0).
9. max-ups([H],Max) ← length([H],N), max(N,0,Max).
10. max-ups([H,H1|T1],Max) ← H>H1, ups([H1|T1],Y), maxlength(Y,M1), length([H],N), max(N,M1,Max).
11. max-ups([H,H1|T1],Max) ← H≤H1, ups([H1|T1],[Y1|Y]), maxlength(Y,M1), length([H|Y1],N), max(N,M1,Max).

Now, clauses 8 and 9 have bodies without calls of ups and maxlength, and in clause 10 the pattern of atoms 'ups([H1|T1],Y), maxlength(Y,M1)' is an instance of the pattern 'ups(L,LL), maxlength(LL,Max)' in the body of clause 1. However, the pattern 'ups([H1|T1],[Y1|Y]), maxlength(Y,M1)' in clause 11 is *not* an instance of any pattern of atoms occurring in its ancestor-clauses. We continue the unfolding process for clause 11 and we get:

12. max-ups([H,H1],Max) ← H≤H1, length([H,H1],N), max(N,M1,Max).
13. max-ups([H,H1,H2|T2],Max) ← H≤H1, H1>H2, ups([H2|T2],Y), maxlength(Y,M1), length([H,H1],N), max(N,M1,Max).
14. max-ups([H,H1,H2|T2],Max) ← H≤H1, H1≤H2, ups([H2|T2],[Y1|Y]), maxlength(Y,M1), length([H,H1|Y1],N), max(N,M1,Max).

At this point we have that:

i) clauses 8, 9, and 12 do not have calls of ups and maxlength,

ii) the bodies of clauses 10 and 13 have calls of ups and maxlength which are instances of the corresponding calls in the body of their ancestor-clause 1 (thus, clause 10 and 13 are foldable), and

iii) the body of clause 14 has calls of ups and maxlength which are an instance of the corresponding calls in the body of the ancestor-clause 11 (thus, clause 14 is foldable).

We say that we obtained two patterns of the atoms ups and maxlength, which are *recurrent*: the first one in the pairs of clauses 1-10 and 1-13, and the second one in the pair 11-14.

We also say that the foldable clauses 10, 13, and 14 generate the three *clause-loops* 1-10, 1-13, and 11-14 in the tree of unfoldings (see the bold up-arrows in Figure 1 in Section 2).

Now, as we have anticipated, we terminate the unfolding process and we introduce two new predicates, which are the eureka predicates, defined by clauses whose bodies are exactly the recurrent patterns of the atoms ups and maxlength, as they occur in the bodies of the ancestor-clauses 1 and 11.

These steps will be later formalized as applications of the *Loop Absorption Strategy*. The name of this strategy derives from the fact that the ancestor-clauses of the clause-loops 1-10, 1-13, and 11-14 have been *absorbed* into the definitions of the predicates new1 and new2 introduced below.
Since in our case the ancestor-clauses are 1 and 11, the eureka predicates are:
15. new1(L,Max) ← ups(L,LL), maxlength(LL,Max).
16. new2(H1,T1,Y1,M1) ← ups([H1|T1],[Y1|Y]), maxlength(Y,M1).

The variables of the predicate new1 and new2 are determined by the following requirements:
- new1 and new2 should be used in the program for max-ups. Thus, the variables for new1 are those which allow a folding step which uses clause 1 as old clause and clause 15 as bridge clause, and the variables for new2 are those which allow a folding step which uses clause 11 as old clause and clause 16 as bridge clause.
- new1 and new2 should have a recursive definition. Thus, the variables for new1 are also those which allow the folding steps which use clauses 10 and 13 as old clauses and clause 15 as bridge clause, and the variables for new2 are also those which allow a folding step which uses clause 14 as old clause and clause 16 as bridge clause.

During the program derivation process we may avoid the introduction of new predicates when their definition is equal to clauses which already exist in the program. Thus, in our case, we do not introduce the predicate new1: indeed clause 15 is equal to clause 1, apart from the name of the head-predicate.

Now, we can fold clauses 10 and 11 by using the clauses 1 and 16 as bridge clauses, and we obtain:
10.1 max-ups([H,H1|T1],Max) ← H>H1, max-ups([H1|T1],M1), length([H],N), max(N,M1,Max).
11.1 max-ups([H,H1|T1],Max) ← H≤H1, new2(H1,T1,Y1,M1), length([H|Y1],N), max(N,M1,Max).

We are now left with the problem of finding the explicit (recursive) definition of the newly introduced predicate new2. This problem can easily be solved by performing again the unfolding steps which have been performed when deriving clauses 12, 13, and 14 starting from clause 11. Two final folding steps will then be required for the clauses which correspond to clauses 13 and 14. (Recall that in the body of clause 12 no atoms for ups and maxlength occur, and thus for the clause corresponding to clause 12 no folding step is required).
This method of finding the recursive definitions of eureka predicates is a general method. It is always successful simply because the Loop Absorption Strategy is applied only when recurrent patterns of atoms have been found.
The derived program for the predicate new2 is:
17. new2(H,[],[H],0).
18. new2(H,[H1|T1],[H],M) ← H>H1, max-ups([H1|T1],M).
19. new2(H,[H1|T1],[H|Y1],M) ← H≤H1, new2(H1,T1,Y1,M).
The final program for max-ups is given by clauses 8, 9, 10.1, 11.1, 17, 18, and 19.
Minor improvements can be performed by simplifying clauses 9 and 10.1, and we get:
9.1 max-ups([H],1).
10.2 max-ups([H,H1|T1],Max) ← H>H1, max-ups([H1|T1],M1), max(1,M1,Max).
In the derived program the construction of the intermediate list LL between the predicates ups and maxlength has been avoided, as we desired.

As in the initial clause 1, we observe that in clause 11.1 the atoms new2(H1,T1,Y1,M1) and length([H|Y1],N) share the common variable Y1, and in order to avoid the construction of the bindings for that shared variable, we may begin another transformation process starting from the clause:
new3(H,H1,T1,M1,N) ← new2(H1,T1,Y1,M1), length([H|Y1],N).
We leave that derivation to the interested reader. The final program which can be derived is as follows:

max-ups([],0).
max-ups([H],1).
max-ups([H,H1|T1],Max) ← H>H1, max-ups([H1|T1],M1), max(1,M1,Max).
max-ups([H,H1|T1],Max) ← H≤H1, new3(H,H1,T1,M1,N), max(N,M1,Max).
new3(H,H1,[],0,2).
new3(H,H1,[H2|T2],M1,2) ← H1>H2, max-ups([H2|T2],M1).
new3(H,H1,[H2|T2],M1,N) ← H1≤H2, new3(H1,H2,T2,M1,N1), N is N1+1.

The improvement of the performances of the derived program is due to the fact that during the unfolding process we have found some clauses whose bodies have recurrent patterns of atoms.

The choice of the unfolding steps to be performed for obtaining such clauses is indeed one of the main problems of the program transformation methodology. We address that problem in this paper, and we solve it by defining a rule, called SDR (short for Synchronized Descent Rule), which selects the atoms to be unfolded. Our SDR rule allows us to find recurrent patterns of atoms, if they exist, for some classes of programs which we will specify later. For those classes of programs, during the derivation process nothing is left to the reader's intuition, because once the recurrent patterns of atoms have been found, then the suitable eureka predicates are derived. For that reason we think that our work is a contribution towards making the transformation methodology more automatic and more useful in practice. □

In Section 2 we formalize the process of unfolding a given clause so that clauses with recurrent patterns of atoms are derived, by introducing the notion of a foldable U-tree. We also formalize the problem of finding foldable U-trees as the *Foldability Problem*, and we show the unsolvability of that problem for various classes of programs, thus providing a theoretical limitation for the transformational approach to program derivation.

In Section 3 we introduce the SDR rule for performing unfolding steps and we also introduce a class of programs for which that rule, together with some applications of the Generalization Strategy, allows us to solve the Foldability Problem. We also consider some interesting subclasses of those programs, in which the Generalization Strategy is not necessary. For those subclasses we extend previously known results.

