A. Pettorossi, M. Proietti

TRANSFORMATION OF LOGIC PROGRAMS: FOUNDATIONS AND TECHNIQUES

R. 369   Novembre 1993

Alberto Pettorossi - Dipartimento di Elettronica, Università di Roma "Tor Vergata", Via della Ricerca Scientifica, 00133 Roma, Italy.
E-mail: adp@iasi.rm.cnr.it

Maurizio Proietti - Istituto di Analisi dei Sistemi ed Informatica del CNR - Viale Manzoni 30, 00185 Roma, Italy.
E-mail: proietti@iasi.rm.cnr.it

This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 89.00026.69, Esprit Project Compulog II, and MURST 40%.

ABSTRACT

We present an overview of some techniques which have been proposed transformation of logic programs. We consider the so-called ‘rules + str approach, and we address the following two issues: the correctness of son transformation rules w.r.t. a given semantics and the use of strategies for gu application of the rules and improving efficiency. We will also show throu examples the use and the power of the transformational approach and we wi illustrate its relationship to other methodologies for program development.

TABLE OF CONTENTS

1. INTRODUCTION

Program transformation is a very important methodology for software development. The basic idea consists in dividing the program development activity in a sequence of small, easy steps. The programmer starts with a problem specification written in some formal language. This specification is then manipulated and transformed into an executable program which in turn is transformed, may be in several steps, with the objective of increasing efficiency. Subsequent program manipulations, such as compilation and code optimization, can also be viewed as an application of some ad hoc transformation techniques. The basic ideas of the program transformation methodology have been introduced in the early seventies for validating various techniques, such as those which remove recursion in favour of iteration [Darlington 72, Walker-Strong 72]. However, the formalization of program transformation has been done some years later [Burstall-Darlington 77] and its extensive use is strongly related to the development of the functional and logic languages, because in those languages we can perform program manipulations using simple tools, such as equational reasoning and logical deduction [Clark-Sickel 77, Hogger 81].

In this paper we will focus our attention on the transformation of *logic programs* into equivalent, more efficient ones. We will not consider in detail the problem of transforming specifications written in richer logical languages into executable logic programs. This problem is often addressed within the area of *program synthesis* [Deville-Lau 94], although in the case of logic programming the border line between 'synthesis' and 'transformation' is very thin. Nor will we consider those techniques which improve program performances by transforming logic programs into lower level languages by translating the programs into WAM code and then optimizing that code.

We will mainly be concerned with the so-called unfold/fold program transformations based on the *rules + strategies* approach. This approach consists in starting from an initial program, say P_0, and then applying one or more elementary transformation rules. Thus, we get a sequence $P_0,\ldots, P_n$ of programs. We want the final program P_n to have the same meaning as the initial one, and this objective can be formalized by the equation: $\mathrm{Sem}(P_0) = \mathrm{Sem}(P_n)$ for some given semantics function Sem. This is normally achieved by considering transformation rules which are semantics preserving, that is, for any given programs P and Q, $\mathrm{Sem}(P) = \mathrm{Sem}(Q)$ holds if Q can be derived from P by a single application of one of the rules.

We usually want P_n to be more efficient than P_0. This efficiency improvement is not ensured by an undisciplined use of the transformation rules. This is the reason why we need to introduce some *transformation strategies*, that is, meta-rules which prescribe suitable sequences of applications of the transformation rules.

In logic programming there are many notions of efficiency which have been used. They are related either to the size of the proofs or to the machine model. For each strategy we will briefly explain in what sense program efficiency is improved, and we refer to the original papers for more detailed information.

In Section 2 we introduce in an informal way the unfold/fold transformation rules for logic programs and, in Section 3, we review some correctness results for these rules w.r.t. various semantics functions.

In Section 4 we consider some basic strategies for program transformation and we show, through some examples, how they can be used for improving efficiency. We also give an overview of many related techniques.

In Section 5 we briefly present partial evaluation and program specialization.

In Section 6 we finally analyze the relationship between program transformation and some other methodologies for program development, such as program analysis and synthesis.

2. A PRELIMINARY EXAMPLE

The 'rules + strategies' approach to program transformation was first introduced in [Burstall-Darlington 77] for functional programs considered as sets of recursive equations. This approach is based on the use of two elementary transformation rules: *unfolding* and *folding*.

The unfolding rule consists in replacing in the right hand side of a given equation an instance of the left hand side of an equation, say E1, by the corresponding instance of the right hand side of E1. The application of the unfolding rule can be viewed as a symbolic computation step. It corresponds to the replacement rule used in [Kleene 71, Chapter XI] for the computation of recursive functions.

The folding rule consists in replacing in the right hand side of a given equation an instance of the right hand side of an equation, say E2, by the corresponding instance of the left hand side of E2. Folding can be viewed as the inverse of unfolding, in the sense that, if we perform an unfolding step followed by a folding step we get back the initial equation. Viceversa, unfolding can be viewed as the inverse of folding.

To the reader who is not familiar with the transformation methodology, the usefulness of inverting a symbolic computation step for improving program efficiency might look somewhat obscure. However, as we will see in Section 4, the folding rule allows us to modify the recursive structure of programs and, by doing so, we will often be able to achieve substantial efficiency improvements.

Every transformation process is meaningful only if we specify what is kept unchanged during the transformation itself. In the case of functional programs the unfolding and folding rules preserve the least fixpoint semantics in the following sense: the program P_n derived from a given initial program P_0 after some unfolding and folding steps computes a function which is less defined than or equal to the one computed by P_0 [Kott 78].

We can formalize the relationship between the functional programs P_0 and P_n by introducing a semantics function Sem whose codomain is the set of functions ordered by inclusion. In this formalization we have that $\mathrm{Sem}(P_0) \supseteq \mathrm{Sem}(P_n)$. Thus, in order to preserve the computed function, we need to ensure also that $\mathrm{Sem}(P_0) \subseteq \mathrm{Sem}(P_n)$, that is, the derived program P_n terminates at least as often as the initial program P_0.

Notice that one could have that $\mathrm{Sem}(P_0) \supseteq \mathrm{Sem}(P_n)$ holds and $\mathrm{Sem}(P_0) \subseteq \mathrm{Sem}(P_n)$ does not hold. Let us consider, for instance, the program $\{f(0) \Leftarrow 0,\ f(n+1) \Leftarrow f(n)\}$ which computes the constant function 0 for $n \geq 0$. If we fold the second equation using itself, then we get the program $\{f(0) \Leftarrow 0,\ f(n+1) \Leftarrow f(n+1)\}$ whose least fixpoint is the function, call it g, such that $g(0) = 0$ and $g(n)$ is undefined for $n > 0$.

The unfold/fold transformation approach has been first adapted to logic programs by [Tamaki-Sato 84]. In that paper it is assumed that an unfolding step is a (symbolic) SLD-resolution step and folding is the inverse of unfolding. The notion of inverse, like in the functional case, has to be understood in the sense that an unfolding step followed by the corresponding folding step (and viceversa) gives us back the initial program.

If from a program P_0 we derive by unfold/fold transformations a program P_1 then the least Herbrand model of P_1 is contained in the one of P_0. In this sense we say that the unfold/fold transformations preserve soundness. In general they do *not* preserve completeness, that is, the least Herbrand model of P_0 may be not contained in the one of P_1. In order to preserve completeness one has to comply with some extra conditions [Tamaki-Sato 84].

The discussion on the various semantics which are preserved by unfold/fold transformations will be the objective of the next section.

Let us now consider a preliminary example where we will see in action the unfold/fold rules for transforming logic programs. In this example we will also see the use of one extra transformation rule, called *definition rule*, and the use of a transformation strategy, called *tupling*. We stress the point that we need strategies for driving the application of the transformation rules and improving efficiency, because, since folding is the inverse of unfolding, we may end up with a final program which is equal to the initial one.

Let us consider the following logic program P_0 for computing the average value A of the elements of a list L:

1. average(L,A) ← length(L,N), sumlist(L,S), div(S,N,A)
2. length([],0) ←
3. length([H|T],s(N)) ← length(T,N)
4. sumlist([],0) ←
5. sumlist([H|T],S1) ← sumlist(T,S), sum(H,S,S1)

where div(S,N,A) holds iff A = S/N and sum(H,S,S1) holds iff S1 = H+S.

Both length(L,N) and sumlist(L,S) visit the same list L and we can avoid this double visit by applying the tupling strategy which suggests the introduction of the following clause for the new predicate newp:

6. newp(L,N,S) ← length(L,N), sumlist(L,S).

By adding clause 6 to P_0 we get a new program P_1 which is equivalent to P_0 w.r.t. all predicates occurring in the initial program P_0, in the sense that each ground atom q(…), where q is a predicate occurring in P_0, belongs to the least Herbrand model of P_0 iff q(…) belongs to the least Herbrand model of P_1.

In order to avoid the double occurrence of the list L in the body of clause 1, we now fold it by using clause 6, that is, we replace 'length(L,N), sumlist(L,S)' which is an instance of the body of clause 6, by the corresponding instance 'newp(L,N,S)' of the head of clause 6. Thus, we get:

1f. average(L,A) ← newp(L,N,S), div(S,N,A).

This folding step is the inverse of unfolding newp in the body of clause 1f. However, if we use the program made out of clauses 1f, 2, 3, 4, 5, and 6 we do not avoid the double visit of the list L, because newp is still defined in terms of the individual predicates length and sumlist. A gain in efficiency is possible if we derive a recursive definition of newp in terms of newp itself.

This recursive definition can be obtained as follows. We first unfold clause 6 w.r.t. length(L,N), that is, we derive the following two resolvents of clause 6 using clauses 2 and 3, respectively:

7. newp([],0,S) ← sumlist([],S)
8. newp([H|T],s(N),S) ← length(T,N), sumlist([H|T],S).

We then unfold clauses 7 and 8 w.r.t. sumlist([],S) and sumlist([H|T],S), respectively, and we get:

9. newp([],0,0) ←
10. newp([H|T],s(N),S1) ← length(T,N), sumlist(T,S), sum(H,S,S1).

We can now fold clause 10 using clause 6 and we get:

10f. newp([H|T],s(N),S1) ← newp(T,N,S), sum(H,S,S1).

At this point we may assume that the transformation process is completed. In the final program made out of clauses 1f, 9, 10f, 2, 3, 4, and 5, the double visit of the input list L is avoided and, thus, the efficiency is improved. The initial and final programs have the same least Herbrand model semantics w.r.t. the predicates average, length, and sumlist.

The crucial step in the above program transformation which improves the program performance, is the introduction of clause 6 defining the new predicate newp. In the literature that step is referred to as a *eureka step* and the predicate newp is also called a *eureka predicate*. It can easily be seen that eureka steps *cannot* in general be mechanically performed, because they require a certain degree of ingenuity. There are, however, very many cases in which the synthesis of eureka predicates can be done in an automatic way and this is the reason why in practice the use of the program transformation methodology turns out to be very powerful.

In the following sections we will present in detail the various transformation rules and the semantics they preserve, and we will also present the various transformation strategies. In this context we will consider the problem of the eureka steps and we will see that they can often be performed on the basis of syntactical properties of the program to be transformed.

3. UNFOLD/FOLD RULES FOR LOGIC PROGRAMS

We now present some of the most relevant transformation rules which have been considered in the literature, and we discuss the restrictions one should impose on their use depending on the semantics one would like to preserve.
The notion of program we will use in this paper is very similar to the one of *normal* programs [Lloyd 87] and it is defined as follows.

We assume that all our programs are written using symbols taken from a fixed language L. The Herbrand universe and the Herbrand base are constructed out of L, independently of the programs. This assumption is mainly motivated by the fact that it is often useful to have that the Herbrand base does not change while transforming programs.

An *atom* is a formula of the form $p(t_1,\dots, t_n)$ where p is a predicate symbol of arity n taken from L and $t_1,\dots, t_n$ are terms constructed out of variables, constants, and function symbols in L. A *literal* is either a *positive literal*, that is, an atom, or a *negative literal*, that is, a formula of the form: $\neg A$, where A is an atom.

A *normal clause* is a formula of the form: $H \leftarrow L_1,\dots, L_n$, where the *head* H is an atom and the *body* $L_1,\dots, L_n$ is a (possibly empty) *sequence* (not a *set*) of literals not necessarily distinct. In particular, if $L_1 \neq L_2$ the clauses $H \leftarrow L_1, L_2$ and $H \leftarrow L_2, L_1$ are different (even though their semantics may be the same). The head and the body of a normal clause C are denoted by hd(C) and bd(C), respectively. Comma will be used to denote the associative concatenation of sequences of literals. Thus, $H \leftarrow (L_1,\dots, L_m), (L_{m+1},\dots, L_n)$ is equal to $H \leftarrow L_1,\dots, L_m, L_{m+1},\dots, L_n$.

A *normal goal* is a formula of the form: $\leftarrow L_1,\dots, L_n$, where $L_1,\dots, L_n$ is a (possibly empty) sequence of literals. If $n = 1$ and L_1 is an atom then a normal goal is said to be *atomic*. When no ambiguity arises, we will feel free to identify the notion of goal with that of sequence of literals.

A *normal program* is a *sequence* (not a *set*) of normal clauses.

Normal clauses, normal goals, and normal programs are called *definite clauses*, *definite goals*, and *definite programs*, respectively, if no negative literals occur in them. The qualifications 'normal' and 'definite' will often be omitted when they are irrelevant or understood from the context.

Given the programs $P_1 = <C_1,\dots, C_r>$ and $P_2 = <D_1,\dots, D_s>$, the concatenation $<C_1,\dots, C_r, D_1,\dots, D_s>$ of P_1 and P_2 is denoted by $P_1 @ P_2$. When denoting programs we will feel free to omit angle brackets and commas if they are understood from the context.

The set of variables occurring in a term (or literal, or sequence of literals, or clause)

T is denoted by vars(T). We assume that the variables occurring in the clauses can be freely renamed, as usually done for bound variables in quantified formulas. This is required to avoid clashes of names, like, for instance, when performing SLDNF-resolution steps.

The program transformation process starting from a given initial program P_0, can be viewed as a sequence of programs $P_0,\dots, P_n$, called *transformation sequence*, such that program P_{k+1}, with $0 \leq k \leq n-1$, is obtained from P_k by the application of a transformation rule, which may depend on $P_0,\dots, P_k$.

During the process of program transformation we need to take into account the semantics which is preserved. For the semantics of a normal program we explicitly consider its dependency on the input goal, also called *query*, and thus, we define a *semantics* for a set Programs of normal programs and a set Queries of atomic queries, to be a function Sem: Programs $\times$ Queries $\rightarrow$ (D,$\leq$), where (D,$\leq$) is a partially ordered set. For instance, if P is a program in the set Programs of definite programs, and Q is a query $\leftarrow$ A in the set Queries of atomic queries, then we may take Sem(P,Q) to be the set of instances of A which belong to the least Herbrand model of P. In this case the set D is the powerset of the Herbrand base of the language of Programs and the ordering $\leq$ is set inclusion.

We say that two programs P_1 and P_2 are *equivalent* w.r.t. Sem iff for all queries Q in Queries Sem(P_1,Q) = Sem(P_2,Q).

We now introduce our formal notion of correctness of a transformation sequer w.r.t. a generic semantics function.

DEFINITION 1. (*Correctness of a Transformation Sequence*) Let Sem: Programs $\times$ Queries $\rightarrow$ (D,$\leq$) be a semantics function. A transformation sequence $P_0,\dots, P_n$ of programs in Programs is *partially correct* w.r.t. Sem iff for each query Q in Queries, containing only predicate symbols which occur in P_0, we have that Sem(P_n,Q) $\leq$ Sem(P_0,Q). $P_0,\dots, P_n$ is *totally correct* w.r.t. Sem iff Sem(P_0,Q) = Sem(P_n,Q).

A transformation rule is *partially correct* (*totally correct*) w.r.t Sem iff for any transformation sequence $P_0,\dots, P_k$ which is partially correct (totally correct) w.r.t. Sem and for any program P_{k+1} obtained from P_k by an application of that rule, we have that the transformation sequence $P_0,\dots, P_k$, P_{k+1} is partially correct (totally correct) w.r.t. Sem. ■

Obviously, if $P_0,\dots, P_k$ and $P_k,\dots, P_n$ are partially correct (totally correct) transformation sequences, also their concatenation $P_0,\dots, P_n$ is partially correct (totally correct). In what follows, by ‘correctness’ we will mean ‘total correctness’.

In the remaining part of this Section 3 we present the basic transformation rules and their relevant properties. These rules are collectively called unfold/fold rules and they are a straightforward generalization of those presented in [Tamaki-Sato 84]. In their presentation we will refer to the transformation sequence $P_0,\dots, P_k$. We assume that the variables of the clauses which are involved in each transformation rule are suitably renamed so that they do *not* have variables in common.

We need the following notions. Given a predicate p occurring in a program P, the *definition* of p in P is the subsequence of all clauses in P whose head predicate is p.

A predicate p *depends on* a predicate q in the program P iff *either* there exists in P a clause of the form: p(…) ← Body such that q occurs in Body *or* there exists in P a predicate r such that p depends on r in P and r depends on q in P.

3.1 TRANSFORMATION RULES

R1. Unfolding. Let P_k be the program <E_1,…, E_r, C, E_{r+1},…, E_s> and C be the clause H ← F, A, G, where A is a *positive* literal and F and G are (possibly empty) sequences of literals. Suppose that:

1) <D_1,…, D_n>, with n>0, is the subsequence of *all* clauses in a program P_j, with $0 \le j \le k$, such that A is unifiable with hd(D_1),…, hd(D_n), with most general unifiers θ_1,…, θ_n, respectively, and
2) C_i is the clause (H ← F, bd(D_i), G)θ_i, for i = 1,…, n.

If we *unfold* C *w.r.t.* A *using* D_1,…, D_n *in* P_j we derive the clauses C_1,…, C_n and we get the new program P_{k+1} = <E_1,…, E_r, C_1,…, C_n, E_{r+1},…, E_s>.

When referring to unfolding steps we will often use a simpler terminology, like, for instance, 'to *unfold* C *w.r.t.* A *using* P_j'.

The unfolding rule corresponds to the application of SLD-resolution to clause C with the selection of the positive literal A and the input clauses D_1,…, D_n.

Some early forms of unfolding used in logic programming can be found in [Clark-Sickel 77, Hogger 81, Komorowski 82] in the context of program synthesis and partial evaluation. We do not consider here the unfolding of a clause w.r.t. a *negative* literal, like the one in [Kanamori-Horiuchi 87, Gardner-Shepherdson 91]. However, that kind of unfolding can be expressed in terms of the goal replacement and clause replacement rules introduced below.

Example 1. Suppose that C = p(X) ← q(t(X)), r(X) is a clause in P_k and the definition of q in P_j, with $0 \le j \le k$, consists of the following clauses:

q(a) ←
q(t(b)) ←
q(t(a)) ← r(a).

Then, by unfolding C w.r.t. q(t(X)) using P_j we derive the following two clauses:

p(b) ← r(b)
p(a) ← r(a), r(a)

which are substituted for C in P_k to obtain P_{k+1}. ■

R2. Folding. Let P_k be the program <E_1,…, E_r, C_1,…, C_n, E_{r+1},…, E_s> and <D_1,…,

D_n> be a subsequence of clauses in a program P_j, with $0 \le j \le k$. Suppose that there exists a positive literal A such that, for i = 1,…, n:

1) $hd(D_i)$ is unifiable with A via a most general unifier θ_i,

2) C_i is the clause $(H \leftarrow F, bd(D_i), G)\theta_i$, where F and G are sequences of literals and

3) for any clause D of P_j not in the sequence $<D_1,\ldots, D_n>$, hd(D) is not unifiable with A.

If we *fold* $C_1,\ldots,C_n$ *using* $D_1,\ldots,D_n$ *in* P_j we derive the clause $H \leftarrow F, A, G$, call it C, and we get the new program $P_{k+1} = <E_1,\ldots, E_r, C, E_{r+1},\ldots, E_s>$.

Our folding rule is similar to the one considered in [Maher 89] and it is the inverse of the unfolding rule, in the sense that given a transformation sequence $P_0,\ldots, P_k$, P_{k+1} where P_{k+1} has been obtained from P_k by unfolding, there exists a transformation sequence $P_0,\ldots, P_k$, P_{k+1}, P_k, where P_k has been obtained from P_{k+1} by folding. Analogously, unfolding can be viewed as the inverse of folding.

We would like to stress the point that the possibility of inverting an unfolding step by a folding step and viceversa, depends on the fact that for transforming programs we can use clauses taken from any program of the transformation sequence constructed so far.

Example 2. The clauses

C_1: $p(t(X)) \leftarrow q(X), r(X)$
C_2: $p(u(X)) \leftarrow s(X), r(X)$

can be folded using

D_1: $a(X, t(X)) \leftarrow q(X)$
D_2: $a(X, u(X)) \leftarrow s(X)$

thereby deriving

C: $p(Y) \leftarrow a(X,Y), r(X)$.

Notice that by unfolding clause C using D_1 and D_2 we get again clauses C_1 and C_2. ■

The folding rules considered in [Tamaki-Sato 84, Tamaki-Sato 86, Kawamura-Kanamori 90] are instances of the above one. In particular the folding rule given in [Tamaki-Sato 84] for definite programs can be presented in the context of normal programs as follows.

R3. T&S-folding. Let P_k be the program $<E_1,\ldots, E_r, C, E_{r+1},\ldots, E_s>$ and D be a clause in the program P_j, with $0 \le j \le k$. Suppose also that:

1) C is the clause $H \leftarrow F, bd(D)\theta, G$, such that F, $bd(D)\theta$, and G are sequences of literals, and

2) θ restricted to the set vars(bd(D)) – vars(hd(D)) is a variable renaming whose image has an empty intersection with the set vars(H, F, $hd(D)\theta$, G), and

3) the predicate symbol of hd(D) occurs in P_j only once, that is, in the head of the clause D (thus, D is not recursive).

If we *T&S-fold* C *w.r.t.* bd(D)θ *using* D *in* P_j we derive the clause H ← F, hd(D)θ, G, call it E, and we get a new program P_{k+1} by replacing C by E in P_k.

By applying the T&S-folding rule the derived program P_{k+1} differs from program P_k because of the replacement of exactly one clause by another one.

It is the case that by applying the T&S-folding rule to clause C using a clause D of P_j and then unfolding the resulting clause using D we obtain again (a variant of) C. To get this relationship between T&S-folding and unfolding, the presence of condition 2) in R3 is essential, as shown by the following example.

Example 3. Let C be p(X) ← q(X) and D be r ← q(Y). Suppose that D is the only clause in P_j with head r. Clauses C and D satisfy conditions 1) and 3) of the T&S-folding rule, but they do not satisfy condition 2) because Y does not occur in the head of r ← q(Y), θ = {Y/X}, and X occurs in the head of C. By unfolding the clause p(X) ← r using P_j we get p(X) ← q(Y), which is not a variant of C. ■

It is not the case, however, that by applying a T&S-folding step to a clause, say C1, we can always get back (a variant of) a given clause, say C, even if C1 has been obtained from C by performing an unfolding step using one clause only.

