An Abstract Strategy for Transforming Logic Programs

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

Alberto PETTOROSSI
Electronics Department, University of Roma Tor Vergata, 00133 Roma, Italy
email: adp@iasi.rm.cnr.it

Abstract. We study the problem of automating some development techniques for logic programs. These techniques are based on the application of semantics preserving transformation rules which are driven by strategies. We propose an *abstract strategy* which is parametrized by three mathematical functions called *definition-folding*, *selection*, and *replacement*.

Once these three functions are supplied, our abstract strategy becomes a concrete one which can be used during program development for driving the application of the Definition, Folding, Unfolding, and Goal Replacement Rules.

We show that the definition-folding function can be determined in an automatic way from the description of the syntactic properties of the program we wish to derive.

We also show through some examples that many program derivation strategies described in the literature, such as the methodology for eliminating unnecessary variables, the tupling strategy, the partial deduction techniques, and the promotion strategy, can be viewed as particular instances of our abstract strategy.

1. Introduction

One of the most popular approaches to the automation of the program transformation methodology is the so called 'rules + strategies' approach [4]. The rules are used for performing elementary transformations which preserve program semantics, while the strategies are used for controlling the application of the transformation rules and improving program efficiency. This approach is similar to the one often used in the area of automated Theorem Proving [3], where the basic inference rules are guided by heuristics.

One could add more levels of control to the program transformation process by introducing, for instance, *meta-strategies* for composing various strategies together. The reader may refer to [6] for a comprehensive survey in the case of functional programs. (In that paper Feather actually uses the name of tactics for what we call here strategies.)

In this paper we consider the case of logic programs and we propose a general framework which can be used for defining several transformation strategies. This framework consists of an abstract strategy which can be instantiated to several concrete strategies by providing three

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. (Italy) under grant n. 89.00026.69.

mathematical functions called *definition-folding*, *selection*, and *replacement*, respectively.

The definition-folding function is used for driving the application of the Definition and Folding Rules, while the selection and replacement functions are used for driving the application of the Unfolding and Goal Replacement Rules.

We will study the class of the definition-folding functions and we will show that they can often be mechanically generated. In particular, we will consider the case when it is required to transform a program into an equivalent one whose text satisfies some given conditions. Various examples of these transformations include the avoidance of multiple occurrences of variables, the transformation to linear recursive or tail-recursive programs, etc.

We will not consider the problem of the automatic generation of the selection and replacement functions, and we will assume that those functions are known before the transformation process begins.

Our abstract strategy is implemented by a procedure, called *(DFUR)** which is an iteration of a compound transformation made out of some definition and folding steps, followed by an unfolding step and a replacement step. The (DFUR)* Procedure can be viewed as a generalization of the procedure for eliminating unnecessary variables introduced in [16], because the (DFUR)* Procedure has two extra parameters which are the definition-folding function and replacement function.

The paper is structured as follows.

In Section 2 we introduce the basic terminology and some preliminary definitions. In Section 3 we state the problem of transforming programs so that some syntactic conditions are satisfied. In order to solve that problem, in Section 4 we introduce our abstract strategy together with the definition-folding, selection, and replacement functions. We also show various examples of its application.

In Sections 5 and 6 we present the (DFUR)* Procedure and we prove some of its properties.

In Section 7 we introduce the so-called *Contraction Procedure* which is used for eliminating some redundant predicates which may be introduced while executing the (DFUR)* Procedure. In Section 8 we show that it is sometimes useful to iterate the applications of the (DFUR)* and Contraction Procedure, thereby arguing in favour of the introduction of meta-strategies.

In Section 9 we present a final example where we show that our proposed transformation strategy may easily incorporate semantics-based conditions (such as those related to the typing information) which the program to be derived has to satisfy.

2. Preliminary Definitions

We consider *definite logic programs* [12]. We assume that all our programs are written using a fixed language $\mathcal{L}$. We also assume that $\mathcal{L}$ contains at least one constant. The *Herbrand universe* associated with $\mathcal{L}$ is the set $H_{\mathcal{L}}$ of all ground terms which can be constructed using constant and function symbols taken from $\mathcal{L}$. Given a predicate p in $\mathcal{L}$ and a program P, we denote by M(P,p) the *least Herbrand model* of p in P, that is:

$$M(P,p) = \{p(t_1,\ldots,t_n) \mid \{t_1,\ldots,t_n\} \subseteq H_{\mathcal{L}} \text{ and } P \models p(t_1,\ldots,t_n)\}.$$

We say that two programs P_1 and P_2 are *equivalent* w.r.t. predicate p iff $M(P_1,p) = M(P_2,p)$.

Given a clause C we denote its head by hd(C) and its body by bd(C). We also denote the sets of all atoms, clauses, and programs by Atoms, Clauses, and Programs, respectively.

Given a term t, we denote by vars(t) the set of variables occurring in t. Similar notation

will be used for variables occurring in atoms, goals, and clauses. Two atoms (also goals and clauses) which differ only for the names of the variables are said to be *variants.*

We now briefly describe the rules which will be used when transforming programs. We assume that when the transformation rules are applied, two distinct clauses do *not* have variables in common.

Definition Rule. Let P be a program and D a clause (belonging or not to P) of the form: $newp(X_1,\ldots,X_m) \leftarrow A_1,\ldots,A_n$, such that: i) $\{X_1,\ldots,X_m\} \subseteq vars(\{A_1,\ldots,A_n\})$, ii) newp does not occur in $P - \{D\}$, and iii) every predicate occurring in the body of D occurs in P as well. D is said to be a *definition clause.*

Given a program P and a definition clause D (not occurring in P) we get a new program by adding D to P.

We say that a clause S_1 is a *synonym* of a clause S_2 iff there exists a renaming substitution ρ, such that $vars(hd(S_1)) = vars(hd(S_2\rho))$, and $bd(S_1) = bd(S_2\rho)$.

If during the program transformation process we are required to consider a definition clause, say S_2, which is a synonym of an already introduced one, say S_1, then we do *not* introduce S_2 and we use S_1, instead.

Unfolding Rule. Given two clauses C and D, and an atom A in bd(C) unifiable with hd(D), the result of *unfolding* C *w.r.t.* A *using* D is the clause obtained by applying an SLD-resolution step to C and D w.r.t. A.

Given a program P and a clause C in P, we get a new program by replacing C by the set of *all* clauses which can be obtained by unfolding C w.r.t. A using clauses of P.

Folding Rule. Given a clause C, a definition clause D, and a subset B of bd(C) such that B is an instance of bd(D) via a substitution θ, the result of *folding* C *using* D *w.r.t.* B is the clause F obtained from C by replacing B by $hd(D)\theta$.

A folding step can be performed only if C is not a definition clause and by unfolding F w.r.t. the atom $hd(D)\theta$ using D we get a variant of clause C.

Given a program P and a clause C in P, we get a new program by replacing C by F.

For introducing the *Goal Replacement Rule* we need the following definition.