In Section 4 we describe the Loop Absorption Strategy, which given a foldable U-tree finds the eureka predicates and their recursive definitions, so that program performances can be improved.

Finally, in Section 5 we discuss some problems one may encounter when the order of atoms and clauses affects the semantics of the programs (as in the Prolog case).

2. THE FOLDABILITY PROBLEM

We now present the notion of U-tree for a given program and a given clause. It is a tree of clauses derived by performing unfolding steps. We also present the notion of foldable U-tree, which is a U-tree where some of the derived clauses have bodies with recurrent patterns of atoms. Those recurrent patterns will suggest the definitions of the auxiliary eureka predicates which allow the improvement of program performances.

Let us introduce some preliminary definitions. A program is a definite logic program [Lloyd 87] in which some *base predicates* (like, for instance, equality, concatenation of lists, arithmetic predicates, etc.) have no explicit definition, that is, they do not occur in the heads of the program clauses. The non-base predicates are called *defined predicates*. In the Maximal-Up-Segments Example above, the base predicates are length, max, ≤, and >, while the defined predicates are max-ups, ups, and maxlength. The meaning of a defined predicate can be given in the usual way in terms of the meanings of the base predicates. An atom with a defined predicate is called a *defined atom*.

Let us start off by formalizing the process of unfolding a clause as a tree of clauses. That tree will be called *Unfolding-tree* (or U-tree for short). Since an unfolding step depends on the choice of the atom in the clause to be unfolded, the formalization of the unfolding process also depends on the choice of a selection function, called *Unfolding-selection rule*, or U-selection rule for short. (Thus, the concepts of U-tree and U-selection rule are analogous to those of SLD-tree and computation rule [Lloyd 87].)

DEFINITION 1. (*U-tree.*) Let Prog be a program, C a clause in Prog, and S a U-selection rule. A *U-tree* for <Prog,C> via S is a tree labelled by clauses and constructed as follows:

i) the root is labelled by the clause C, and

ii) let M be a node labelled by a clause of the form: $H \leftarrow A_1,\ldots, A_h,\ldots, A_k$, where A_h is the defined atom selected by the U-Selection rule S. For each clause $A \leftarrow B_1,\ldots, B_s$ 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:

$(H \leftarrow A_1,\ldots, A_{h-1}, B_1,\ldots, B_s, A_{h+1},\ldots, A_k)\,\sigma$.

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. (*Upper portion of a tree.*) Given a tree T we say that a subtree T1 of T is an *upper portion* of T iff i) if a node N is in T1 then every ancestor of N in T is also in T1, and ii) if a node N is in T1 then every brother of N in T is in T1 as well. ■

One can easily show that the least Herbrand model of a program Prog is equal to the least Herbrand model of $(Prog-\{C\})\cup L$, where C is a clause of Prog 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. The unfolding process for the Maximal-Up-Segments program given in Example 1 can be represented by a U-tree whose finite upper portion is depicted in Figure 1 below (The underlined atoms are the ones selected for unfolding). ❑

DEFINITION 3. (*Foldable clause in a U-tree.*) Let Prog be a program, C a clause in Prog, 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 the set of defined atoms in the body of D is an instance of the set of 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. (*Foldable U-tree.*) Let Prog be a program, C a clause in Prog, 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 which satisfies the following condition: for each leaf-clause L such that: i) its body has at least one defined atom, and ii) each defined atom of its body unifies at least one head in Prog, we have that L is a foldable clause.

That portion will be called *foldable upper portion* of the U-tree. ■

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 <Maximal-Up-Segments program, clause 1> is depicted in Figure 1.

Now we can formalize the *Foldability Problem* as follows: Given a program Prog and a clause C in Prog, is there a U-selection rule S and a foldable U-tree for <Prog,C> via S?

THEOREM 5. The Foldability Problem is partially solvable and it is not solvable.

Proof. (Sketch) Any Turing Machine M and any word w can be encoded into a logic program ProgM and clause C_w, respectively, such that there is a foldable U-tree for $<ProgM\cup\{C_w\},C_w>$ if and only if either M halts or M enters into an infinite cycle when starting on the input word w. Thus the Halting Problem for

Turing Machines can be reduced to the Foldability Problem. (See [Proietti-Pettorossi 89] for a complete proof.) ■

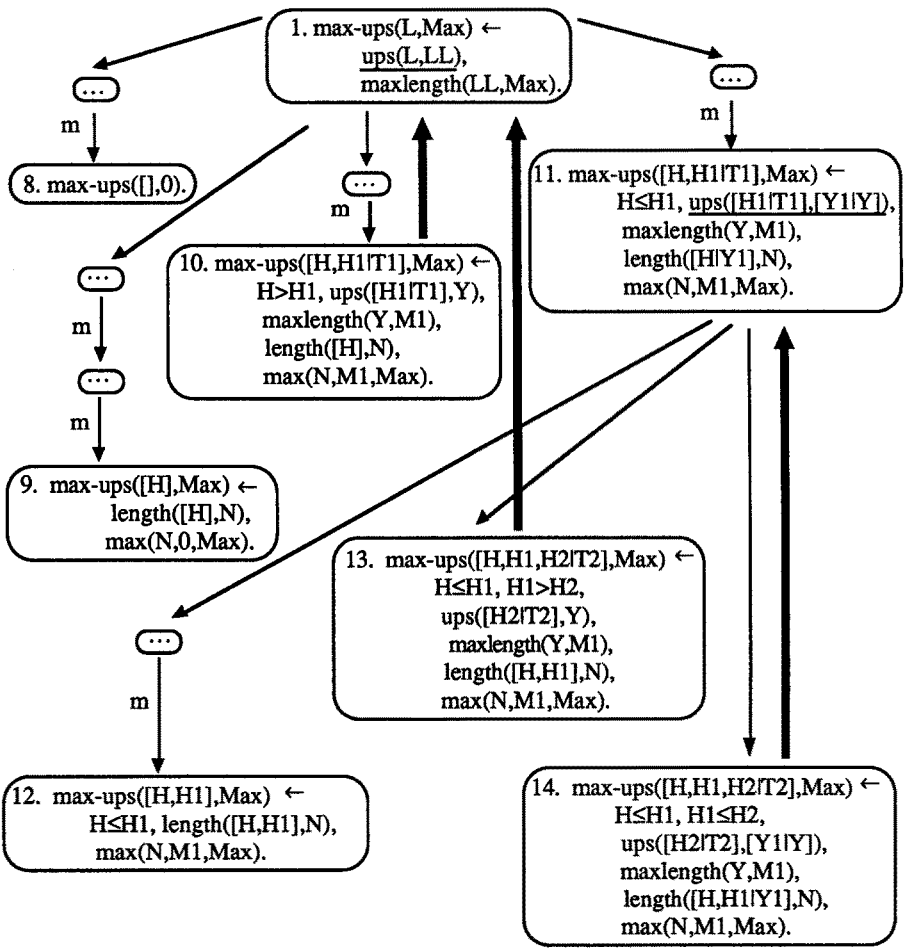
We will now show that the Foldability Problem is unsolvable also for some restricted classes of programs.

DEFINITION 6. (*Linear recursive programs.*) A program is said to be linear recursive iff there is at most one defined predicate in the body of each clause. ■

Example 3. The program {p ← p,q.} is linear, while the program {p ← p,q. q ← q.} is not. The max-ups program of Example 1 is not linear because two defined predicates occur in the body of clause 1. ❑

THEOREM 7. The Foldability Problem for linear recursive programs is not solvable.

Proof. The program ProgM∪{C_w} mentioned in the proof of Theorem 5 is a linear recursive program without base predicates. ■



8, 9, and 12 have no defined atoms.
10, 13 and 14 are foldable.
m means 'unfolding maxlength'.
Bold up-arrows denote <ancestor-clause, foldable clause> pairs.

