Transformation of Logic Programs

Alberto Pettorossi and Maurizio Proietti

Contents

This paper appears as a chapter of: D. M. Gabbay, C.J. Hogger, and J. A. Robinson (eds.) *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5, Oxford University Press, 1998, pp. 697–787.

1 Introduction

Program transformation is a methodology for deriving correct and efficient programs from specifications.

In this chapter, we will look at the so called 'rules + strategies' approach, and we will report on the main techniques which have been introduced in the literature for that approach, in the case of logic programs. We will also present some examples of program transformation, and we hope that through those examples the reader may acquire some familiarity with the techniques we will describe.

The program transformation approach to the development of programs has been first advocated in the case of functional languages by Burstall and Darlington [1977]. In that seminal paper the authors give a comprehensive account of some basic transformation techniques which they had already presented in [Darlington, 1972; Burstall and Darlington, 1975].

Similar techniques were also developed in the case of logic languages by Clark and Sickel [1977], and Hogger [1981], who investigated the use of predicate logic as a language for both program specification and program derivation.

In the transformation approach the task of writing a correct and efficient program is realized in two phases. The first phase consists in writing an initial, maybe inefficient, program whose correctness can easily be shown, and the second phase, possibly divided into various subphases, consists in transforming the initial program with the objective of deriving a new program which is more efficient.

The separation of the correctness concern from the efficiency concern, is one of the major advantages of the transformation methodology. Indeed, using this methodology one may avoid some difficulties often encountered in other approaches. One such difficulty, which may occur when following the *stepwise refinement* approach, is the design of the invariant assertions, which may be quite intricate, especially when developing very efficient programs.

The experience gained during the past two decades or so, shows that the methodology of program transformation is very valuable and attractive, in particular for the task of programming 'in the small', that is, for writing single modules of large software systems.

Program transformation has also the advantage of being adaptable to various programming paradigms, and although in this chapter we will focus our attention to the case of logic languages, in the relevant literature one can find similar results for the case of functional languages, and also for the case of imperative and concurrent languages.

The basic idea of the program transformation approach can be pictorially represented as in Fig. 1. From the initial program P_0, which can be viewed as the initial specification, we want to obtain a final program P_n with the same semantic value, that is, we want that $\mathbf{SEM}[P_0] = \mathbf{SEM}[P_n]$ for some given semantic function $\mathbf{SEM}$. The program P_n is often derived in various steps, that is, by constructing a sequence $P_0, \ldots, P_n$ of programs, called *transformation sequence*, such that for $0 \leq i < n$, $\mathbf{SEM}[P_i] = \mathbf{SEM}[P_{i+1}]$, where P_{i+1} is obtained from P_i by applying a *transformation rule*.

In principle, one might obtain a program P_n such that $\mathbf{SEM}[P_0] = \mathbf{SEM}[P_n]$ by deriving intermediate programs whose semantic value is com-

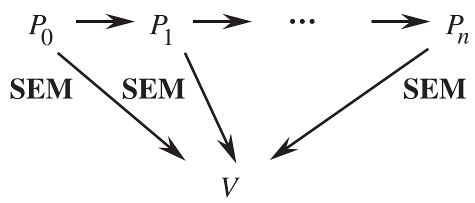

Fig. 1. The program transformation idea: from program P_0 we derive program P_n preserving the semantic value V.

pletely unrelated to $\mathbf{SEM}[P_0]$. This approach, however, has not been followed in practice, because it enlarges in an unconstrained way the search space to be explored when looking for those intermediate programs.

Sometimes, if the programs are nondeterministic, as it is the case of most logic programs which produce a *set* of answers for any input query, we may allow transformation steps which are *partially correct*, but not *totally correct*, in the sense that for $0 \leq i < n$ and for every input query Q, $\mathbf{SEM}[P_i, Q] \subseteq \mathbf{SEM}[P_{i+1}, Q]$. (Here, and in what follows, the semantic function **SEM** is assumed to depend both on the program and the input query.)

As already mentioned, during the program transformation process one is interested in reducing the complexity of the derived program w.r.t. the initial program. This means that for the sequence $P_0, \ldots, P_n$ of programs there exists a cost function C which measures the computational complexity of the programs, such that $C(P_0) \geq C(P_n)$.

Notice that we may allow ourselves to derive a program, say P_i, for some $i > 0$, such that $C(P_0) < C(P_i)$, because subsequent transformations may lead us to a program whose cost is smaller than the one of P_0. Unfortunately, there is no general theory of program transformations which deals with this situation in a satisfactory way in all possible circumstances.

The efficiency improvement from program P_0 to program P_n is not ensured by an undisciplined use of the transformation rules. This is the reason why we need to introduce the so-called *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 which is assumed for the execution of programs. In what follows we will briefly explain how the strategies which have been proposed in the literature may improve program efficiency, and we will refer to the original papers for more details on these issues.

So far we have indicated two major objectives of the program transformation approach, namely the preservation of the semantic value of the initial program and the reduction of the computational complexity of the derived program w.r.t. the initial one.

There is a third important objective which is often given less attention: the formalization of the program transformation process itself. The need for this formalization derives from the desire of making the program transformation approach a powerful programming methodology. In particular, via this formalization it is possible for the programmer to perform 'similar' transformations when starting from 'similar' initial programs, thus avoiding the difficulty of deciding which transformation rule should be applied at every step. It is also possible to make alternative program transformations by considering any program of a previously constructed sequence of programs and deriving from it a different sequence by applying different transformation rules.

The formalization of the program transformation process allows us to define various transformation strategies and, through them, to give suggestions to the programmer on how to transform the program at hand on the basis of the sequence of programs constructed so far.