Definition 1. (*Linking Variables*) Let C be a clause and B a subset of bd(C), the *linking* variables of B in C are the variables occurring in B and also in $\{hd(C)\} \cup (bd(C) - B)$. ■

Example 1. Let C be the clause: $h(X) \leftarrow p(b,Y), q(X,Y,Z), s(Z)$. The linking variables of $\{p(b,Y), q(X,Y,Z)\}$ in C are X and Z. ■

Goal Replacement Rule. Let P be a program and C a clause in P. Suppose that:
i) $G_1 \subseteq bd(C)$ and the linking variables of G_1 in C are $X_1,\ldots,X_v$, and
ii) G_2 is a set of atoms such that $vars(G_2) \cap vars(C) = \{X_1,\ldots,X_v\}$.

Suppose that G_1 and G_2 are *equivalent*, in the sense that we have:

$$M(P \cup \{D_1\}, new) = M(P \cup \{D_2\}, new),$$

where D_1 and D_2 are the clauses: $new(X_1,\ldots,X_v) \leftarrow G_1$ and $new(X_1,\ldots,X_v) \leftarrow G_2$, respectively, and new is a predicate symbol not occurring in P.

A *goal replacement* step consists in replacing the clause C by the clause C_1 such that $hd(C_1) = hd(C)$ and $bd(C_1) = (bd(C) - G_1) \cup G_2$.

Clause Deletion Rule. Given a program P and a clause C in P, we say that C is a *failing clause* iff its body contains an atom which is *not* unifiable with the head of any clause in P.

If C is a failing clause then it can be deleted from P.

The application of the rules presented above preserves partial correctness, in the sense that if we derive program P_2 from program P_1 then for each predicate p in P_1 we have that: $M(P_2,p) \subseteq M(P_1,p)$. By forcing some restrictions on the use of those rules [19], we can preserve total correctness, that is, we get: $M(P_1,p) = M(P_2,p)$.

The reader can easily verify that the use of the rules in the transformation techniques presented in this paper always preserves total correctness.

Variants of the above transformation rules can be shown to be correct w.r.t. other logic languages and program semantics (see, for instance, [2, 8, 10, 15, 17, 18]).

For simplicity reasons we will not discuss those variants here and we will focus our attention on the problem of mechanizing the transformation strategies when using the rules we have described above. However, we believe that the techniques we will introduce, can easily be extended to more sophisticated logic languages and more complex transformation rules.

3. The Transformation Problem and an Introductory Example

Our program transformation techniques can be used for solving instances of the following *Transformation Problem*:

> given a program P and a property Φ, we are required to find a program TransfP such that: i) P is equivalent to TransfP w.r.t. every predicate occurring in P and ii) Φ(TransfP) holds.

We assume that Φ is a decidable property. Some examples of Φ will be given below.

Example 2. Eliminating Unnecessary Variables.

An unnecessary variable of a clause C is a variable which *either* occurs in the body of C and not in its head, *or* it occurs more than once in the body of C.

Given a program P we want to obtain an equivalent program without unnecessary variables, that is, we want to solve an instance of the Transformation Problem, where Φ(TransfP) holds iff no clause in TransfP contains unnecessary variables.

We may apply the strategy presented in [16] which is based on the so-called *Elimination Procedure.* This procedure consists in the repeated application of a *compound transformation* made out of an unfolding step, followed by some definition and folding steps.

The abstract transformation strategy we will present in this paper is an improvement of the strategy for eliminating unnecessary variables, and in order to clarify our presentation, let us briefly recall the essential features of the Elimination Procedure by looking at the following program derivation.

Let us consider the problem of eliminating the unnecessary variables from the following program, called RL:

1. rotate_leftdepth(Tree1,N) ← rotate(Tree1,Tree2), leftdepth(Tree2,N).
2. rotate(leaf,leaf).
3. rotate(tree(L,R),tree(L1,R1)) ← rotate(L,L1), rotate(R,R1).
4. rotate(tree(L,R),tree(R1,L1)) ← rotate(L,L1), rotate(R,R1).
5. leftdepth(leaf,0).
6. leftdepth(tree(L,R),succ(N)) ← leftdepth(L,N).

where: i) rotate(T_1,T_m) holds iff either $T_1 = T_m$ or there exists a sequence of binary trees $T_1,\ldots,T_m$ such that for i=1,...,m-1, T_{i+1} is obtained from T_i by interchanging the left and right subtrees of a node, and ii) leftdepth(T,N) holds iff N is the length of the path from the root of the binary tree T to its leftmost leaf.

The objective of our transformation process is to replace clause 1 by a set of clauses without unnecessary variables, thereby avoiding the construction of the intermediate tree Tree2.

By performing an unfolding step followed by some definition and folding steps we can indeed do so at the expense of introducing some new definitions *with* unnecessary variables, as the following derivation shows.

Unfolding step. By unfolding clause 1 w.r.t. the atom rotate(Tree1,Tree2) we get the three clauses:

7. rotate_leftdepth(leaf, N) ← leftdepth(leaf, N).
8. rotate_leftdepth(tree(L,R), N) ← rotate(L, L1), rotate(R, R1), leftdepth(tree(L1,R1), N).
9. rotate_leftdepth(tree(L,R), N) ← rotate(L, L1), rotate(R, R1), leftdepth(tree(R1,L1), N).

Definition step. We introduce the following new clause, whose body is made out of the atoms with unnecessary variables in clause 8:

10. new1(L, R, N) ← rotate(L, L1), rotate(R, R1), leftdepth(tree(L1,R1), N).

Folding steps. We fold clauses 8 and 9 using the newly introduced clause 10. We get:

8f. rotate_leftdepth(tree(L, R), N) ← new1(L, R, N).
9f. rotate_leftdepth(tree(L, R), N) ← new1(R, L, N).

Now we have that the program RL is equivalent to (RL – {clause 1}) ∪ {7, 8f, 9f, 10}, where clauses 7, 8f, and 9f do *not* contain unnecessary variables.

The application of the Elimination Procedure continues by applying to the new definition clause 10 with the unnecessary variables L1 and R1 the compound transformation consisting of an unfolding step followed by some definition and folding steps. This transformation may determine the introduction of new definitions containing unnecessary variables. Thus, these definitions must, in turn, be processed by the Elimination Procedure.

According to the Definiton Rule presented in Section 2, we comply with the following condition: if a definition, say N, to be introduced during the execution of the Elimination Procedure, is a synonym of an already existing definition, say M, we do not add N to the set of the definitions to be processed by the procedure, and we use, instead, the old clause M.

The transformation process terminates when no definitions to be processed are left.

From the above description of the Elimination Procedure, one can see that it is parametrized by two functions: the first one, called *definition-folding function*, is used for introducing the new definition clauses and performing the folding steps, while the second one, called *selection function*, is used for selecting the atom for unfolding.

The selection function can be effectively constructed for a class of programs which includes the one we consider here, as indicated in [16], and the definition-folding function can be constructed as follows.