Example 4. Let C be the clause p(X) ← r(X) and D be the clause r(t(X)) ← q(X). From program <C,D>, by unfolding C using D we get the clause C1 = p(t(X)) ← q(X). There are only two ways of applying the T&S-folding rule to C1. The first one is to use clause C1 itself, thereby getting the clause p(t(X)) ← p(t(X)). The second one is to use clause D and if we do so we get the clause p(t(X)) ← r(t(X)). In neither case we get a variant of C. Obviously, from the program <C1,D> we can get again the program <C,D> by the general folding rule R2. ■

R4. Definition Introduction (or **Definition**, for short). We may get program P_{k+1} by concatenating program P_k with a sequence of clauses <p(…) ← $Body_i$ | i = 1,…, n> such that the predicate symbol p does not occur in P_0,…, P_k.

This definition rule is similar to the one in [Maher 89] and it is more general than the definition rule considered in [Tamaki-Sato 84], where only one non-recursive new clause can be introduced (see R15 below).

R5. Definition Elimination. We may get program P_{k+1} by deleting from program P_k all clauses of the definition of a predicate q such that q does not occur in P_0 and no predicate different from q depends on q in P_k.

The definition elimination rule can be viewed as an inverse of the definition introduction rule (modulo the name of the predicate which has been eliminated). It has been presented in [Maher 87, Maher 89] and also in [Bossi-Cocco 90] where it has been called *restriction*.

R6. Goal Replacement. A *replacement law* is a pair S ≡ T, where S and T are sequences of literals. Let $\{X_1,\ldots, X_n\}$ be the set vars(S) ∩ vars(T), and let us consider the following two clauses:

D_S. $p(X_1,\ldots, X_n) \leftarrow S$
D_T. $p(X_1,\ldots, X_n) \leftarrow T$

where p is any new predicate symbol. We say that S ≡ T is *valid* w.r.t. the semantics Sem and program P_k iff Sem(P_k @ <D_S>, ← $p(X_1,\ldots, X_n)$) = Sem(P_k @ <D_T>, ← $p(X_1,\ldots, X_n)$).
Let C = H ← F, S, G be a clause in P_k such that:

1) S ≡ T is a valid replacement law w.r.t. Sem and P_k, and
2) vars(H,F,G) ∩ vars(S) = vars(H,F,G) ∩ vars(T) = $\{X_1,\ldots, X_n\}$.

By *replacement of* S *in* C *using* S ≡ T we derive the clause H ← F, T, G, call it R, and we get P_{k+1} by replacing C by R in P_k.

The relation ≡ defined in the above rule R6 is an equivalence relation.

Our goal replacement rule has been adapted from various versions presented in [Tamaki-Sato 84, Tamaki-Sato 86, Maher 87, Maher 89, Maher 90, Gardner-Shepherdson 91, Bossi et al. 92]. In order to summarize some different cases, our presentation of rule R6 is parametric w.r.t. the semantics function Sem.

Example 5. Let us consider the following clauses in P_k:

```
C.  sublist(N, X, Y) ← length(X, N), append(V, X, W), append(W, Z, Y).
    append([], L, L) ←
    append([H|T], L, [H|TL]) ← append(T, L, TL).
```

Let us assume that, for any definite program P and atomic query ← A, Sem(P, ← A) is the set of instances of A which belong to the least Herbrand model of P.

The replacement law ‘append(V, X, W), append(W, Z, Y) ≡ append(X, L, M), append(K, M, Y)’ (which expresses a weak form of associativity of append) is valid w.r.t. Sem and P_k. Indeed, if we consider the following two clauses:

D_S. p(X, Y) ← append(V, X, W), append(W, Z, Y)
D_T. p(X, Y) ← append(X, L, M), append(K, M, Y)

we have that Sem(P_k @ <D_S>, ← p(X, Y)) = Sem(P_k @ <D_T>, ← p(X, Y)). Thus, by goal replacement of ‘append(V, X, W), append(W, Z, Y)’ in C we derive the following clause:

sublist(N, X, Y) ← length(X, N), append(X, L, M), append(K, M, Y). ■

The validity of a replacement law is in general undecidable. However, if we use totally correct transformation rules only, then for any transformation sequence we need to prove a replacement law only once. Indeed, if $S \equiv T$ is valid w.r.t. a semantics Sem and program P, then $S \equiv T$ is valid w.r.t. Sem and Q for every program Q derived from P by using totally correct transformation rules.

In order to prove the validity of a replacement law, there are some ad hoc proof methods depending on the specific semantics which is considered (see Section 6).

A simple method which is parametric w.r.t. the chosen semantics is based on unfold/fold transformations. It was introduced by Kott for recursive equation programs [Kott 82] and its application to logic programs is described in [Boulanger-Bruynooghe 93, Proietti-Pettorossi 93b]. Given the replacement law $S \equiv T$, we consider the two clauses D_S and D_T which are defined from S and T as specified above, and we construct two correct transformation sequences: P @ $<D_S>,\ldots, P_Z$ and P @ $<D_T>,\ldots, P_Z$, for some final program P_Z. Thus, Sem(P @ $<D_S>, \leftarrow p(X_1,\ldots, X_n)) =$ Sem($P_Z, \leftarrow p(X_1,\ldots, X_n)) =$ Sem(P @ $<D_T>, \leftarrow p(X_1,\ldots, X_n)$), and the validity of $S \equiv T$ w.r.t. Sem and P is proved.

For instance, the validity of the replacement law considered in Example 5 can be proved as shown in the following example.

Example 6. Consider again program P_k of Example 5 and the clauses:

D_S. p(X, Y) ← append(V, X, W), append(W, Z, Y)
D_T. p(X, Y) ← append(X, L, M), append(K, M, Y).

In order to prove that Sem(P_k @ $<D_S>$, ← p(X, Y)) = Sem(P_k @ $<D_T>$, ← p(X, Y)) we will construct two transformation sequences, the first one starting from P_k @ $<D_S>$ and the second one starting from P_k @ $<D_T>$. Their correctness w.r.t. suitable semantics functions will be shown in Section 3.2. As a consequence the replacement law 'append(V, X, W), append(W, Z, Y) ≡ append(X, L, M), append(K, M, Y)' is valid w.r.t. those semantics functions.

The first transformation sequence starting from P_k @ $<D_S>$ is constructed as follows. We unfold clause D_S w.r.t. append(V, X, W) and we derive the following two clauses:

D1. p(X, Y) ← append(X, Z, Y)
D2. p(X, Y) ← append(T, X, T1), append([H|T1], Z, Y).

We now unfold clause D2 w.r.t. append([H|T1], Z, Y) and we get:

D3. p(X, [H|T2]) ← append(T, X, T1), append(T1, Z, T2).

Then we fold clause D3 using clause D_S and we derive:

D3f. p(X, [H|T2]) ← p(X, T2).

Thus, from P_k @ $<D_S>$ we derive the final program of the transformation sequence, which is P_k @ <D1, D3f>.

The second transformation sequence starts from P_k @ $<D_T>$. We first unfold clause D_T w.r.t. append(K, M, Y). We derive two clauses:

D4. p(X, Y) ← append(X, L, Y)
D5. p(X, [H|U]) ← append(X, L, M), append(T, M, U).

By folding clause D5 using clause D_T we get:

D5f. p(X, [H|U]) ← p(X, U).

Thus, the final program of this transformation sequence is P_k @ <D4, D5f>, which is equal to P_k @ <D1, D3f> up to variable renaming. ■

We finally present the clause replacement transformation rule. Similarly to the goa replacement rule, we have chosen a presentation which is parametric w.r.t. th semantics function Sem, so that we can give account of the different rules considere in the literature [Tamaki-Sato 84, Maher 87, Maher 89, Maher 90, Bossi-Cocco 90 Gardner-Shepherdson 91, Proietti-Pettorossi 91a]. Its applicability condition is ir general undecidable, as for the goal replacement rule. However, in the followin Section 3.2 we will show some useful instances of the clause replacement rule whos applicability conditions can be effectively tested.

R7. Clause Replacement. From $P_k = <E_1,\ldots, E_r, C_1,\ldots, C_n, E_{r+1},\ldots, E_s>$ we get $P_{k+1} = <E_1,\ldots, E_r, D_1,\ldots, D_m, E_{r+1},\ldots, E_s>$ if for every query Q containing only predicates occurring in $P_k \cup P_{k+1}$ we have that $Sem(P_k, Q) = Sem(P_{k+1}, Q)$.

3.2 SEMANTICS PRESERVING TRANSFORMATIONS FOR DEFINITE PROGRAMS

We now consider programs without negative literals in the bodies of their clauses and we discuss the correctness of the transformation rules w.r.t. various semantics functions. We will first review the correctness properties of unfold/fold transformations w.r.t. both the least Herbrand model and the computed answer substitution semantics. We then take into account semantics functions related to program termination, such as the finite failure semantics and the answer substitutions semantics computed by the depth-first search strategy of Prolog.

3.2.1 LEAST HERBRAND MODEL

In this section we assume that the semantics function is based on the concept of least Herbrand model of a definite program. This function, call it Sem_H, has type: Programs × Queries → (D,≤), where Programs is the set of definite programs, Queries is the set of atomic queries, and (D,≤) is the powerset of the Herbrand base ordered by set inclusion.

As already mentioned, when considering least Herbrand models of programs we assume that we are given a fixed language L, so that the Herbrand base does not change during the transformation sequences. In particular, if we introduce a predicate, say p, not occurring in a given program, by applying rule R4, then we assume that p is in L.

$\mathrm{Sem}_H(P, \leftarrow A)$ is defined as the set of atoms in the least Herbrand model of P which are instances of A.

Let us now consider the following instances of the goal and clause replacement rules, whose correctness w.r.t. Sem_H can easily be checked.

R8. Goal Rearrangement. We get P_{k+1} from P_k by replacing the goal G,H in a clause of P_k by the goal H,G using the replacement law G,H ≡ H,G.

R9. Deletion of Duplicate Goals. We get P_{k+1} from P_k by replacing the goal G,G in a clause of P_k by the goal G using the replacement law G,G ≡ G.

From rules R8 and R9 it follows that the body of a clause can be considered as a set of atoms. (Recall that we have already assumed that comma is associative.)

R10. Clause Rearrangement. We get P_{k+1} by replacing the sequence of clauses <C, D> in P_k by <D, C>.

R11. Deletion of Subsumed Clauses. A clause C is *subsumed* by D iff there exist a substitution θ and a sequence of atoms S such that $hd(C) = hd(D)\theta$ and $bd(C) = bd(D)\theta$, S. We get P_{k+1} by deleting from P_k a clause which is subsumed by another clause in P_k.

Obviously, rule R11 allows us to delete duplicate clauses.

R12. Deletion of Clauses with Finitely Failed Body. Let C be a clause in P_k of the form $H \leftarrow A_1,\ldots, A_m, L, B_1,\ldots, B_n$ with $m,n \geq 0$. If L has a finitely failed SLD-tree in P_k, then we say that C has a *finitely failed body in* P_k and we get P_{k+1} by deleting C from P_k.

The above five replacement rules R8,…, R12 will collectively be called *boolean rules*.

Notice that, rules R8, R9, R10, and R11, are implicitly used when considering programs as *sets* of clauses, and bodies of clauses as *sets* of literals. On the contrary, as already mentioned, in this paper we consider programs as sequences of clauses and bodies of clauses as sequences of literals, and we have to make an explicit use of rules R8, R9, R10, and R11, when needed. Our choice is motivated by the fact that some instances of these rules are not correct when considering the computed answer substitution semantics (see Section 3.2.2) or the pure Prolog semantics (see Section

3.2.4).

THEOREM 2 [Tamaki-Sato 84]. Every transformation sequence constructed by using the rules of unfolding, definition introduction, definition elimination, and clause replacement is totally correct w.r.t. Sem_H.

PROOF. By the soundness and completeness of SLD-resolution we get th correctness of unfolding. The one of the definition introduction and defi elimination is a consequence of the fact that the notion of correctnes transformation sequence is defined w.r.t. queries containing only predicate syn which occur in the initial program. The total correctness of the clause replacem is a straightforward consequence of the definitions. ■

Nothing can be said about the total correctness of a transformation sequence $P_0,\ldots, P_n$ containing folding and goal replacement steps different from R8 and R9. Indeed, for P_k and P_{k+1}, with $0 \leq k < n$, it may happen that either $Sem_H(P_k, \leftarrow A) < Sem_H(P_{k+1}, \leftarrow A)$ or $Sem_H(P_k, \leftarrow A) > Sem_H(P_{k+1}, \leftarrow A)$, where $<$ means $\leq$ and $\neq$.

For instance, consider the following transformation sequence:

P_0: $p \leftarrow q$ $\quad q \leftarrow$
P_1: $p \leftarrow p$ $\quad q \leftarrow$ $\quad$ (by folding, or goal replacement, since $p \equiv q$ is valid in P_0)
P_2: $p \leftarrow q$ $\quad q \leftarrow$ $\quad$ (by unfolding using clauses in P_0)

Thus, we have derived the program P_2 equal to program P_0.

We have that $Sem_H(P_0, \leftarrow p) > Sem_H(P_1, \leftarrow p)$ and $Sem_H(P_1, \leftarrow p) < Sem_H(P_2, \leftarrow p)$.

However, the reader may verify that, by applying the folding rule or the goal replacement rule to the program P_k of a transformation sequence $P_0,\ldots,P_k$, we derive clauses which are true in the least Herbrand model of P_0. Thus, we have the following result, which generalizes the one of [Tamaki-Sato 84] which was stated for a weaker version of the transformation rules.

THEOREM 3. (*Partial Correctness of Transformations w.r.t.* Sem_H) Every transformation sequence constructed by using the rules R1,…, R12 is partially correct w.r.t. the semantics Sem_H. ■

From Theorem 3 it follows that, if there exist a transformation sequence $P_0, P_1,\ldots, P_{n-1}, P_n$ constructed by using the set of rules R1,…, R12 and a transformation sequence $P_n, Q_1,\ldots, Q_k, P_0$, with $k \geq 0$, constructed by using the same set of rules, then both sequences are totally correct w.r.t. Sem_H. This property suggests the introduction of the notion of *reversible transformation sequence*, which can be stated w.r.t. any set R of transformation rules.

DEFINITION 4. Let R be a set of transformation rules. A transformation sequence P_0, $P_1,\ldots, P_{n-1}, P_n$ is said to be *reversible* w.r.t. R iff there exists a transformation sequence $P_n, Q_1,\ldots, Q_k, P_0$, with $k\geq 0$, which can be constructed by using the same set R.

A transformation rule which belongs to R, is said to be reversible w.r.t. R iff every transformation sequence P_0, P_1 obtained by applying that rule to any given program P_0, is reversible w.r.t. R. ■

Notice that, the construction of the transformation sequence $P_n, Q_1,\ldots, Q_k, P_0$ should be independent of the construction of the transformation sequence $P_0, P_1,\ldots, P_{n-1}, P_n$. This independence condition is crucial because, in general, we can derive a new program by using clauses occurring in a program which is not the last one of the transformation sequence at hand. Thus, it may be the case that there exists a transformation sequence $P_0, P_1,\ldots, P_{n-1}, P_n, \ldots, P_0$, but there is no transformation sequence $P_n, Q_1,\ldots, Q_k, P_0$, that is, $P_0, P_1,\ldots, P_{n-1}, P_n$ is not reversible.

THEOREM 5. Let Sem be a semantics function and R a set of transformation rules which are partially correct w.r.t. Sem. If a transformation sequence constructed using R is reversible, then it is totally correct w.r.t. Sem. ■

In general, it is hard to check whether or not a transformation sequence is reversible. However, we have that the clause replacement rule is reversible w.r.t. itself (as a simple consequence of its definition), and rules R13 and R14 which we will introduce below, are reversible w.r.t. any set of rules including {R1, R2, R6}. Therefore, by Theorems 3 and 5 every transformation sequence P_0, P_1 obtained by applying rule R13 or rule R14 to a given program P_0 is totally correct w.r.t. Sem_H.

These rules R13 and R14 are instances of the folding and goal replacement rules, respectively.

R13. Reversible Folding. A folding step of clauses $C_1,\ldots,C_n$ in P_k using $D_1,\ldots,D_n$ in program P_j is said to be a *reversible folding* iff j=k and $\{C_1,\ldots,C_n\} \cap \{D_1,\ldots,D_n\} = \emptyset$.

Let C be the clause derived by applying reversible folding to clauses $C_1,\ldots, C_n$ in P_k using $D_1,\ldots, D_n$, and let P_k, P_{k+1} be the resulting transformation sequence. We have that also P_{k+1}, P_k is a transformation sequence, because $D_1,\ldots, D_n$ are in P_{k+1} and by unfolding C using $D_1,\ldots, D_n$ in P_{k+1} we get P_k again.

Example 7. By reversible folding from the following program:

P_0: $p \leftarrow q, r$ $\quad q \leftarrow$ $\quad r \leftarrow$ $\quad s \leftarrow q$

we get:

P_1: $p \leftarrow s, r$ $\quad q \leftarrow$ $\quad r \leftarrow$ $\quad s \leftarrow q$.

Notice that by unfolding the first clause of P_1 w.r.t. s we get again P_0. ∎

Various instances of the reversible folding rule have been proposed in [Maher 8 Maher 89, Gardner-Shepherdson 91].

As already mentioned, folding is not a totally correct rule w.r.t. Sem_H, and thus, it not reversible w.r.t. the set of rules made out of unfolding, folding, definition introduction, definition elimination, and boolean rules R8,…, R12, which are partially correct w.r.t. Sem_H. Analogous statement holds if we refer to T&S-folding, instead of folding.

In rule R13 we have indicated some sufficient conditions which make the folding rule to be a reversible rule. These conditions are particularly useful because they can be syntactically checked.

In what follows, by an application of the reversible folding rule we will mean an application of rule R13, rather than an application of the folding rule which produces a reversible transformation sequence.

Unfortunately, the reversibility restriction to the folding rule seriously limits its power. For instance, for the derivation of the recursive definition of the predicate average in the example of Section 2, we have performed a folding step which is not a reversible folding. The following example shows that the set of rules consisting of unfolding and reversible folding is strictly less powerful than the one consisting of unfolding and (totally correct) folding.

Example 8. Let us consider the following two programs:

P_0:	$p \leftarrow q, r$	$q \leftarrow q$	$r \leftarrow r$
P_1:	$p \leftarrow p$	$q \leftarrow q$	$r \leftarrow r$

P_1 can be obtained from P_0 by a folding step (which is a T&S-folding step). This folding step is totally correct because P_0 and P_1 are equivalent w.r.t. Sem_H. On the other hand, it is impossible to derive P_1 from P_0 by using unfolding and reversible folding only. Indeed, if we apply unfolding or reversible folding to any clause in P_0 we get again P_0. ∎

R14. Reversible Goal Replacement. Let C be a clause in P_k and $S \equiv T$ be a valid replacement law w.r.t. the semantics Sem and program P_k. The replacement of S in C using $S \equiv T$ is said to be a Sem-*reversible goal replacement* if $S \equiv T$ is valid w.r.t. Sem and the derived program P_{k+1}.

Suppose that by replacement of S in C using $S \equiv T$ we derive a clause R and we get the program $P_{k+1} = (P_k - \{C\}) \cup \{R\}$ and $S \equiv T$ is valid w.r.t. Sem and P_{k+1}. Then $T \equiv S$ is valid w.r.t. Sem and P_{k+1}, and by replacement of T in R using $T \equiv S$ we get P_k again. Thus, the conditions indicated in R14 are sufficient to ensure that goal replacement is reversible w.r.t. {R14}.

Similarly to the case of folding, by an application of the reversible goal replacement rule we will mean an application of the rule R14.

Rules R8 and R9 are particular instances of Sem_H-reversible goal replacements.

In what follows we will feel free to say 'reversible goal replacement', instead of 'Sem-reversible goal replacement', when the semantics function Sem is understood from the context.

Example 9. Consider again the programs P_0 and P_1 of Example 8. The folding step which produces P_1 from P_0 can also be viewed as a goal replacement step, because $p \equiv q, r$ is valid w.r.t. Sem_H and P_0. Since $p \equiv q, r$ is valid also w.r.t. Sem_H and P_1, the transformation step from P_0 to P_1 can be viewed as an application of the Sem_H-reversible goal replacement rule, and therefore, it is correct w.r.t. Sem_H. ■

As a summary of the above considerations we have the following result.

THEOREM 6. [Maher 87] Let $P_0,\ldots, P_n$ be a transformation sequence of definite programs, constructed by using the following transformation rules: unfolding, reversible folding, definition introduction, definition elimination, Sem_H-reversible goal replacement (including R8 and R9), clause replacement (including R10, R11, and R12). Then $P_0,\ldots, P_n$ is correct w.r.t. the semantics Sem_H. ■

We have seen that the reversible folding rule R13 has the advantage of be totally correct transformation rule, but as already mentioned, it is a weak rule Example 8). In order to overcome this limitation we now present a more pov folding rule which is *not* an instance of R13 and yet it is totally correct w.r.t. Sem_I The correctness of this new rule is ensured by an easily verifiable condition on the transformation sequence.

Let us first notice that by performing a folding step we may introduce recursive clauses from non-recursive ones and some infinite computations may replace finite ones, thereby affecting the semantics of the program and loosing total correctness.

A simple example of this undesired introduction of infinite computations is *folding,* where all clauses in a predicate definition can be folded using themselves. For instance, the definition $p \leftarrow q$ of a predicate p can be replaced (using T&S-folding) by $p \leftarrow p$.

This inconvenience can be avoided by ensuring that 'enough' unfolding steps have been performed before folding, so that 'going backward in the computation' (as folding does) does not prevail over 'going forward in the computation' (as unfolding does). This idea is the basis of various techniques in which total correctness is ensured by counting the number of unfolding and folding steps performed during the transformation sequence [Kott 78, Kanamori-Fujita 86, Bossi et al. 92b, Amtoft 92].

For the presentation of the powerful folding rule we have promised above, we need the following assumptions [Seki 91]. We assume that all predicate symbols occurring in each program of a transformation sequence $P_0,\ldots, P_n$ are partitioned into the set $Pred_{new}$ of *new predicates* and the set $Pred_{old}$ of *old predicates.* New predicates are the ones which *either* occur in the head of exactly one clause of P_0 and not elsewhere in P_0 *or* they are in the head of clauses introduced by applying the T&S-definition rule (see below).

The distinction between new and old predicates could be generalized in a way similar to the one presented in [Tamaki-Sato 86], where the set of predicates of the initial program is partitioned into an arbitrary number of levels so that the level of a predicate symbol in the body of a clause is not greater than the level of the head of the clause.

R15. T&S-definition. Given a transformation sequence $P_0,\ldots, P_k$, we may get a new program P_{k+1} by adding to program P_k a clause $H \leftarrow$ Body such that:

1) the predicate of H does not occur in $P_0,\ldots, P_k$, and
2) Body is made out of literals with old predicates occurring in $P_0,\ldots, P_k$.

Notice that, due to T&S-folding steps a new predicate may occur in the body of a clause whose head has an old predicate.

In order to describe some conditions which ensure the total correctness of T&S-folding (see Theorem 8 below) we need to take into account whether or not an atom has been generated by unfolding during a transformation sequence. This motivates the introduction of the following notion which we describe in the case of normal programs because we need to use it also later in Section 3.3.