However, it is not always simple to derive from a given program transformation sequence the strategies which can successfully be applied to similar initial programs. Research efforts are currently being made in this direction.

We often refer to the above three major objectives of the program transformation approach, that is, the preservation of semantics, the improvement of complexity, and the formalization of the transformation process itself, as the *transformation triangle*, with the following three vertices: (i) Semantics, (ii) Complexity, and (iii) Methodology.

Finally, the program transformation methodology should be supported by an automatic (or semiautomatic) system, which both guides the programmer when applying the transformation rules, and also acquires in the form of transformation strategies, some knowledge about successful transformation sequences, while it is in operation.

Together with the 'rules + strategies' approach to program transformation, in the literature one also finds the so-called 'schemata' approach. We will not establish here the theoretical difference between these two approaches, and indeed that difference is mainly based on pragmatic issues. We will briefly illustrate the schemata approach in Section 5.3.

In Section 2 we will present a preliminary example of logic program transformation. It will allow the reader to have a better understanding of the various transformation techniques which we will introduce later. In Section 3 we will describe the transformation rules for logic programs, and in Section 4 we will study their correctness w.r.t. the various semantics which may be preserved. Section 5 is devoted to the introduction of the transformation strategies which are used for guiding the application of the rules. Then, in Section 6 we will consider the partial evaluation technique, and finally, in Section 7 we will briefly indicate some methodologies for program derivation which are related to program transformation.

2 A Preliminary Example

The 'rules + strategies' approach to program transformation as it was first introduced in [Burstall and Darlington, 1977] for recursive equation programs, is based on the use of two elementary transformation rules: the *unfolding rule* and the *folding rule*.

The unfolding rule consists in replacing an instance of the left hand side of a recursive equation by the corresponding instance of the right hand side. This rule corresponds to the 'replacement rule' used in [Kleene, 1971] for the computation of recursively defined functions. The application of the unfolding rule can also be viewed as a symbolic computation step.

The folding rule consists in replacing an instance of the right hand side of a recursive equation by the corresponding instance of the left hand side. 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 expression. Vice versa, unfolding can be viewed as the inverse of folding.

The reader who is not familiar with the transformation methodology, may wonder about the usefulness of performing a folding step, that is, of inverting a symbolic computation step, when one desires to improve program efficiency. However, as we will see in some examples below, the folding rule allows us to modify the recursive structure of the programs to be transformed and, by doing so, we will often be able to achieve substantial efficiency improvements.

Program derivation techniques following the 'rules + strategies' approach, have been presented in the context of logic programming in [Clark and Sickel, 1977; Hogger, 1981], where the basic derivation rules consist of the substitution of a formula by an equivalent formula.

Tamaki and Sato [1984] have adapted the unfolding and folding rules to the case of logic programs. Following the basic ideas relative to functional programs, they take an application of the unfolding rule to be equivalent to a computation step, that is, an application of SLD-resolution, and the folding rule to be the inverse of unfolding.

As already mentioned, during the transformation process we want to keep unchanged, at least in a weak sense, the semantic value of the programs which are derived, and in particular, we want the final program to be partially correct w.r.t. 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, as defined in [van Emden and Kowalski, 1976], is contained in the least Herbrand model of P_0 [Tamaki and Sato, 1984]. Thus, the unfold/fold transformations are partially correct w.r.t. the least Herbrand model semantics.

In general, unfold/fold transformations are not totally correct w.r.t. the least Herbrand model semantics, that is, the least Herbrand model of P_0 may be not contained in the one of P_1. In order to get total correctness one has to comply with some extra conditions [Tamaki and Sato, 1984].

The study of the various semantics which are preserved when using the unfold/fold transformation rules will be the objective of Section 4.

Let us now consider a preliminary example of program transformation where we will see in action some of the rules and strategies for transforming logic programs. In this example, together with the unfolding and folding rules, we will also see the use of two other transformation rules, called *definition rule* and *goal replacement rule*, and the use of a transformation strategy, called *tupling strategy*.

As already mentioned, the need for strategies which drive the application of the transformation rules and improve efficiency, comes from the fact that folding is the inverse of unfolding, and thus, we may construct a useless transformation sequence where the final program is equal to the initial program.

Let us consider the following logic program P_0 for testing whether or not a given list is palindrome:

1. $pal([\,]) \leftarrow$
2. $pal([H]) \leftarrow$
3. $pal([H|T]) \leftarrow append(Y,[H],T), pal(Y)$
4. $append([\,],Y,Y) \leftarrow$
5. $append([H|X],Y,[H|Z]) \leftarrow append(X,Y,Z)$

We have that, given the lists X, Y, and Z, $append(X,Y,Z)$ holds in the least Herbrand model of P_0 iff Z is the concatenation of X and Y.

Both $pal(Y)$ and $append(Y,[H],T)$ visit the same list Y and we may avoid this double visit by applying the tupling strategy which suggests the introduction of the following clause for the new predicate *newp*:

6. $newp(L,T) \leftarrow append(Y,L,T), pal(Y)$

Actually, clause 6 has been obtained by a simultaneous application of the tupling strategy and the so-called *generalization strategy*, in the sense that in the body of clause 3 the argument $[H]$ has been generalized to the variable L. In Section 5, we will consider the tupling and the generalization strategies and we will indicate in what cases they may be useful for improving program efficiency.

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 $q(\ldots)$, where q is a predicate occurring in P_0, belongs to the least Herbrand model of P_0 iff $q(\ldots)$ belongs to the least Herbrand model of P_1.