Figure 1. The minimal foldable upper portion of a U-tree for the Maximal-Up-Segments program.

DEFINITION 8. (*Linear terms and linear atoms.*) A term or an atom is linear iff each variable occurs in it at most once. ■

DEFINITION 9. (*Linear-Term clauses and programs.*) A clause C of a program Prog is said to be a linear-term clause (or LT-clause) iff each defined atom of C is linear. A program Prog is a linear-term program (or LT-program) iff it is made of LT-clauses only. ■

Example 4. The following clause of the Maximal-Up-Segments program (see Example 1) is a LT-clause:

maxlength([H|T],M) ← maxlength(T,M1), length(H,N), max(N,M1,M).

The following clause of the Maximal-Up-Segments program is *not* a LT-clause:

ups([H,H1|T1],[[H]|Y]) ← H>H1, ups([H1|T1],Y).

because the variable H occurs twice in the head. However, it can be transformed into a LT-clause introducing some equalities (to be considered as base predicates) as follows:

ups([H,H1|T1],[[K]|Y]) ← H=K, H>H1, ups([H1|T1],Y). ❑

Notice that the linearity notions introduced in Definitions 6 and 9 are completely unrelated. The program {p(a). p(t(X,X)) ← p(X).} is linear recursive but not LT, while the program {p(a). p(t(X,Y)) ← p(X), p(Y).} is LT but it is not linear recursive.

THEOREM 10. The Foldability Problem for linear recursive LT-programs is not solvable.

Proof. The program Prog$M \cup \{C_w\}$ mentioned in the proof of Theorem 5 is a linear recursive LT-program without base predicates. ■

3. SOLVING THE FOLDABILITY PROBLEM FOR NON-ASCENDING PROGRAMS

In this Section we introduce the class of non-ascending programs, and for the programs in that class we will show that we can produce foldable U-trees, if we allow ourselves to apply some generalization steps (they will be formally defined later). Those foldable U-trees will be used for synthesizing suitable eureka predicates and their recursive definitions. We will also present particular subclasses of programs in which the generalization steps are *not* necessary, thus simplifying the program derivation process.

DEFINITION 11. 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:

- depth(X,X) = 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 var(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 $var(t) \cap var(u)$ we have: $depth(X,t) \leq depth(X,u)$.

The above definitions of depth, maxdepth, and var, and of the ≤ relation for terms are extended from terms to atoms in the obvious way, by considering the predicate symbols as term constructors. ■

Notice that the ≤ relation of terms is *not* a partial order. Indeed, it does not satisfy the antisymmetric and transitive properties.

DEFINITION 12. (*Non-Ascending programs.*) Let Prog be a LT-program, and C a clause in 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.

The clause C is said to be non-ascending iff $A_i \leq H$ for i=1,...,m.

The program Prog is said to be non-ascending iff all its clauses are non-ascending. ■

DEFINITION 13. A *Synchronized Descent Rule* (called simply SDR) for a LT-program Prog is a partial function defined as follows. Let D: H ← Body be a 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. ■

Often when referring to a SDR, we will not specify the corresponding clause D and the program Prog, if 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 uniquely determined. For simplicity reasons, unless otherwise specified, we may refer to 'the' SDR instead of 'a' SDR, when the statement at hand is valid for any choice of SDR.

However, different choices may determine the derivation of different programs with different behaviours. The problem of anticipating which of the several choices is more advantageous is outside the scope of this paper and we leave it for future research.

We give now the following two lemmas whose easy proofs can be found in [Proietti-Pettorossi 89].

LEMMA 14. By resolving two clauses of a non-ascending program we get a non-ascending clause. ■

LEMMA 15. Let us consider a non-ascending program and two of its clauses of the form: $H \leftarrow A_1,\ldots,A_m,\ldots,A_p$ and $K \leftarrow B_1,\ldots,B_n,\ldots,B_q$, where $A_1,\ldots,A_m$ and $B_1,\ldots,B_n$ only are the defined atoms. Let us also consider a defined atom A_s in $\{A_1,\ldots,A_m\}$. Suppose that $A_r \leq A_s$ for r=1,...,m and assume that there exists the most general unifier σ of A_s and K. We have:

- $maxdepth(A_r\sigma) \leq \max\{maxdepth(A_s), maxdepth(K), maxdepth(A_r)\}$ for r=1,...,m,
- $maxdepth(B_r\sigma) \leq \max\{maxdepth(A_s), maxdepth(K), maxdepth(B_r)\}$ for r=1,...,n. ■

THEOREM 16. Let Prog be a non-ascending program, C a clause in Prog, and S a SDR for Prog. For each defined atom G in the body of each clause of the U-tree for <Prog, C> via S, we have:

$$maxdepth(G) \leq \max\{maxdepth(A) \mid A \text{ is a defined atom in Prog}\}. \quad (*)$$

Proof. By induction on the construction of the U-tree using Lemmas 14 and 15. ■

DEFINITION 17. Given a program Prog and a clause C, we denote by G(Prog,C) the context-free grammar with start symbol S and productions Prods defined as the least set such that (we use upper case letters for the non-terminal symbols and lower case letters for the terminal ones):

- if C has the form: $p(\ldots) \leftarrow p_1(\ldots),\ldots,p_m(\ldots),A_1,\ldots,A_n$, where $p_1,\ldots,p_m$ are the defined predicates (w.r.t. Prog), the production: $S \rightarrow P_1\ldots P_m$ is in Prods,
- for each clause of Prog of the form: $q(\ldots) \leftarrow q_1(\ldots),\ldots,q_r(\ldots),B_1,\ldots,B_s$, where $q_1,\ldots,q_r$ ($r \geq 1$) are the defined predicates, the production: $Q \rightarrow Q_1\ldots Q_r$ is in Prods,
- for each defined predicate d occurring in the body of some clause, the production: $D \rightarrow d$ is in Prods. ■

THEOREM 18. Let Prog be a non-ascending program and C a clause in Prog such that the language generated by G(Prog,C) is finite. Let S be a SDR for Prog.

The U-tree T for <Prog, C> via S is foldable if S is defined for all clauses in T with at least a defined atom in their bodies.

Proof. Since S is (always) defined, the set of maximal paths from the root of T to its leaves can be partitioned into: i) paths with leaf-clause whose body does not contain a defined atom, ii) paths with leaf-clause where the atom selected by S cannot be unified with any head in Prog, and iii) infinite paths. For case iii), by Theorem 16 and the hypothesis on G(Prog,C) in any infinite path there is at least one clause which is foldable. Thus, a foldable upper portion of a U-tree, as required by Definition 4, can be constructed by taking paths of type i) and type ii), and minimal initial segments of infinite paths, ending in a foldable clause. ■

We would like now to revisit the Maximal-Up-Segments Example and show that the technique suggested by the above Theorem 18 can automatically derive for us a foldable U-tree.

Example 5. Maximal-Up-Segments revisited.
The grammar G(Max-ups, clause 1) constructed as indicated by Definition 17 consists of the following set of productions: {S → Ups Maxlength, Ups → Ups, Ups → ups, Maxlength → Maxlength, Maxlength → maxlength}. Therefore the language accepted by G(Max-ups, clause 1) consists of the word 'ups maxlength' only (and thus, it is finite).

In order to use the above Theorem 18, which ensure the existence of a foldable U-tree, we first transform the program for max-ups into a non-ascending one, by using the equality predicates as indicated in Example 4. We then construct the U-tree using a SDR for that program and clause 1. The reader may verify that SDR turns out to be always defined and thus it we get a foldable upper portion of that U-tree. That portion is exactly the one depicted in Figure 1, except for the base atoms in the clauses. In particular, the following clauses should replace the ones in Figure 1 with the same first numbers. (We also have listed here clauses 1 and 8 for the reader's convenience.)