DEFINITION 7. (*Fold-allowing Occurrences of Literals*) Let $P_0,\ldots, P_n$ be a transformation sequence of normal programs constructed by using the following rules: unfolding, T&S-folding, T&S-definition, definition elimination, and boolean rules (that is, R8, …, R12). Let D be a clause in P_i with $0 \leq i \leq n$.

Case i = 0. An occurrence of a literal L in bd(D) is *fold-allowing* iff L is positive and hd(D) has an old predicate.

Case 0<i ≤n.

1) For each clause C in P_i which is not involved in the derivation from P_{i-1} to P_i each literal of bd(C) in P_i is *fold-allowing* iff so is the same literal of bd(C) in P_{i-1}.
2) Suppose that D has been derived by *unfolding* a clause C in P_{i-1} w.r.t. a positive literal A. Thus, C and D are of the form: $H \leftarrow B_1,\ldots, B_k, A, B_{k+1},\ldots, B_m$ and $(H \leftarrow B_1,\ldots, B_k, \mathrm{bd}(E), B_{k+1},\ldots, B_m)\theta$, respectively, where E is a clause such that hd(E) is unifiable with A via a most general unifier θ. In D the literals occurring in $\mathrm{bd}(E)\theta$ are *fold-allowing* and for $r = 1,\ldots, m$ the literal $B_r\theta$ is *fold-allowing* iff so is B_r in bd(C).
3) Suppose that D has been derived by *T&S-folding* C in P_{i-1}. Thus, C and D are of the form: $H \leftarrow B_1,\ldots, B_k, \mathrm{bd}(E)\theta, B_{k+1},\ldots, B_m$ and $H \leftarrow B_1,\ldots, B_k, \mathrm{hd}(E)\theta, B_{k+1},\ldots, B_m$, respectively, for a clause E and a substitution θ. In bd(D) the literal $\mathrm{hd}(E)\theta$ is *fold-allowing* and for $r = 1,\ldots, m$ the literal B_r is *fold-allowing* iff so is B_r in bd(C).

4) Suppose that D has been derived by applying the T&S-definition rule. Then no literal in bd(D) is *fold-allowing*.
5) Suppose that D has been derived by applying rule R8 (goal rearrangement) to C in P_{i-1}. Thus, C and D are of the form: $H \leftarrow B_1,\ldots, B_{k-1}, B_k, B_{k+1}, B_{k+2},\ldots, B_m$ and $H \leftarrow B_1,\ldots, B_{k-1}, B_{k+1}, B_k, B_{k+2},\ldots, B_m$, respectively. For $r = 1,\ldots, m$ the literal B_r in bd(D) is *fold-allowing* iff so is B_r in bd(C).
6) Suppose that D has been derived by applying rule R9 (deletion of duplicate goals) to C in P_{i-1}. Thus, C and D are of the form: $H \leftarrow B_1,\ldots, B_k, L, L, B_{k+1},\ldots, B_m$ and $H \leftarrow B_1,\ldots, B_k, L, B_{k+1},\ldots, B_m$, respectively. In bd(D) the literal L is *fold-allowing* iff so is at least one occurrence of L in bd(C). For $r = 1,\ldots, m$ the literal B_r in bd(D) is *fold-allowing* iff so is B_r in bd(C).
7) Suppose that P_i has been derived from P_{i-1} by applying rule R10 (clause rearrangement). Thus, they are of the form: $\ldots, C_1, C_2, \ldots$ and $\ldots, C_2, C_1, \ldots$, respectively. For j=1,2 each literal occurring in bd(C_j) in P_i is *fold-allowing* iff so is the same literal occurring in bd(C_j) in P_{i-1}. ■

One can easily show that *inherited literals* defined in [Seki 91] are exactly the literals which are not fold-allowing according to the above Definition 7.

THEOREM 8. [Tamaki-Sato 84] (*Correctness of T&S-folding w.r.t.* Sem_H) Let $P_0,\ldots, P_n$ be a transformation sequence of definite programs, constructed by using the following transformation rules: unfolding, T&S-folding, T&S-definition, definition elimination, and boolean rules. Suppose that no T&S-folding step is performed after a definition elimination step. Suppose also that we apply T&S-folding to a clause C using a clause D only if i) hd(D) has a new predicate and ii) either hd(C) has an old predicate or *at least one* atom in bd(C) is fold-allowing. Then $P_0,\ldots, P_n$ is correct w.r.t. the semantics Sem_H. ■

The hypothesis that no T&S-folding step is performed after a definition elimination step is needed to prevent that a T&S-folding step is applied using a clause with a head predicate whose definition has been eliminated. This point is illustrated by the following example.
Example 10. Let us consider the transformation sequence:

$p \leftarrow q$	$p \leftarrow fail$	$q \leftarrow$		
$p \leftarrow q$	$p \leftarrow fail$	$q \leftarrow$	$newp \leftarrow q$	(by T&S-definition)
$p \leftarrow q$	$p \leftarrow fail$	$q \leftarrow$		(by definition elimination)
$p \leftarrow newp$	$p \leftarrow fail$	$q \leftarrow$		(by T&S-folding)

According to our definitions newp is a new predicate and p is an old one. Thus, hypotheses i) and ii) of Theorem 8 above are fulfilled. However, the transformation sequence is not correct w.r.t. Sem_H. ■

Notice that when we T&S-fold clause C w.r.t. a sequence of atoms in bd(C), no atom in that sequence is required to be fold-allowing.

The above theorem can be used to show the correctness of the transformation process presented in Section 2, where average is the only new predicate. Unfortunately, Theorem 8 does not ensure the correctness of a transformation sequence where we allow general Sem_H-reversible goal replacement steps. However, we may construct transformation sequences containing both Sem_H-reversible goal replacement steps and folding steps which are not instances of R13, by concatenating several transformation sequences, each of them being proved correct either by Theorem 6 or by Theorem 8.

The reader may find in [Tamaki-Sato 84, Tamaki-Sato 86, Gardner-Shepherdson 91] some other restricted forms of the folding and goal replacement rules which are correct w.r.t. Sem_H.

3.2.2 COMPUTED ANSWER SUBSTITUTIONS

We now consider the semantics function Sem based on the notion of *cc answer substitutions* [Lloyd 87]. It captures the procedural behaviour of definite programs more accurately than it is done by the least Herbrand model semantics.

Two substitutions η and θ are said to be *equal modulo renaming* iff there exists a renaming substitution ρ such that η is equal to the restriction of $\theta\,\rho$ to the domain of θ. In what follows we will always consider substitutions modulo renaming.

The computed answer substitution semantics is a function Sem_{CA}: Programs $\times$ Queries $\rightarrow$ $(D,\leq)$, where Programs is the set of definite programs, Queries is the set of atomic queries, and $(D,\leq)$ is the powerset of the set of all substitutions (modulo renaming) ordered by set inclusion. By definition, we have that $\text{Sem}_{CA}(P, \leftarrow A) = \{\theta \mid$ there exists an SLD-refutation of $\leftarrow A$ with computed answer substitution $\theta\}$.

By soundness and completeness of SLD-resolution, we have that the equivalence w.r.t. Sem_{CA} implies the equivalence w.r.t. Sem_H. However, the converse is not true. For instance, consider the following two programs:

P_1: $p(a) \leftarrow$
P_2: $p(X) \leftarrow$ $\qquad$ $p(a) \leftarrow$

We have that P_1 and P_2 have the same least Herbrand model $\{p(a)\}$. However, $\text{Sem}_{CA}(P_1, \leftarrow p(X)) = \{\{X/a\}\}$, while $\text{Sem}_{CA}(P_2, \leftarrow p(X)) = \{\{\},\{X/a\}\}$, where $\{\}$ is the *identity* substitution.

Various researchers have addressed the problem of proving the correctness of some transformation rules w.r.t. Sem_{CA} [Kawamura-Kanamori 90, Bossi-Cocco 90, Bossi et al. 92b]. It can easily be shown that rules R8,…, R12 preserve the Sem_{CA} semantics with the exception of rule R9 (deletion of duplicate goals) and rule R11 (deletion of subsumed clauses), as it is shown in the following example.

Example 11. Let us consider the program:

P_1: $p(X) \leftarrow q(X), q(X)$ $\qquad$ $q(t(Y,a)) \leftarrow$ $\qquad$ $q(t(a,Z)) \leftarrow$

By deleting an occurrence of q(X) in the body of the first clause we get:

P_2: p(X) ← q(X) q(t(Y,a)) ← q(t(a,Z)) ←

The substitution {X/t(a,a)} belongs to $Sem_{CA}(P_1$, ← p(X)) and not to $Sem_{CA}(P_2$, ← p(X)).

Let us now consider the program:

P: p(X) ← p(a) ←

The clause p(a) ← is subsumed by p(X) ←. However, if we delete p(a) ← the Sem_{CA} semantics is not preserved, because {X/a} is no longer a computed answer substitution for the query ← p(X). ■

There are particular cases where the deletion of duplicate goals and the deletion of subsumed clauses are correct w.r.t. Sem_{CA}, and indeed the following two rules are correct w.r.t. Sem_{CA}.

R16. Deletion of Duplicate Ground Goals. We get program P_{k+1} from progran P_k by replacing a ground goal G,G in a clause of P_k using the replacement law G,G ≡ G.

This rule is an instance of the Sem_{CA}-reversible goal replacement rule.

R17. Deletion of Duplicate Clauses. We get program P_{k+1} by replacing t sequence of clauses <C, C> in program P_k by <C>.

For the correctness of a transformation sequence w.r.t. Sem_{CA} we have the following results, corresponding to Theorems 6 and 8, respectively.

THEOREM 9. Let $P_0,\ldots, P_n$ be a transformation sequence of definite programs, constructed by using the following transformation rules: unfolding, reversible folding, definition introduction, definition elimination, reversible goal replacement R14 (in particular, rules R8 and R16), and clause replacement R7 (in particular, rules R10, R12, and R17). Then $P_0,\ldots, P_n$ is correct w.r.t. Sem_{CA}. ■

THEOREM 10. [Kawamura-Kanamori 90, Bossi-Cocco 90]. (*Correctness of T&S-folding w.r.t.* Sem_{CA}) Let $P_0,\ldots, P_n$ be a transformation sequence constructed by using the following transformation rules: unfolding, T&S-folding, T&S-definition, definition elimination, goal rearrangement, deletion of duplicate ground goals, clause rearrangement, deletion of duplicate clauses, and deletion of clauses with finitely failed body. Suppose that no T&S-folding step is performed after a definition elimination step. Suppose also that we apply T&S-folding to a clause C using a clause D only if i) hd(D) has a new predicate and ii) either hd(C) has an old predicate or *at least one* atom in bd(C) is fold-allowing. Then $P_0,\ldots, P_n$ is correct w.r.t. the semantics Sem_{CA}. ■

3.2.3 FINITE FAILURE

In Theorems 6, 8, 9, and 10 we have shown that the *set* of the atomic consequences of a program and the *set* of answer substitutions that are computed by a program are preserved by a number of transformations. However, the use of the rules according to the hypotheses of Theorem 8 may transform a finitely failing program into an infinitely failing one (and viceversa), as shown by the following example.

Example 12. Let us consider the transformation sequence, where p is the only new predicate:

P_0: $p(X) \leftarrow q(X), r(X)$ $q(a) \leftarrow$ $r(b) \leftarrow r(b)$
P_1: $p(b) \leftarrow q(b), r(b)$ $q(a) \leftarrow$ $r(b) \leftarrow r(b)$
(by unfolding the first clause w.r.t. r(X))
P_2: $p(b) \leftarrow p(b)$ $q(a) \leftarrow$ $r(b) \leftarrow r(b)$
(by T&S-folding the first clause).

This transformation sequence satisfies the conditions stated in Theorem 8, but P_0 finitely fails for the query $\leftarrow p(b)$, while P_2 does not. ■

In order to reason about the preservation of finite failure during program transformation we now consider the semantics function Sem_{FF} from Programs $\times$ Queries to $(D,\leq)$, where Programs is the set of definite programs, Queries is the set of atomic queries, and $(D,\leq)$ is the powerset of the set of (possibly not ground) atoms ordered by set inclusion. By definition, $Sem_{FF}(P, \leftarrow A) = \{B \mid B \text{ is an instance of } A$ and there exists a finitely failed SLD-tree for P and $\leftarrow B\}$.

As a result of soundness and completeness of (fair) SLD-resolution w.r.t. finite failure [Lloyd 87], we easily get the partial correctness of our transformation rules R1, R4, …, R14 w.r.t. Sem_{FF}. (Example 10 shows that rules R2 and R3, together with definition elimination, are not partially correct w.r.t. Sem_{FF}.) Thus, similarly to the cases of Sem_H and Sem_{CA}, we have the following result, basically due to [Maher 87].

THEOREM 11. Let $P_0,\dots, P_n$ be a transformation sequence of definite programs, constructed by using the transformation rules: unfolding, reversible folding, definition introduction, definition elimination, reversible goal replacement (in particular, rules R8 and R9), and clause replacement (in particular, rules R10, R11, and R12). Then $P_0,\dots, P_n$ is correct w.r.t. Sem_{FF}. ■

If we allow folding steps which are not reversible foldings, it may be the case that a folding step affects the fairness of SLD-derivations, because as we will show below, it imposes a ‘synchronized’ evaluation of a sequence of atoms. Thus, given a program P_1 and a query Q, by applying folding steps which are not reversible, we may derive a program P_2 such that a fair SLD-derivation for Q using P_2 *encodes* an unfair SLD-derivation for Q using P_1.

Let us consider, for instance, program P_1 of Example 12 and the infinite sequence of goals: $\leftarrow$ p(b), $\leftarrow$ p(b),… , which describes the SLD-derivation for the program P_2 and the query $\leftarrow$ p(b). Since the folding step which produced P_2 from P_1 replaces 'q(b), r(b)' by 'p(b)', this derivation can be viewed as an encoding of the unfair SLD-derivation for P_1:

$\leftarrow$ p(b), $\leftarrow$ q(b), r(b), $\leftarrow$ q(b), r(b), …

which is obtained by selecting for SLD-resolution always the atom r(b).

The following Theorem 12 is a modification of Theorem 8. Its proof is based on the fact that unfair SLD-derivations cannot be introduced if *all* atoms replaced in a folding step, have previously been derived by unfolding. This condition is not fulfilled by the folding step shown in Example 12 because in the body of the clause p(b) $\leftarrow$ q(b), r(b) in P_1 the atom q(b) has *not* been derived by unfolding, or in the sense of Definition 7, q(b) is not fold-allowing.

THEOREM 12. [Seki 91]. (*Correctness of T&S-folding w.r.t.* Sem_{FF}) Let $P_0,\ldots,P_n$ be a transformation sequence of definite programs, constructed by using the following transformation rules: unfolding, T&S-folding, T&S-definition, definition elimination, and boolean rules. Suppose that no T&S-folding step is performed after a definition elimination step. Suppose also that we apply T&S-folding to a clause C using a clause D only if i) hd(D) has a new predicate and ii) either hd(C) has an old predicate or *all* atoms of bd(C) w.r.t. which T&S-folding steps are performed, are fold-allowing. Then $P_0,\ldots,P_n$ is correct w.r.t. the semantics Sem_{FF}. ■

3.2.4 PURE PROLOG

In this section we consider the case where a definite program is evaluated using a Prolog evaluator. Its control strategy can be described as follows. The SLD-tree for a given program and a given query, is constructed by using the *left-to-right rule* for selecting the atom w.r.t. which SLD-resolution should be applied in a given goal. In this SLD-tree, the sons of a given goal are ordered from left to right according to the order of the clauses used for performing the corresponding SLD-resolution step. Thus, we have an ordered SLD-tree which is visited in a depth-first manner. The use of the Prolog control strategy has two consequences: i) the answer substitutions are generated in a fixed order, possibly with repetitions, and ii) there may be some answer substitutions which cannot be obtained in a finite number of computation steps, because in the depth-first visit they are 'after' branches of infinite length. Therefore the completeness of SLD-resolution is lost.

We will define a semantics function Sem_{Prolog} by taking into consideration the generation order, the multiplicity, and the finite time computability of the answer substitutions. Thus, given a program P and a query Q, we consider the ordered SLD-tree T constructed as specified above. The left-to-right ordering of the brother nodes in T determines the left-to-right ordering of the branches and leaves.

If T is finite then $Sem_{Prolog}(P,Q)$ is the sequence of the computed answer substitutions (modulo renaming) corresponding to the non-failed leaves of T in the left-to-right order.

If T is infinite we consider a (possibly infinite) sequence F of computed answer substitutions (modulo renaming), each substitution being associated with a leaf of T. F is obtained by visiting from left to right the non-failed leaves which are on branches to the left of the leftmost infinite branch. There are two cases: *either* F is infinite, in which case $Sem_{Prolog}(P,Q)$ is F *or* F is finite in which case $Sem_{Prolog}(P,Q)$ is F followed by the symbol $\perp$, which is called the *undefined* substitution. All substitutions different from $\perp$ are said to be *defined*.

Thus, Sem_{Prolog} is defined as a function from Programs $\times$ Queries to $(D,\leq)$, where Programs and Queries are the sets of definite programs and atomic queries, respectively. $(D,\leq)$ is the set SubstSeq of finite or infinite sequences of defined substitutions, and finite sequences of defined substitutions followed by the undefined substitution $\perp$. Similar approaches to the semantics of Prolog can be found in [Jones-Mycroft 84, Debray-Mishra 88, Deville 90, Baudinet 92]).

The sequence consisting of the substitutions $\theta_1, \theta_2, \ldots$ is denoted by $<\theta_1, \theta_2, \ldots>$, and the concatenation of two sequences S_1 and S_2 in SubstSeq is denoted by $S_1 @ S_2$ and it is defined as the usual monoidal concatenation of finite or infinite sequences, with the extra property: $<\perp> @ S = <\perp>$.

Example 13. Consider the following three programs:

P_1:	$p(a) \leftarrow$	$p(b) \leftarrow$	$p(a) \leftarrow$
P_2:	$p(a) \leftarrow$	$p(X) \leftarrow p(X)$	$p(b) \leftarrow$
P_3:	$p(a) \leftarrow$	$p(b) \leftarrow p(b)$	$p(a) \leftarrow$

We have that:

$Sem_{Prolog}(P_1, \leftarrow p(X)) = <\{X/a\}, \{X/b\}, \{X/a\}>$,
$Sem_{Prolog}(P_2, \leftarrow p(X)) = <\{X/a\}, \{X/a\}, \ldots >$, and
$Sem_{Prolog}(P_3, \leftarrow p(X)) = <\{X/a\}, \perp>$. ■

The order $\leq$ over SubstSeq expresses a *less defined than or equal to* relation between sequences which can be introduced as follows. For any two sequences of substitutions S_1 and S_2, the relation $S_1 \leq S_2$ holds iff *either* $S_1 = S_2$ *or* $S_1 = S_3 @ <\perp>$ and $S_2 = S_3 @ S_4$, for some S_3 and S_4 in SubstSeq. For instance, $<\perp> \leq S$, for any (possibly empty) sequence S, and for all substitutions η_1, η_2, η_3 with $\eta_1 \neq \perp$ and $\eta_2 \neq \perp$, $<\eta_1, \perp> \leq <\eta_1, \eta_2, \eta_3>$. The sequences $<\eta_1>$ and $<\eta_1, \eta_2>$ are not comparable w.r.t. the order $\leq$.

Unfortunately, most transformation rules presented in the previous sections are not even partially correct w.r.t. Sem_{Prolog}. Indeed, it is easy to see that the application of a boolean rule may affect the order, or the multiplicity, or the finite time computability of the computed answer substitutions.

An unfolding step may affect the order of the computed answer substitutions as well as the termination of a program, as it is shown by the following examples.

Example 14. By unfolding w.r.t. r(Y) the first clause of the following program:

P_0: p(X,Y) ← q(X), r(Y)   q(a) ←   q(b) ←   r(a) ←   r (b) ←

we get:

P_1: p(X,a) ← q(X)   p(X,b) ← q(X)   q(a) ←   q(b) ←   r(a) ←   r (b) ←

The order of the computed answer substitutions is changed. Indeed, we have that:

$\mathrm{Sem}_{\mathrm{Prolog}}(P_0, \leftarrow p(X,Y)) = $ <{X/a,Y/a}, {X/a,Y/b}, {X/b,Y/a}, {X/b,Y/b}>,

and

$\mathrm{Sem}_{\mathrm{Prolog}}(P_1, \leftarrow p(X,Y)) = $ <{X/a,Y/a}, {X/b,Y/a}, {X/a,Y/b}, {X/b,Y/b}>. ■

Example 15. By unfolding w.r.t. r the first clause of the following program:

P_0: p ← q, r   q ←   q ← q   r ← fail   r ←

we get:

P_1: p ← q, fail   p ← q   q ←   q ← q   r ← fail   r ←

P_1 is *less defined* than P_0. Indeed, we have that:

$\mathrm{Sem}_{\mathrm{Prolog}}(P_0, \leftarrow p) = $ <{}, {}, … >, and
$\mathrm{Sem}_{\mathrm{Prolog}}(P_1, \leftarrow p) = \langle\perp\rangle$. ■

Example 16. By unfolding w.r.t. r(X) the first clause of the following program:

P_0: p ← q(X), r(X)   q(a) ← q(a)   r(b) ←

we get:

P_1: p ← q(b)   q(a) ← q(a)   r(b) ←

P_1 is *more defined* than P_0. Indeed, we have that:

$\mathrm{Sem}_{\mathrm{Prolog}}(P_0, \leftarrow p) = \langle\perp\rangle$, and
$\mathrm{Sem}_{\mathrm{Prolog}}(P_1, \leftarrow p) = \langle\rangle$. ■

We also have that the use of the folding rule does not always preserve $\mathrm{Sem}_{\mathrm{Prolog}}$. In order to overcome this inconvenience, several researchers have proposed restricted versions of the unfolding and folding rules [Proietti-Pettorossi 91a, Sahlin 91]. The following rules R18 and R19 are two instances of the unfolding rule which can be shown to be totally correct w.r.t. $\mathrm{Sem}_{\mathrm{Prolog}}$.

R18. Leftmost Unfolding. A *leftmost unfolding* step of clause C consists of an unfolding step of C w.r.t. the leftmost atom of its body.

R19. Deterministic Non-left-propagating Unfolding. The unfolding of a clause $H \leftarrow F, A, G$ w.r.t. the atom A is *deterministic non-left-propagating* iff i) there exists exactly one clause D such that A is unifiable with hd(D) via a most general unifier θ, and ii) $H \leftarrow F$ is a variant of $(H \leftarrow F)\theta$.

Also the definition introduction, definition elimination, and clause replacement rules are totally correct w.r.t. $\mathrm{Sem}_{\mathrm{Prolog}}$. We have that the goal replacement rule is partially correct w.r.t. $\mathrm{Sem}_{\mathrm{Prolog}}$, and so is the T&S-folding rule if we allow the use of clauses of the current program only. Thus, we can state a result which is analogous to Theorems 6, 9, and 11 for $\mathrm{Sem}_{\mathrm{H}}$, $\mathrm{Sem}_{\mathrm{CA}}$, and $\mathrm{Sem}_{\mathrm{FF}}$, respectively.