In order to avoid the double occurrence of the list Y in the body of clause 3, we now fold that clause using clause 6, that is, we replace '$append(Y,[H],T)$, $pal(Y)$' which is an instance of the body of clause 6, by the corresponding instance '$newp([H],T)$' of the head of clause 6. Thus, we get:

3f. $pal([H|T]) \leftarrow newp([H],T)$

This folding step is the inverse of the step of unfolding clause 3f w.r.t. $newp([H],T)$.

Unfortunately, if we use the program made out of clauses 1, 2, 3f, 4, 5, and 6, we do not avoid the double visit of the input list, because *newp* is defined in terms of the two predicates *append* and *pal*. As we will show, a gain in efficiency is possible if we derive a definition of *newp* in terms of *newp* itself. This recursive definition of *newp* can be obtained as follows.

We first unfold clause 6 w.r.t. $pal(Y)$, that is, we derive the following three resolvents of clause 6 using clauses 1, 2, and 3, respectively:

7. $newp(L,T) \leftarrow append([\,],L,T)$
8. $newp(L,T) \leftarrow append([H],L,T)$
9. $newp(L,T) \leftarrow append([H|Y],L,T), append(R,[H],Y), pal(R)$

We then unfold clauses 7, 8, and 9 w.r.t. the atoms $append([\,],L,T)$, $append([H],L,T)$, and $append([H|Y],L,T)$, respectively, and we get

10. $newp(L,L) \leftarrow$
11. $newp(L,[H|L]) \leftarrow$
12. $newp(L,[H|U]) \leftarrow append(Y,L,U), append(R,[H],Y), pal(R)$

Now, in order to get a recursive definition of the predicate *newp* where no multiple visits of lists are performed, we would like to fold the entire body of clause 12 using clause 6, and in order to do so we need to have only one occurrence of the atom *append*. We can do so by applying the goal replacement rule (actually, the version formalized by rule R5.1 on page 18) which allows us to replace the goal '$append(Y,L,U)$, $append(R,[H],Y)$' by the equivalent goal '$append(R,[H|L],U)$'. Thus, we get the following clause:

13. $newp(L,[H|U]) \leftarrow append(R,[H|L],U), pal(R)$

We can fold clause 13 using clause 6, and we get:

13f. $newp(L,[H|U]) \leftarrow newp([H|L],U)$

Having derived a recursive definition of *newp* the transformation process is completed. The final program we have obtained, is as follows:

1. $pal([\,]) \leftarrow$
2. $pal([H]) \leftarrow$

3f. $pal([H|T]) \leftarrow newp([H],T)$

4. $append([\,],Y,Y) \leftarrow$
5. $append([H|X],Y,[H|Z]) \leftarrow append(X,Y,Z)$

10. $newp(L,L) \leftarrow$
11. $newp(L,[H|L]) \leftarrow$

13f. $newp(L,[H|U]) \leftarrow newp([H|L],U)$

In this final program no double visits of lists are performed, and the time complexity is improved from $O(n^2)$ to $O(n)$, where n is the size of the input list. The initial and final programs have the same least Herbrand model semantics w.r.t. the predicates *pal* and *append*.

Notice that if we are interested in the computation of the predicate *pal* only, in the final program we can discard clauses 4 and 5, which are unnecessary.

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, many cases in which the synthesis of eureka predicates can be performed in an automatic way, and this is the reason why in practice the use of the program transformation methodology is very powerful.

In Section 5, we will consider the problem of inventing the eureka predicates and we will see that it can often be solved on the basis of syntactical properties of the program to be transformed by applying suitable transformation strategies.

3 Transformation Rules for Logic Programs

In this section, we will present the most frequently used transformation rules for logic programs considered in the literature.

As already mentioned, the rules for transforming logic programs are directly derived from those used in the case of functional programs, but in adapting those rules, care has been taken, because logic programs compute *relations* rather than functions, and the *nondeterminism* inherent in relations does affect the various transformation techniques.

Moreover, for logic programs a rich variety of semantics can be defined and the choice of a particular semantics to be preserved affects the transformation rules to be used.

These facts motivate the large amount of research work which has been devoted to the extension of the transformation methodology to logic programs. Surveys of the work on logic program transformation have appeared in [Shepherdson, 1992; Pettorossi and Proietti, 1994].

3.1 Syntax of Logic Programs

Let us first briefly recall the syntax of logic programs and let us introduce the terminology which we will use. For other notions concerning logic programming not explicitly stated here we will refer to [Lloyd, 1987]. We assume that all our logic 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 which is sometimes made in the theory of logic programs (see, for instance, [Kunen, 1989]), in our case is motivated by the convenience of having a constant Herbrand universe while transforming programs.

An *atom* is a formula of the form: $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol taken from **L**, and $t_1, \ldots, t_n$ 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 *goal* is a finite, possibly empty, *sequence* of literals not necessarily distinct. For clarity, a goal may also be written between round parentheses. In particular, if $L_1 \neq L_2$ the goals (L_1, L_2) and (L_2, L_1) are different, even though their semantics may be the same. Commas will be used to denote the associative concatenation of goals. Thus, $(L_1, \ldots, L_m), (L_{m+1}, \ldots, L_n)$ is equal to $(L_1, \ldots, L_m, L_{m+1}, \ldots, L_n)$.

A *clause* is a formula of the form: $H \leftarrow B$, where the *head* H is an atom and the *body* B is a (possibly empty) goal. The head and the body of a clause C are denoted by $hd(C)$ and $bd(C)$, respectively.

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

A *query* is a formula of the form: $\leftarrow G$, where G is a (possibly empty) goal. Notice that our notion of query corresponds to that of goal considered in [Lloyd, 1987].