1. max-ups(L,Max) ← ups(L,LL), maxlength(LL,Max).
8. max-ups([],0).
9.2 max-ups([H],Max) ← H=K, length([K],N), max(N,0,Max).
10.3 max-ups([H,H1|T1],Max) ← H=K, H>H1, ups([H1|T1],Y), maxlength(Y,M1), length([K],N), max(N,M1,Max).
11.2 max-ups([H,H1|T1],Max) ← H=K, H≤H1, ups([H1|T1],[Y1|Y]), maxlength(Y,M1), length([K|Y1],N), max(N,M1,Max).
12.1 max-ups([H,H1],Max) ← H=K, H≤H1, H1=K1, length([K,K1],N), max(N,M1,Max).
13.1 max-ups([H,H1,H2|T2],Max) ← H=K, H≤H1, H1=K1, H1>H2, ups([H2|T2],Y), maxlength(Y,M1), length([K,K1],N), max(N,M1,Max).
14.1 max-ups([H,H1,H2|T2],Max) ← H=K, H≤H1, H1=K1, H1≤H2, ups([H2|T2],[Y1|Y]), maxlength(Y,M1), length([K,K1|Y1],N), max(N,M1,Max). □

As already mentioned, we now state a few more facts which ensure the existence of foldable U-trees for various classes of programs, whereby allowing in those cases the automatic derivation of the auxiliary eureka predicates which are necessary for synthesizing efficient programs, and guaranteeing also the success of our transformation methodology.

From the above Theorem 18 we can easily get the following Theorem.
THEOREM 19. For each linear recursive non-ascending program Prog and for each clause C in Prog there exists a U-selection rule S and a foldable U-tree for <Prog,C> via S.
Proof. In the body of each clause of the U-tree the set of defined atoms is either empty or a singleton. If it is not empty we consider the selection rule which chooses the only defined atom occurring in the body. That rule is a SDR (because ≤ is reflexive), and it is defined for all clauses with at least a defined atom in their bodies. The thesis follows from the fact that G(Prog,C) generates a finite language. ■

Now we will prove that foldable U-trees exist for two more classes of programs. The subterm relation is assumed to be reflexive.
THEOREM 20. Let Prog be a linear recursive non-ascending program and C be a non-ascending clause (*not* in Prog) of the form: $h(\ldots) \leftarrow p_1(\ldots), p_2(\ldots), A_1, \ldots, A_n$, where the predicates p_1 and p_2 are the only defined predicates (w.r.t. Prog), and h does not occur in Prog. Assume that:
i) for each clause D in Prog of the form: $p(\ldots) \leftarrow \ldots, q(\ldots), \ldots$, where q is the only defined predicate in the body of D, we have that:
- for each argument r and s of q and p, respectively, if r has a variable in common with s then r is a subterm of s,
- distinct arguments of q 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, or $var(t_1) \cap var(t_2) = \emptyset$, and

ii.2) $var(p_1(\ldots)) \cap var(p_2(\ldots) = var(t_1) \cap var(t_2)$.

Then there is a foldable U-tree for $<Prog \cup \{C\}, C>$.

Proof. First notice that the language accepted by $G(Prog \cup \{C\}, C)$ is finite. Indeed, it is made out of a production of the form: $S \rightarrow P_1\ P_2$ (corresponding to C), together with productions of the form $P \rightarrow Q$ (corresponding to the clauses in Prog with exactly one defined atom in their bodies), and productions of the form $P \rightarrow p$ (corresponding to the defined predicates of Prog).

Moreover, condition ii) implies that a SDR is defined for C. Indeed, by the hypothesis that C is a non-ascending clause we have that t_1 and t_2 are linear terms. Therefore, if t_1 is a subterm of t_2 then $t_1 \leq t_2$, and, by hypothesis ii.2, we have that $p_1 \leq p_2$. The case when t_2 is a subterm of t_1 is analogous, and the case when $var(t_1) \cap var(t_2) = \emptyset$ is trivial. We will now show that condition ii) is preserved by unfoldings. Thus SDR is defined for all clauses with at least a defined atom in their bodies, and the thesis follows from Theorem 18.

Let us consider a clause E: $h(\ldots) \leftarrow f(\ldots,i,\ldots), g(\ldots,j,\ldots), \ldots$, which satisfies hypothesis ii) replacing f, i, g, and j by p_1, t_1, p_2, and t_2, respectively. Let $f(\ldots,i,\ldots)$ be the atom selected for unfolding. (The case for $g(\ldots,j,\ldots)$ is analogous.) Let $f(\ldots,s,\ldots) \leftarrow \ldots, q(\ldots,u,\ldots), \ldots$ be a clause, call it D, in Prog where s is the argument which occurs in the same position of i, and u is the argument of q with at least one variable in common with s, if any, otherwise u is *any* argument of q. Notice that by hypothesis i) at most one argument of q may have common variables with s.

By unfolding E using D we get the following clause E1: $(h(\ldots) \leftarrow q(\ldots,u,\ldots), g(\ldots,j,\ldots), \ldots)\sigma$, where σ is the computed most general unifier. Since $f(\ldots,i,\ldots)$ and $f(\ldots,s,\ldots)$ are linear atoms, and as usual, we assume that D does not have common variables with E, the most general unifier σ can be partitioned into four sets of bindings S_1, S_2, S_3, and S_4 such that:

- S_1 consists of bindings of the form V/a, where $V \in var(i)$ and a is a subterm of s,
- S_2 consists of bindings of the form W/b, where $W \in var(f(\ldots,i,\ldots))$-var(i) and b is a subterm of an argument of $f(\ldots,s,\ldots)$ different from s (therefore var(b) is a subset of $var(f(\ldots,s,\ldots))$-var(s)), and symmetrically:
- S_3 consists of bindings of the form X/c, where $X \in var(s)$ and c is a subterm of i,
- S_4 consists of bindings of the form Y/d, where $Y \in var(f(\ldots,s,\ldots))$-var(s) and d is a subterm of an argument of $f(\ldots,i,\ldots)$ different from i (therefore var(d) is a subset of $var(f(\ldots,i,\ldots))$-var(i)).

By hypothesis ii) and by the non-ascending property, we have that in clause E for each argument x and y of f and g different from i and j, respectively, $var(x) \cap var(y) = \emptyset$.

If $var(i) \cap var(j) = \emptyset$, by hypothesis ii.2) we have that $var(f(\ldots,i,\ldots)) \cap var(g(\ldots,j,\ldots)) = \emptyset$, and therefore $var(q(\ldots,u,\ldots)\sigma) \cap var(g(\ldots,j,\ldots)\sigma) = \emptyset$. Hence condition ii) is preserved.

Let us now consider the case when $var(i) \cap var(j) \neq \emptyset$, and j is a subterm of i. (The case when i is a subterm of j is analogous.)

If $var(u) \cap var(s) \neq \emptyset$, u is a subterm of s, by hypothesis i). Therefore $u\sigma$ and $j\sigma$ are subterms of the linear term $i\sigma$ and, thus, condition ii.1) holds for $u\sigma$ and $j\sigma$. Otherwise, if $var(u) \cap var(s) = \emptyset$, the bindings that may apply to u are the ones in S_4 only, and therefore the set of the variables of $u\sigma$ is a subset of those occurring either in an argument of $f(\ldots,i,\ldots)$ different from i or in u. On the other hand, the bindings that may apply to j are the ones in S_1 only, and therefore $var(j\sigma)$ is a subset of $var(j) \cup var(s)$. Hence we have: $var(u\sigma) \cap var(j\sigma) = \emptyset$. Thus, condition ii.1) is preserved by unfoldings.

Notice now that, by hypothesis ii) each variable X occurring in the arguments of $g(\ldots)$ in E different from j occurs neither in $f(\ldots,i,\ldots)$ nor in D. Therefore, X does not occur in $q(\ldots)\sigma$ and $X\sigma = X$. Thus, $var(q(\ldots)\sigma) \cap var(g(\ldots)\sigma)$ is a subset of $var(j\sigma)$.