For any clause C and any two atoms A_1 and A_2 in bd(C), we assume that $A_1 \downarrow A_2$ holds

iff $vars(A_1) \cap vars(A_2) \neq \emptyset$. Let $\Downarrow$ denote the reflexive and transitive closure of the relation $\downarrow$ over bd(C).

We consider the partition of bd(C) into the equivalence classes (or *blocks*) determined by $\Downarrow$, and for each equivalence class B containing unnecessary variables we consider a (possibly new) definition D_B such that: $bd(D_B) = B$ and $hd(D_B)$ contains exactly the linking variables of B in C. It is clear that by folding B using D_B all unnecessary variables occurring in B are eliminated.

We consider the set DefC of all new definition clauses constructed as we have now described. We also consider the clause FoldC obtained from C by performing a folding step for each block B in bd(C) containing unnecessary variables. FoldC does *not* contain unnecessary variables.

The value of the definition-folding function we need to construct is given by the pair <DefC, FoldC>.

As an example of use of the definition-folding function, the reader may verify that in our case, during the execution of the Elimination Procedure, we have to consider the following clause:

11. new1(L, R, succ(N)) ← rotate(L, L1), rotate(R, R1), leftdepth(L1, N).

whose body is partitioned into the blocks:

b1. {rotate(R, R1)}, with linking variable R, and
b2. {rotate(L, L1), leftdepth(L1, N)}, with linking variables L and N.

Block b1 determines the introduction of the following definition clause:

12. new2(R) ← rotate(R, R1).

while block b2 determines a definition clause which is a synonym of clause 1. Thus, according to our Definition Rule we do not introduce a new clause for folding b2 and we use clause 1 instead. By folding clause 11 using clauses 1 and 12 we get the clause:

13. new1(L, R, succ(N)) ← rotate_leftdepth(L, N), new2(R).

which does not contain unnecessary variables. Thus, the value of the definition-folding function for clause 11 is the pair: <{clause 12}, clause 13>.

We leave to the reader to check that, for a suitable selection function (see also Example 4), the Elimination Procedure terminates and it produces the following set of clauses:

7. rotate_leftdepth(leaf, N) ← leftdepth(leaf, N).
8f. rotate_leftdepth(tree(L, R), N) ← new1(L, R, N).
9f. rotate_leftdepth(tree(L, R), N) ← new1(R, L, N).
13. new1(L, R, succ(N)) ← rotate_leftdepth(L, N), new2(R).
14. new2(leaf).
15. new2(tree(L,R)) ← new2(L), new2(R).

Now our task of eliminating all unnecessary variables has been completed because in the initial program clause 1 can be replaced by the above set of clauses. The derived program does not construct any tree to be passed from the computation of rotate to the computation of leftdepth.

Moreover, the derived program can be simplified by eliminating the intermediate predicate new1. This simplification can be done by performing some unfolding steps, which can be considered to be an application of the *Partial Evaluation* technique (also called *Partial Deduction* in the case of logic programs) [7, 11, 13]. These final unfolding steps will be determined in our approach by the *Contraction Procedure* introduced in Section 7. ■

4. The Abstract Transformation Strategy

In order to deal with more general Transformation Problems, we will now extend the basic ideas underlying the Elimination Procedure presented in Example 2. We will consider two enhancements which consist in:

i) allowing the application of the Goal Replacement Rule, and

ii) considering more general ways of performing the definition and folding steps, in the sense that the new definitions are not necessarily determined by the equivalence relation $\Downarrow$ over the bodies of the clauses.

The abstract transformation strategy which derives from the above two enhancements is parametrized by the definition-folding, selection, and replacement functions. It can be outlined as follows.

> Suppose that we have an instance of the Transformation Problem for a given program P and property Φ.
>
> We partition P into two sets of clauses TransfP and RestofP, such that Φ(TransfP) holds and RestofP is the complement of TransfP w.r.t. P.
>
> We then consider a clause C in RestofP and we perform some definition and folding steps on C (according to the definition-folding function) so that Φ(TransfP $\cup$ {F}) holds, where F is the clause derived by folding C.
>
> If these definition and folding steps are not possible because either C itself is a definition clause or we cannot find the appropriate clauses for folding, we maintain unchanged the values of TransfP and RestofP, otherwise we add F to TransfP and replace C in RestofP by the set of new definition clauses used for folding C.
>
> We then perform some unfolding steps (according to the selection function) and we apply the Goal Replacement Rule (according to the replacement function) on the clauses of RestofP. If for a set T of clauses generated by these transformation steps we have that Φ(TransfP $\cup$ T) holds, then we add T to TransfP.
>
> Finally we recursively solve the Transformation Problem for the current values of TransfP and RestofP.
>
> The transformation process terminates when RestofP is empty. In that case the initial program is equivalent to TransfP, for which the property Φ holds by construction.

In Section 5 we will present a procedure, called (DFUR)*, which implements the abstract strategy we have now outlined.

In order to clarify the reader's ideas let us now see how our abstract strategy works in the following example.

Example 3. (*Towers of Hanoi*) Consider the following program for the familiar Towers of Hanoi problem:

1. hanoi(0,A,B,C,[ ]).
2. hanoi(s(N),A,B,C,M) ← hanoi(N,A,C,B,M1), hanoi(N,C,B,A,M2),
   append(M1,[m(A,B) | M2],M).
3. append([ ], L, L).
4. append([H | T], L, [H | TL]) ← append(T, L, TL).

We want to derive an equivalent *linear recursive* program for the hanoi relation, where linear recursiveness is defined as follows.

Given two predicates p and q in a program P, we say that p *depends on* q iff in P *either* there exists a clause of the form p(...) ← ..., q(...), ... *or* there exists a clause of the form p(...) ← ..., r(...), ... and r depends on q. A clause E in P is said to be *linear recursive w.r.t. program* P iff in bd(E) there exists at most one predicate depending on the predicate which occurs in hd(E). We say that program P is linear recursive iff all its clauses are linear recursive w.r.t. P.

A linear recursive program for the hanoi relation can be achieved by using the *tupling* transformation strategy [14]. We now show that the same goal can also be achieved by using our abstract transformation strategy, where Φ(TransfP) holds iff TransfP is a linear recursive program.

We assume that during the derivation process the goal replacement steps are determined by the functionality of hanoi(N,A,B,C,M) w.r.t. the first four arguments, that is, for any term n, a, b, c, m1, and m2, we may replace the goal 'hanoi(n,a,b,c,m1), hanoi(n,a,b,c,m2)' by the goal 'hanoi(n,a,b,c,m1), m1=m2'.

The subset of the initial program made out of all clauses different from clause 2 is linear recursive. Thus, initially we have: TransfP = {clause 1, clause 3, clause 4} and RestofP = {clause 2}.

We now perform a definition and a folding step on clause 2 so that the derived clause is linear recursive. Indeed, we introduce the following definition clause:

5. new1(N,A,B,C,M1,M2) ← hanoi(N,A,C,B,M1), hanoi(N,C,B,A,M2).

and, by folding clause 2 using clause 5 we get:

2f. hanoi(s(N),A,B,C,M) ← new1(N,A,B,C,M1,M2), append(M1,[m(A,B) | M2],M).