Goals, clauses, logic programs, and queries are called *definite goals*, *definite clauses*, *definite logic programs*, and *definite queries*, respectively, if no negative literals occur in them. When we want to stress the fact that occurrences of negative literals are allowed, we will follow [Lloyd, 1987] and use the qualification *normal*. Thus, 'normal goal', 'normal clause', 'normal program', and 'normal query' are synonyms of 'goal', 'clause', 'program', and 'query', respectively. We will feel free to omit both qualifications 'definite' and 'normal' when they are irrelevant or understood from the context.

Given a term t, the set of variables occurring in t is denoted by $vars(t)$. A similar notation will also be adopted for the variables occurring in literals, goals, clauses, and queries.

A *substitution* is a finite mapping from variables to terms of the form: $\{X_1/t_1, \ldots, X_n/t_n\}$. The application of a substitution θ to a term t will be denoted by $t\theta$. Similar notation will be used for substitutions applied to literals, goals, clauses, and queries.

For other notions related to substitutions, such as *instance* and *most general unifier*, we refer to [Apt, 1990].

A *variable renaming* is a bijection from the set of variables of the language **L** onto itself. We assume that the variables occurring in a clause 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 resolution steps. Two clauses which differ only for a variable renaming are called *variants*.

For simplicity reasons, we will identify any two computed answer substitutions obtained by resolution steps which differ only by the renaming of the clauses involved.

By a *predicate renaming* we mean a bijective mapping of the set of predicate symbols of **L** onto itself. Given a predicate renaming ρ and a program P, by $\rho(P)$ we denote the program obtained by replacing each predicate symbol p in P by $\rho(p)$. A similar notation will be adopted for the predicate renaming in queries.

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

We will say that a predicate p *depends on* a predicate q in the program P iff either there exists in P a clause of the form: $p(\ldots) \leftarrow B$ such that q occurs in the goal B or there exists in P a predicate r such that p depends on r in P and r depends on q in P.

We say that a query (or a goal) Q depends on a clause of the form: $p(\ldots) \leftarrow B$ in a program P iff either p occurs in Q or there exists a predicate occurring in Q which depends on p. The *relevant part* $P_{rel}(Q)$ of program P for the query (or the goal) Q is the subsequence of clauses in P on which Q depends.

3.2 Semantics of Logic Programs

When deriving new programs from old ones, we need to take into account the semantics which is preserved. For the formal definition of the semantics of a logic program we explicitly consider the dependency on the input query, and thus, we define a *semantics* of a set $\mathbf{P}$ of logic programs w.r.t. a set $\mathbf{Q}$ of queries, to be a function $\mathbf{SEM}$: $\mathbf{P} \times \mathbf{Q} \rightarrow (\mathbf{D}, \leq)$, where $(\mathbf{D}, \leq)$ is a partially ordered set.

We will assume that every semantics $\mathbf{SEM}$ we consider, is *preserved by predicate renaming*, that is, for every predicate renaming ρ, program P, and query Q, $\mathbf{SEM}[P, Q] = \mathbf{SEM}[\rho(P), \rho(Q)]$.

We will also assume that every semantics $\mathbf{SEM}$ is preserved by interchanging the order of two adjacent clauses with different head predicates. Thus, we may always assume that all clauses constituting the definition of a predicate are adjacent.

We say that two programs P_1 and P_2 in $\mathbf{P}$ are *equivalent* w.r.t. the semantics function $\mathbf{SEM}$ and the set $\mathbf{Q}$ of queries iff for every query Q in $\mathbf{Q}$ we have that $\mathbf{SEM}[P_1, Q] = \mathbf{SEM}[P_2, Q]$.

An example of semantics function can be provided by taking $\mathbf{P}$ to be the set of definite programs (denoted by $\mathbf{P}^+$), $\mathbf{Q}$ the set of definite queries (denoted by $\mathbf{Q}^+$), and $\mathbf{D}$ the powerset of the set of substitutions (denoted by $\mathcal{P}(\mathbf{Subst})$), ordered by set inclusion.

We now define the *least Herbrand model semantics* as the function $\mathbf{LHM}\colon \mathbf{P}^+ \times \mathbf{Q}^+ \to (\mathcal{P}(\mathbf{Subst}), \leq)$ such that, for every $P \in \mathbf{P}^+$ and for every $\leftarrow G \in \mathbf{Q}^+$,

$$\mathbf{LHM}[P, \leftarrow G] = \{\theta \mid \text{every ground instance of } G\theta \text{ is a logical consequence of } P\}$$

where we identify the program P and the goal G with the logical formulas obtained by interpreting sequences of clauses (or atoms) as conjunctions and interpreting $\leftarrow$ as the implication connective. As usual, the empty sequence of atoms in the body of a clause is interpreted as *true*.

The least Herbrand model $\mathbf{M}(P)$ of a definite program P defined according to [van Emden and Kowalski, 1976] can be expressed in terms of the function $\mathbf{LHM}$ as follows:

$$\mathbf{M}(P) = \{A \mid A \text{ is a ground atom and } \mathbf{LHM}[P, \leftarrow A] = \mathbf{Subst}\}.$$

Thus, two programs P_1 and P_2 in $\mathbf{P}^+$ are equivalent w.r.t. $\mathbf{LHM}$ iff $\mathbf{M}(P_1) = \mathbf{M}(P_2)$.

In order to state various correctness results concerning our transformation rules w.r.t. different semantics functions (see Section 4), we need the following notion of *relevance* [Dix, 1995].

Definition 3.2.1 (Relevance). A semantics function $\mathbf{SEM}\colon \mathbf{P} \times \mathbf{Q} \to (\mathbf{D}, \leq)$ is *relevant* iff for every program P in $\mathbf{P}$ and query Q in $\mathbf{Q}$, we have that $\mathbf{SEM}[P, Q] = \mathbf{SEM}[P_{rel}(Q), Q]$.