THEOREM 13. Let $P_0,\ldots, P_n$ be a transformation sequence of definite programs, constructed by using the transformation rules: leftmost unfolding, deterministic non-left-propagating unfolding, T&S-folding, definition introduction, definition elimination, reversible goal replacement, and clause replacement. Suppose that each T&S-folding is an instance of the reversible folding rule R13. Then $P_0,\ldots, P_n$ is correct w.r.t. $\mathrm{Sem}_{\mathrm{Prolog}}$. ■

For the case of T&S-folding which is not an instance of reversible folding, we have the following result which is analogous to Theorems 8, 10, and 12, and it is based on the fact that an application of the leftmost unfolding rule is 'a step forward in the computation' using the left-to-right computation rule.

THEOREM 14. [Proietti-Pettorossi 91a] (*Correctness of T&S-folding w.r.t.* $\mathrm{Sem}_{\mathrm{Prolog}}$) Let $P_0,\ldots, P_n$ be a transformation sequence of definite programs, constructed by using the following transformation rules: leftmost unfolding, deterministic non-left-propagating unfolding, T&S-folding, and T&S-definition. Suppose that no T&S-folding step is performed after a definition elimination step. Suppose also that we apply T&S-folding to a clause C using a clause D only if i) hd(D) has a new predicate and ii) either hd(C) has an old predicate or the *leftmost* atom of bd(C) is fold-allowing. Then $P_0,\ldots, P_n$ is correct w.r.t. the semantics $\mathrm{Sem}_{\mathrm{Prolog}}$. ■

The following example shows that in Theorem 14 we cannot replace 'the *leftmost* atom' by 'an atom'.

Example 17. Let us consider the following initial program:

P_0: $\quad p \leftarrow q(X), r(X) \qquad q(X) \leftarrow \mathrm{fail} \qquad r(X) \leftarrow r(X)$

We have that: i) p is a new predicate and q, r, and fail are old predicates, and i occurrences of q(X) and r(X) in the first clause are not fold-allowing. By determini non-left-propagating unfolding of p $\leftarrow$ q(X), r(X) w.r.t. r(X), we get the followin program which is equal to P_0:

P_1: p $\leftarrow$ q(X), r(X) q(X) $\leftarrow$ fail r(X) $\leftarrow$ r(X)

Now, the occurrence of r(X) in the first clause is fold-allowing. If we fold the first clause of P_1 using that same clause, we get:

P_2: p $\leftarrow$ p q(X) $\leftarrow$ fail r(X) $\leftarrow$ r(X)

P_2 is not equivalent to P_0 w.r.t. Sem_{Prolog}. Indeed, we have that:

$Sem_{Prolog}(P_0, \leftarrow p) = \langle\rangle$, and
$Sem_{Prolog}(P_2, \leftarrow p) = \langle\perp\rangle$. ■

In this paper we have considered only the case of *pure* Prolog, where the SLD-resolution steps have no side-effects. Properties preserved by unfold/fold rules when transforming Prolog programs with side-effects, including cuts, can be found in [Deville 90, Sahlin 91, Prestwich 93a].

3.3 SEMANTICS PRESERVING TRANSFORMATIONS FOR NORMAL PROGRAMS

In this section we consider the case where the bodies of the clauses contain negative literals. There is a large number of papers dealing with transformations which preserve the various semantics which have been proposed for logic programs with negation. In particular, some restricted forms of unfolding and folding have been shown to be correct w.r.t. various semantics, such as the success set and finite failure set semantics [Gardner-Shepherdson 91, Seki 91, Shepherdson 92], Clark's completion [Gardner-Shepherdson 91, Shepherdson 92], Fitting's and Kunen's three-valued extensions of Clark's completion [Fitting 85, Kunen 87, Bossi et al. 92a, Sato 92], perfect model semantics [Przymusinsky 87, Maher 89, Seki 91], stable model semantics [Gelfond-Lifschitz 88, Maher 90], and well-founded model semantics [Van Gelder et al. 88, Maher 90, Seki 93]. For limitations of space, we will report here only on the results concerning the following three semantics [Lloyd 87]: i) success set, ii) finite failure set, and iii) Clark's completion.

The success set semantics for normal programs, denoted Sem_{SS} is a function from Programs $\times$ Queries to $(D,\leq)$, where Programs is the set of normal programs, Queries is the set of atomic queries, and $(D,\leq)$ is the powerset of the set of (possibly not ground) atoms ordered by set inclusion. By definition we have that $Sem_{SS}(P, \leftarrow A) = \{B \mid B$ is an instance of A and there exists an SLDNF-refutation for P and $\leftarrow B\}$.

The finite failure semantics for normal programs, denoted Sem_{FF}, has the same domain and codomain of Sem_{SS}. By definition we have that $Sem_{FF}(P, \leftarrow A) = \{B \mid B$

is an instance of A and there exists a finitely failed SLDNF-tree for P and $\leftarrow$ B}.

For the correctness of a transformation sequence w.r.t. Sem_{SS} and Sem_{FF} there are results which are analogous to Theorem 12. Indeed, the statement of that theorem is valid if we replace 'definite programs' by 'stratified normal programs' and we consider any of the two semantics Sem_{SS} or Sem_{FF} [Seki 91].

Notice also that the hypotheses for the version of Theorem 12 for normal programs and Sem_{SS} are more restrictive than the hypotheses of Theorem 8 for definite programs and Sem_H. This is due to the fact that in order to preserve the success set of normal programs, we may need to preserve their finite failure sets as well, because the evaluation of positive goals may require the evaluation of negative goals.

Now we consider a definition of the semantics function based on the completion of a normal program. This function is from Programs $\times$ Queries to $(D,\leq)$, where Programs is the set of normal programs, Queries is the set of atomic queries, and $(D,\leq)$ is the powerset of the set of (possibly not ground) atoms ordered by set inclusion. By definition we have that $Sem_{Comp}(P, \leftarrow A) = \{B \mid B$ is an instance of the atom A and the universal closure of B is a logical consequence of the completion Comp(P) of the program P$\}$.

The partial correctness of the unfolding and folding rules can easily be established, as illustrated by the following example.

Example 18. Let us consider the program:

$$\begin{array}{lllllll} P_0: & p \leftarrow q, \neg r & q \leftarrow s, t & v \leftarrow t & s \leftarrow & u \leftarrow \\ & & q \leftarrow s, u & v \leftarrow u & & \end{array}$$

whose completion is:

$$Comp(P_0):\quad p \leftrightarrow q \wedge \neg r \qquad q \leftrightarrow (s \wedge t) \vee (s \wedge u) \qquad v \leftrightarrow t \vee u \qquad s \qquad u \qquad \neg r \qquad \neg t$$

By unfolding the first clause of P_0 w.r.t. q we get:

$$\begin{array}{llllll} P_1: & p \leftarrow s, t, \neg r & q \leftarrow s, t & v \leftarrow t & s \leftarrow & u \leftarrow \\ & p \leftarrow s, u, \neg r & q \leftarrow s, u & v \leftarrow u & & \end{array}$$

whose completion is:

$$Comp(P_1):\quad p \leftrightarrow (s \wedge t \wedge \neg r) \vee (s \wedge u \wedge \neg r) \qquad q \leftrightarrow (s \wedge t) \vee (s \wedge u) \qquad v \leftrightarrow t \vee u \qquad s \qquad u \qquad \neg r \qquad \neg t$$

$Comp(P_1)$ can be obtained by replacing q in $p \leftrightarrow q \wedge \neg r$ of $Comp(P_0)$ by $(s \wedge t) \vee (s \wedge u)$ and then applying the distributive and associative laws. Since $q \leftrightarrow (s \wedge t) \vee (s \wedge u)$ holds in $Comp(P_0)$, we have that $Comp(P_1)$ is a logical consequence of $Comp(P_0)$.
From P_1 by folding the definition of p using the definition of v in P_1 itself, we get:

$$\begin{array}{llllll} P_2: & p \leftarrow s, v, \neg r & q \leftarrow s, t & v \leftarrow t & s \leftarrow & u \leftarrow \\ & & q \leftarrow s, u & v \leftarrow & & \end{array}$$

whose completion is:

$$Comp(P_2):\quad p \leftrightarrow s \wedge v \wedge \neg r \qquad q \leftrightarrow (s \wedge t) \vee (s \wedge u) \qquad v \leftrightarrow t \vee u \qquad s \qquad u \qquad \neg r \qquad \neg t$$

$\mathrm{Comp}(P_2)$ can be obtained from $\mathrm{Comp}(P_1)$ by first using the associative, commutative, and distributive laws for replacing the formula $p \leftrightarrow (s \wedge t \wedge \neg r) \vee (s \wedge u \wedge \neg r)$ by $p \leftrightarrow (t \vee u) \wedge (s \wedge \neg r)$, and then replacing $t \vee u$ by v. Since $v \leftrightarrow t \vee u$ holds in $\mathrm{Comp}(P_1)$, we have that $\mathrm{Comp}(P_2)$ is a logical consequence of $\mathrm{Comp}(P_1)$. ■

In general, if a program P_{k+1} can be obtained from a program P_k by folding steps which use clauses in P_k only or by unfolding steps, then $\mathrm{Comp}(P_{k+1})$ can be obtained from $\mathrm{Comp}(P_k)$ by one or more replacements of a formula F by a formula G such that $F \leftrightarrow G$ is a logical consequence of $\mathrm{Comp}(P_k)$. Thus, $\mathrm{Comp}(P_{k+1})$ is a logical consequence of $\mathrm{Comp}(P_k)$.

A similar statement holds if P_{k+1} can be obtained from P_k by applying the goal replacement rule or the clause replacement rule. Thus, we have the following result, analogous to Theorem 3 for Sem_H.

THEOREM 15. (*Partial Correctness of Transformations w.r.t.* $\mathrm{Sem}_{\mathrm{Comp}}$) Let $P_0, \ldots, P_n$ be a transformation sequence constructed by using the rules R1,…, R3, R5,…, R12, R15. Suppose that each folding step is performed by using clauses in the current program only. Then $P_0, \ldots, P_n$ is partially correct w.r.t. the semantics $\mathrm{Sem}_{\mathrm{Comp}}$. ■

Unfortunately, the unfolding rule is not totally correct w.r.t. $\mathrm{Sem}_{\mathrm{Comp}}$ as shown by the following example adapted from [Maher 89].

Example 19. Let us consider the program:

$$P_0: \quad p(X) \leftarrow q(X) \qquad p(X) \leftarrow \neg q(succ(X)) \qquad q(X) \leftarrow q(succ(X))$$

whose completion is (equivalent to):

$$\mathrm{Comp}(P_0): \quad \forall X\, (p(X) \leftrightarrow q(X) \vee \neg q(succ(X))) \qquad \forall X\, (q(X) \leftrightarrow q(succ(X)))$$

together with the axioms of Clark's Equality Theory (CET) [Lloyd 87, Apt 90]. CET is a first order complete (and hence decidable) equality theory which axiomatizes the identity relation on the Herbrand universe. By self-unfolding of the last clause of P_0 we get:

$$P_1: \quad p(X) \leftarrow q(X) \qquad p(X) \leftarrow \neg q(succ(X)) \qquad q(X) \leftarrow q(succ(succ(X)))$$

whose completion is (equivalent to):

$$\mathrm{Comp}(P_1): \quad \forall X\, (p(X) \leftrightarrow q(X) \vee \neg q(succ(X))) \qquad \forall X\, (q(X) \leftrightarrow q(succ(succ(X))))$$

together with the axioms of CET.

We have that $\forall X\, p(X)$ is a logical consequence of $\mathrm{Comp}(P_0)$. On the other hand, $\forall X\, p(X)$ is not a logical consequence of $\mathrm{Comp}(P_1)$. Indeed, let us consider the interpretation I whose universe is the set of integers, p(x) holds iff q(x) holds iff x is an even integer, and succ is the successor function. I is a model of $\mathrm{Comp}(P_1)$ while it is not a model of $\forall X\, p(X)$. ■

We may restrict the use of the unfolding rule so to make it totally correct w.r.t. $\mathrm{Sem}_{\mathrm{Comp}}$, by requiring that during a transformation sequence no *self-unfolding* steps are performed, that is, we never unfold a clause using itself (and possibly other ones).

Indeed, if program P_1 is derived from program P_0 by performing an unfolding step which is not a self-unfolding, then the transformation sequence P_0, P_1 is reversible (w.r.t. any set of rules including R1 and R13), because we may get P_0 from P_1 by reversible folding (rule R13). Thus, by Theorems 5 and 15, we have that the unfolding rule is totally correct w.r.t. $\mathrm{Sem}_{\mathrm{Comp}}$.

As a consequence, we have the following result where by *reversible unfolding* we mean an unfolding step which is not a self-unfolding.

THEOREM 16. Let $P_0,\ldots, P_n$ be a transformation sequence constructed by using the transformation rules: reversible unfolding, reversible folding, T&S-definition, definition elimination, reversible goal replacement, and clause replacement. Then $P_0,\ldots, P_n$ is correct w.r.t. $\mathrm{Sem}_{\mathrm{Comp}}$. ■

We end this section by showing, through the following example hypotheses of Theorem 12 are not sufficient to ensure the correctness of w.r.t. Sem $_{\mathrm{Comp}}$.

Example 20. Let us consider the following transformation sequence:

P_0:	$p \leftarrow q$	$q \leftarrow q$	$r \leftarrow p$	$r \leftarrow \neg q$	
P_1:	$p \leftarrow q$	$q \leftarrow q$	$r \leftarrow p$	$r \leftarrow \neg q$	(by reversible unfolding of $p \leftarrow q$)
P_2:	$p \leftarrow p$	$q \leftarrow q$	$r \leftarrow p$	$r \leftarrow \neg q$	(by T&S-folding $p \leftarrow q$).

By Theorem 12 P_0 and P_2 are equivalent w.r.t. $\mathrm{Sem}_{\mathrm{FF}}$. Let us now consider the completions of P_0 and P_2, respectively:

$\mathrm{Comp}(P_0)$:	$p \leftrightarrow q$	$q \leftrightarrow q$	$r \leftrightarrow p \vee \neg q$
$\mathrm{Comp}(P_2)$:	$p \leftrightarrow p$	$q \leftrightarrow q$	$r \leftrightarrow p \vee \neg q$.

We have that r is a logical consequence of $\mathrm{Comp}(P_0)$. On the contrary, r is not a logical consequence of $\mathrm{Comp}(P_2)$. Indeed, the interpretation where p is false, q is true, and r is false is a model of $\mathrm{Comp}(P_2)$, but not of r. ■

It should be noted that in the above Example 20, P_0 is equivalent to P_2 w.r.t. other two-valued or three-valued semantics for normal programs such as the already mentioned Fitting's and Kunen's extensions of Clark's completion, perfect model, stable model, and well-founded model semantics. The reader may find various correctness results of T&S-folding w.r.t. these semantics in [Sato 92, Seki 91, Seki 93].

4. STRATEGIES FOR TRANSFORMING LOGIC PROGRAMS

The transformation process should be directed by some metarules, which we call *strategies*, because, as we have seen in the previous section, the transformation rules have inverses, and thus, they allow for final programs which are equal to the initial ones. Obviously, we are not interested in such useless transformations.

In this section we present an overview of some transformation strategies which have been proposed in the literature. They are used, in particular, for solving one of the most crucial problems of the transformation methodology, that is, the use of the definition rule for the introduction of the so-called *eureka predicates*.

The reader may refer to [Feather 87, Partsch 90, Deville 90, Pettorossi-Proietti 93] for a treatment of transformation strategies for functional and logic programs.

For simplicity reasons, we only consider the case of definite programs with the least Herbrand model semantics Sem_H. We will use in our examples the following rules, whose correctness w.r.t. Sem_H is ensured when they are used according to the hypotheses of Theorems 6 and 8: unfolding (R1), T&S-folding (R3), T&S-definition (R15), definition elimination (R5), reversible goal replacement (R14), and boolean rules (R8, R9, R10, R11, and R12).

In some examples below we will construct transformation sequences by using both T&S-folding and reversible goal replacement not according to the hypotheses of Theorems 6 and 8. In these examples, however, the correctness w.r.t. Sem_H continues to hold, as the reader may check by referring to [Tamaki-Sato 84, Tamaki-Sato 86].

In order to simplify our presentation we will usually avoid to mention the use of rule R8 (goal rearrangement) and rule R9 (deletion of duplicate goals).

If we allow the use of boolean rules then the concatenation of sequences of literals and the concatenation of sequences of clauses are associative, commutative, and idempotent. Therefore, in that case, when dealing with collections of literals or programs, we will feel free to use set-theoretic notations, such as $\{\ldots\}$ and $\cup$, instead of $<\ldots>$ and @.

Before presenting the technical details of the transformation strategies we would like to give an informal explanation of the main ideas which justify their use. We are given an initial program and we want to apply the transformation rules to improve its efficiency. In order to do so, we usually need a preliminary analysis of the initial program by which we discover that the evaluation of a goal, say $A_1,\ldots, A_n$, in the body of a program clause, is inefficient because it evokes some redundant computations.

For example, by analyzing the initial program P_0 given in Section 2, we may discover that the evaluation of the conjunction of atoms ‘length(L,N), sumlist(L,S)’ in the body of the clause:

1. average(L,A) $\leftarrow$ length(L,N), sumlist(L,S), div(S,N,A)

is inefficient because it determines a double traversal of the list L.

In order to improve the performance of program P_0, we can apply the technique which consists in introducing a new predicate, say newp, by means of a clause, say N, with body $A_1,\ldots, A_n$. This initial transformation step has been formalized as an application of the *tupling strategy* (see Section 4.1). We then unfold clause N one or more times, thereby generating some new clauses. This process can be viewed as a symbolic evaluation of a query which is an instance of $A_1,\ldots, A_n$. This unfolding gives us the opportunity of improving our program, because, for instance, we may delete some clauses with finitely failed body, thus avoiding failures at run-time, or we may delete duplicate atoms, thus avoiding repeated computations, and so on.

With reference to the example of Section 2, we recall that by unfolding clause 1 w.r.t length and sumlist we derived the clauses:

9. newp([],0,0) ←
10. newp([H|T],s(N),S1) ← length(T,N), sumlist(T,S), sum(H,S,S1)

which avoid multiple traversals of the input list when it is empty.

The efficiency improvements due to the unfoldings, can be iterated *at each level of recursion*, and thus, they become computationally significant, only if we find a *recursive definition* of newp. In that case the multiple traversals of the input will be avoided for any given list. This recursive definition can often be achieved by performing a folding step using the clause N introduced by tupling.

In our case, by folding we get:

10f. newp([H|T],s(N),S1) ← newp(T,N,S), sum(H,S,S1)

and indeed, this recursive clause together with clause 9 avoids multiple traversals of *any* input list.

In some unfortunate cases we may not be able to perform the desired final folding steps and derive the recursive definition of newp. In those cases we may use some auxiliary strategies and we may introduce some extra eureka predicates which allow us to perform the required folding steps. Two of those auxiliary strategies are the *loop absorption* and *generalization* strategies described in Section 4.1.

In [Darlington 81] the expression *need for folding* is introduced to refer to the need of performing the final folding steps for improving program efficiency. This need plays an important role in the program transformation methodology, and it can be regarded as a meta-strategy. It is the need for folding that often suggests the suitable strategy to apply at each step of the derivation.

Need for folding in program transformation is related to similar ideas in the field of automated theorem proving [Boyer-Moore 74] and program synthesis [Deville-Lau 94], where inductive proofs and inductive synthesis tactics are driven by the need of applying an inductive hypothesis.

4.1 BASIC TRANSFORMATION STRATEGIES

We now describe some of the basic strategies which have been introduced in the literature for transforming logic programs. They are: tupling, loop absorption, and generalization.

The basic ideas underlying these strategies come from the early days of program transformation and they were already present in [Burstall-Darlington 77]. The tupling strategy was formally defined in [Pettorossi 77] where it is used for tupling together different function calls which require common subcomputations or visit the same data structure. The name 'loop absorption' was introduced in [Proietti-Pettorossi 90], for indicating a strategy which derives a new predicate definition when a goal is recurrently evaluated in the program to be transformed. This strategy is present in various forms in a number of different transformation techniques, such as the above mentioned tupling, supercompilation [Turchin 86], compiling control [Bruynooghe et al. 89], as well as various techniques for partial evaluation (see Section 5). Finally, the generalization strategy has its origin in the automated theorem proving context [Boyer-Moore 74], where it is used to generate a new generalized conjecture allowing the application of an inductive hypothesis.

The tupling, loop absorption, and generalization strategies will be used in this paper as building blocks to describe a (non-exhaustive) number of more complex transformation techniques.

For a formal description of the strategies and their possible mechanization we now introduce the notion of *unfolding tree*. It represents the process of unfolding a given clause using a given program. This notion is also related to the one of symbolic trace tree of [Bruynooghe et al. 89], where, however, goal replacement is not taken into account.

DEFINITION 17. Let P be a program and C a clause. An *unfolding tree* for <P,C> is a (finite or infinite) non-empty labelled tree such that:

i) the root is labelled by the clause C,

ii) if M is a node labelled by a clause D then
either M has no sons,
or M has n (≥ 1) sons labelled by the clauses $D_1,\ldots,D_n$ obtained by unfolding D w.r.t. an atom of its body using P,
or M has one son labelled by a clause obtained by goal replacement from D. ■

In an unfolding tree we also have the usual relations of descendant node (or clause) and ancestor node (or clause).

Given a program P and a clause C, the construction of an unfolding tree for <P,C> is non-deterministic. In particular, during the process of constructing an unfolding tree we need to decide whether or not a node should have son-nodes, and in case we decide that son-nodes should be constructed by unfolding, we need to choose the atom w.r.t. which the unfolding step should be performed. These choices can be realized by using a function defined as follows.

DEFINITION 18. An *unfolding selection rule* (or *u-selection rule*, for short) is a function that given an unfolding tree and one of its leaves, tells us whether or not to unfold the clause in that leaf and in the affirmative case, it tells us the atom w.r.t. which that clause should be unfolded. ■

DEFINITION 19. Given a clause C of the form $H \leftarrow A_1,\dots, A_m, B_1,\dots, B_n$, the *linking variables* of the sequence of atoms $A_1,\dots, A_m$ in C are the variables in $\mathrm{vars}(A_1,\dots, A_m) \cap \mathrm{vars}(H, B_1,\dots, B_n)$. ■

We now formally introduce the tupling, loop absorption, and gene strategies.

S1. Tupling. Let $A_1,\dots, A_n$, with $n \geq 1$, be some atoms occurring in the body of a clause C of a given initial program. We introduce a new predicate newp defined by a clause T of the form:

$$\mathrm{newp}(X_1,\dots, X_k) \leftarrow A_1,\dots, A_n$$

where $X_1,\dots, X_k$ are the linking variables of $A_1,\dots, A_n$ in C. We then look for the recursive definition of the eureka predicate newp by performing some unfolding steps followed by suitable folding steps using clause T. We finally fold clause C w.r.t. the atoms $A_1,\dots, A_n$ using clause T. ■

The tupling strategy is often applied when $A_1,\dots, A_n$ share some variables. The program improvements which can be achieved by using this strategy are based on the fact that we need to evaluate only once the subgoals which are common to the computations determined by the tupled atoms $A_1,\dots, A_n$. By tupling we can also avoid multiple visits of data structures and the construction of intermediate bindings.