Now, TransfP = {clause 1, clause 2f, clause 3, clause 4} and RestofP = {clause 5}.

The definition clause 5 is obtained by applying the following general technique for deriving linear recursive programs. Given a clause C which is not linear recursive w.r.t. TransfP ∪ {C} we consider a definition clause D_C of the form:

$p(X_1,\ldots,X_n)$ ← NewBody

such that: i) p is a new predicate symbol iff D_C is not a synonym of an already introduced definition, ii) $X_1,\ldots,X_n$ are the linking variables of NewBody in C, and iii) NewBody is the set of all atoms in bd(C) whose predicates depend on the predicate of hd(C).

D_C is then used for performing a folding step on C, thereby obtaining a linear recursive clause FoldC which is added to TransfP. We have that TransfP ∪ {FoldC} is a linear recursive program.

Now we unfold clause 5 and we get:

6. new1(0,A,B,C,[ ],M) ← hanoi(0,C,B,A,M).
7. new1(s(N),A,B,C,M1,M2) ← hanoi(N,A,B,C,M11), hanoi(N,B,C,A,M12),
   hanoi(s(N),C,B,A,M2),
   append(M11,[m(A,C) | M12],M1).

Clause 6 is linear recursive while clause 7 is not. The functionality rule *cannot* be applied in our case. Thus, no replacement is performed on clause 7.

The abstract transformation strategy is then applied to the current values of TransfP and RestofP, that is, {1, 2f, 6, 3, 4} and {7}, respectively.

By definition, we introduce the predicate new2 by the following clause:

8. new2(N,A,B,C,M11,M12,M2) ← hanoi(N,A,B,C,M11), hanoi(N,B,C,A,M12),
hanoi(s(N),C,B,A,M2).

By folding clause 7 using clause 8 we get the following linear recursive clause:

9. new1(s(N),A,B,C,M1,M2) ← new2(N,A,B,C,M11,M12,M2),
append(M11,[m(A,C) | M12],M1).

By unfolding clause 8 we get:

10. new2(N,A,B,C,M11,M12,M2) ← hanoi(N,A,B,C,M11), hanoi(N,B,C,A,M12),
hanoi(N,C,A,B,M21), hanoi(N,A,B,C,M22),
append(M21,[m(C,B) | M22], M2).

By applying the Goal Replacement Rule we get:

11. new2(N,A,B,C,M11,M12,M2) ← hanoi(N,A,B,C,M11), hanoi(N,B,C,A,M12),
hanoi(N,C,A,B,M21), M11=M22,
append(M21,[m(C,B) | M22], M2).

At this point of the derivation we have: TransfP = {1, 2f, 6, 9, 3, 4} and RestofP = {11}.

We are now left with the problem of transforming clause 11 into a linear recursive one. Thus, we may continue the application of our abstract transformation strategy by introducing some more definition clauses for performing folding steps on clause 11.

We cannot show the remaining derivation for lack of space, however the reader may easily verify that eventually no new definitions will be introduced in RestofP and the derivation process halts producing the following linear recursive clauses:

1. hanoi(0,A,B,C,[ ]).
2f. hanoi(s(N),A,B,C,M) ← new1(N,A,B,C,M1,M2), append(M1,[m(A,B) | M2],M).
6. new1(0,A,B,C,[ ],M) ← hanoi(0,C,B,A,M).
9. new1(s(N),A,B,C,M1,M2) ← new2(N,A,B,C,M11,M12,M2),
append(M11,[m(A,C) | M12],M1).
12. new2(N,A,B,C,M11,M12,M2) ← new3(N,A,B,C,M11,M12,M21), M11=M22,
append(M21,[m(C,B) | M22], M2).
13. new3(0,A,B,C,[ ],M12,M21) ← new1(0,B,A,C,M12,M21).
14. new3(s(N),A,B,C,M11,M12,M21) ← new4(N,A,B,C,M111,M112,M12,M21),
append(M111,[m(A,B) | M112],M11).
15. new4(N,A,B,C,M111,M112,M12,M21) ← new5(N,A,B,C,M111,M112,M121,M21),
M111=M122,
append(M121,[m(B,C) | M122],M12).
16. new5(N,A,B,C,M111,M112,M121,M21) ← new3(N,A,C,B,M111,M112,M121),
M211=M112, M212=M121,
append(M211,[m(C,A) | M212],M21).

The program we have derived so far can be further simplified by performing some unfolding steps for eliminating the intermediate predicates new1, new2, new4 and new5 (see the Contraction Procedure in Section 7) and for partially evaluating the equalities and some atoms with the predicates append and hanoi. By doing so, we get the following program:

```
hanoi(0,A,B,C,[ ]).
hanoi(s(0),A,B,C,[m(A,B)]).
hanoi(s(s(N)),A,B,C,M) ← new3(N,A,B,C,M1,M2,M3),
                          append(M3,[m(C,B) | M1],M4),
                          append(M1,[m(A,C) | M2],M5),
                          append(M5, [m(A, B) | M4],M).
new3(0,A,B,C,[ ],[ ],[ ]).
new3(s(N),A,B,C,M1,M2,M3) ← new3(N,A,C,B,M4,M5,M6),
                             append(M5,[m(C,A) | M6],M3),
                             append(M6,[m(B,C)|M4],M2),
                             append(M4,[m(A,B)|M5],M1).
```
■

As this example shows, the abstract transformation strategy we have outlined at the beginning of this section, becomes a concrete procedure for program derivation when we provide three mathematical functions:

i) the *definition-folding function* (*df-function*, for short) α for determining the set NewD of new definition clauses to be introduced and the clause obtained from a given clause by performing some folding steps using either clauses in NewD or definition clauses introduced in a previous application of the Definition Rule. We assume that the value of α may depend on:
   - the given clause,
   - the program derived so far by the abstract transformation strategy, and
   - the set of definition clauses introduced so far during the transformation process.

   Thus, α is a partial function from Clauses × Programs × $\mathcal{P}$(Clauses) to $\mathcal{P}$(Clauses) × Clauses. For any given clause C, program P, and set of clauses Defs, the first and second components of α(C, P, Defs) will be written α_d(C, P, Defs) and α_f(C, P, Defs), respectively. Thus, $\alpha = <\alpha_d, \alpha_f>$.

   Since our restrictions on the use of the Definition Rule forbid the introduction of two synonym clauses, we will assume that for every clause C, program P, and set of clauses Defs, α_d(C, P, Defs) does not contain a synonym of a clause in Defs.

ii) the *selection function* S for selecting the atom to be unfolded in the body of the clause in hand. S is a total function from Clauses to Atoms.

iii) the *replacement function* R for specifying the goal replacements (possibly none) to be applied to the body of a clause. R is a total function from Clauses to Clauses such that hd(C) = hd(R(C)) for every clause C.