Thus, a semantics is relevant iff its value for a given program and a given query is determined only by the query and the clauses on which the query depends.

The least Herbrand model semantics and many other semantics we will consider, are relevant, but well-known semantics, such as the *Clark's completion* [Lloyd, 1987] and the *stable model semantics* [Gelfond and Lifschitz, 1988], are not.

3.3 Unfold/Fold Rules

As already mentioned, the program transformation process starting from a given initial program P_0, can be viewed as a sequence of programs $P_0, \ldots, P_k$, called *transformation sequence* [Tamaki and Sato, 1984], such that program P_{j+1}, with $0 \leq j \leq k-1$, is obtained from program P_j by the application of a *transformation rule*, which may depend on $P_0, \ldots, P_j$. An application of a transformation rule is also called a *transformation step*.

Since most transformation rules can be viewed as the replacement of a given clause C by some new clauses $C_1, \ldots, C_n$, the transformation process can also be represented by means of trees of clauses [Pettorossi and Proietti, 1989], where the clauses $C_1, \ldots, C_n$ are viewed as the son-nodes of C. This tree-based representation will be useful for describing the transformation strategies (see Section 5).

The transformation rules we will present in this chapter are collectively called *unfold/fold rules* and they are a generalization of those introduced by [Tamaki and Sato, 1984]. Several special cases of these rules will be introduced in the subsequent sections, when discussing the correctness of the transformation rules w.r.t. different semantics of logic programs.

In the presentation of the rules we will refer to the transformation sequence $P_0, \ldots, P_k$ and we will 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.

Rule R1. Unfolding. Let P_k be the program $E_1, \ldots, E_r, C, E_{r+1}, \ldots, E_s$ and C be the clause $H \leftarrow F, A, G$, where A is a *positive* literal and F and G are (possibly empty) goals. Suppose that

1. $D_1, \ldots, D_n$, with $n > 0$, is the subsequence of all clauses of a program P_j, for some j, with $0 \leq j \leq k$, such that A is unifiable with $hd(D_1), \ldots, hd(D_n)$, with most general unifiers $\theta_1, \ldots, \theta_n$, respectively, and
2. C_i is the clause $(H \leftarrow F, bd(D_i), G)\theta_i$, for $i = 1, \ldots, n$.

If we *unfold C w.r.t. A using P_j* we derive the clauses $C_1, \ldots, C_n$ and we get the new program $P_{k+1} = E_1, \ldots, E_r, C_1, \ldots, C_n, E_{r+1}, \ldots, E_s$.

The unfolding rule corresponds to the application of a resolution step to clause C with the selection of the positive literal A and the input clauses $D_1, \ldots, D_n$.

Example 3.3.1. Let P_k be the following program:

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

Then, by unfolding $p(X) \leftarrow q(t(X)), r(X), r(b)$ w.r.t. $q(t(X))$ using P_k itself we derive the following program P_{k+1}:

$p(b) \leftarrow r(b), r(b)$
$p(X) \leftarrow r(t(X)), r(X), r(b)$

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

Remark 3.3.2. There are two main differences between the unfolding rule in the case of logic programs and the unfolding rule in the case of functional programs.

The first difference is that, when we unfold a clause C w.r.t. an atom A in $bd(C)$ using a program P_j, it is not required that A be an instance of the head of a clause in P_j. We only require that A be unifiable with the head of at least one clause in P_j. This is related to the fact that a resolution step produces a *unifying* substitution, not a *matching* substitution, as it happens in a rewriting step for functional programs.

The second difference is that in the functional case it is usually assumed that the equations defining a program are mutually exclusive. Thus, by unfolding a given equation we may get at most one new equation. On the contrary, in the logic case there may be several clauses in a program P_j whose heads are unifiable with an atom A in the body of a clause C. As a result, by unfolding C w.r.t. A, we may derive more than one clause.

The unfolding rule is one of the basic transformation rules in all transformation systems for logic programs proposed in the literature. Our presentation of the rule follows [Tamaki and Sato, 1984] where, however, it is required that the program used for unfolding a clause is the one where this clause occurs, that is, with reference to our rule R1, P_j is required to be P_k.

Some derivation rules for logic programs similar to the unfolding rule have been considered in [Komorowski, 1982] and in [Clark and Sickel, 1977; Hogger, 1981], in the context of partial evaluation and program synthesis (see also Sections 6 and 7).

Gardner and Shepherdson [1991] have defined a transformation rule which can be considered as the unfolding of a clause w.r.t. a *negative* literal. Given the clause $C = H \leftarrow F, \neg A, G$ in a program P, where A is a ground atom, Gardner and Shepherdson's unfolding rule transforms P as follows:

1. either $\neg A$ is deleted from the body of C, if the query $\leftarrow A$ has a finitely failed SLDNF-tree in P,
2. or C is deleted from P, if the query $\leftarrow A$ has an SLDNF-refutation in P.

Gardner and Shepherdson's unfolding rule w.r.t. negative literals can be expressed in terms of the *goal replacement rule* (in case 1) and the *clause deletion rule* (in case 2). These rules will be introduced below.

Also Kanamori and Horiuchi [1987] and Sato [1992] allow unfolding steps w.r.t. negative literals. However, their notion of program goes beyond the definition of logic program which we consider here, and their transformation rules may more properly be regarded as rules for logic program synthesis starting from first order logic specifications (see Section 7).