S2. Loop Absorption. Suppose that a non-root clause C in an unfolding tree has the form: $H \leftarrow A_1,\dots, A_m, B_1,\dots, B_n$, and the body of a descendant D of C contains (as a subsequence of atoms) an instance $(A_1,\dots, A_m)\theta$ of $A_1,\dots, A_m$ for some substitution θ. Suppose also that the clauses in the path from C to D have been generated by applying no transformation rule, except for R8 and R9, to $B_1,\dots, B_n$. We introduce a new predicate defined by the following clause A:

$$\mathrm{newp}(X_1,\dots, X_k) \leftarrow A_1,\dots, A_m$$

where $\{X_1,\dots, X_k\}$ is the minimum subset of $\mathrm{vars}(A_1,\dots, A_m)$ which is necessary to fold both C and D using a clause whose body is $A_1,\dots, A_m$. (See Point 2 of R3 for the conditions on $\{X_1,\dots, X_k\}$ and θ which should be satisfied for allowing folding.) We fold clause C using clause A and we then look for the recursive definition of the eureka predicate newp. This can be done by performing the unfolding steps corresponding to the ones which lead from clause C to clause D and then folding using clause A again. ■

S3. Generalization. Given a clause C of the form: $H \leftarrow A_1,\dots, A_m, B_1,\dots, B_n$, we define a new predicate genp by a clause G of the form:

$$\mathrm{genp}(X_1,\dots, X_k) \leftarrow \mathrm{GenA}_1,\dots, \mathrm{GenA}_m$$

where $(GenA_1,\ldots, GenA_m)\theta = A_1,\ldots,A_m$, for a given substitution θ, and $\{X_1,\ldots,X_k\}$ is a superset of the variables which are necessary to fold C using a clause whose body is $GenA_1,\ldots, GenA_m$. We then fold C using G and we get:

$$H \leftarrow genp(X_1,\ldots,X_k)\theta,\ B_1,\ldots,B_n.$$

We finally look for the recursive definition of the eureka predicate genp. ■

A suitable form of the clause G introduced by generalization can often be obtaine by matching clause C against one of its descendants, say D, in the unfolding tree whi is considered during program transformation (see Example 23 below). In particular, w will consider the case where:

1. D is the clause $K \leftarrow E_1,\ldots,E_m, F_1,\ldots,F_r$, and D has been obtained from C b applying no transformation rule, except for R8 and R9, to $B_1,\ldots, B_n$,
2. $E_1,\ldots,E_m$ is not an instance of $A_1,\ldots,A_m$,
3. the goal $GenA_1,\ldots,GenA_m$ is the most specific generalization of $A_1,\ldots,A_m$ and $E_1,\ldots,E_m$, and
4. $\{X_1,\ldots, X_k\}$ is the minimum subset of $vars(GenA_1,\ldots, GenA_m)$ which is necessary to fold both C and D using a clause whose body is $GenA_1,\ldots, GenA_m$.

4.2 TECHNIQUES WHICH USE BASIC TRANSFORMATION STRATEGIES

In this section we will present some techniques for improving program efficiency by using the tupling, loop absorption, and generalization strategies.

4.2.1 COMPILING CONTROL

One of the advantages of logic programming over conventional imperative programming is that by writing a logic program one may separate the ‘logic’ part of an algorithm from the ‘control’ part [Kowalski 79]. By doing so, the correctness of an algorithm w.r.t. a given specification is often easier to prove. Obviously, we are then left with the problem of providing an efficient control.

Unfortunately, the standard top-down, depth-first, and left-to-right Prolog strategy for controlling SLD-resolution does not always give us the desired level of efficiency, because of the amount of non-determinism during the evaluation of a program. Much work has been done in the direction of improving the control strategy of logic languages (see, for instance, [Bruynooghe-Pereira 84, Naish 86]).

We consider here a transformation technique, called *Compilin* [Bruynooghe et al. 89], which follows a different approach. Instead of enhancing the naive Prolog evaluator using a better (and often more complex) control strategy, we transform the given program so that the derived one behaves under the naive evaluator as the given program would behave under an enhanced evaluator.

The main advantage of the Compiling Control technique is that we can use relatively simple evaluators which have small and efficient compilers.

The Compiling Control approach can also be followed to 'compile' bottom-up and mixed evaluation strategies [De Schreye et al. 91, Sato-Tamaki 88] as well as lazy evaluation and coroutining [Narain 86]. In this paper we only show the use of the Compiling Control technique in the case where the control to be 'compiled' is a computation rule different from the left-to-right Prolog one. In this case, by applying Compiling Control one can improve generate-and-test programs by simulating a computation rule which selects test predicates as soon as the relevant data are available.

A similar idea has also been investigated in the area of functional programming, within the so-called Filter Promotion strategy [Darlington 81, Bird 84]. Some other transformation techniques for improving generate-and-test logic programs which are closely related to the Compile Control technique and the Filter Promotion strategy, can be found in [Seki-Furukawa 87, Brough-Hogger 91, Träff-Prestwich 92].

The problem of 'compiling' a given computation rule C can be described as follows: given a program P_1 and a set Q of queries, we want to derive a new program P_2 which, for any query in Q, is equivalent to P_1 w.r.t. Sem_H and behaves under the left-to-right computation rule as P_1 does under the rule C [Bruynooghe et al. 89, De Schreye-Bruynooghe 89].

By 'equal behaviour' we mean that for a query in Q, the SLD-tree, say T_1, constructed by using P_1 and the computation rule C is equal to the SLD-tree, say T_2, constructed by using P_2 and the left-to-right computation rule, if i) we look at T_1 and T_2 as directed trees with leaves labeled by 'success' or 'failure' and arcs labeled by most general unifiers, and ii) we possibly replace non-branching paths of T_1 by single arcs, each of which is labeled by the composition of the most general unifiers labeling the corresponding path to be replaced. Thus, when comparing the trees T_1 and T_2 we disregard the goals in the nodes.

Basic forms of Compiling Control can be formulated as follows. Given a program P_1, a set Q of queries, and a computation rule C, Compiling Control derives the new program P_2 by first constructing a suitable unfolding tree, say T, and then applying the loop absorption strategy. (Some more complex forms of Compiling Control require the use of generalization strategies possibly more powerful than S3.)

Without loss of generality, we assume that every query in Q is of the form $\leftarrow$ q(…) and in P_1 there exists only one clause, say R, whose head predicate is q. (We can use the T&S-definition rule to comply with this condition.) The root clause of T is R and new nodes are generated by using a suitable u-selection rule.

It is required that the unfolding tree T constructed from P_1, Q, and C satisfies the condition that each SLD-tree generated by a query in Q using the program P_1 and the computation rule C, is a *concretization tree* T_γ which is derived from T as follows.

Let q(…) be unifiable with hd(R) via a substitution θ. T_γ is derived from T by: i) deleting each node (and the subtree rooted in that node) whose clause head is not unifiable with hd(R)θ, ii) replacing for each remaining node the clause in that node, say D, by $\leftarrow$ bd(D)μ, where μ is the most general unifier of hd(R)θ and hd(D), and finally,

iii) adding the root $\leftarrow q(\ldots)\theta$.

Example 21. Let us consider the program:

P_1:	$q(X) \leftarrow r(X), s(X)$	$r(a) \leftarrow$	$r(X) \leftarrow t(X)$	$r(X) \leftarrow u(X)$
	$s(a) \leftarrow$	$s(b) \leftarrow$	$t(b) \leftarrow$	$u(a) \leftarrow$

An unfolding tree T starting from the clause $q(X) \leftarrow r(X), s(X)$ is as depicted in Fig. 1, where (as we will also do in all figures) we have underlined the atoms which have been selected for unfolding.

Figure 2 shows the concretization tree T_γ of the tree T for the query $\leftarrow q(b)$ and the substitution $\theta = \{X/b\}$. ■

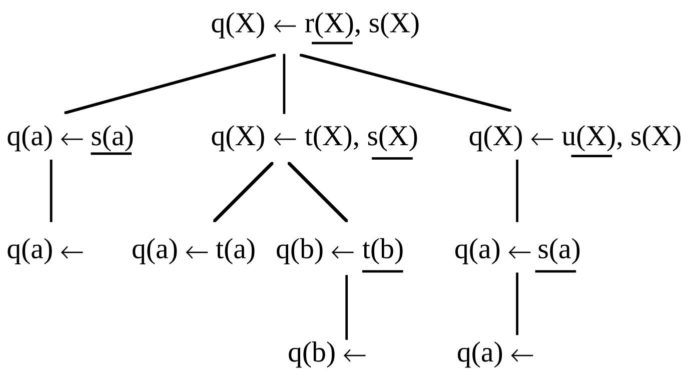

Figure 1. The unfolding tree T for $<P_1, q(X) \leftarrow r(X), s(X)>$.

Notice that in general, the unfolding tree T may correspond to an infinite set of concretization trees (because Q may be infinite). Thus, T itself may be an infinite tree and in this case the Compiling Control technique is applied by looking for a finite-graph representation (if any) of T itself. In our Example 22 below, this representation is obtained by identifying any two nodes N_1 and N_2 iff N_2 is a descendant of N_1 and the body of the clause in N_2 is an instance of the body of the clause in N_1.

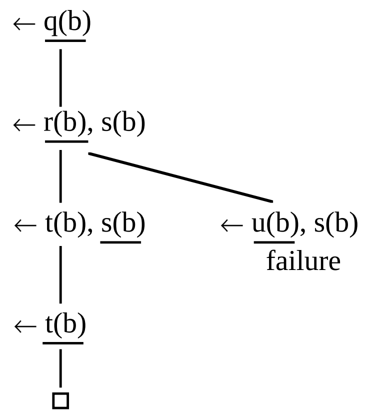

Figure 2. The concretization tree T_γ of the unfolding tree T for $<P_1, \leftarrow q(b)>$.

This finite-graph representation will allow us to apply the loop absorption strategy w.r.t. the clauses corresponding to each pair of identified nodes, and we will get the final program, where the given computation rule has been 'compiled'.

In general, the construction of the unfolding tree T from the given program P_1, the set Q of queries, and the computation rule C can be viewed as the evaluation of an 'abstract query' which represents the whole set Q, by using the program P_1 and the rule C. We will not give here the formal notion of abstraction which may allow us to effectively construct the tree T and we refer to [Cousot-Cousot 77] where abstract interpretation techniques are presented.

Example 22. (*Common Subsequences*) Let sequences be represented as lists of items. We assume that subseq(X,Y) holds iff X is a subsequence of Y in the sense that X can be obtained from Y by deleting some (possibly not contiguous) elements. Suppose that we want to verify whether or not a sequence X is a common subsequence of the two sequences Y and Z. The following program Csub does so by first verifying that X is a subsequence of Y, and then by verifying that X is a subsequence of Z.

1. csub(X,Y,Z) ← subseq(X,Y), subseq(X,Z)
2. subseq([],X) ←
3. subseq([A|X],[A|Y]) ← subseq(X,Y)
4. subseq([A|X],[B|Y]) ← subseq([A|X],Y)

where csub(X,Y,Z) holds iff X is a subsequence which is common to both Y and Z.

Let Q be the set of queries {← csub(X,s1,s2) | s1 and s2 are ground lists and X is an unbound variable} and the computation rule C be the following one: *if* the body of the clause to be unfolded is subseq(w,x), subseq(y,z) and w is a proper subterm of y *then* C selects the atom subseq(y,z) *else* C selects the leftmost atom in the body.

We first construct the infinite unfolding tree T corresponding to Csub, Q, and C. A finite-graph representation of T is depicted in Fig. 3, where dashed arrows denote identifications of nodes. The tree T has as its root clause 1 which is the only one whose head unifies with csub(X,s1,s2).

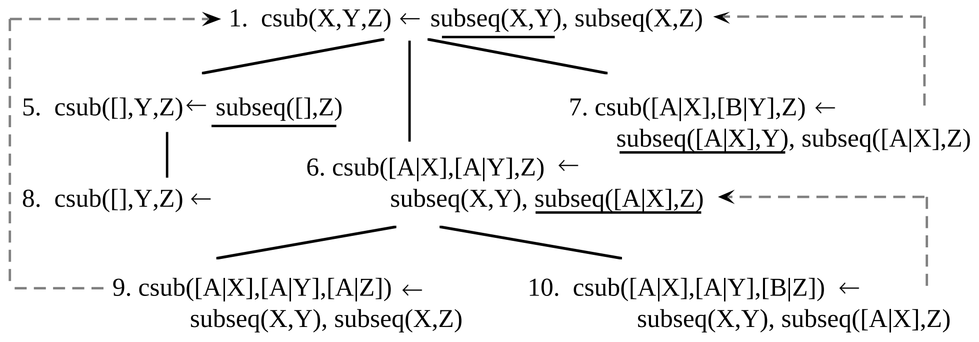
1. csub(X,Y,Z) ← subseq(X,Y), subseq(X,Z)
5. csub([],Y,Z) ← subseq([],Z)
7. csub([A|X],[B|Y],Z) ← subseq([A|X],Y), subseq([A|X],Z)
6. csub([A|X],[A|Y],Z) ← subseq(X,Y), subseq([A|X],Z)
8. csub([],Y,Z) ←
9. csub([A|X],[A|Y],[A|Z]) ← subseq(X,Y), subseq(X,Z)
10. csub([A|X],[A|Y],[B|Z]) ← subseq(X,Y), subseq([A|X],Z)

Figure 3. An unfolding tree for <Csub, csub(X,Y,Z) ← subseq(X,Y), subseq(X,Z)> using the computation rule C.

We leave to the reader the task of verifying that for every query ← csub(X,s1,s2) in Q the SLD-tree rooted in ← csub(X,s1,s2) and constructed by using the computation rule C is a concretization tree of T.

Since the body of clause 10 is an instance of the body of clause 6 we apply the loop absorption strategy. We introduce a eureka predicate newcsub by the following clause:

11. newcsub(A,X,Y,Z) ← subseq(X,Y), subseq([A|X],Z)

and we fold clause 6, whereby obtaining:

6f. csub([A|X],[A|Y],Z) ← newcsub(A,X,Y,Z).

We also have that the body of clause 7 is an instance of the body of clause 1. We fold clause 7 and we get:

7f. csub([A|X],[B|Y],Z) ← csub([A|X],Y,Z).

We now have to look for the recursive definition of the predicate newcsub. Starting from clause 11, we perform the unfolding steps corresponding to the ones which lead from clause 6 to clause 9 and 10. We get clauses:

12. newcsub(A,X,Y,[A|Z]) ← subseq(X,Y), subseq(X,Z)
13. newcsub(A,X,Y,[B|Z]) ← subseq(X,Y), subseq([A|X],Z)

and by folding we get:

12f. newcsub(A,X,Y,[A|Z]) ← csub(X,Y,Z)
13f. newcsub(A,X,Y,[B|Z]) ← newcsub(A,X,Y,Z).

The final program is made out of clauses: 8, 6f,7f,12f, and 13f.

Let us now compare the SLD-tree, say T_1, for Csub, a query ← csub(X,s1,s2) in Q, and the computation rule C, with the SLD-tree, say T_2, for the final program, the query

$\leftarrow$ csub(X,s1,s2), and the left-to-right computation rule. The trees T_1 and T_2 are equal except that: i) if a node of T_1 is labeled by a goal of the form: $\leftarrow$ subseq(…), subseq(…) then the corresponding node of T_2 is labeled by either $\leftarrow$ csub(…) or $\leftarrow$ newcsub(…), and ii) some paths of T_1 have been replaced according to the rewritings shown in Fig. 4 for any unbound variable X and ground lists t1 and t2.

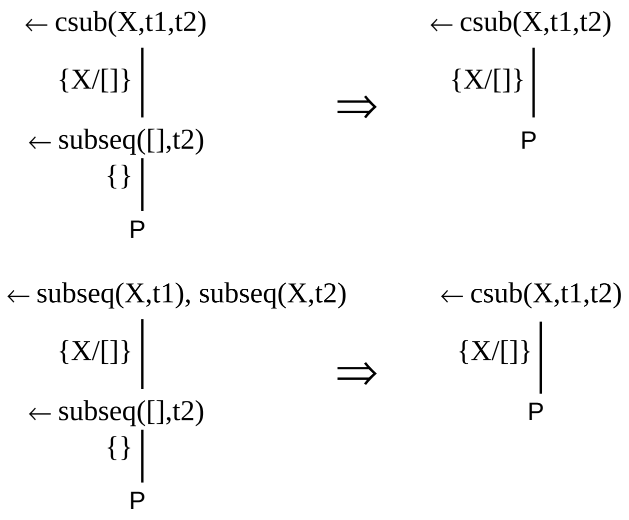

Figure 4. Path replacements for the SLD-tree T_1.

■

4.2.2 COMPOSING PROGRAMS

A popular style of programming, which can be called *compositional*, consists in decomposing a given goal in smaller and easier subgoals, then writing pieces of programs which solve these smaller subgoals, and finally, composing the various pieces together. The compositional style of programming is often helpful for writing programs which can easily be understood and proved correct w.r.t. their specifications.

Unfortunately, this programming style often produces inefficient programs, because the composition of the various subgoals does not take into account the interactions which may occur among the evaluations of these subgoals. For instance, let us consider a logic program with a clause of the form:

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

where in order to solve the goal p(X) we are required to solve q(X,Y) and r(Y). The binding of the variable Y is not explicitly needed because it does not occur in the head of the clause. If the construction and the destruction of that binding is expensive, then our program is likely to be inefficient.

Similar problems occur when the compositional style of programming is applied for writing programs in other programming languages, different from logic. In imperative languages one may construct several procedures which are then combined together by using various kinds of sequential or parallel composition operators. In functional languages, the small subgoals in which a given goal is decomposed are solved by means of individual functions which are then combined together by using function application or tupling.

There are various papers in the literature which present techniques for improving the efficiency of the evaluation of programs written according to the compositional style of programming. Similarly to the case discussed in Section 4.2.1, two approaches have been followed: 1) the improvement of the evaluator by using, for instance, garbage collection, memoization, and various forms of laziness and coroutining, and 2) the transformation of the given program into a semantically equivalent one which can be more efficiently evaluated by a non-improved evaluator.

In the imperative and functional cases, various transformation methods have been proposed, such as, for instance: *finite differencing* [Paige-Koenig 82], *composition* or *deforestation strategy* [Feather 82, Wadler 88], and *tupling* [Pettorossi 77]. (See also [Feather 87, Partsch 90] for a survey.)

For logic programs two main methods have been considered: *loop fusion* [Debray 88] and *unnecessary variable elimination* [Proietti-Pettorossi 91b]. The aim of loop fusion is to transform a program for computing a predicate which is defined as the composition of two independent recursive predicates, into a program where the computations corresponding to these two predicates are performed by one predicate only. The benefits one may expect from loop fusion are the avoidance of multiple traversals of data structures and the avoidance of the construction of intermediate data structures.

The transformational methods for composing logic programs are closely related to methods for *logic program construction* [Sterling-Lakhotia 88], where complex programs are developed by enhancing and composing together simpler programs.

The method presented in [Proietti-Pettorossi 91b] may be used for deriving programs without unnecessary variables. A variable X of a clause C is said to be *unnecessary* if at least one of the following two conditions holds: 1) X occurs more than once in the body of C (in this case we say that X is a *shared* variable), 2) X does not occur in the head of C (in this case we say that X is an *existential* variable). Since unnecessary variables often determine multiple traversals of data structures and construction of intermediate data structures, the results of unnecessary variable elimination are often similar to those of loop fusion.

In the following example we recast loop fusion and unnecessary variable elimination in terms of the basic strategies presented in Section 4.1.

Example 23. (*Minimal Leaf Replacement*) Suppose that we are given a binary tree, say InTree, whose leaves are labelled by numbers. We want to obtain another tree, say OutTree, of the same shape with all its leaves replaced by their minimal value. This can be done by first computing the minimal leaf value, say Min, of InTree, and then visiting again Intree for replacing its leaves by Min. A program which realizes this algorithm is as follows:

```
1.  mintree(InTree, OutTree) ← minleaves(InTree,Min),
                               replace(Min,InTree,OutTree)
2.  minleaves(tip(N),N) ←
3.  minleaves(tree(L,R),Min) ←  minleaves(L,MinL), minleaves(R,MinR),
                                min(MinL,MinR,Min)
4.  replace(M,tip(N),tip(M)) ←
5.  replace(Min,tree(InL,InR),tree(OutL,OutR)) ←  replace(Min,InL,OutL),
                                                  replace(Min,InR,OutR)
```

where min(M1,M2,M) holds iff M is the minimum number between M1 and M2.

We would like to derive a program which traverses InTree only once. This could be done by applying the loop fusion method and obtaining a new program where the computations corresponding to minleaves and replace are performed by one predicate only. The same result can be achieved by eliminating the shared variables whose bindings are binary trees, and in particular, the variable InTree in clause 1.

To this aim we may apply the tupling strategy to the predicates minleaves and replace which share the argument InTree. Since the atoms to be tupled together constitute the whole body of clause 1 defining the predicate mintree, we do not need to introduce a new predicate, and we only need to look for the recursive definition of the predicate mintree. After some unfolding steps, we get:

```
6.  mintree(tip(N),tip(N)) ←
7.  mintree(tree(InL,InR),tree(OutL,OutR)) ←
        minleaves(InL,MinL), minleaves(InR,MinR), min(MinL,MinR,Min),
        replace(Min,InL,OutL), replace(Min,InR,OutR)
```

As suggested by the tupling strategy, we may now look for a fold of the goal 'minleaves(InL,MinL), replace(Min,InL,OutL)' using clause 1. Unfortunately, no matching is possible because this goal is not an instance of 'minleaves(InTree,Min), replace(Min,InTree,OutTree)'. Thus, we apply the generalization strategy and we introduce the following clause:

```
8.  genmintree(InTree,M1,M2,OutTree) ← minleaves(InTree,M1),
                                       replace(M2,InTree,OutTree)
```

whose body is the most specific generalization of the two goals to be folded, that is, 'minleaves(InL,MinL), replace(Min,InL,OutL)' and the body of clause 1. By folding clause 1 we get:

```
1f. mintree(InTree,OutTree) ← genmintree(InTree,Min,Min,OutTree).
```

We are now left with the problem of finding the recursive definition of the predicate genmintree introduced in clause 8. This is an easy task, because we can perform the unfolding steps corresponding to those leading from clause 1 to clauses 6 and 7, and then we can use clause 8 for folding. After those steps we get the following final program:

```
1f.  mintree(InTree,OutTree) ← genmintree(InTree,Min,Min,OutTree)
9.   genmintree(tip(N),N,M,tip(M)) ←
10.  genmintree(tree(InL,InR),M1,M2,tree(OutL,OutR)) ←
                    genmintree(InL,ML1,M2,OutL),
                    genmintree(InR,MR1,M2,OutR), min(ML1,MR1,M1)
```

This program performs the desired tree transformation in one visit. Indeed, let us consider the evaluation of a query of the form ← mintree(t,T), where t is a ground binary tree and T is an unbound variable. During the visit of the input tree t, the predicate genmintree both computes the minimal leaf value M1 and replaces the leaves using the unbound variable M2. The instantiation of M2 to the minimal leaf value is performed by the unification of the variables M1 and M2 due to clause 1f of our final program.