On the other hand, by hypothesis i), each variable X occurring in the arguments of $q(\ldots)$ in D different from u does not occur in s. Therefore the only bindings that may apply to X are the ones in S_4 and the

variables occurring in the arguments of $q(\ldots)\sigma$ different from $u\sigma$ occur either in D or in an argument of $f(\ldots,i,\ldots)$ different from i. Thus, they do not occur in $g(\ldots)\sigma$ and therefore $var(q(\ldots)\sigma)\cap var(g(\ldots)\sigma)$ is a subset of $var(u\sigma)$.
Therefore $var(q(\ldots)\sigma)\cap var(g(\ldots)\sigma)$ is a subset of $var(u\sigma)\cap var(j\sigma)$. The inverse inclusion is obvious. Thus, also condition ii.2) is preserved by unfoldings. This completes the proof. ■

REMARK. The class of programs which satisfy the hypotheses of the above theorem extends the one of unilinear programs presented in [Pettorossi-Proietti 89]. ❑

THEOREM 21. Let Prog be a linear recursive non-ascending program and C be a non-ascending clause (*not* in Prog) of the form: $h(\ldots) \leftarrow p_1(\ldots), p_2(\ldots),A_1,\ldots,A_n$, where the predicates p_1 and p_2 are the only defined predicates (w.r.t. Prog), and h does not occur in Prog. Assume that:
i) for each clause D in Prog of the form: $p(\ldots) \leftarrow \ldots,q(\ldots),\ldots$ where q is the (only) defined predicate in the body of D, we have that:
- for each argument s of $p(\ldots)$ and for each argument r of $q(\ldots)$ $|var(s)\cap var(r)| \leq 1$,
- distinct arguments of q do not have common variables with the same argument of p, and
ii) there exist two arguments t_1 and t_2 of $p_1(\ldots)$ and $p_2(\ldots)$, respectively, such that:
ii.1) $|var(t_1)\cap var(t_2)| \leq 1$ (where $|\ldots|$ denotes cardinality), and
ii.2) $var(p_1(\ldots))\cap var(p_2(\ldots) = var(t_1)\cap var(t_2)$.
Then there is a foldable U-tree for <Prog$\cup${C},C>.
Proof. It is easy to see that the language accepted by G(Prog$\cup${C},C) is finite and that condition ii) is preserved by unfoldings. Therefore a SDR is defined for all clauses in any U-tree for <Prog$\cup${C},C> with at least one defined atom in their bodies, and the thesis follows from Theorem 18. ■
Notice that the above theorem is a consequence of Theorem 20 if we enforce condition i) by requiring that the arguments of $q(\ldots)$ are variables only.

In the following Example 6 we will apply the results of Theorem 20 and 21, which ensure the existence of a foldable upper portion of a U-tree via SDR, and we will construct that tree.
Example 6. Common-Sublists.
The following program Comsub tests whether or not a list X is a sublist of both Y and Z.
1. comsub(X,Y,Z) ← sub(X,Y), sub(X,Z).
2. sub([],X).
3. sub([A|X],[A|Y]) ← sub(X,Y).
4. sub(X,[A|Y]) ← sub(X,Y).

where sub(X,Y) holds iff X is a sublist of Y. The order of the elements should be preserved, but the elements selected by X need not to be consecutive in Y. For instance, [b,d] is a sublist of [a,b,c,d], but [d,b] is not.

Unfortunately, the above program does *not* satisfy the non-ascending hypothesis of Theorems 20 and 21, because the variable A occurs twice in the head of clause 3. We can transform it into a non-ascending program by introducing equalities. Clauses 1, 2, and 4 remain unchanged, while clause 3 becomes:
3.1 sub([A|X],[A1|Y]) ← A=A1, sub(X,Y).

Now we can apply Theorem 20 (or 21) because: i) in clauses 3 and 4 each argument of sub in the bodies is a subterm of the corresponding one in the heads (indeed, it is a single variable), ii.1) in clause 1 sub(X,Y) and sub(X,Z) share the variable X only, and ii.2) $var(sub(X,Y))\cap var(sub(X,Z)) = X = var(X) \cap var(X)$. Thus, we are ensured of the existence of a foldable U-tree for <Comsub, clause 1> via a SDR. The minimal foldable upper portion of that U-tree is depicted in the Figure 2 below. (The underlined atoms are the ones selected for unfolding.) In the following Section we will consider again this example and we will see that by applying the Loop Absorption Procedure, we can derive the suitable eureka predicates and

the improved program version. ❑

Unfortunately, Theorem 18 stated above does not ensure us that we can construct a foldable U-tree in the case when, at some point during the unfolding process, we get a clause for which no SDR is defined. For overcoming that difficulty we will now introduce the so called *Generalization Strategy* (see [Boyer-Moore 75] for a similar strategy used in functional programs). However, in order to preserve correctness, we need to use it together with its 'inverse', which is the *Equality Introduction*. This technique will allow us to get foldable U-trees in all cases (but unfortunately, we cannot be sure that the derived programs always have better performances).

DEFINITION 22. (*Generalization + Equality Introduction Rule.*) 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: $H \leftarrow Gen A_1,\ldots, Gen A_n,\ X_1=t_1,\ldots,\ X_r=t_r$, where: $(Gen A_1,\ldots, Gen A_n)\,\theta = (A_1,\ldots, A_n)$ and $\theta = \{X_1/t_1,\ldots, X_r/t_r\}$. ■

Obviously, when applying the Generalization + Equality Introduction 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 (otherwise the generalization step will be without effect). The notion of foldability is not changed.

By performing suitable generalization steps, it is possible to transform any clause into an equivalent one for which a SDR is defined. Indeed, we can use generalizations so that no pair of defined atoms in the body of a clause has common variables and in that case an SDR is trivially defined. Notice that for doing this, we can restrict ourselves to consider a particular kind of generalization step, whose inverse instantiation is a substitution $\theta = \{X_1/t_1,\ldots, X_r/t_r\}$, where all terms $t_1,\ldots, t_r$ are variables. Thus, by Theorem 18 we have the following fact.

THEOREM 23. Given any non-ascending program Prog and clause C in Prog such that the language generated by G(Prog,C) is finite, there exists a foldable U-tree for <Prog, C>, and it can be constructed using unfolding steps and 'Generalization + Equality Introduction' steps. ■

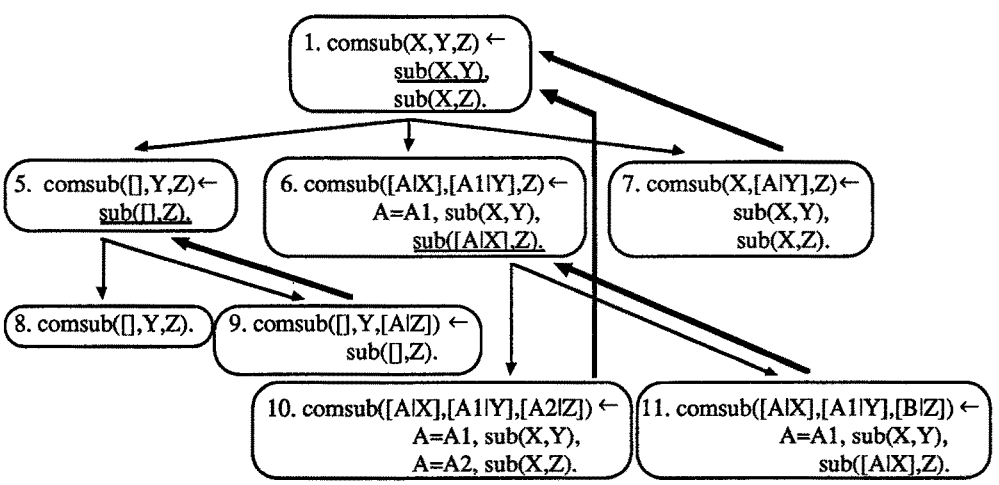


Figure 2. The minimal foldable upper portion of a U-tree for the Comsub program.