Rule R2. Folding. Let P_k be the program $E_1, \ldots, E_r, C_1, \ldots, C_n, E_{r+1}, \ldots, E_s$ and $D_1, \ldots, D_n$ be a subsequence of clauses in a program P_j, for some j, with $0 \leq j \leq k$. Suppose that there exist an atom A and two goals F and G such that for each i, with $1 \leq i \leq n$, there exists a substitution θ_i which satisfies the following conditions:

1. C_i is a variant of the clause $H \leftarrow F, bd(D_i)\theta_i, G$,
2. $A = hd(D_i)\theta_i$,
3. for every clause D of P_j not in the sequence $D_1, \ldots, D_n$, $hd(D)$ is not unifiable with A, and
4. for every variable X in the set $vars(D_i) - vars(hd(D_i))$, we have that
   - $X\theta_i$ is a variable which does not occur in (H, F, G) and
   - the variable $X\theta_i$ does not occur in the term $Y\theta_i$, for any variable Y occurring in $bd(D_i)$ and different from X.

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

The folding rule 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} is obtained from P_k by folding, there exists a transformation sequence $P_0, \ldots, P_k, P_{k+1}, P_k$, where P_k can be obtained from P_{k+1} by unfolding.

Notice that the possibility of inverting a folding step by performing an unfolding step, depends on the fact that for unfolding (as for folding) we can use clauses taken from any program of the transformation sequence constructed so far.

Example 3.3.3. Let us consider a transformation sequence P_0, P_1 where P_1 includes the clauses

C_1. $p(X) \leftarrow q(t(X), Y), r(X)$
C_2. $p(Z) \leftarrow s(Z), r(Z)$

and the definition of predicate a in P_0 consists of the clauses

D_1. $a(U) \leftarrow q(U, V)$
D_2. $a(t(W)) \leftarrow s(W)$

Clauses C_1, C_2 can be folded using D_1, D_2. Indeed, the conditions listed in the folding rule R2 are satisfied with $\theta_1 = \{U/t(X), V/Y\}$ and $\theta_2 = \{W/Z\}$. The derived clause is

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

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

The following example shows that condition 4 in rule R2 is necessary for ensuring that, after a folding step, we can perform an unfolding step which leads us back to the program we had before the folding step.

Example 3.3.4. Let C be $p(Z) \leftarrow q(Z)$ and D be $r \leftarrow q(X)$. Suppose that D is the only clause in P_j with head r. Clauses C and D satisfy conditions 1, 2, and 3 of the folding rule with $n = 1$ and $\theta_1 = \{X/Z\}$. However, they do not satisfy condition 4 because X does not occur in the head of D, and $X\theta_1$, which is Z, occurs in the head of C. By replacing the body of C by the head of D we get the clause $p(Z) \leftarrow r$. If we then unfold $p(Z) \leftarrow r$ using P_j we get $p(Z) \leftarrow q(X)$, which is *not* a variant of C.

Notice that, if a program P_k can be transformed into a program P_{k+1} by an unfolding step, it is not always possible to derive again P_k by means of a folding step applied to P_{k+1} (see the following example). Thus, we may say that folding is only a 'right-inverse' of unfolding.

Example 3.3.5. Let C be the clause $p(X) \leftarrow r(X)$ and D be the clause $r(t(X)) \leftarrow q(X)$. From program C, D, by unfolding C using the program C, D itself, we get the clause $C_1 = p(t(X)) \leftarrow q(X)$ and the program C_1, D. In order to get back C, D by folding, we would like to fold C_1 and derive (a variant of) C. There are only two ways of applying the folding rule to C_1. The first one is to use clause C_1 itself, thereby getting the clause $p(t(X)) \leftarrow p(t(X))$. The second one is to use clause D and if we do so we get the clause $p(t(X)) \leftarrow r(t(X))$. In neither case we get a variant of C.

Our presentation of the folding rule is similar to the one in [Gergatsoulis and Katzouraki, 1994] where, however, during the application of the folding rule, the introduction of some equality atoms is also allowed. We will deal with the equality introduction in a separate rule.

Several folding rules which are special cases of rule R2 have been considered in the literature. These folding rules have various restrictions depending on: i) the choice of the program in the transformation sequence from where the clauses used for folding (i.e. $D_1, \ldots, D_n$) are taken, ii) the number of these clauses, and iii) whether or not these clauses are allowed to be recursive (i.e. predicates in the bodies of the clauses depend on predicates in the heads of those clauses).

We will present some of these special cases of the folding rule in subsequent sections. In particular, the rule originally introduced by Tamaki and Sato [1984] will be presented in Section 4.4.1 (see rule R2.2, page 30), when dealing with the preservation of the least Herbrand model semantics while transforming programs.

Rule R3. Definition Introduction (or **Definition**, for short)**.** We may get program P_{k+1} by adding to program P_k n clauses of the form:

$p_i(\ldots) \leftarrow B_i$, for $i = 1, \ldots, n$, such that the predicate symbol p_i does not occur in $P_0, \ldots, P_k$.

The definition rule is said to be *non-recursive* iff every predicate symbols occurring in the bodies B_i's occurs in P_k as well.

Our presentation of the definition rule is similar to the one in [Maher, 1987] and it allows us to introduce one or more new predicate definitions each of which may consist of more than one clause.

Rule R4. Definition Elimination. We may get program P_{k+1} by deleting from program P_k the clauses constituting the definitions of the predicates $q_1, \ldots, q_n$ such that, for $i = 1, \ldots, n$, q_i does not occur in P_0 and every predicate in P_k which depends on q_i is in the set $\{q_1, \ldots, q_n\}$.

The definition elimination rule can be viewed as an inverse of the definition introduction rule. It has been presented in [Maher, 1987], where it has been called *deletion*, and also in [Bossi and Cocco, 1993], where it has been called *restricting operation*.