Notice also that no shared variable whose binding is a binary tree occurs in the clauses defining mintree and genmintree. Thus, we have been successful in eliminating unnecessary variables. ■

4.2.3 CHANGING DATA REPRESENTATIONS

The choice of appropriate data structures is usually very important for the design of efficient programs. In essence, this is the meaning of Wirth’s motto “Algorithms + Data Structures = Programs” [Wirth 76].

However, it is sometimes difficult to identify the data structures which allow a very efficient execution of our algorithms before actually writing the programs. Moreover, complex data structures may complicate correctness proofs.

Program transformation has been proposed as a methodology for providing appropriate data structures in a *dynamic* way [Partsch 90, Chapter 8]: first the programmer writes a preliminary version of the program implementing a given algorithm using simple data structures, and then he transforms their representations while preserving program semantics and improving efficiency.

An example of transformational change of data representations is the transformation of logic programs which use lists into equivalent programs which use *difference-lists*. Difference-lists are data structures which are sometimes used for implementing algorithms that manipulate sequences of elements. The advantage of using difference-lists is that the concatenation of two sequences represented as difference-lists can often be performed in constant time, while the concatenation of standard lists takes linear time w.r.t. the length of the first list.

A difference-list can be thought of as a pair <L,R> of lists, denoted by L\R, such that there exists a third list X for which the concatenation of X and R is L [Clark-Tärnlund 77]. In that case we say that X is *represented* by the difference-list L\R. Obviously, a single list can be represented by many difference-lists.

Programs that use lists are often simpler to write and understand than the equivalent ones which make use of difference-lists. Several (semi)automatic methods for the transformation of programs which use lists into programs which use difference-lists have been proposed in the literature [Hansson-Tärnlund 82, Brough-Hogger 87, Zhang-Grant 88, Marriot-Søndergaard 93, Proietti-Pettorossi 93b].

The problem of obtaining programs which use difference-lists, instead of lists, can be formulated as follows. Let p(X,Y) be a predicate defined in a program P where Y is a list. We want to define the new predicate diff_p(X,L\R) which holds iff p(X,Y) holds and Y is represented by the difference-list L\R.

Let us assume that the concatenation of lists is defined in P by means of a predicate append(X,Y,Z) which holds iff the concatenation of X and Y is Z. Then, the desired transformation can often be achieved by applying the T&S-definition rule and introducing the following definition for the predicate diff_p [Zhang-Grant 88]:

D. diff_p(X,L\R) ← p(X,Y), append(Y,R,L).

Then we have to look for a recursive definition of the predicate diff_p, which should depend neither on p nor on append.

This can be done, as clarified by the following example, by starting from clause D and performing some unfolding and goal replacement steps, based on the associativity property of append, followed by folding steps using D.

We can then express p in terms of diff_p by observing that in the least Herbrand model of P $\cup$ {D}, diff_p(X,Y\[]) holds iff p(X,Y) holds. Thus, in our transformed program the clauses for predicate p can be replaced by the single clause:

E. p(X,Y) ← diff_p(X,Y\[]).

Example 24. (*List Reversal Using Difference-lists*) Let us consider the following program for reversing a list:

1. reverse([],[]) ←
2. reverse([H|T],R) ← reverse(T,R1), append(R1,[H],R)
3. append([],L,L) ←
4. append([H|T],L,[H|TL]) ← append(T,L,TL)

Given a list L of length n, the answer to the query ← reverse(L,R) is obtained in O(n^2) SLD-resolution steps. Indeed, for the evaluation of reverse(L,R) clause 2 is invoked n–1 times. Thus, n–1 calls to append are generated, and the evaluation of each of those calls requires O(n) SLD-resolution steps.

The above program can be improved by using a difference-list for representing the second argument of reverse. This is motivated by the fact that by clause 2 the list which appears as second argument of reverse is constructed by the predicate append, and as already mentioned, concatenation of difference-lists can be much more efficient than concatenation of lists.

We start off by applying the T&S-definition rule and introducing the clause:

D1. diff_rev(X,L\R) ← reverse(X,Y), append(Y,R,L).

The recursive definition of diff_rev can easily be derived as follows. We unfold clause D1 w.r.t. reverse(X,Y) and we get:

D2. diff_rev([],L\R) ← append([],R,L)
D3. diff_rev([H|T],L\R) ← reverse(T,R1), append(R1,[H],Y), append(Y,R,L).

By unfolding, clause D2 is replaced by:

D4. diff_rev([],R\R) ←

By using the unfold/fold proof method described in Section 3.1 we can prove the validity of the replacement law:

F. append(R1,[H],Y), append(Y,R,L) ≡ append(R1,[H|R],L)

w.r.t. Sem_H and the current program made out of clauses D3, D4, 1, 2, 3, and 4. Thus, we apply the goal replacement rule to clause D3 and we get:

D5. diff_rev([H|T],L\R) ← reverse(T,R1), append(R1,[H|R],L).

The above step is an application of the reversible goal replacement rule because law F is valid also w.r.t. the program we have obtained by this replacement step.

We now fold D5 using D1 and we get:

D6. diff_rev([H|T],L\R) ← diff_rev(T, L\[H|R])

which, together with clause D4, provides the desired recursive definition of diff_rev.

Notice that this last folding step is an application of T&S-folding and it is not an instance of the reversible folding rule (R13). Its correctness is not ensured by Theorem 8, because the transformation sequence corresponding to the above derivation is constructed by using the goal replacement rule. This folding step, however, is correct w.r.t. Sem_H as shown in [Tamaki-Sato 86].

Our final program which uses difference-lists, is obtained by replacing the clauses defining reverse by the following single clause (see clause E above):

D7. reverse(X,Y) ← diff_rev(X,Y\[]).

The derived program (made out of clauses D4, D6, and D7) takes O(n) SLD-resolution steps for reversing a list of length n. ■

A crucial step in the derivation of programs which use difference-lists introduction of the clause of the form:

D. diff_p(X,L\R) ← p(X,Y), append(Y,R,L)

which defines the eureka predicate diff_p. This eureka predicate can also be viewed as the invention of an *accumulator* variable, in the sense of the *accumulation strategy* [Bird 84]. Indeed, as indicated in Example 24 above, the third argument of diff_rev(X,L\R) can be viewed as an *accumulator* which at each SLD-resolution step stores the result of reversing the list visited so far.

In the following example we show that the invention of accumulator variables can be derived by using the basic strategies described in Section 4.1.

Example 25. (*Inventing Difference-lists by Generalization*) Let us consider again the initial program of Example 24. We would like to derive a program for list reversal which does *not* use the append predicate. We can do so by applying the tupling strategy to clause 2 (because of the shared variable R1) and introducing a eureka predicate new_rev:

N. new_rev(T,H,R) ← reverse(T,R1), append(R1,[H],R).

As suggested by the tupling strategy, we then look for a recursive definition of new_rev by performing unfolding and goal replacement steps followed by folding steps using N. We have the additional requirement that the recursive definition of new_rev should not contain any call to append. This requirement can be fulfilled if the final folding steps are performed w.r.t. a conjunction of the atoms of the form 'reverse(…), append(…)' and no other calls to append occur in the folded clauses.

The unfolding tree generated by some unfolding and goal replacement steps starting from clause N are depicted in Fig. 5.

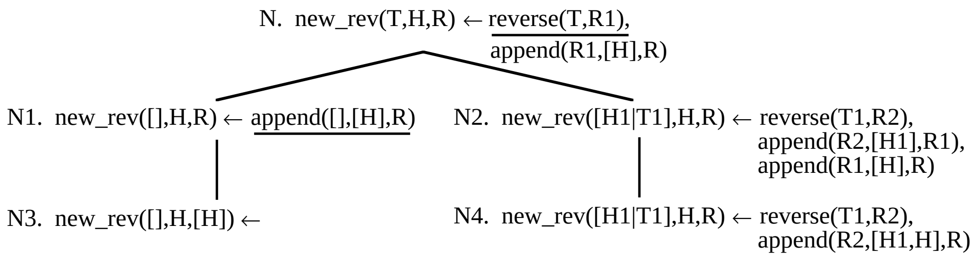
N. new_rev(T,H,R) ← reverse(T,R1), append(R1,[H],R)
N1. new_rev([],H,R) ← append([],[H],R)
N2. new_rev([H1|T1],H,R) ← reverse(T1,R2), append(R2,[H1],R1), append(R1,[H],R)
N3. new_rev([],H,[H]) ←
N4. new_rev([H1|T1],H,R) ← reverse(T1,R2), append(R2,[H1,H],R)

Figure 5. An unfolding tree for the reverse program.

Let us now consider clause N4 in the unfolding tree of Fig. 5. If we were able to fold it using the root clause N, we would have obtained the required recursive definition of new_rev. Unfortunately, that folding step is not possible because the argument [H1,H] of the call of append in clause N4 is not an instance of [H] in N (even if we rename the variables of the clauses).

Since N4 is a descendant of N we are in a situation where we can apply the generalization strategy. By doing so we introduce a new eureka predicate gen_rev defined by the following clause:

G1. gen_rev(T1,X,Y,R) ← reverse(T1,R2), append(R2,[X|Y],R).

where the bd(G1) is the most specific generalization of bd(N) and bd(N1).

The recursive definition of gen_rev can be found by replaying the transformation steps which lead from N to N4 in the unfolding tree. We get the following program:

G2. gen_rev([],X,Y,[X|Y]) ←
G3. gen_rev([H|T],X,Y,R) ← gen_rev(T,H,[X|Y],R).

We can then fold clause 2 using G1 and we get:

2f. reverse([H|T],R) ← gen_rev(T,H,[],R).

The final program which is made out of clauses 1, 2f, G2, and G3, has a computational behaviour similar to the program derived in Example 24. In particular, the third argument of gen_rev is used as an accumulator. ■

4.3 OVERVIEW OF OTHER TECHNIQUES

In this section we would like to give a brief account of some more techniques which have been presented in the literature for improving the efficiency of logic programs by using transformation methods.

Schematic Transformations

A common feature of the strategies we have described in Section 4.2 is that they made out of sequences of transformation rules which are not predefined; on contrary they depend on the structure of the programs derived during transformation process. The schema-based approach to progran transformation is complementary to the strategy-based one and it consists in providi a catalogue of predefined transformations of *program schemata*.

A program schema is an *abstraction* of a program, where some terms, conjunctions of literals, and clauses are replaced by meta-variables. If a schema S is an abstraction of a program P, then we say that P is an *instance* of S. Two schemata S_1 and S_2 are *equivalent* (w.r.t. a given semantics function Sem) iff for all the values of the meta-variables the corresponding instances P_1 and P_2 are equivalent (w.r.t. Sem). The transformation of a schema S_1 into a schema S_2 is *correct* (w.r.t. Sem) iff S_1 and S_2 are *equivalent* (w.r.t. Sem). Usually, we are interested in a schema transformation if each instance of the derived schema is more efficient of the corresponding instance of the initial one.

Given an initial program P_1, the schema-based program transformation technique works as follows. We first choose a schema S_1 which is an abstraction of P_1, then we choose a transformation of schema S_1 into schema S_2 in a given catalogue of correct schema transformations, and finally we instantiate S_2 to get the transformed program P_2.

The problem of proving the equivalence of program schemata has been addressed within various contexts (see, for instance, [Paterson-Hewitt 70, Walker-Strong 72, Huet-Lang 78]). Some methodologies for developing logic programs using program schemata are proposed in [Deville-Burnay 89, Kirschenbaum et al. 89, Fuchs-Fromherz 92] and some examples of logic program schema transformations can be found in [Brough-Hogger 87, Seki-Furukawa 87, Brough-Hogger 91]. The schema transformations presented in these papers are useful for *recursion removal* (see below) and for reducing nondeterminism in generate-and-test programs (see Section 4.2.1).

The main advantage of the schema-based approach over the strategy-based one is that the application of a schema transformation can be performed in constant time; however, the choice of a suitable schema transformation in the catalogue of the available ones does require some extra time. On the other hand, one of the drawbacks of the schema-based approach is the space requirements and the fact that, when the program to be transformed is not an instance of any schema in the catalogue, then no action can be performed.

Recursion Removal
Recursion is the main control structure for declarative (functional or logic) programs. Unfortunately, the extensive use of recursively defined procedures may lead to inefficiency in time and space. In the case of imperative programs some program transformation techniques that remove recursion in favour of iteration have been studied, for instance, in [Paterson-Hewitt 70, Walker-Strong 72].

In logic programming languages, where no iterative constructs are available, recursion removal can be understood as the derivation of tail-recursive clauses from recursive clauses. A definite clause is said to be *recursive* iff its head predicate also occurs in an atom of its body. A recursive clause is said to be *tail-recursive* iff it is of the form:

$$p(t) \leftarrow L, p(u)$$

where L is a conjunction of atoms. (For simplicity reasons in presenting this issue we restrict ourselves to definite programs.) A program is said to be tail-recursive iff all its recursive clauses are tail-recursive.

The elimination of recursion in favour of iteration can be achieved in two steps. First the given program is transformed into an equivalent, tail-recursive one, and then the derived tail-recursive program is executed in an efficient, iterative way by using an ad-hoc compiler optimization, called *tail-recursion optimization* (see [Bruynooghe 82] for a detailed description and the applicability conditions in the case of Prolog implementations).

Tail-recursion optimization (also called *last-call optimization*) makes sense only if we assume the left-to-right computation rule, so that, for instance, when the clause $p(t) \leftarrow L, p(u)$ is invoked, the recursive call p(u) is the last call to be evaluated.

In principle, any recursive clause can be transformed into a tail-recursive one by simply rearranging the order of the atoms in the body. This transformation is correct w.r.t. Sem_H (see rule R8). However, goal rearrangements can increase the amount of non-determinism, thus making useless the efficiency improvement gains due to tail-recursion optimization. Moreover, goal rearrangements do not preserve Prolog semantics (see Section 3.2.4), and tail-recursion optimization is usually applied to Prolog.

Thus, many researchers have elaborated more complex transformation strategies for obtaining tail-recursion without increasing the non-determinism. We would like to mention the following three approaches.

The first approach consists in transforming *almost-tail-recursive* clauses into tail-recursive ones [Debray 85, Azibi 87, Debray 88] by using unfold/fold rules. A clause is said to be almost-tail-recursive iff it is of the form:

$$p(t) \leftarrow L, p(u), R$$

where L is a conjunction of atoms and R, called the *tail-computation*, is a conjunction of atoms whose predicates do not depend on p. Usually, the tail-computation contains calls to 'primitive' predicates, such as the ones for computing concatenation of lists and arithmetic operations, such as addition and multiplication of integers. The transformation methods considered in [Debray 85, Azibi 87, Debray 88] are closely related to the ones considered by [Arsac-Kodratoff 82] for functional programs. They use the generalization strategy and some replacement laws which are valid for the primitive predicates, like, for instance, associativity of list concatenation, associativity and commutativity of addition, and distributivity of multiplication over addition.

The second approach is based on schema transformations [Bloch 84, Brough-Hogger 87, Brough-Hogger 91], where some almost-tail recursive program schemata are shown to be equivalent to tail-recursive ones.

The third approach consists in transforming a given program into a *binary* one, that is, a program whose clauses have only one atom in their bodies [Tarau-Boyer 90]. This transformation method is applicable to all programs and it is in the style of continuation-based transformations for functional programs [Wand 80]. The transformation works by adding to each predicate an extra argument (representing the so-called *continuation*), which encodes the next goal to be evaluated.

For instance, the clauses:

$$p \leftarrow$$
$$p \leftarrow p, q$$

are transformed into:

$$p \leftarrow p1(true)$$
$$p1(G) \leftarrow G$$
$$p1(G) \leftarrow p1((q,G)).$$

This transformation by itself does not improve efficiency. However, it allows us to use a specialized version of the Warren Abstract Machine [Warren 83], and to perform further efficiency improving transformations [Demoen 93, Neumerkel 93].

Annotations and Memoing

In this paper we have mainly considered transformations which do not make use of the extra-logical features of logic languages, like cuts, asserts, delay declarations, etc. In the literature, however, there are various papers which deal with transformation rules which preserve the operational semantics of full Prolog (see Section 3.2.4), and there are also some transformation strategies which work by inserting in a given Prolog program extra-logical predicates for improving efficiency by taking advantage of suitable properties of the evaluator. These strategies are related to some techniques which have been first introduced in the case of functional programs and are referred to as *program annotations* [Schwarz 82].

In the case of Prolog, a typical technique which produces annotated programs consists in adding a cut operator '!' in a point where the execution of the program can be performed in a deterministic way. For instance, the following two Prolog clauses:

```
p(X) ← C, Body1
p(X) ← not(C), Body2
```

can be transformed (if C has no side-effects) into:

```
p(X) ← C, !, Body1
p(X) ← Body2
```

The derived clauses are more efficient than the initial ones and behave like an if-then-else statement.

Prolog program transformations based on the insertion of cuts are reported in [Sawamura-Takeshima 85, Debray-Warren 89, Deville 90].

Other techniques which introduce annotations for the evaluator are related to the automatic generation of *delay declarations* [Naish 86, Wiggins 92], which procrastinate calls to predicates until they are suitably instantiated.

A last kind of annotation technique which has been used for improving program efficiency is the so-called *memoization* [Michie 68]. Results of previous computations are stored in a table together with the program itself, and when a query has to be evaluated that table is looked up first. This technique has been implemented in logic programming by enhancing the SLDNF-resolution compiler through tabulations [Warren 92] or by using the 'assert' predicate for the run-time updating of the programs [Sterling-Shapiro 86].

5. PARTIAL EVALUATION AND PROGRAM SPECIALIZATION

Partial evaluation (also called *partial deduction* in the case of logic programming) is a program transformation technique which allows us to derive a new program from an old one when part of the input data is known at compile time. This technique which can be considered as an application of the s-m-n theorem [Kleene 71, Chapter IX], has been extensively applied in the field of imperative and functional languages [Futamura 71, Ershov 77, Bjørner et al. 88, Jones et al. 93] and first used in logic programming by [Komorowski 82] (see also [Venken 84, Gallagher 86, Safra-Shapiro 86, Takeuchi 86, Takeuchi-Furukawa 86, Ershov et al. 88] for early papers on partial deduction, with special emphasis on the problem of partially evaluating meta-interpreters).

The resulting program may be more efficient than the initial one because by using the partially known input, it is possible to avoid some run-time computations which are performed at compile time.

Partial evaluation can be viewed as a particular case of *program specialization*

[Scherlis 81], which is aimed at transforming a given program by exploiting the knowledge of the context where that program is used. This knowledge can be expressed as a precondition which is satisfied by the input values of the program.

No much work has been done in the area of logic program specialization, apart from the particular case of partial deduction. Noteworthy exceptions are [Bossi et al. 90] and various papers by Gallagher and others [Gallagher et al. 88, Gallagher-Bruynooghe 91, de Waal-Gallagher 92]. In the latter papers the use of the abstract interpretation methodology has a crucial role. Within this methodology it is possible to represent and manipulate a possibly infinite set of input values which satisfies a given precondition, by considering, instead, an element of a finite abstract domain.

Abstract interpretations can be used *before* and *after* the application of program specialization during the so-called *preprocessing* phase and *postprocessing* phase, respectively. During the preprocessing phase, using abstract interpretations we may collect information depending on the control flow, such as groundness of arguments and determinacy of predicates. This information can then be exploited for directing the specialization process. Examples of this preprocessing are the binding time analysis performed by the Logimix partial evaluator of [Mogensen-Bondorf 93] and the determinacy analysis performed by Mixtus [Sahlin 91].

During the postprocessing phase, abstract interpretations may be used for improving the program obtained by the specialization process, as indicated, for instance, in [Gallagher 93] where it is shown how one can get rid of the so-called *useless clauses*.

The idea of partial evaluation can be presented as follows [Lloyd-Shepherdson 91]. Let us consider a normal program P and an atomic query $\leftarrow$ A. We construct a finite SLDNF-tree for P $\cup$ {$\leftarrow$ A} containing at least one non-root node. For this construction we use an *unfolding strategy* U which tells us the atoms which should be unfolded and when to terminate the construction of the tree.

The notion of unfolding strategy is analogous to the one of u-selection rule (see Section 4.1), but it applies to goals, instead of clauses. We then construct the set of clauses $\{A\theta_i \leftarrow G_i \mid i = 1, \ldots, n\}$, called *resultants*, obtained by collecting from eac non-failed leaf of the SLDNF-tree, the goal $\leftarrow G_i$ and the corresponding computed answer substitution θ_i.

A *partial evaluation of* P *w.r.t.* the atom A is the program P_A obtained from P by first replacing the clauses of P which constitute the definition of the predicate symbol, say p, occurring in A by the set of resultants $\{A\theta_i \leftarrow G_i \mid i = 1, \ldots, n\}$, and then keeping only the definitions of the predicates on which p depends.

The generation of *finite* SLDNF-trees can be performed within general frameworks for dealing with termination of unfolding as the ones described in [Bruynooghe et al. 92, Bol 93].

Example 26. Let us consider the following program P:

p([],Y) $\leftarrow$

```
p([H|T],Y) ← q(T,Y)
q(T,Y) ← Y = b
q(T,Y) ← p(T,Y)
```

and the atom A = p(X,a). Let us use the unfolding strategy U which performs unfolding steps starting from the query ← p(X,a) until each leaf of the SLDNF-tree is either a success or a failure or it has the same predicate of the root node. We get the tree depicted in Fig. 6.

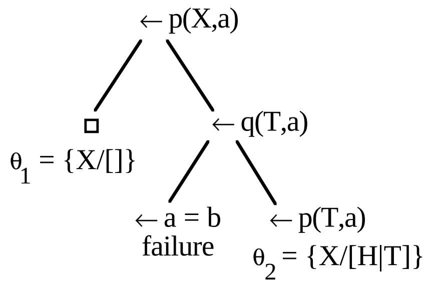

Figure 6. An SLDNF-tree for P ∪ {← p(X,a)} using U.

By collecting the goals and the substitutions corresponding to the leaves of that tree we have the following set of resultants:

```
p([],a) ←
p([H|T],a) ← p(T,a)
```

which constitute the partial evaluation P_A of P w.r.t. A. The clauses for q have been discarded because p does not depend on q in the derived program.

If we use the program P_A, the evaluation of an instance of the query ← p(X,a) is more efficient than the one using the initial program because the calls to the predicate q need not be computed and some failure branches are avoided. ■

The notion of partial evaluation of a program w.r.t. an atom can be extended to the one w.r.t. a set S of atoms by considering the set union of all resultants of the atoms in S. Theorem 23 below establishes the correctness of partial evaluation. First we need the following definitions.

DEFINITION 21. Given a set R of normal clauses and a set S of atoms, we say that R is S-*closed* iff an atom in R with predicate symbol occurring in S is an instance of an atom in S. ■

DEFINITION 22. Given a set S of atoms, we say that S is *independent* iff no two atoms in S have a common instance. ■

THEOREM 23. (*Correctness of Partial Evaluation*) [Lloyd-Shepherdson 91]. Given a normal program P and an independent set S of atoms, let us consider a partial evaluation P_S of P w.r.t. S. Then for every atomic query $\leftarrow$ A such that A is an instance of an atom in S and P_S is S-closed, we have that:

i) $Sem_{SS}(P, \leftarrow A) = Sem_{SS}(P_S, \leftarrow A)$, and

ii) $Sem_{FF}(P, \leftarrow A) = Sem_{FF}(P_S, \leftarrow A)$. ■

This theorem can be extended to the case of computed answer substitution semantics and normal queries [Lloyd-Shepherdson 91].