An example of the value of the pair $<\alpha_d, \alpha_f>$ is provided by <DefC, FoldC> constructed in Example 2 for solving the problem of eliminating unnecessary variables from the program defining the relation rotate_leftdepth. Some more examples of df-functions will be given in the sequel.

The df-function α is often suggested by the form of the property Φ given in the specification of the Transformation Problem. Indeed, during the application of our transformation strategy the folding steps due to the function α should produce a clause F such that Φ(TransfP ∪ {F}) holds. Thus, we will consider only α's which are *consistent* with Φ, in the sense specified by the following definition.

Definition 2. (*Consistent df-functions*) Given a property Φ over Programs, we say that a df-function α is *consistent* with Φ iff for every clause C, program TransfP, and set of clauses Defs, *if* there exists a set N of definition clauses which are not synonym of clauses in Defs and there exists a clause T such that T can be obtained by folding C using clauses in Defs ∪ N and Φ(TransfP ∪ {T}) holds *then* Φ(TransfP ∪ {α_f(C, TransfP, Defs)}) holds *else* α(C, TransfP, Defs) is undefined. ■

Obviously, for each Φ there exists an α which is consistent with Φ.

The assumption of decidability of the property Φ implies that for any given C the value of a df-function α consistent with Φ can be constructed by first generating the set of all clauses derivable from C by definition and folding steps, and then selecting from that set a clause which satisfies Φ. This generate-and-test process terminates because according to our Definition Rule, for any clause C the set of definition clauses which can be used for folding C is finite. (Recall, in particular, that synonym clauses cannot be introduced and the arguments of the head of a definition clause are distinct variables which also occur in its body.)

5. The (DFUR)* Procedure

The procedure which implements our abstract transformation strategy is the following one.

(DFUR)* Procedure.
Input: An instance of the Transformation Problem, that is, a program P without failing clauses and a predicate Φ over Programs, together with a df-function α which is consistent with Φ, a selection function S, and a replacement function R.
Output: a program TransfP such that: i) TransfP is equivalent to P w.r.t. every predicate occurring in P, and ii) Φ(TransfP) holds.

Initially, let TransfP be a maximal subset of P such that Φ(TransfP) holds, RestofP be the complement of TransfP w.r.t. P, and Defs be the set of definition clauses occurring in P.
while RestofP $\neq \emptyset$ *do*
consider a clause C in RestofP and create a set of clauses TransfC;
1: (*Definition and Folding Steps*)
if C does not occur in Defs and α(C,TransfP,Defs) is defined
then i) TransfC := α_d(C, TransfP,Defs),
ii) TransfP := TransfP $\cup$ {α_f(C, TransfP,Defs)},
iii) Defs := Defs $\cup$ α_d(C, TransfP,Defs)
else TransfC := {C};
2: (*Unfolding Step*)
TransfC := {E | E is a non-failing clause which can be obtained by unfolding a clause D in TransfC w.r.t. the atom S(D) using clauses in P};
consider a maximal subset T of TransfC such that Φ(TransfP $\cup$ T) holds;
TransfP := TransfP $\cup$ T;
TransfC := TransfC – T;
3: (*Replacement Steps*)
for every clause E in TransfC replace E by R(E);
consider a maximal subset T of TransfC such that Φ(TransfP $\cup$ T) holds;
TransfP := TransfP $\cup$ T;
TransfC := TransfC – T;
4: RestofP := (RestofP – {C}) $\cup$ TransfC. ■

Example 4. (Eliminating Unnecessary Variables Revisited)
Let us consider again the instance of the Transformation Problem presented in Example 2. In that case Φ(TransfP) is: 'TransfP does not contain any clause with unnecessary variables'.

The derivation of Example 2 above is an application of our (DFUR)* Procedure.

i) The df-function α is the one introduced in Example 2 by means of the equivalence relation $\Downarrow$. That df-function is consistent with Φ.

ii) The selection function S is constructed as follows (see also the SDR rule in [16]).
Let us assume the usual tree representation of atoms where predicate, function, constant, and variable symbols are labels of nodes. Let A be an atom and let X be a variable occurring in A. The *depth* of X in A, denoted by depth(X, A), is the length of the *shortest* path from the root of A to a leaf labelled by X. Given two atoms A and B, we write $A \leq_{var} B$ iff for each variable X in vars(A) $\cap$ vars(B) we have that depth(X, A) $\leq$ depth(X, B).
For a given clause C the selection function S returns the leftmost atom M such that for every atom A in bd(C) we have that $A \leq_{var} M$. If such an atom M does not exist S returns the leftmost atom in bd(C).

iii) The replacement function R is in this case the identity function, that is, we do not make any goal replacement.

The reader may verify that the (DFUR)* Procedure with the definition-folding, selection, and replacement functions described above, does terminate and it produces the program given at the end of Example 2. ■

Example 5. (Towers of Hanoi Revisited)
The derivation presented in Example 3 is obtained by applying the (DFUR)* Procedure with the following input functions.

i) For any given clause C which is not linear recursive w.r.t. TransfP $\cup$ {C} the value of the df-function is the pair by $<D_C$,FoldC>. This df-function is consistent with the property Φ which for any program P states that P is linear recursive.
ii) The selection function is the SDR [16].
iii) The replacement function is the one determined by the functionality of the relation hanoi(n,a,b,c,m) w.r.t. <n,a,b,c>. ■

6. Properties of the (DFUR)* Procedure

In this section we prove the partial correctness of the (DFUR)* Procedure as well as some properties about its termination.

Theorem 1. The (DFUR)* Procedure is partially correct, that is, for any program P and property Φ we have that, if the (DFUR)* terminates with output TransfP then:
i) P is equivalent to TransfP w.r.t. every predicate occurring in P and
ii) Φ(TransfP) holds.

Proof. Point i) follows from the correctness of the transformation rules [19]. Point ii) follows from the fact that initially Φ(TransfP) holds and for every set T of clauses which is added to TransfP, Φ(TransfP $\cup$ T) holds. In particular, since the df-function α is consistent with Φ, we have that for any clause C, Φ(TransfP $\cup$ {α_f(C, TransfP, Defs)}) holds. ■

Theorem 2. The problem of deciding whether or not the (DFUR)* Procedure terminates is undecidable.

Proof. The (DFUR)* Procedure is a generalization of the Elimination Procedure whose termination is not decidable [16]. ■

We now present a result which can be useful for proving the termination of the (DFUR)* Procedure in many practical cases.

Let us consider an execution of the (DFUR)* Procedure for a program P, a property Φ, a df-function α, a selection function S, and a replacement function R. Let $Defs_i$ be the value of

the variable Defs immediately after the i-th execution of the assignment of point 1.iii) of that procedure. By definition $Defs_0 = \{D \mid D$ is a definition clause occurring in the initial program $P\}$.

Lemma 3. *(Termination of (DFUR)*)* Suppose that the df-function α is total. Then the (DFUR)* Procedure terminates iff the set $BdDefs = \{bd(D) \mid D \in Defs_i$ for some $i \geq 0\}$ is finite.