The next rule we will introduce is the *goal replacement rule*, which allows us to replace a goal in the body of a clause by an *equivalent* goal. Equivalence between goals, as it is usually defined, depends on the semantics of the program P_k where the replacement takes place.

We now introduce a simple notion of goal equivalence which is parametric w.r.t. the semantics considered. A more complex notion will be given later.

Definition 3.3.6 (Goal Equivalence). Two goals G_1 and G_2 are *equivalent* w.r.t. a semantics **SEM** and a program P iff $\mathbf{SEM}[P, \leftarrow G_1] = \mathbf{SEM}[P, \leftarrow G_2]$. (We will feel free to omit the references to **SEM** and/or P when they are understood from the context.)

Rule R5. Goal Replacement. Let **SEM** be a semantics function, G_1 and G_2 two equivalent goals w.r.t. **SEM** and P_k, and $C = H \leftarrow L, G_1, R$ a clause in P_k. By *replacement* of goal G_1 by goal G_2 in C we derive the clause $D = H \leftarrow L, G_2, R$ and we get P_{k+1} from P_k by replacing C by D.

Example 3.3.7. Let **SEM** be the least Herbrand model semantics **LHM** defined in Section 3.2 and P_k be the following program:

$C.\ p(X,Y) \leftarrow q(X,Y)$
$\quad q(a,Y) \leftarrow$
$\quad q(b,Y) \leftarrow q(b,Y)$
$\quad r(X,X) \leftarrow$

We have that $q(X,Y)$ is equivalent to $r(X,a)$ w.r.t. **LHM** and P_k. Thus, by replacement of $q(X,Y)$ in C we derive the clause

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

The above definition of goal equivalence does not take into account the clause where the goal replacement occurs. As a result, many substitutions of goals by new goals which produce from a program P_k a new program P_{k+1}, cannot be viewed as applications of our rule R5, even though P_k and P_{k+1} are equivalent programs.

Example 3.3.8. Let us consider the following clauses in a program P_k:

$C.\ sublist(N,X,Y) \leftarrow length(X,N), append(V,X,I), append(I,Z,Y)$
$A_1.\ append([\,],L,L) \leftarrow$
$A_2.\ append([H|T],L,[H|U]) \leftarrow append(T,L,U)$

If we replace the goal '$append(V,X,I), append(I,Z,Y)$' in the body of C by the goal '$append(X,Z,J), append(V,J,Y)$' we get a program, say P_{k+1}, which is equivalent to P_k w.r.t. the least Herbrand model semantics **LHM**. However, the two goals '$append(V,X,I), append(I,Z,Y)$' and '$append(X,Z,J), append(V,J,Y)$' are not equivalent in the sense of Definition 3.3.6.

Indeed, the substitution $\theta = \{V/[\,], X/[\,], I/[\,], Z/[\,], Y/[\,], J/[a]\}$ belongs to $\mathbf{LHM}[P_k, \leftarrow (append(V,X,I), append(I,Z,Y))]$ and does not belong to $\mathbf{LHM}[P_k, \leftarrow (append(X,Z,J), append(V,J,Y))]$.

In order to overcome the above mentioned limitation of the goal replacement rule R5, we now consider a weaker notion of goal equivalence which depends on a given set of variables. A similar notion was introduced, for definite programs and the computed answer substitution semantics, by Cook and Gallagher [1994]. We will then consider a version of the goal replacement rule based on this weaker notion of goal equivalence.

Definition 3.3.9 (Goal equivalence w.r.t. a set of variables). Let the program P_k be $C_1, \ldots, C_n$ and let **SEM** be a semantics function. Let us consider the following two clauses:

$D_1.\ newp_1(X_1, \ldots, X_m) \leftarrow G_1$
$D_2.\ newp_2(X_1, \ldots, X_m) \leftarrow G_2$

where $newp_1$ and $newp_2$ are distinct predicate symbols not occurring in P_k, $\{X_1, \ldots, X_m\}$ is a set of m variables, and G_1 and G_2 are two goals. Let V denote the set $\{X_1, \ldots, X_m\}$. We say that G_1 and G_2 are equivalent w.r.t. **SEM**, P_k and V, and we write $G_1 \equiv_V G_2$, iff

$$\mathbf{SEM}[(C_1, \ldots, C_n, D_1), \leftarrow newp_1(X_1, \ldots, X_m)] =$$
$$= \mathbf{SEM}[(C_1, \ldots, C_n, D_2), \leftarrow newp_2(X_1, \ldots, X_m)].$$

If G_1 and G_2 are equivalent w.r.t. **SEM**, P_k, and V, we also say that the *replacement law* $G_1 \equiv_V G_2$ is *valid* w.r.t. **SEM** and P_k.

Since we have assumed that **SEM** is preserved by predicate renaming, we have that, for any set V of variables, $\equiv_V$ is an equivalence relation.

Rule R5.1 Clausal Goal Replacement. Let $C = H \leftarrow L, G_1, R$ be a clause in P_k. Suppose that goals G_1 and G_2 are equivalent w.r.t. **SEM**, P_k, and the set of variables $vars(H, L, R) \cap vars(G_1, G_2)$. By *clausal goal replacement* of G_1 by G_2 in C we derive the clause $D = H \leftarrow L, G_2, R$ and we get P_{k+1} by substituting D for C in P_k.

In Example 4.3.1 below we will see that rule R5.1 overcomes the above mentioned limitation of rule R5.