In Example 26, the correctness of program P_A resulting from the partial evaluation process, follows from Theorem 23, because for the singleton {p(X,a)} the independence property trivially holds. The closedness property also holds because p([],a), p([H|T],a), and p(T,a) are all instances of p(X,a).

The closedness and independence hypotheses cannot be dropped from Theorem 23, as it is shown by the following example.

Example 27. Suppose we want to partially evaluate the following program P:

p(a) $\leftarrow$ p(b)
p(b) $\leftarrow$

w.r.t. the atom p(a). We can derive the resultant p(a) $\leftarrow$ p(b). Thus, a partial evaluatio of P w.r.t. p(a) is the program P_a:

p(a) $\leftarrow$ p(b)

obtained by replacing the definition of p in P by the resultant p(a) $\leftarrow$ p(b). P_a is not {p(a)}-closed and we have that $Sem_{SS}(P_a, \leftarrow p(a)) = \emptyset$, while $Sem_{SS}(P, \leftarrow p(a)) =$ {p(a)}.

Now, consider the following program Q:

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

and the set S of atoms {p, q(X), q(a)} which is not independent. A partial evaluation of Q w.r.t. S is the following program Q_S:

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

Q_S is S-closed and $Sem_{SS}(Q_S, \leftarrow p) = \{p\}$, while $Sem_{SS}(Q, \leftarrow p) = \emptyset$, because the unique SLDNF-derivation for $Q \cup \{\leftarrow p\}$ flounders. ■

Various strategies have been proposed in the literature for computing from a giv program P and atomic query $\leftarrow$ A, the set S of atoms with the independence an closedness properties required by Theorem 23 [Benkerimi-Lloyd 90, Bruynooghe et al. 92, Gallagher 91, Martens et al. 92]. Some of them require generalization steps and the use of abstract interpretations.

Other techniques for partial evaluation and program specialization are based on the unfold/fold rules [Fujita 87, Bossi-Cocco 90, Sahlin 91, Prestwich 93b, Proietti-Pettorossi 93a]. By using those techniques, given a program P and a set of atoms S = $\{A_1,\ldots,A_m\}$, for i=1,…,m we introduce a new predicate $newp_i$ defined by:

D_i: $newp_i(X_1,\ldots,X_n) \leftarrow A_i$

where $X_1,\ldots,X_n$ are the variables occurring in A_i.

When using the definition, unfolding, and folding rules the correctness of the derived programs is ensured by the results presented in Section 3, instead of Theorem 23. In particular, as discussed in Section 3.3, for any program P and atomic query $\leftarrow newp_i(t_1,\ldots,t_n)$ we have that:

i) $Sem_{SS}(P \cup \{D_1,\ldots,D_m\}, \leftarrow newp_i(t_1,\ldots,t_n)) = Sem_{SS}(Q, \leftarrow newp_i(t_1,\ldots,t_n))$, and

ii) $Sem_{FF}(P \cup \{D_1,\ldots,D_m\}, \leftarrow newp_i(t_1,\ldots,t_n)) = Sem_{FF}(Q, \leftarrow newp_i(t_1,\ldots,t_n))$

where Q is any program derived from $P \cup \{D_1,\ldots,D_m\}$ by applying the transformation rules according to the restrictions of Theorems 11 and 12.

Let us now briefly compare the two approaches to partial evaluation we have mentioned above, that is, the one based on Theorem 23 and the one based on the unfold/fold rules. In the approach based on Theorem 23 the efficiency gains are obtained by constructing SLDNF-trees and extracting resultants. This process corresponds to the application of some unfolding steps, and since efficiency gains are obtained without using the folding rule, it may seem that this is an exception to the 'need for folding' meta-strategy of Section 4. However, in order to guarantee the correctness of the partial evaluation of a given program P w.r.t. a set of atoms S, for each element of S we are required to find an SLDNF-tree whose leaves contain *instances* of atoms in S (see the closedness condition), and as the reader may easily verify, this requirement exactly corresponds to the 'need for folding'.

Conversely, the second approach based on the unfold/fold rules, does not require the closedness and independence conditions, but as we show in Example 28 below, we need to perform some final folding steps using the clauses $D_1,\ldots,D_m$ corresponding to the atoms in S.

In this second approach the use of the *renaming* technique for structure specialization [Benkerimi-Lloyd 90, Gallagher-Bruynooghe 90] which is often required in the first approach, is not needed, as indicated by the following example. In this example we derive by unfold/fold essentially the same program obtained by renaming in [Gallagher 93]. For other issues concerning the use of folding during partial evaluation the reader may refer to [Owen 89].

We now present an example of derivation of a partial evaluation of a program by applying the unfold/fold transformation rules and the loop absorption strategy.

Example 28. (*String Matching*) [Sahlin 91, Gallagher 93]. Let us consider the following program M for string matching:

1. match(P,T) ← match1(P,T,P,T)
2. match1([],X,Y,Z) ←
3. match1([A|Ps],[A|Ts],P,T) ← match1(Ps,Ts,P,T)
4. match1([A|Ps],[B|Ts],P,[C|T]) ← ¬ (A=B), match1(P,T,P,T)

where the pattern P and the string T are represented as lists, and match(P,T) holds iff the pattern P occurs in the string T.

We want to partially evaluate the given program w.r.t. the atom match([a,a,b],X). In order to do so we first introduce the following definition:

5. newp1(X) ← match([a,a,b],X)

whose body is the atom w.r.t. which the partial evaluation should be performed. As usual when applying the definition rule, the name of the head predicate is a new symbol, newp1 in our case. Then we construct the unfolding tree for <M, clause 5> using the u-selection rule which unfolds the leftmost positive atom, if any.

The u-selection rule terminates the construction of the unfolding tree when for each clause C at a leaf of the tree at hand we have that:

i) the predicates match and match1 do not occur in bd(C), or

ii) C is a clause with finitely failed body, or

iii) all atoms in bd(C) with predicate match or match1 can be folded using one of the definitions introduced so far.

The u-selection rule also terminates the construction of the unfolding tree when we can apply the loop absorption strategy, that is, an atom in the body of a clause at a leaf L is an instance of an atom in the body of a clause occurring in an ancestor node of L.

By using this u-selection rule we get the tree depicted in Fig. 7.

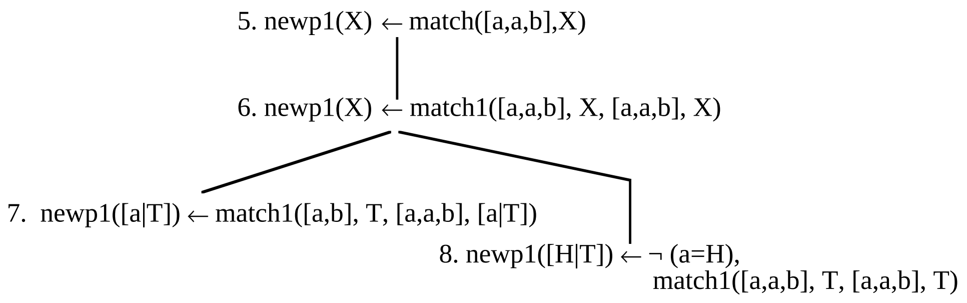
5. newp1(X) ← match([a,a,b],X)
6. newp1(X) ← match1([a,a,b], X, [a,a,b], X)
7. newp1([a|T]) ← match1([a,b], T, [a,a,b], [a|T])
8. newp1([H|T]) ← ¬ (a=H),
match1([a,a,b], T, [a,a,b], T)

Figure 7. An unfolding tree for <M, newp1(X) ← match([a,a,b],X)>.

In clause 8 the atom match1([a,a,b], T, [a,a,b], T) is an instance of the body of clause 6. Thus, we can apply the loop absorption strategy and we introduce the new definition:

9. newp2(T) ← match1([a,a,b], T, [a,a,b], T).

We fold clause 6 using clause 9 and we get:

6f. newp1(X) ← newp2(X).

Now the unfold/fold derivation continues by constructing the unfolding tree for <M, clause 9>, which is depicted in Fig. 8 (where we used the same u-selection rule described above).

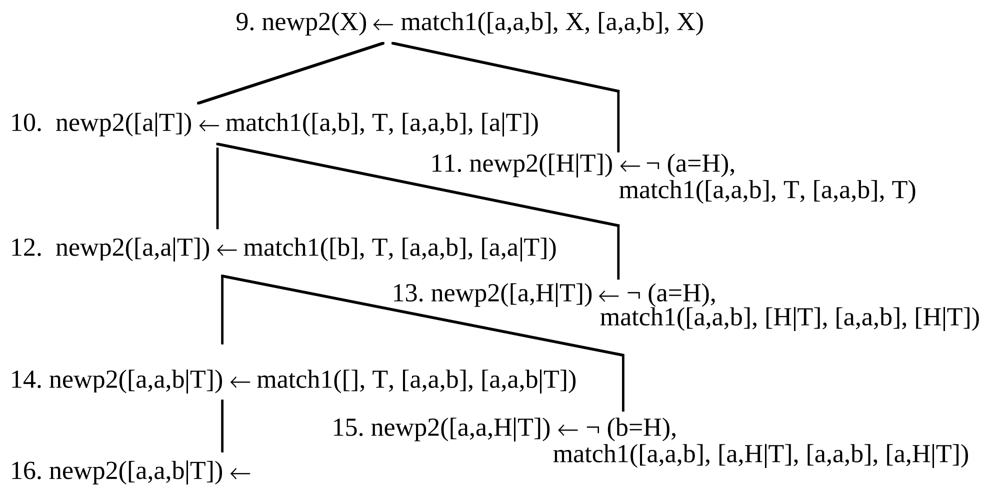
9. newp2(X) ← match1([a,a,b], X, [a,a,b], X)
10. newp2([a|T]) ← match1([a,b], T, [a,a,b], [a|T])
11. newp2([H|T]) ← ¬ (a=H), match1([a,a,b], T, [a,a,b], T)
12. newp2([a,a|T]) ← match1([b], T, [a,a,b], [a,a|T])
13. newp2([a,H|T]) ← ¬ (a=H), match1([a,a,b], [H|T], [a,a,b], [H|T])
14. newp2([a,a,b|T]) ← match1([], T, [a,a,b], [a,a,b|T])
15. newp2([a,a,H|T]) ← ¬ (b=H), match1([a,a,b], [a,H|T], [a,a,b], [a,H|T])
16. newp2([a,a,b|T]) ←

Figure 8. An unfolding tree for <M, newp2(X) ← match1([a,a,b], X, [a,a,b], X)>.

By folding clauses 15, 13, and 11, we get the following program:

6f.	newp1(X) ← newp2(X)
16.	newp2([a,a,b|T]) ←
15f.	newp2([a,a,H|T]) ← ¬(b=H), newp2([a,H|T])
13f.	newp2([a,H|T]) ← ¬(a=H), newp2([H|T])
11f.	newp2([H|T]) ← ¬(a=H), newp2(T)

which is exactly the program produced by the Mixtus partial evaluator of [Sahlin 91, p. 124]. ■

One of the most relevant motivations for developing the partial evaluation methodology, is that it can be used for compiling programs and for deriving compilers from interpreters, via the Futamura projections technique [Futamura 71]. For this last application it is necessary that the partial evaluator be *self-applicable*, that is, able to partially evaluate itself. The interested reader may refer to [Jones et al. 93] for a general overview, and to [Fujita-Furukawa 88, Fuller-Abramsky 88, Mogensen-Bondorf 93, Gurr 93] for more details on the problem of self-applicatbility of partial evaluators in the logic languages Prolog and Gödel.

Partial evaluation has also been used in the area of deductive databases for deriving very efficient techniques for recursive query optimization. Some results in this direction can be found in [Bry 89].

6. RELATED METHODOLOGIES FOR PROGRAM DEVELOPMENT

From what we have presented above it should be clear that program transformation is a methodology for program development which is very much related to various fields of Theoretical Computer Science and Software Engineering. Here we want to briefly indicate some of the techniques and methods which are used in those fields which are of some relevance to the transformation methodology and its application.

Let us begin by considering some of the *analysis* techniques by which the programmer may investigate various properties of the programs at hand. Those properties may then be used for improving efficiency by applying transformation methods. Program properties which are often useful for program transformation concern, for instance, the flow of computation, the use of data structures, the propagation of bindings, the sharing of information among arguments, the termination for a given class of queries, the groundness and freeness of arguments, and the functionality (or determinacy) of a predicate.

Perfect knowledge about these properties is in general impossible to obtain, because of undecidability limitations. However, it is often the case that approximate reasoning can be carried out by using abstract interpretation techniques [Debray 92]. They make use of finite interpretation domains where information can be obtained by a finite amount of computation. The interpretation domains vary according to the property to be analyzed and the degree of information one would like to obtain [Cortesi et al. 92].

A general framework where program transformation strategies are supported by abstract interpretation techniques is defined in [Boulanger-Bruynooghe 93]. Among the many transformation techniques which strongly depend on program analysis techniques we would like to mention: i) Compiling Control (see Section 4.2.1), where the information about the flow of computation is used for generating the unfolding tree, ii) the specialization method of [Gallagher-Bruynooghe 91] which is based on a technique for approximating the set of all possible calls generated during the evaluation of a given class of queries, iii) various techniques which insert cuts on the basis of a determinacy information (see Section 4.3), and iv) various techniques implemented in the Spes system [Alexandre et al. 92] in which mode analysis is used to mechanize several transformation strategies.

Very much related with these methodologies for the analysis of programs are the ones for the proof of properties of programs. They have been used for program verification, and in particular for making sure that a given set of clauses satisfies a given specification, or a given first order formula is true in a chosen semantic domain.

These proofs may be used for driving the application of suitable instances of the goal replacement rule.

Many proofs techniques can be found in the literature, and in particular, in the field of Theorem Proving and Computer Aided Deduction. For the ones which have been used for logic programs and can be adapted for program transformation we may recall those in [Drabent-Maluszynski 88, Bossi-Cocco 89, Deransart 89].

The field of program transformation partially overlaps with that of *program synthesis*. Indeed, if we consider the given initial program as a program specification then the final program derived by transformation can be considered as an implementation of such specification. However, it is usually understood that program synthesis differs from program transformation because the specification is a somewhat implicit description of the program to be derived. Such implicit description often does not allow us to get the desired program by simple manipulations, like the one obtainable by standard transformation rules. Moreover, it is often the case that the specification language differs from the executable language in which the final program should be written. This language barrier can be overcome by using transformation rules, but these techniques, we think, go beyond the area of traditional program transformation and more precisely belong to the field of logic program synthesis for which we refer to [Deville-Lau 94].

Finally, we would like to mention that the transformation and specialization techniques considered in this paper have been partially extended to concurrent logic programs [Ueda-Furukawa 88] and constraint logic programs [Hickey-Smith 91].

7. CONCLUSIONS

We have looked at the theoretical foundations of the so-called 'rules + strategies' approach to logic program transformation. We established a unified framework where to consider and compare the various rules which have been proposed in the literature. That framework is parametric with respect to the semantics which is preserved during transformation. We have presented various sets of transformation rules and the corresponding correctness results w.r.t. the following semantics: the least Herbrand model, the computed answer substitutions, the finite failure, and the pure Prolog semantics. We have also considered the case of normal programs, and using the proposed framework we presented the rules which preserve finite failure, success set, and Clark's completion semantics. Our presentation could have been extended by considering various other semantics for normal programs available in the literature, but space limitations prevented us from doing so.

We have also presented a unified framework in which it is possible to describe some of the most significant methods which have been proposed for the transformation of logic programs. We have singled out a few strategies, such as the tupling, the loop absorption, and the generalization strategies and we have shown that the basic techniques related to compiling control, program composition, change of data representation, partial evaluation, and program specialization can be viewed as suitable applications of those strategies.

An area of further investigation is the characterization of the power of the transformation rules and strategies, both in the 'completeness' sense, that is, their capability of deriving equivalent programs, and in the 'complexity' sense, that is, their capability of deriving programs which are more efficient than the given ones. No conclusive results are available in these directions.

A line of research that can be pursued in the future is the integration of tools, like abstract interpretations, proofs of properties, and program synthesis, within the rules + strategies approach to program transformation.

The impact of the transformational methodology in the practice of logic program development is still small. However, it is recognized that the automation of transformation techniques and their use in a system for software development is of crucial importance. There is a growing interest in the mechanization of transformation strategies, the production of interactive tools for implementing program transformers, and the development of optimizing compilers which make use of the transformation techniques.

The importance of the transformation methodology will substantially increase by extending its theory and applications to the case of complex logic languages which include features, like constraints, parallelism, concurrency, and object-orientation.

ACKNOWLEDGEMENTS

We express our warm gratitude to M. Bruynooghe, J.P. Gallagher, M. Leuschel, M. Maher, T. Sato, H. Seki and the anonymous referees for their helpful suggestions and comments. Many thanks also to O. Aioni and M. Gaspari for their careful reading of a previous draft of the paper.

REFERENCES

[Aerts-Van Besien 91] Aerts, K., and Van Besien, D.: “Autolap: A System for Transforming Logic Programs” Department of Electronics, Technical Report, University of Rome II, 1991.

[Alexandre et al. 92] Alexandre, F., Bsaïes, K., Finance, J.P., and Quéré, A.: “Spes: A System for Logic Program Transformation” Proceedings of the International Conference on Logic Programming and Automated Reasoning, LPAR ’92, Lecture Notes in Computer Science No. 624, 1992, pp. 445–447.

[Amtoft 92] Amtoft, T.: “Unfold/fold Transformations Preserving Termination Properties” Proc. PLILP ’92, Leuven, Belgium, LNCS n. 631, Springer-Verlag, 1992, pp.187–201.

[Apt 90] Apt, K.R.: “Introduction to Logic Programming”, in: Handbook of Theoretical Computer Science, van Leeuwen, J. (Ed.), Elsevier, 1990, pp. 493–576.

[Arsac-Kodratoff 82] Arsac, J. and Kodratoff, Y.: “Some Techniques for Recursion Removal From Recursive Functions” ACM Toplas 4 (2), 1982, pp. 295–322.

[Azibi 87] Azibi, N.: “TREQUASI: Un système pour la transformation automatique de programmes PROLOG récursifs en quasi-itératifs” PhD thesis, Université de Paris-Sud, Centre D’Orsay, France, 1987.

[Baudinet 92] Baudinet, M.: “Proving Termination Properties of Prolog Programs: A Semantic Approach” J. Logic Programming, 14, 1992, pp. 1–29.

[Benkerimi-Lloyd 90] Benkerimi, K. and Lloyd, J.W.: “ A Partial Evaluation Procedure for Logic Programs”, in: Proc. 1990 North American Conference on Logic, Austin, Texas (USA), MIT Press, 1990, pp. 343-358.

[Bird 84] Bird, R.S.: “The Promotion and Accumulation Strategies in Transformational Programming”, Toplas ACM Vol. 6, No. 4, 1984, pp. 487–504.

[Bjørner et al. 88] Bjørner, D., Ershov, A.P., and Jones, N.D. (Eds.): “Partial Evaluation and Mixed Computation” IFIP TC2 Workshop on Partial and Mixed Computation, Gammel Avernæs, Denmark, Elsevier Science Publishers, North Holland, 1988.

[Bloch 84] Bloch, C.: “Source-to-Source Transformations of Logic Programs” Master Thesis, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1984.

[Bol 93] Bol, R.: “Loop Checking in Partial Deduction” J. Logic Programming 16, 1993, pp. 25–46.

[Bossi-Cocco 89] Bossi, A. and Cocco, N.: “ Verifying Correctness of Logic Programs”, Proc. TAPSOFT 89, Vol.2, LNCS n.352, Springer-Verlag, 1989, pp.

96–110.

[Bossi-Cocco 93] Bossi, A. and Cocco, N.: “Basic Transformation Operations for Logic Programs which Preserve Computed Answer Substitutions” Journal of Logic Programming, 16, 1993, pp. 47–87.

[Bossi et al. 90] Bossi, A. Cocco, N., and Dulli, S.: “ A Method for Specializing Logic Programs”, ACM TOPLAS, vol. 12, 2, 1990, pp. 253-302.

[Bossi et al. 92a] Bossi, A., Cocco, N., and Etalle, S.: “Transforming Normal Programs by Replacement” In: A. Pettorossi (Ed.), Proc. 3rd International Workshop on Meta-Programming in Logic (Meta ’92), Uppsala, Sweden, LNCS n. 649, Springer-Verlag, 1992, pp. 265–279.

[Bossi et al. 92b] Bossi, A., Cocco, N., and Etalle, S.: “On Safe Folding” Proc. PLILP ’92, Leuven, Belgium, LNCS n. 631, Springer-Verlag, 1992, pp.172–186.

[Boulanger-Bruynooghe 93] Boulanger, D. and Bruynooghe, M.: “Deriving Unfold/Fold Transformations of Logic Programs Using Extended OLDT-based Abstract Interpretation” J. Symbolic Computation, 15, 1993, pp. 495–521.

[Boyer-Moore 74] Boyer R.S. and Moore, J.S.: “Proving Theorems about LISP Functions” JACM, 22 (1), 1974, pp. 129–144.

[Brough-Hogger 87] Brough, D.R. and Hogger, C.J.: “Compiling Associativity into Logic Programs” J. Logic Programming, 4, 1987, pp. 345–359.

[Brough-Hogger 91] Brough, D.R. and Hogger, C.J.: “Grammar-Related Transformations of Logic Programs” New Generation Computing, 9, 1991, pp. 115–134.

[Bruynooghe 82] Bruynooghe, M.: “The Memory Management of Prolog Implementations” In: Clark, K.L. and Tärnlund, S.-Å. (Eds.): “Logic Programming” Academic Press, 1982, pp. 83–98.

[Bruynooghe et al. 89] Bruynooghe, M., De Schreye, D., and Krekels, B.: “Compiling Control” Journal of Logic Programming, 6 (1&2), 1989, pp. 135–162.

[Bruynooghe et al. 92] Bruynooghe, M., De Schreye, D., and Martens, B.: “ A General Criterion for Avoiding Infinite Unfolding during Partial Deduction of Logic Programs”, New Generation Computing, 11, 1992, 47–79.

[Bruynooghe-Pereira 84] Bruynooghe, M. and Pereira, L.M.: “Deduction Revision by Intelligent Backtracking” In: Campbell, J.A. (Ed.): Implementations of Prolog, Ellis Horwood, 1984, pp. 253–266.

[Bry 89] Bry, F.: “Query Evaluation in Recursive Data Bases: Bottom-up and Top-down Reconciled” Proc. 1st Int. Conf. on Deductive and Object- Oriented Databases, Kyoto, Japan, 1989.

[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 78] Clark, K.: "Negation as Failure", in Logic and Data Bases, Gallaire, H. and Minker, J. (Eds.), Plenum Press, New York, 1978, pp. 293–322.

[Clark-Sickel 77] Clark, K. and Sickel, S.: "Predicate Logic: A Calculus for Deriving Programs" Proc. 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts (USA), 1977.