Of course, it is not very useful to construct a foldable U-tree at the expense of removing all shared variables among atoms, because if no shared variables occur in the bodies of the foldable clauses of the U-tree, the program we derive does not avoid redundant computations of bindings. Therefore we should use generalization steps with parsimony, so that we may maintain some shared variables among the atoms of the generalized clause. This is always possible because the $\leq$ relation (see Definition 11) is defined between to atoms with exactly one common variable.

Let us assume, for instance, that a SDR is not defined because a 'cycling' situation among variables occurs, as indicated by the following clause:

h(X,Y) ← p(t(X,r(Y))), q(t(Y,r(X))).

The depth of X in p is smaller than the depth of X in q and viceversa for Y. In order to break that cycle we introduce by generalization a new variable, and we get:

h(X,Y) ← X=X1, p(t(X1,r(Y))), q(t(Y,r(X))).

Now for that resulting clause SDR is defined and selects p.

Another simple strategy to avoid unnecessary generalization steps in the U-tree for <Prog,C> is the following one: before generalizing, one may try to unfold one of the defined atoms of the clause where SDR is not defined. If that unfolding step does not produce any atom in the body of a clause with maxdepth greater than the maximal maxdepth of a defined atom in Prog, then we can avoid a generalization step. (In that case, in fact, each atom G obtained after that unfolding step satisfies the property (*) in the statement of Theorem 16, which is used for proving Theorem 18.)

Let us now present an example of the construction of a foldable U-tree and the corresponding improved program version by using the Generalization Strategy.

Example 7. Prefix-Suffix of a list.

A list P is a prefix of a list L if P is either empty or it is an initial segment of L. A list S is a suffix of L if S is either empty or it is a final segment of L. Suppose we want to check whether or not a list PS is both a prefix and a suffix of the list L. We define a relation presuf by means of the following program Presuf:

1. presuf(PS,L) ← prefix(PS,L), suffix(PS,L).
2. prefix([],L).
3. prefix([X|P],[X|L]) ← prefix(P,L).
4. suffix(L,L).
5. suffix(S,[X|L]) ← suffix(S,L).

We would like to avoid the double visit of both the list PS and the list L, while the test of presuf is performed. That objective may be achieved by finding a foldable U-tree for <program Presuf, clause 1>. The existence of that U-tree is not guaranteed by our results because Presuf does *not* satisfy the non-ascending property. Indeed, the variable X occurs twice in the head of clause 3 and the variable L occurs twice in the head of clause 4. However, we can easily obtain a semantically equivalent program, called Presuf1, which is non-ascending, by using equalities for transforming clauses 3 and 4 into the following clauses 3.1 and 4.1:

3.1 prefix([X|P],[X1|L]) ← X=X1, prefix(P,L).

4.1 suffix(L,L1) ← L=L1.

We can now apply the SDR rule for constructing a foldable U-tree. The unfolding process via SDR generates a U-tree which has a path (starting from the root-clause 1) made out of the following clauses:

1. presuf(PS,L) ← prefix(PS,L), suffix(PS,L).
6. presuf([X|Xs],[X1|Ys]) ← X=X1, prefix(Xs,Ys), suffix([X|Xs],[X1|Ys]).
7. presuf([X|Xs],[X1|Ys]) ← X=X1, prefix(Xs,Ys), suffix([X|Xs],Ys).
8. presuf([X|Xs],[X1,X2|Ys]) ← X=X1, prefix(Xs,[X2|Ys]), suffix([X|Xs],Ys).

(We get 6 from 1 by unfolding prefix, 7 from 6 and 8 from 7 by unfolding suffix).

At this point no SDR is defined because of the cycling situation between the arguments of prefix(Xs,[X2|Ys]) and suffix([X|Xs],Ys); in fact, we have:

depth(Xs, prefix(Xs,[X2|Ys])) ≤ depth(Xs, suffix([X|Xs],Ys)), and

depth(Ys, prefix(Xs,[X2|Ys])) ≥ depth(Ys, suffix([X|Xs],Ys)).

Moreover, if we unfold either prefix or suffix in clause 8 we cause the maxdepth of the other atom to grow bigger than max{ maxdepth(A) | A is a defined atom in Presuf1 } = 2.

In fact, we get either:

presuf([X,X3|Xs],[X1,X2|Ys]) ← X=X1, X3=X2, prefix(Xs,Ys), suffix([X,X3|Xs],Ys)

where maxdepth(suffix([X,X3|Xs],Ys))=3, or:

presuf([X|Xs],[X1,X2,Y|Ys]) ← X=X1, prefix(Xs,[X2,Y|Ys]), suffix([X|Xs],Ys)

where maxdepth(prefix(Xs,[X2,Y|Ys]))=3.

Thus, in order to construct a foldable U-tree we need to apply a generalization step. We generalize the variable Xs in the atom suffix([X|Xs],Ys) in clause 8 and we obtain:

8.1 presuf([X|Xs],[X1,X2|Ys]) ← X=X1, prefix(Xs,[X2|Ys]), Xs=Z, suffix([X|Z],Ys).

For clause 8.1 SDR is defined and it selects prefix. By continuing the construction of the U-tree from clause 8.1 we get a path made out of the following clauses:

9. presuf([X,X3|Xs],[X1,X2|Ys]) ← X=X1,X3=X2, prefix(Xs,Ys), [X3|Xs]=Z, suffix([X|Z],Ys).

10. presuf([X,X3,X4|Xs],[X1,X2,X5|Ys]) ← X=X1, X3=X2, X4=X5, prefix(Xs,Ys), [X3,X4|Xs]=Z, suffix([X|Z],[X5|Ys]).

11. presuf([X,X3,X4|Xs],[X1,X2,X5|Ys]) ← X=X1, X3=X2, X4=X5, prefix(Xs,Ys), [X3,X4|Xs]=Z, suffix([X|Z],Ys).

(We get 9 from 8.1 and 10 from 9 by unfolding prefix, and 11 from 10 by unfolding suffix).

Now clause 11 is foldable because the defined atoms 'prefix(Xs,Ys), suffix([X|Z],Ys)' are an instance (actually, a renaming) of the corresponding ones in the ancestor-clause 9. For lack of space we do not show here the construction of the complete minimal foldable upper portion of a U-tree for <Presuf1, clause 1>. (It is enough to use the SDR and no more generalization steps will be required.)

Notice that, a different generalization step was possible for clause 8. Indeed, we could derive the following clause:

presuf([X|Xs],[X1,X2|Ys]) ← X=X1, prefix(Xs,[X2|Z]), Z=Ys, suffix([X|Xs],Ys)

where we have generalized the occurrence of the variable Ys instead of Xs. By continuing the derivation process we get a different foldable U-tree and (by applying the Loop Absorption Procedure as indicated in the next Section) a different final program. We do not present any method for anticipating which of the two choices is more advantageous. □

4. SYNTHESIS OF EUREKA PREDICATES AND DERIVATION OF IMPROVED PROGRAMS

The theoretical results presented in the previous Section can be used as a formal justification of the procedure for the synthesis of the eureka predicates indicated in Steps 1-3 below. The success of that technique is guaranteed within the class of non-ascending programs specified by Theorems 19, 20, and 21. If we perform some generalization steps for making the SDR always defined, the success is guaranteed also for the larger class specified by Theorem 23.

Loop Absorption Procedure.

Step 1. *Synthesis of eureka predicates: application of the Loop Absorption Strategy.*

Let us consider a program Prog and a clause C in Prog. For each foldable leaf-clause L of the minimal foldable upper portion T of the foldable U-tree for <Prog, C>: i) we consider the corresponding ancestor-clause A, and ii) we introduce a new eureka predicate newp by a clause N whose body

consists of the set of defined atoms in A. The set of variables of newp will be the minimal set which allows to fold both A and L, using N as a bridge clause.