Proof. If the (DFUR)* Procedure terminates then the sequence of values assigned to the variable Defs is finite and this implies the finiteness of BdDefs.

In order to prove the inverse implication let us notice that the execution of each definition, folding, unfolding, and replacement step terminates. Thus, it is enough to show that the body of the *while-do* loop is executed a finite number of times.

Let us assume that BdDefs is a finite set. The set $Defs^* = \{D \in Defs_i$ for some $i \geq 0\}$ is finite as well, because no two synonym definition clauses are introduced and the set of variables occurring in the head of a definition clause is a subset of the variables occurring in its body.

Each unfolding step of point 2 is performed on a clause belonging to Defs*, because the df-function α is total. Thus, the thesis follows from the fact that point 2 of the procedure can be executed a finite number of times only and it is executed each time the body of the *while-do* loop is entered. ■

The following result is an immediate consequence of the above lemma.

Theorem 4. Let α be a total df-function. If the set $\{bd(\alpha_d(C, P, D)) \mid C \in$ Clauses, $P \in$ Programs, $D \in \mathcal{P}$(Clauses)$\}$ is finite modulo renaming of variables then every execution of the (DFUR)* Procedure with α as input terminates independently of the other input values.■

7. The Contraction Procedure

As it is demonstrated by Example 2, the (DFUR)* Procedure may introduce some predicates which can be eliminated from the derived program by means of unfolding steps. These unfolding steps may be performed according to the *Contraction Procedure* presented below.

In that procedure we use the following terminology.

Given a program P, a predicate r in P is said to be *recursive* iff there exists a program clause of the form: $r(\ldots) \leftarrow \ldots, r(\ldots), \ldots$.

Given a set Rec of predicates, the Contraction Procedure always terminates and it produces a program in which every predicate occurring in both Rec and the body of a clause, is recursive. Notice that our Contraction Procedure does not realize all predicate eliminations which are possible by performing unfolding steps. However, it is impossible to exhibit a procedure which is substantially better than ours, because the problem of deciding for any given program whether or not a predicate can be eliminated by unfolding is undecidable.

Contraction Procedure.

Input: a program P, a set Rec of predicates occurring in P, and a predicate p in P.

Output: a program P_{Contr} such that: i) P_{Contr} is equivalent to P w.r.t. p, ii) for every predicate q occurring in P_{Contr} if q is different from p then p depends on q, and iii) every predicate belonging to Rec and occurring in the body of a clause of P_{Contr} is recursive.

while there exists a predicate q in Rec *do*

1: *if* q is not recursive

then *while* there exists a clause C in P and an atom A of the form q(...) in bd(C) *do*
if C is a failing clause
then P := P – {C}
else P = (P – {C}) ∪ {C' | C' is the result of unfolding C w.r.t. A using a clause in P};
2: Rec := Rec – {q};
P_{Contr} := {Q | Q ∈ P *and* p depends on the predicate symbol of hd(Q)}. ■

Theorem 5. The Contraction Procedure is totally correct.

Proof. i) The equivalence of P_{Contr} and P w.r.t. p follows from the fact that we use transformation rules which preserve equivalence w.r.t. p. Indeed, the Unfolding Rule, the deletion of a failing clause, and the deletion of a clause C such that p does not depend on the predicate symbol occurring in hd(C), preserve the equivalence w.r.t. p. (Recall that the language from which we take the symbols for writing our programs is fixed and therefore, these deletion operations do not change the Herbrand universe.)
ii) Obvious, by the last assignment of the Contraction Procedure.
iii) Let us consider the predicate q processed during the i-th execution of the body of the outer *while-do* loop. After the execution of point 1 q is either recursive or it does not occur in the body of any clause. Thus, after the execution of the *while-do* loop p does not depend on the non-recursive predicates. Every occurrence of a non-recursive predicate different from p is then eliminated by the last assignment of our Contraction Procedure.

The inner *while-do* loop terminates. Indeed, suppose that during its execution we have in hand a non-recursive predicate q and a clause C. If C is a failing clause then, after the execution of the body of the *while-do* loop, the number of occurrences of q in P decreases by at least one. If C is not a failing clause then, by unfolding, C is replaced by a set of clauses such that in each of them q has one occurrence less than in C. (Recall that, since q is not recursive no occurrence of q can be introduced by unfolding a clause w.r.t. q(...).)

The termination of the Contraction Procedure is then obvious, because the body of the outer *while-do* loop is executed once for every predicate in Rec which is a finite set. ■

Example 6. We apply the Contraction Procedure to the final program derived in Example 2. We write here again that program for the reader's convenience:

7. rotate_leftdepth(leaf, N) ← leftdepth(leaf, N).
8f. rotate_leftdepth(tree(L, R), N) ← new1(L, R, N).
9f. rotate_leftdepth(tree(L, R), N) ← new1(R, L, N).
12. new1(L, R, succ(N)) ← rotate_leftdepth(L, N), new2(R).
13. new2(leaf).
14. new2(tree(L,R)) ← new2(L), new2(R).
2. rotate(leaf,leaf).
3. rotate(tree(L,R),tree(L1,R1)) ← rotate(L,L1), rotate(R,R1).
4. rotate(tree(L,R),tree(R1,L1)) ← rotate(L,L1), rotate(R,R1).
5. leftdepth(leaf,0).
6. leftdepth(tree(L,R),succ(N)) ← leftdepth(L,N).

We take Rec to be the set {new1, new2, rotate_leftdepth}, and p to be the predicate rotate_leftdepth.

Since new1 is not recursive and it belongs to Rec, we unfold clauses 8f and 9f w.r.t. new1(...) and we get:

15. rotate_leftdepth(tree(L, R), succ(N)) ← rotate_leftdepth(L, N), new2(R).
16. rotate_leftdepth(tree(L, R), succ(N)) ← rotate_leftdepth(R, N), new2(L).

Now the predicates new2 and rotate_leftdepth are recursive and we exit from the *while-do* loop with clauses: 7, 15, 16, 12, 13, 14, 2, 3, 4, 5, and 6.

Then we get rid of clauses: 12, 2, 3, and 4, because the predicate rotate_leftdepth does not depend on the predicates occurring in their heads. Thus, the Contraction Procedure terminates with the final program P_{Contr} made out of clauses: 7, 15, 16, 13, 14, 5, and 6. ■

8. Iterating (DFUR)* and Contraction: An Example of Partial Deduction

In Section 6 we proved that the output of the (DFUR)* Procedure is a set of clauses which satisfy a given property Φ. It is easy to see that the Contraction Procedure may affect the validity of Φ (see also Example 7 below). When this happens, we may start again the transformation process by applying the (DFUR)* Procedure followed by the Contraction Procedure until the derived program satisfies property Φ.

We would like to illustrate the usefulness of iterating the (DFUR)* and Contraction Procedures by an example of *Partial Deduction*. Partial Deduction is a transformation technique which specializes a given program to a given goal. The efficiency improvements one can expect from the application of this technique are due to the fact that some parts of the input are processed at compile time, and thus, during the execution of the transformed program (which in this context is called the *residual* program) some computations can be avoided.