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

[Cortesi et al. 92] Cortesi, A., Filé, G., and Winsborough, W.: "Comparison of Abstract Interpretations" In: Proc. Nineteenth ICALP, Lecture Notes in Computer Science 623, 1992.

[Cousot-Cousot 77] Cousot, P. and Cousot, R.: "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints" 4th POPL, 1977, pp. 238-252.

[Darlington 72] Darlington, J.: " A Semantic Approach to Automatic Program Improvement" Ph.D. Thesis, Department of Artificial Intelligence, Edinburgh University, Edinburgh, U.K., 1972.

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

[Debray 85] Debray, S. K.: "Optimizing Almost-Tail-Recursive Prolog Programs" Proc. IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France, LNCS n. 201, Springer-Verlag, 1985, pp. 204–219.

[Debray 88] Debray, S. K.: "Unfold/Fold Transformations and Loop Optimization of Logic Programs", Proceedings SIGPLAN 88, Conference on Programming Language Design and Implementation, Atlanta, Georgia (U.S.A.), SIGPLAN Notices, 23, (7), 1988, pp. 297–307.

[Debray 92] Debray, S.K. (Ed.), Special Issue of the Journal of Logic Programming on Abstract Interpretation, Vol. 12, No. 2&3, 1992.

[Debray-Mishra 88] Debray, S.K. and Mishra, P.: "Denotational and Operational Semantics for Prolog", J. Logic Programming, Vol. 5, 1988, pp. 61-91.

[Debray-Warren 89] Debray, S.K., and Warren, D.S.: "Functional Computations in Logic Programs", ACM TOPLAS, 11 (3), 1989, pp. 451-481.

[Demoen 93] Demoen, B.: "On the Transformation of a Prolog Program to a More Efficient Binary Program" In: K.K. Lau and T. Clement (Eds.) Logic Program Synthesis and Transformation, Proc. Lopstr '92, Workshops in Computing, Springer-Verlag, 1993.

[Deransart 89] Deransart, P.: “Proof Methods of Declarative Properties of Logic Programs”, Proc. TAPSOFT 89, Vol.2, LNCS n.352, Springer-Verlag, 1989, pp. 207–226.

[De Schreye-Bruynooghe 89] De Schreye, D. and Bruynooghe, M.: “On the Transformation of Logic Programs with Instantiation Based Computation Rules” J. Symbolic Computation, 7, 1989, 125–154.

[De Schreye et al. 91] De Schreye, D., Martens, B., Sablon, G., and Bruynooghe, M.: “Compiling Bottom-up and Mixed Derivations into Top-down Executable Logic Programs” Journal of Automated Reasoning, 7, 1991, pp. 337–358.

[Deville 90] Deville, Y.: “Logic Programming: Systematic Program Development” Addison-Wesley, 1990.

[Deville-Burnay 89] Deville, Y. and Burnay, J.: “Generalization and Program Schemata” In: Proc. NACLP 89, MIT Press, 1989, pp. 409–425.

[Deville-Lau 94] Deville, Y. and Lau, K.-K.: “Logic Program Synthesis”, J. Logic Programming, this issue.

[de Waal-Gallagher 92] de Waal, D.A and Gallagher, J P.: “Specialization of a Unification Algorithm” In: T. Clement and K.-K. Lau (Eds.): Logic Program Synthesis and Transformation (Proc. Lopstr ’91), Workshops in Computing, Springer-Verlag, 1992, pp. 205–221.

[Drabent-Maluszynski 88] Drabent, W. and Maluszynski, J.: “Inductive Assertion Method for Logic Programs” Theoretical Computer Science, 59, 1988, pp. 133–155.

[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.

[Ershov 77] Ershov, A.P.: “On the Partial Computation Principle” Information Processing Letters, Vol. 6, No. 2, 1977, pp. 38–41.

[Ershov et al. 88] Ershov, A.P. et al.: Special Issue: Workshop on Partial Evaluation and Mixed Computation, New Generation Computing, Vol. 6, Nos. 2,3, 1988.

[Feather 82] Feather, M.S.: “ A System for Assisting Program Transformation”, ACM Trans. Program. Lang. Syst. 4 (1), 1982, pp. 1–20.

[Feather 87] Feather, M.S.: “ A Survey and Classification of Some Program Transformation Techniques”, in: Meertens, L.G.L.T. (Ed.), Program Specification and Transformation, Proc. IFIP TC2/W.G.2.1 Working Conference, Bad Tölz, Germany, 1986, Elsevier Science Publishers, North Holland, 1987, pp. 165–195.

[Fitting 85] Fitting, M.: “ A Kripke-Kleene Semantics for Logic Programs” Journal of

Logic Programming, 2(4), 1985, pp. 295–312.

[Fuchs-Fromherz 92] Fuchs, N.E. and Fromherz, M.P.J.: “Schema-based Transformations of Logic Programs” In: T. Clement and K.-K. Lau (Eds.) Logic Program Synthesis and Transformation, Proc. Lopstr ’91, Workshops in Computing, Springer-Verlag, 1992, pp. 111–125.

[Fujita 87] Fujita H.: “An Algorithm for Partial Evaluation with Constraints” ICOT Technical Memorandum TM–0367, Japan, 1987.

[Fujita-Furukawa 88] Fujita, H. and Furukawa, K.: “ A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation” In [Ershov et al. 88], pp. 91–118.

[Fuller-Abramsky 88] Fuller, D.A. and Abramsky, S.: “Mixed Computation of Prolog Programs” In [Ershov et al. 88], pp. 119–141.

[Futamura 71] Futamura, Y.: “Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler” Systems, Computers, Controls 2 (5): 45-50, 1971.

[Gallagher 86] Gallagher, J.P.: “Transforming Programs by Specializing Interpreters” In: Proceedings Seventh European Conference on Artificial Intelligence, ECAI ’86, 1986, pp. 109–122.

[Gallagher 91] Gallagher, J.P.: “ A System for Specializing Logic Programs” Technical Report TR-91-32, University of Bristol, November 1991.

[Gallagher 93] Gallagher, J.P.: “Tutorial on Specialization of Logic Programs” Proceedings of PEPM ‘93 ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, Copenhagen, Denmark, 1993, pp. 88-98.

[Gallagher-Bruynooghe 90] Gallagher, J.P. and Bruynooghe, M.: “Some Low-level Source Transformations for Program Specialization” Proceedings Meta ‘90 Workshop on Meta-Programming in Logic (Bruynooghe, M., Ed.), Leuven, Belgium, 1990, pp. 229-244.

[Gallagher-Bruynooghe 91] Gallagher, J.P. and Bruynooghe, M.: “The Derivation of an Algorithm for Program Specialisation” New Generation Computing 6(2), 1991, pp. 305-333.

[Gallagher et al. 88] Gallagher, J.P., Codish, M., and Shapiro, E.: “Specialization of Prolog and FCP Programs Using Abstract Interpretation” In [Ershov et al. 88], pp. 159–186.

[Gardner-Shepherdson 91] Gardner, P.A. and Shepherdson, J.C.: “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.

[Gelfond-Lifschitz 88] Gelfond, M. and Lifschitz, V.: “The Stable Model Semantics for Logic Programming” Proceedings of the Fifth International Conference and Symposium, The MIT Press, 1988, pp. 1070–1080.

[Gurr 93] Gurr, C.A.: “ A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel”, Ph. D. Thesis, Department of Computer Science, University of Bristol, U.K. 1993.

[Hansson-Tärnlund 82] Hansson, Å. and Tärnlund, S.-Å.: “Program Transformation by Data Structure Mapping” Logic Programming (Clark, K.L. and Tärnlund, S.-Å., Eds.), Academic Press, 1982, pp.117–122.

[Hickey-Smith 91] Hickey, T.J., and Smith, D.A.: “ Towards the Partial Evaluation of CLP Languages” Proc. ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM ’91, New Haven, CT (U.S.A.) 1991, SIGPLAN NOTICES, 26, 9, 1991, pp. 43–51.

[Hogger 81] Hogger C J.: “Derivation of Logic Programs”, JACM, 28, (2) (1981), pp. 372–392.

[Huet-Lang 78] Huet, G. and Lang, B.: “Proving and Applying Program Transformation Expressed with Second-Order Patterns”, Acta Informatica 11, 1978, pp. 31–55.

[Jones et al. 93] Jones, N.D., Gomard, C.K., and Sestoft, P.: “Partial Evaluation and Automatic Program Generation” Prentice Hall, 1993.

[Jones-Mycroft 84] Jones, N. and Mycroft, A.: “Stepwise Development of Operational and Denotational Semantics for Prolog”, in Proc. 1984 Int. Symp. on Logic Programming, Atlantic City, New Jersey, 1984, pp. 289-298.

[Kanamori-Fujita 86] Kanamori, T. and Fujita, H.: "Unfold/Fold Transformation of Logic Programs with Counters", ICOT Technical Report 179, Tokio, 1986.

[Kanamori-Horiuchi 87] Kanamori, T. and Horiuchi, K.: “Construction of Logic Programs Based on Generalized Unfold/Fold Rules” Proceedings of the Fourth International Conference on Logic Programming, The MIT Press, 1987, pp. 744–768.

[Kanamori-Maeji 86] Kanamori, T. and Maeji, M.: “Derivation of Logic Programs from Implicit Definition” Technical Report TR-178, ICOT, Tokyo, Japan, 1986.

[Kawamura-Kanamori 90] Kawamura, T. and Kanamori, T.: “Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation”, Theoretical Computer Science 75, 1990, 139–156.

[Kirschenbaum et al. 89] Kirschenbaum, M., Lakhotia A., and Sterling, L.: “Skeletons and Techniques for Prolog Programming”, TR 89–170, Case Western Reserve University, 1989.

[Kleene 71] Kleene, S.C.: “Introduction to Metamathematics” North-Holland, 1971.

[Komorowski 82] Komorowski, H.J.: “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.

[Kott 78] Kott, L: “About Transformation System: A Theoretical Study”, 3ème Colloque International sur la Programmation, Dunod, Paris, 1978, pp. 232–247.

[Kott 82] Kott, L.: “The McCarthy’s Recursion Induction Principle: “Oldy” but “Goody””, Calcolo, 19, 1, 1982, pp. 59–59.

[Kott 85] Kott, L: “Unfold/Fold Program Transformation” Algebraic Methods in Semantics, Cambridge University Press, 1985, pp. 411–434.

[Kowalski 79] Kowalski, R.A.: “Algorithm = Logic + Control” CACM 22(7), July 1979, pp. 424–436.

[Kunen 87] Kunen, K.: “Negation in Logic Programming” Journal of Logic Programming, 4(4) 1987, pp.289–308.

[Kursawe 88] Kursawe, P.: “Pure Partial Evaluation and Instantiation”, in: Bjørner, D., Ershov, A.P., and Jones, N.D. (Eds.), Partial Evaluation and Mixed Computation, North Holland, 1988, pp. 283–298.

[Lakhotia-Sterling 88] Lakhotia, A. and Sterling, L.: “Composing Recursive Logic Programs with Clausal Join” In [Ershov et al. 88], pp. 211–226.

[Lloyd 87] Lloyd, J.W.: “Foundations of Logic Programming” Springer-Verlag, 2nd edition, 1987.

[Lloyd-Shepherdson 91] Lloyd, J.W. and Shepherdson, J.C.: “Partial Evaluation in Logic Programming” Journal of Logic Programming 11, 1991, pp. 217–242.

[Maher 87] Maher, M.J.: “Correctness of a Logic Program Transformation System”, IBM Research Report RC 13496, T.J. Watson Research Center, 1987.

[Maher 89] Maher, M.J.: “ A Transformation System for Deductive Database Modules with Perfect Model Semantics”, Proc. 9th Conf. on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India, LNCS n. 405, Springer-Verlag, 1989, 89–98, Full version in IBM Research Report RC 15062 (also appeared in Theoretical Computer Science, 110, 1993, 377–403).

[Maher 90] Maher, M.J.: “Reasoning About Stable Models (and other Unstable Semantics)”, IBM Research Report, T.J. Watson Research Center, 1990.

[Marriot-Søndergaard 93] Marriot, K. and Søndergaard, H.: “Difference-list Transformation for Prolog” New Generation Computing, 11, 1993, pp. 125–177.

[Martens et al. 92] Martens, B., De Schreye, D., and Bruynooghe, M.: “Sound and Complete Partial Deduction with Unfolding Based on Well-Founded Measures”

In: Proceedings of the International Conference on Fifth Generation Computer Systems, Ohmsha Ltd. IOS Press, 1992, pp.473–480.

[Michie 68] Michie, D.: “Memo functions and Machine Learning” Nature 218 (5136), 1968, pp.19–22.

[Mogensen-Bondorf 93] Mogensen, T. and Bondorf, A.: “Logimix: A Self-Applicable Partial Evaluator for Prolog” In: K.K. Lau and T. Clement (Eds.) Logic Program Synthesis and Transformation, Proc. LOPSTR ’92, Workshops in Computing, Springer-Verlag, 1993, pp. 214–227.

[Naish 86] Naish, L.: “Negation and Control in Prolog” Lecture Notes in Computer Science n. 238, Springer-Verlag, 1985.

[Nakagawa 85] Nakagawa, H.: "Prolog Program Transformations and Tree Manipulation Algorithms", J. Logic Programming, 2, 1985, pp. 77-91.

[Narain 86] Narain, S.: “ A Technique for Doing Lazy Evaluation in Logic” J. Logic Programming, 3 (3), 1986, pp. 259–276.

[Neumerkel 93] Neumerkel, U.W.: “Specialization of Prolog Programs with Partially Static Goals and Binarization” Ph.D. Thesis, Technical University Wien, Austria, 1993.

[Owen 89] Owen, S.: “Issues in the Partial Evaluation of Meta-Interpreters” In: Abramson, H. and Rogers, M.H. (Eds.): Meta-Programming in Logic Programming, MIT press, 1989, pp. 319–339.

[Paige-Koenig 82] Paige, R. and Koenig, S.: “Finite Differencing of Computable Expressions” ACM Toplas 4 (3) July 1982, pp. 402–454.

[Partsch 90] Partsch, H. A.: “Specification and Transformation of Programs” Springer-Verlag, New York, 1990.

[Paterson-Hewitt 70] Paterson, M. S. and Hewitt, C. E.: “Comparative Schematology” Conference on Concurrent Systems and Parallel Computation Project MAC, Woods Hole, Mass. 1970, pp. 119–127.

[Pettorossi 77] Pettorossi, A.: “Transformation of Programs and Use of Tupling Strategy”, Proc. Informatica 77, Bled, Yugoslavia, 1977, pp. 1–6.

[Pettorossi 84] Pettorossi, A.: “Methodologies for Transformations and Memoing in Applicative Languages” Ph. D. Thesis, Edinburgh University, Edinburgh, Scotland, 1984.

[Pettorossi-Proietti 93] Pettorossi, A. and Proietti, M.: “Rules and Strategies for Program Transformation” In “State-of-the-Art Seminar on Program Specification and Transformation” Rio de Janeiro (Brazil), Springer-Verlag, 1993, pp. 263–304.

[Prestwich 93a] Prestwich, S.: “An Unfold Rule for Pure Prolog” In: K.K. Lau and

T. Clement (Eds.) Logic Program Synthesis and Transformation, Proc. LOPSTR '92, Workshops in Computing, Springer-Verlag, 1993, pp. 199–213.

[Prestwich 93b] Prestwich, S.: "Online Partial Deduction of Large Programs" In: Proceedings ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93, Copenhagen, Denmark, ACM Press, 1993, pp. 111–118.

[Proietti-Pettorossi 90] Proietti, M. and Pettorossi, A.: "Construction of Efficient Logic Programs by Loop Absorption and Generalization" Proceedings Meta '90 (M. Bruynooghe, Ed.), Leuven, Belgium, 1990, pp. 57-81.

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

[Proietti-Pettorossi 91b] Proietti, M. and Pettorossi, A.: "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, LNCS n. 528, Springer-Verlag, 1991, pp. 347–358.

[Proietti-Pettorossi 93a] Proietti, M. and Pettorossi, A.: "The Loop Absorption and the Generalization Strategies for the Development of Logic Programs and Partial Deduction" Journal of Logic Programming, 1993:16:123–161.

[Proietti-Pettorossi 93b] Proietti, M. and Pettorossi, A.: "Synthesis of Programs from Unfold/Fold Proofs" to be published in: Proceedings LOPSTR '93, Louvain-la-Neuve, Belgium, 1993.

[Przymusinsky 87] Przymusinsky, T.: "On the Declarative Semantics of Stratified Deductive Databases and Logic Programs" In Minker, J. (Ed.): "Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, CA, 1987, pp. 193–216.

[Safra-Shapiro 86] Safra, S. and Shapiro, E.: "Meta Interpreters for Real" In: Kugler,H.J. (Ed.) Proceedings Information Processing 86, North Holland, 1986, pp. 271–278.

[Sahlin 91] Sahlin, D.: "An Automatic Partial Evaluator for Full Prolog", PhD. Thesis, SICS, Sweden, March 1991.

[Sato 92] Sato, T.: "An Equivalence Preserving First Order Unfold/Fold Transformation System" Theoretical Computer Science, 105, 1992, pp. 57–84.

[Sato-Tamaki 88] Sato, T. and Tamaki, H.: "Deterministic Transformation and Deterministic Synthesis" In: "Future Generation Computers" North-Holland, 1988.

[Sawamura-Takeshima 85] Sawamura, H. and Takeshima, T.: "Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and their Application to Prolog

Optimization” In Proc. Symposium on Logic Programming, Boston, IEEE Computer Society Press, 1985, pp. 200–207.

[Scherlis 81] Scherlis, W. L.: “Program Improvement by Internal Specialization” Proc. 8th ACM Symposium on Principles of Programming Languages, Williamsburg, Va. (1981), pp. 41–49.

[Schwarz 82] Schwarz, J.: “Using Annotations to Make Recursive Equations Behave” IEEE Transactions on Software Engineering SE-8 (1), 1982.

[Seki 91] Seki, H.: “Unfold/Fold Transformation of Stratified Programs”, Theoretical Computer Science 86, 1991, pp. 107–139.

[Seki 93] Seki, H.: “Unfold/Fold Transformation of General Logic Programs for the Well-founded Semantics”, Journal of Logic Programming, Special Issue on Partial Deduction, Vol.16, 1&2, 1993, pp. 5–23.

[Seki-Furukawa 87] [Seki, H. and Furukawa, K.: “Notes On Transformation Techniques for Generate and Test Logic Programs”, Proceedings Symposium on Logic Programming, San Francisco, USA, 1987, pp. 215–223.

[Shepherdson 92] Shepherdson, J.C.: “Unfold/Fold Transformations of Logic Programs”, Math. Struct. in Comp. Science, vol. 2, 1992, pp. 143–157.

[Sterling-Lakhotia 88] Sterling, L. and Lakhotia A.: “Composing Prolog Meta-Interpreters”, in: Proc. Fifth Int. Conf. on Logic Programming, Seattle, WA (USA), MIT Press, 1988, pp. 386–403.

[Sterling-Shapiro 86] Sterling, L. and Shapiro, E.: “The Art of Prolog” The MIT Press, 1986.

[Takeuchi 86] Takeuchi, A.: “Affinity between Meta-Interpreters and Partial Evaluation” In: Kugler,H.J. (Ed.) Proceedings Information Processing 86, North Holland, 1986, pp. 279–282.

[Takeuchi-Furukawa 86] Takeuchi, A. and Furukawa, K.: “Partial Evaluation of Prolog Programs and Its Application to Meta-Programming” In: Kugler,H.J. (Ed.) Proceedings Information Processing 86, North Holland, 1986, pp. 279–282.

[Tamaki-Sato 84] Tamaki, H. and Sato, T.: “Unfold/Fold Transformation of Logic Programs” In: S.-Å. Tärnlund (Ed.), Proc. 2nd International Conference on Logic Programming, Uppsala, Sweden, 1984, pp. 127–138.

[Tamaki-Sato 86] Tamaki, H. and Sato, T.: “ A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation” Technical Report No. 86-4, Ibaraki University, Japan, 1986.

[Tarau-Boyer 90] Tarau, P. and Boyer, M.: “Elementary Logic Programs” In: Deransart, P. and Maluszynski, J. (Eds.) Proceedings PLILP ’90, Linköping, Sweden, August 1990, Lecture Notes in Computer Science n. 456, Springer-Verlag, 1990, pp. 159-173.

[Träff-Prestwich 92] Träff, J.L. and Prestwich, S.D.: “Meta-Programming for Reordering Literals in Deductive Databases” In: A. Pettorossi (Ed.), Proc. 3rd International Workshop on Meta-Programming in Logic (Meta ’92), Uppsala, Sweden, LNCS n. 649, Springer-Verlag, 1992, pp. 280–293.

[Turchin 86] Turchin, V.F.: “The Concept of a Supercompiler” ACM TOPLAS 8 (3) 1986, pp. 292–325.

[Ueda-Furukawa 88] Ueda, K. and Furukawa K.: “Transformation Rules for GHC Programs” Proceedings Intern. Conf. Fifth Generation Computer Systems, ICOT, 1988, pp. 582–591.

[Van Gelder et al. 88] Van Gelder, A., Ross, K., and Schlipf, J.: “Unfounded Sets and Well-founded Semantics for General Logic Programs” Proceedings of the Symposium on Principles of Database Systems, ACM Sigact-Sigmod, 1989, pp. 221–230.

[Venken 84] Venken, R.: “ A Prolog Meta-Interpretation for Partial Evaluation and Its Application to Source-to-Source Transformation and Query Optimization” In: O’Shea, T. (Ed.) Proceedings ECAI ’84, Pisa, Italy, North-Holland, 1984, pp. 91–100.

[Wadler 88] Wadler, P. L.: “Deforestation: Transforming Programs to Eliminate Trees”, Proc. ESOP 88, Nancy, France, Lecture Notes in Computer Science n. 300, 1988, pp. 344–358.

[Walker-Strong 72] Walker, S.A. and Strong, H.R.: "Characterization of Flowchartable Recursions" Proc. 4th Annual ACM Symp. on Theory of Computing, Denver, Co. (1972).

[Wand 80] Wand, M.: “Continuation-based Program Transformation Strategies” Journal ACM 27 (1), 1980, pp.164–180.

[Warren 83] Warren, D.H.D.: “An Abstract Prolog Instruction Set” Technical Report 309, SRI International, 1983.

[Warren 92] Warren, D.S.: “Memoing for Logic Program” CACM, Vol.35, No.3, 1992.

[Wiggins 92] Wiggins, G.A.: “Negation and Control in Automatically Generated Logic Programs” In: A. Pettorossi (Ed.), Proc. 3rd International Workshop on Meta-Programming in Logic (Meta ’92), Uppsala, Sweden, LNCS n. 649, Springer-Verlag, 1992, pp. 250–264.

[Wirth 76] Wirth, N.: “Algorithms + Data Structures = Programs” Prentice-Hall, Inc. 1976.

[Zhang-Grant 88] Zhang, J. and Grant, P.W.: “An Automatic Difference-list Transformation Algorithm for Prolog” Proceedings 1988 European Conference on Artificial Intelligence (ECAI ’88), Pitman, 1988, pp. 320–325.