Step 2. *Derivation of the new program version.*

(a) For each clause N introduced at Step 1, relative to the leaf-clause L and the ancestor-clause A, we perform all unfolding and generalization steps corresponding to the subtree of T rooted in A, with the following exception: if during the unfolding and generalization process we have derived a clause, whose corresponding clause in T is *either* a foldable clause *or* the ancestor-clause of a foldable one, then we stop the unfolding and generalization process and we fold that clause by using as bridge clause one which defines a eureka predicate introduced at Step 1. This unfolding-generalization-folding process gives the explicit definition of the predicate defined by the clause N.

(b) We perform the actions described in the above Step 2(a) also for the clause C. (The subtree of T to be considered is T itself, which is rooted in C.)

Step 3. *Simplification.*

We simplify the base predicates, whenever possible. We also eliminate clauses which are subsumed by other clauses and clauses whose bodies contain defined atoms which do not match any head. ■

REMARK. We know that if during Step 1 a eureka predicate, say p, turns out to be defined exactly as one of the predicates already present in our program, we may avoid the introduction of p. In that case:

- the folding step relative to a foldable clause which could have been performed at Step 2(a) and Step 2(b) using p, will be performed using instead the corresponding predicate of our program,
- the folding step relative to the ancestor-clause of a foldable clause which could have been performed at Step 2(a) and 2(b) using p, will *not* be performed.

On the contrary, if we choose to introduce the predicate p, nothing goes wrong: we get a final program with redundant clauses (which could be discarded by performing folding and unfolding steps). ❑

Example 8. Maximal-Up-Segments revisited.

The reader may easily verify that for the max-ups program, the technique described by the above steps 1-3 gives the same final program derived in Example 1. In particular, with reference to the clauses of Example 5 the pairs <ancestor-clause, foldable clause> are: <1, 10.3>, <1, 13.1>, and <11.2, 14.1>.

The defined atoms in the bodies of the ancestor-clauses 1 and 11.2 give us exactly the definitions of the eureka predicates which we introduced in Example 1 (see clauses 15 and 16). ❑

Example 9. Common-Sublists revisited.

Step 1. As the reader may verify from Figure 2, the <ancestor-clause, foldable clause> pairs are the following ones: <5,9>, <1,10>, <6,11>, and <1,7>, where:

1. comsub(X,Y,Z) ← sub(X,Y), sub(X,Z).
5. comsub([],Y,Z) ← sub([],Z).
6. comsub([A|X],[A1|Y],Z) ← A=A1, sub(X,Y), sub([A|X],Z).
7. comsub(X,[B|Y],Z) ← sub(X,Y), sub(X,Z).
9. comsub([],Y,[B|Z]) ← sub([],Z).
10. comsub([A|X],[A1|Y],[A2|Z]) ← A=A1, sub(X,Y), A=A2, sub(X,Z).
11. comsub([A|X],[A1|Y],[B|Z]) ← A=A1, sub(X,Y), sub([A|X],Z).

The eureka predicates we introduce by Loop Absorption Strategy are:

newcomsub(A,X,Y,Z) ← sub(X,Y), sub([A|X],Z) due to the pair <6,11>, and

newsub(Z) ← sub([A|X],Z) due to the pair <5,9>.

We do not introduce the eureka predicates relative to the other pairs, because they should have been defined exactly as the predicate comsub, which is already in the initial program version.

Step 2. By performing the unfolding and folding steps (no generalization step is necessary) which

correspond to the subtrees rooted in clause 1, 6 and 5 in the U-tree of Figure 2, we get the following recursive definitions of comsub, newcomsub, and newsub, respectively:

12. comsub([],Y,Z).
13. comsub([],Y,[A|Z]) ← newsub(Z).
14. comsub([A|X],[A1|Y],Z) ← A=A1, newcomsub(A,X,Y,Z).
15. comsub(X,[A|Y],Z) ← comsub(X,Y,Z).
16. newcomsub(A,X,Y,[A2|Z]) ← A=A2, comsub(X,Y,Z).
17. newcomsub(A,X,Y,[B|Z]) ← newcomsub(A,X,Y,Z).
18. newsub(Z).
19. newsub([B|Z]) ← newsub(Z).

Step 3. We simplify the equalities in clauses 14 and 16. Moreover, we can eliminate clause 13 which is subsumed by clause 12, and we also eliminate clauses 18 and 19 which become unnecessary. The final program is:

12. comsub([],Y,Z).
14.1 comsub([A|X],[A|Y],Z) ← newcomsub(A,X,Y,Z).
15. comsub(X,[A|Y],Z) ← comsub(X,Y,Z).
16.1 newcomsub(A,X,Y,[A|Z]) ← comsub(X,Y,Z).
17. newcomsub(A,X,Y,[B|Z]) ← newcomsub(A,X,Y,Z).

This program is equal to the one derived in [Tamaki-Sato 84], but through our methodology we have now made all derivation steps easily mechanizable and not based upon the programmer's intuition. □

Example 10. Prefix-Suffix of a list revisited.

Where we show only the derivation which is relative to the path of the foldable U-tree constructed in the above Example 7, leading from clause 1 to clause 11, via clauses 8, 8.1, and 9.

Step 1. From the path under consideration we derive the <ancestor-clause, foldable clause> pair <clause 9, clause 11>, which gives us the following eureka predicate definition:

12. newpresuf(Xs,Ys,X,Z) ← prefix(Xs,Ys), suffix([X|Z],Ys).

Step 2. By performing the unfolding, generalizaton, and folding steps which correspond to the subpath from clause 1 to clause 9, and to the subpath from clause 9 to clause 11, we get the following two clauses:

13. presuf([X,X3|Xs],[X1,X2|Ys]) ← X=X1, X3=X2, [X3|Xs]=Z, newpresuf(Xs,Ys,X,Z).
14. newpresuf([X1|Xs],[Y1|Ys],X,Z) ← X1=Y1, newpresuf(Xs,Ys,X,Z).

Step 3. After the simplification of the equalities, we get the following two clauses:

13.1 presuf([X,X2|Xs],[X,X2|Ys]) ← newpresuf(Xs,Ys,X,[X2|Xs]).
14.1 newpresuf([X1|Xs],[X1|Ys],X,Z) ← newpresuf(Xs,Ys,X,Z).

The reader may complete the derivation of the program for presuf(PS,L) by considering the other paths of the foldable upper portion of the U-tree for <Presuf1, clause 1>. The program which can be derived avoids the double visit of the list L, but because of the generalization step we have performed, it does not avoid the double visit of PS. □

5. FINAL DISCUSSION

We have presented a program transformation methodology and we have applied it to the derivation of some logic programs. We have also characterized our methodology by establishing some theorems which ensure the existence of foldable upper portions of U-trees.

If we have to apply the proposed techniques in the case of Prolog programs (or other programs written in any other logic language in which the order of the atoms and clauses is significant), it may be the case that

we have to perform some extra transformation steps for maintaining the efficiency improvements. Let us consider, in fact, the following situations.

i) 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 be increased, because some predicates could be evaluated before they are sufficiently instantiated. In those cases we can safely perform the necessary transformations, which interchange the relevant 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].

ii) The unfolding process may increase the number of generated clauses (see for instance, our Example 10) and therefore at run time each resolution step may need more time when searching for the unifying clause-head. That phenomenon may limit the improvement of efficiency determined by the use of the eureka predicates, but in that case 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 MacProlog optimizing compiler [Clark et al. 87]).

Now we present the results of some experiments we have performed in our C-Prolog implementation on a VAX 780 under VMS. In the tables below, we compare the time and space performances of the initial program versions with respect to the final ones obtained by introducing eureka predicates.

The Maximal-Up-Segments program.

Computing a solution of max-ups(L,M) where L is a ground list and M is a free variable.
Case (1): |L|=200, Case (2): |L|=400, where |L| denotes from now on the length of the list L.