Example 7. (Double Append)
Let us consider the following program, called Double_Append:

double_append(L1, L2, L3, L4) ← append(L2, L3, A), append(L1, A, L4).
append([ ], L, L).
append([H | T], L, [H | TL]) ← append(T, L, TL).

Suppose that we want to specialize the above program to queries which are instances of the goal double_append(L1, [a], L3, L4). In order to do so we introduce the new clause:

D1. query(L1, L3, L4) ← double_append(L1, [a], L3, L4).

and we would like to derive from the program Double_Append ∪ {D1} a new program in which the binding [a] has been already processed.

In this example, and more generally in the case of Partial Deduction, a possible definition of the property Φ in the specification of the Transformation Problem is as follows: Φ(TransfP) holds iff for any given clause C in TransfP no atom in bd(C) is instantiated, that is, all arguments of the predicates are unbound variables.

Let us consider the df-function $\alpha = \langle \alpha_d, \alpha_f \rangle$ defined as follows:

i) α_d(C, P, Defs) is the set of definition clauses which are not synonyms of clauses in Defs and are of the form new(X_1,...,X_n) ← A, where A is an instantiated atom in bd(C) and X_1,...,X_n are the linking variables of A, and
ii) α_f(C, P, Defs) is the clause obtained by folding each instantiated atom using the corresponding definition clause in α_d(C, P, Defs) ∪ Defs.

The df-function α is total and consistent with Φ. The selection function returns the leftmost atom in the body of the clause in hand. The Goal Replacement Rule is not applied, that is, we have: R(C) = C for each clause C.

We now apply the (DFUR)* Procedure to Double_Append ∪ {D1}, the df-function, the

selection function, and replacement function given above. The (DFUR)* Procedure terminates with the following program as output:

1. query(L1, L3, L4) ← new1(L3, A), append(L1, A, L4).
2. new1(L3, [a | T]) ← new2(L3, T).
3. new2(L3, L3).

together with the clauses for append. Notice that the derived program satisfies the given property Φ.

We apply the Contraction Procedure with input program P = Double_Append ∪ {1, 2, 3}, set of predicates Rec = {query, new1, new2}, and predicate p = query. We get the following clause:

D2. query(L1, L3, L4) ← append(L1, [a | L3], L4).

together with the clauses for append.

The program made out of the above clause D2 and the clauses for append does *not* satisfy Φ because the second argument of append in bd(D2) is instantiated. Thus, we apply again the (DFUR)* Procedure. We perform the following steps:

(*Definition and Folding*) Since D2 is a definition clause we do not perform any definition or folding step.

(*Unfolding*) We unfold clause D2 and we get:

4. query([ ], L3, [a | L3]).
5. query([H | L1], L3, [H | L4]) ← append(L1, [a | L3], L4).

(*Folding*) No new definitions are necessary, because we can fold clause 5 using clause D2, and we get:

6. query([H | L1], L3, [H | L4]) ← query(L1, L3, L4).

The (DFUR)* Procedure terminates and the Contraction Procedure eliminates the clauses for append. The derived program satisfies Φ and consists of clauses 4 and 6 only. ■

9. Using Semantic Information During Transformation: An Example

In this section we show that some semantics-based transformation techniques can easily be incorporated into the framework of our transformation technique. As an example we consider the case where the transformations take advantage of the type information provided by the programming language.

For our type declarations we will follow the syntax of the Gödel language [9].
We assume that each predicate pred with arity n in a program P is associated with a type declaration of the form 'pred : $type_1$* ... *$type_n$', where $type_1$, ... ,$type_n$ are base types. Similarly, each constant and function symbol occurring in the program is associated with a type declaration.

Example 8. (*Leaves of Binary Trees*)
Let us consider the following program Leaves which computes the list of integers which label the leaves of a binary tree (we do not write the clauses for the predicate append: they can be

found in Example 3):

1. leaves(leaf(N), [N]).
2. leaves(tree(L,R), LT) ← leaves(L, LL), leaves(R, LR), append(LL, LR, LT).

where leaves : Btree * ListOfInt and append : ListOfInt * ListOfInt * ListOfInt.

The type declarations for the constant [ ] and the function symbols [_|_], leaf, and tree are the following:

[ ] : ListOfInt;
[_|_] : Integer * ListOfInt → ListOfInt;
leaf : Integer → Btree;
tree : Btree * Btree → Btree.

Notice that for every term occurring in a clause it is possible to infer at most one type assignment which is consistent with the given type declarations. Thus, when we introduce a definition clause newp(X_1,...,X_m) ← A_1,...,A_n, the type declaration of the predicate newp can be inferred in a unique way from the already known types, because the variables which are arguments of newp occur in A_1,...,A_n.

We would like to transform the program Leaves into a program TransfLeaves in which every clause C satisfies the following properties:

i) each atom of C has precisely one argument of type Btree, if an argument of this type occurs in bd(C) at all,
ii) each argument of type Btree in bd(C) is a proper subterm of the argument of type Btree in hd(C), and
iii) two arguments of type Btree in bd(C) do not have variables in common.

A clause C satisfying i), ii), and iii) is said to be *strictly linear w.r.t. Btree.*
Thus, we would like to solve a Transformation Problem where Φ(TransfLeaves) holds iff every clause in TransfLeaves is strictly linear w.r.t. Btree.

In program Leaves clause 2 is the only clause which is not strictly linear w.r.t. Btree, because append has no arguments of type Btree. Thus, we start our transformation process by partitioning the program Leaves into the following two subsets: i) TransfLeaves containing clause 1 and the clauses for append, and ii) RestofLeaves containing clause 2 only.

(*Definition and Folding*) According to our transformation strategy, we now introduce a new definition clause such that by folding clause 2 using that definition clause we get a clause which is strictly linear w.r.t. Btree. Indeed, we introduce the following definition clause:

3. new1(T, A, LTA) ← leaves(T, LT), append(LT, A, LTA).

For the new predicate new1 we have the following type declaration:

new1 : Btree * ListOfInt * ListOfInt.

By folding clause 2 using clause 3 we get:

2f. leaves(tree(L,R), LT) ← new1(L, LR, LT) , leaves(R, LR).

which is strictly linear w.r.t. Btree.

(*Unfolding*) By unfolding clause 3 we get:

4. new1(leaf(N), A, LTA) ← append([N], A, LTA).
5. new1(tree(L,R),A,LTA) ← leaves(L,LL), leaves(R,LR), append(LL,LR,LT), append(LT,A,LTA).

Clause 4 is strictly linear w.r.t. Btree because no arguments of type Btree occur in its body.

(*Replacement*) By the associativity of append, clause 5 can be replaced by the following clause:

6. new1(tree(L,R),A,LTA) ← leaves(L,LL), leaves(R,LR), append(LL,B,LTA), append(LR,A,B).

(*Definition and Folding*) No new definition clauses are needed because by folding clause 6 using clause 3 we get the following clause which is strictly linear w.r.t. Btree:

6f. new1(tree(L,R),A,LTA) ← new1(L,B,LTA), new1(R,A,B).

RestofLeaves is now empty and we stop our transformation process. The derived program, which satisfies Φ, is the following one:

1. leaves(leaf(N), [N]).
2f. leaves(tree(L,R), LT) ← new1(L, LR, LT) , leaves(R, LR).
4. new1(leaf(N), A, LTA) ← append([N], A, LTA).
6f. new1(tree(L,R),A,LTA) ← new1(L,B,LTA), new1(R,A,B).

together with the clauses for append.

Now there is no need to apply the Contraction Procedure because all predicates in the above program are recursive. However, further simplifications can be performed by unfolding the atom append([N], A, LTA) in clause 4.

In the final program we have derived, the list concatenation performed by the predicate append is interleaved with the computation of the predicate leaves, and the construction of the list of leaves can be performed while the members of that list are generated. The second argument of the predicate new1 can be viewed as an accumulator, because it contains the portion of the list of leaves constructed so far. The reader may verify as a simple exercise that a derivation similar to ours can be performed by applying the so-called *promotion* and *accumulation* strategies [1].

Finally, we would like to notice that in the *Definition and Folding* transformations of our derivation above, the folding steps have all produced clauses which are strictly linear w.r.t. Btree. Therefore, those steps can be considered to have been performed according to a df-function which is consistent with Φ. However, for this program derivation there exist more than one df-function consistent with Φ. We will not address here the problem of choosing among all possible df-functions: we simply remark that its solution may depend on the particular transformation problems to be solved. ■

10. Conclusions

We have presented an abstract strategy for the automatic derivation of logic programs by transformation. We assume that the programmer suggests the desired 'shape' of the program to be derived (linear recursive, tail recursive, without unnecessary variables, etc.) by supplying a syntactic property Φ which should be satisfied by the transformed program.

Given the property Φ, we are required to generate three parameters (the definition-folding

function, the selection function, and the replacement function) which turn the abstract strategy into a concrete one, and also drive the application of the transformation rules (Definition, Unfolding, Folding, Goal Replacement, and Clause Deletion).

We have shown that in many cases the definition-folding function can be automatically generated.

We did not investigate on the formal relationship between the success of our abstract strategy in achieving the desired program shapes, and the increase of program efficiency. However, in all examples we have considered in this paper the final program is, under a Prolog execution, more efficient than the initial one.

We have also shown through some examples that several transformation strategies which have been proposed in the literature for improving efficiency, can be viewed as instances of our abstract strategy for suitable choices of the definition-folding, selection, and replacement functions. In particular, in Examples 2, 3, 7, and 8 we have considered the strategy for eliminating unnecessary variables [16], the tupling strategy [14], the partial deduction techniques [7, 11, 13], and the promotion strategy [1].

Related work in the area of functional programming has been presented in [5]. In the transformation system described there, some derivation steps are driven by the recursive shape of the programs to be obtained. Here we have considered the case of logic programs and we have generalized that approach by considering a generic syntactic property Φ and formalizing our transformation strategy as a higher order procedure which takes other functions as arguments.

11. Acknowledgements

We want to thank Prof. C. Rauszer and Prof. A. Skowron of the Mathematics Department of Warsaw University (Poland) for their kind invitation to take part in the Banach Semester during November 1991.

We also thank Dr. O. Aioni for her careful reading of a previous draft of this paper and our colleagues of the Department of Pure and Applied Mathematics of the University of Padova (Italy) for their stimulating and helpful discussions.

The IASI Institute of the National Research Council of Italy and the University of Rome Tor Vergata provided the necessary computing and research facilities.

References

[1] R.S. Bird, The Promotion and Accumulation Strategies in Transformational Programming. *ACM-TOPLAS* 6 (4):487-504, October 1984.

[2] A. Bossi and N. Cocco, Basic Transformation Operations for Logic Programs which Preserve Computed Answer Substitutions of Logic Programs. Tech. Rep. University of Padova, Italy, 1990. To appear in: *Special Issue of the Journal of Logic Programming on Partial Deduction*, 1992.

[3] R.S. Boyer and J.S. Moore, Proving Theorems about LISP Functions. *Journal of the ACM*, 22 (1), 129-144, 1974.

[4] R.M. Burstall and J. Darlington, A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24 (1):44-67, January 1977.

[5] M.S. Feather, A System for Assisting Program Transformation. *ACM-TOPLAS* 4 (1):1-20, January 1982.

[6] M.S. Feather, A Survey and Classification of Some Program Transformation Techniques. In: Proc. TC2 IFIP Working Conference on Program Specification and Transformation, Bad Tölz, Germany, 1986, 165-195.

[7] Y. Futamura, Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems, Computers, Controls* 2 (5): 45-50, 1971.

[8] P.A. Gardner and J.C. Shepherdson, Unfold/Fold Transformations of Logic Programs. In: J.-L. Lassez

and G. Plotkin (Eds.), Computational Logic, Essays in Honor of Alan Robinson. MIT Press, 1991, 565-583.

[9] P.M. Hill and J.W. Lloyd, The Gödel Report. TR 91-02, Department of Computer Science, University of Bristol, March 1991.

[10] T. Kawamura and T. Kanamori, Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation. *Theoretical Computer Science* 75:139-156, 1990.

[11] H.J. Komorowski, Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In: Proc. Ninth ACM Symp. on Principles of Programming Languages, Albuquerque, New Mexico, 1982, pp. 255-267.

[12] J.W. Lloyd, Foundations of Logic Programming. Springer-Verlag, 2nd edition, 1987.

[13] J.W. Lloyd and J.C. Shepherdson, Partial Evaluation in Logic Programming. *Journal of Logic Programming* 11:217-242, 1991.

[14] A. Pettorossi, Transformation of Programs and Use of Tupling Strategy. In: Proc. Informatica '77, Bled, Yugoslavia, 1977, pp. 1-6.

[15] M. Proietti and A. Pettorossi, Semantics Preserving Transformation Rules for Prolog. In: Proc. ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '91, New Haven, CT (U.S.A.) 1991, *SIGPLAN NOTICES* 26 (9):274-284, 1991.

[16] M. Proietti and A. Pettorossi, Unfolding-Definition-Folding, in This Order, for Avoiding Unnecessary Variables in Logic Programs. In: J. Maluszynski and M. Wirsing (Eds.), Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming, PLILP '91, Passau, Germany, 1991. *Lecture Notes in Computer Science* 528:347-358, 1991.

[17] T. Sato, An Equivalence Preserving First Order Unfold/Fold Transformation System. In: Proc. 2nd International Conference on Algebraic and Logic Programming, ALP '90, Nancy, France, 1990. *Lecture Notes in Computer Science* 463:175-188, 1990.

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

[19] H. Tamaki and T. Sato, Unfold/Fold Transformation of Logic Programs, In: S.-Å. Tarnlünd (Ed.) Proc. 2nd International Conference on Logic Programming, Uppsala, Sweden, 1984, pp. 243-251.