Program:		Initial (1)	Final (1)	Initial (2)	Final (2)
Time (sec):		0.110	0.070	0.220	0.140
Space (bytes):	global stack	17828	9808	35672	19628
	local stack	28108	16224	55560	31916

The Common-Sublists program.

Test 1. The goal is: comsub(X,Y,Z) where X, Y and Z are ground lists, and X is a sublist of Y and Z.
Case (1): |X|=40, |Y|=310, |Z|=231. Case (2): |X|=1000, |Y|=1270, |Z|=1191.

Program:		Initial (1)	Final (1)	Initial (2)	Final (2)
Time (sec):		0.041	0.043	0.165	0.130
Space (bytes):	global stack	8744	8092	63380	47500
	local stack	17200	19812	81052	51828

Test 2. The goal is: comsub(X,Y,Z) where X, Y, Z are ground lists, and the list X is *not* a sublist of the list Y. |X|=11, |Y|=28, |Z|=28.

Program:		Initial	Final
Time (sec):		0.041	0.064
Space (bytes):	global stack	588	980
	local stack	1356	2644

Test 3. Computing all solutions of the goal: comsub(X,Y,Z) where X is free variable while Y and Z are ground lists. Case (1): |Y|=5, |Z|=5. Case (2): |Y|=10, |Z|=10.

Program:		Initial (1)	Final (1)	Initial (2)	Final (2)
Time (sec):		0.207	0.063	6.516	0.817
Space (bytes):	global stack	2388	804	25684	10844
	local stack	9340	3260	89260	36860

The Prefix-Suffix program.

Test 1. The goal is: presuf(X,Y) where both X and Y are ground lists. Case (1): |X|=330, |Y|=942. Case (2): |X|=990, |Y|=2262.

Program:		Initial (1)	Final (1)	Initial (2)	Final (2)
Time (sec):		0.075	0.065	0.228	0.203
Space (bytes):	global stack	19052	13772	48092	32252
	local stack	33040	24852	77920	51252

Test 2. Computing all solutions of the goal: presuf(X,Y) where X is a free variable and Y is a ground list. Case (1): |Y|=40, Case (2): |Y|=96.

Program:		Initial (1)	Final (1)	Initial (2)	Final (2)
Time (sec):		0.23	0.16	0.83	0.53
Space (bytes):	global stack	1780	1388	4244	3180
	local stack	5316	5316	11812	11812

As the reader may notice from the above tables, the efficiency improvements one may expect by the application of our transformation methodology are confirmed by the experimental results. When the evaluations of the predicates tupled together by the introduction of the eureka predicates share common subcomputations then the improvements are particularly substantial.

Moreover, it is often the case that the derived programs realize suitable *synchronizations* among the computations relative to various predicates, thus realizing the so-called *filter promotion* [Bird 84] and improving efficiency.

For instance, in the case of the Common-Sublists example, if using the initial program version we want to generate all common sublists of two lists, say Y and Z, we have first to construct a sublist of Y and then we have to test whether or not it is a sublist of Z.

Instead, if we use the derived program, when an element of a sublist of Y is generated we can check whether or not it occurs also in Z (in the suitable order). That synchronization of the computations allows us to prevent the generation of the entire sublist of Y before discovering, for instance, that it is not a sublist of Z, thus drastically reducing the search space.

Unfortunately, there are some cases in which the performances of the final program versions are *not* better than the initial ones. That phenomenon can be best illustrated by Test 2 of the Common-Sublists program. In that case we have that the goal comsub(X,Y,Z) cannot be satisfied because X is not a sublist of Y. Therefore no computation is necessary for verifying whether or not X is a sublist of Z. The initial program gives the negative result faster because sub(X,Y) immediately fails (indeed X is not a sublist of Y), while in the final program comsub(X,Y,Z) takes some extra time for verifying whether X is a sublist of Z (indeed, it calls the recursively defined predicate newcomsub(A,X,Y,Z)).

Finally, we want to address a point related to the fact that the derived programs may have a large number of clauses, especially for non-toy examples. This is actually the price one pays for applying automatic methods for the development of programs by transformation. For instance, in the Prefix-Suffix example a 'clever transformer' could have invented the new predicate genpresuf, defined by the clause:

genpresuf(PS1,PS2,L) ← prefix(PS1,L), suffix(PS2,L).

starting from the initial clause:

1. presuf(PS,L) ← prefix(PS,L), suffix(PS,L).

Then the derivation process would have been much simpler than the one we have presented using our techniques, but the derived program would have *not* been more efficient. However, we do not see how the invention of the above genpresuf predicate can be derived by easily mechanizable techniques like the ones we have presented here.

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.

Thanks to the anonymous referees, and also to Dr. M. Bruynooghe, Dr. N. Jones, Dr. A. Lakhotia, Dr. J. Maluszynski, and Dr. L. Sterling, who showed much interest in our work.

7. REFERENCES

[Azibi 87] Azibi, N.: "TREQUASI: Un système pour la transformation automatique de programmes PROLOG récursifs en quasi-itératifs, These, Université de Paris-Sud, Centre d'Orsay, 1987.

[Bird 84] Bird, R.S.:"The Promotion and Accumulation Strategies in Transformational Programming", Toplas ACM 6 (4) 1984, pp. 487-504.

[Boyer-Moore 75] Boyer, R.S. and Moore, J.S.: "Proving Theorems about LISP Functions", JACM 22 (1) 1975, pp. 129-144.

[Bossi et al. 88] Bossi, A., Cocco, N. and Dulli S.:"A Method for Specializing Logic Programs", Proc. 3rd Italian Conf. on Logic Programming, GULP 88, Roma, 1988, pp. 97-114.

[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.

[Bruynooghe et al. 89] Bruynooghe, M., De Raedt, L., and De Schreye D.: "Explanation Based Program Transformation", R. CW-89, Katholieke Universiteit Leuven, Belgium, 1989 (also in Proc IJCAI 89).

[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].

[Darlington 81] Darlington, J.: "An Experimental Program Transformation and Synthesis System", Artificial Intelligence 16, 1981, pp. 1-46.

[Debray 88] Debray, S.K.: "Unfold/Fold Transformations and Loop Optimization of Logic Programs", Proc. SIGPLAN 88 Conf. on Programming Language Design and Implementation, Atlanta, 1988.

[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.

[Deransart-Maluszynski 85] Deransart, P. and Maluszynski, J.: "Relating Logic Programs and Attribute Grammars" INRIA Report n.393, 1985.

[van Emden-Kowalski 76] van Emden, M.H. and Kowalski, R.: "The Semantics of Predicate Logic as a Programming Language", JACM, Vol. 23, No. 4, October 1976, pp. 733-742.

[Feather 86] Feather, M.S.: "A Survey and Classification of Some Program Transformation Techniques", Proc. TC2 IFIP Working Conf. 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.

[Kawamura-Kanamori 88] Kawamura, T. and Kanamori, T.: "Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation", Proc. Int. Conf. on FGCS, Tokyo, 1988, pp. 413-422.

[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 Int. Conf. on Logic Programming, Lisboa (Portugal) 1989, pp. 539-553.

[Proietti-Pettorossi 89] Proietti, M. and Pettorossi, A.: "The Loop Absorption and the Generalization Strategies for the Development of Logic Programs", Report IASI-CNR, Roma, Italy, (Dec. 1989).

[Sterling-Lakhotia 88] Sterling, L. and Lakhotia, A.: "Composing Prolog Meta-Interpreters", Proc. 5th Int. Conf. on Logic Programming, Seattle, WA (USA), 1988.

[Tamaki-Sato 84] Tamaki, H. and Sato, T.: "Unfold/Fold Transformation of Logic Programs", Proc. 2nd Int. Conf. on Logic Programming, Uppsala, 1984.

[Wadler 84] Wadler, P. L.: "Listless is Better than Laziness" Ph.D. Dissertation, Carnegie-Mellon University, August 1984.