In the next section we will show that the clausal goal replacement rule R5.1 can be viewed as a derived rule, because its application can be mimicked by suitable applications of the transformation rules R1, R2, R3, R4, and R5 defined above. Thus, without loss of generality, we may consider rule R5 as the only goal replacement rule, when also rules R1, R2, R3, and R4 are available.

Various notions of goal equivalence and goal replacement have been introduced in the literature [Tamaki and Sato, 1984; Maher, 1987; Gardner and Shepherdson, 1991; Bossi *et al.*, 1992b; Bossi *et al.*, 1992a; Cook and Gallagher, 1994]. Each of these notions have been defined in terms of a particular semantics, while in our presentation we introduced a notion which has the advantage of being parametric w.r.t. the given semantics **SEM**.

We finally present a class of transformation rules which will collectively be called *clause replacement* rules and referred to as rule R6.

Rule R6. Clause Replacement. From program P_k we get program P_{k+1} by applying one of the following four rules.

Rule R6.1 Clause Rearrangement. We get P_{k+1} by replacing in P_k the sequence C, D of two clauses by D, C.

This clause rearrangement rule is implicitly used by many authors who consider a program as a set or a multiset of clauses.

Rule R6.2 Deletion of Subsumed Clauses. A clause C is *subsumed* by a clause D iff there exist a substitution θ and a (possibly empty) goal G such that $hd(C) = hd(D)\theta$ and $bd(C) = bd(D)\theta, G$. We may get program P_{k+1} by deleting from P_k a clause which is subsumed by another clause in P_k.

In particular, the rule for the deletion of subsumed clauses allows us to remove duplicate clauses.

Rule R6.3 Deletion of Clauses with Finitely Failed Body. Let C be a clause in program P_k of the form: $H \leftarrow A_1, \ldots, A_m, L, B_1, \ldots, B_n$ with $m, n \geq 0$. If literal L has a finitely failed SLDNF-tree in P_k, then we say that C has a *finitely failed body* in P_k and we get program P_{k+1} by deleting C from P_k.

The rules for the deletion of subsumed clauses and the deletion of clauses with finitely failed body are instances of the clause deletion rule introduced by Tamaki and Sato [1984] for definite programs. Other rules for deleting clauses from a given program, or adding clauses to a given program are studied in [Tamaki and Sato, 1984; Gardner and Shepherdson, 1991; Bossi and Cocco, 1993; Maher, 1993]. The correctness of those rules strictly depends on the semantics considered. For further details the reader may look at the original papers.

Rule R6.4 Generalization + Equality Introduction. Let us assume that the equality predicate '=' (written in infix notation) is defined by the clause $X = X \leftarrow$ in every program of the transformation sequence $P_0, \ldots, P_k$. Let us also consider a clause

$C.\;\; H \leftarrow A_1, \ldots, A_m$

in P_k, a substitution $\theta = \{X/t\}$, with X not occurring in t, and a clause

$GenC.\;\; GenH \leftarrow GenA_1, \ldots, GenA_m$

such that $C = GenC\,\theta$.

By *generalization + equality introduction* we derive the clause

$D.\;\; GenH \leftarrow X = t, GenA_1, \ldots, GenA_m$

and we get P_{k+1} by replacing C by D in P_k.

This transformation rule was formalized in [Proietti and Pettorossi, 1990].

4 Correctness of the Transformation Rules

In this section we will first present some correctness properties of the transformation rules which we introduced in the previous section. These properties are parametric w.r.t. the semantics function **SEM**. We will then give an overview of the main results presented in the literature concerning the correctness of the transformation rules w.r.t. various semantics for definite and normal programs.

We first introduce the notion of correctness of a transformation sequence w.r.t. a generic semantics function **SEM**.

Definition 4.0.1 (Correctness of a Transformation Sequence). Let $\mathbf{P}$ be a set of programs, $\mathbf{Q}$ a set of queries, and $\mathbf{SEM}$: $\mathbf{P} \times \mathbf{Q} \rightarrow (\mathbf{D}, \leq)$ be a semantics function. A transformation sequence $P_0, \ldots, P_k$ of programs in $\mathbf{P}$ is *partially correct* (or *totally correct*) w.r.t. **SEM** iff for every query Q in $\mathbf{Q}$, containing only predicate symbols which occur in P_0, we have that $\mathbf{SEM}[P_k, Q] \leq \mathbf{SEM}[P_0, Q]$ (or $\mathbf{SEM}[P_k, Q] = \mathbf{SEM}[P_0, Q]$).

A transformation rule is *partially correct* (or *totally correct*) w.r.t. **SEM** iff for every transformation sequence $P_0, \ldots, P_k$ which is partially correct (or 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 extended transformation sequence $P_0, \ldots, P_k, P_{k+1}$ is partially correct (or totally correct) w.r.t. **SEM**.

A transformation step which allows us to derive P_{k+1} from a transformation sequence $P_0, \ldots, P_k$ is said to be *partially correct* (or *totally correct*) w.r.t. **SEM** iff for every query Q in $\mathbf{Q}$, containing only predicate symbols which occur in P_k, we have that $\mathbf{SEM}[P_{k+1}, Q] \leq \mathbf{SEM}[P_k, Q]$ (or $\mathbf{SEM}[P_{k+1}, Q] = \mathbf{SEM}[P_k, Q]$).

Notice that, if a transformation sequence is constructed by performing a sequence of partially correct transformation steps, then it is partially correct. However, it may be the case that not all transformation steps realizing a partially correct transformation sequence, are partially correct. Also, the application of a partially correct transformation rule may generate a transformation step which is not partially correct.

Similar remarks also hold for total correctness, instead of partial correctness.

Obviously, if $P_0, \ldots, P_k$ and $P_k, P_{k+1}, \ldots, P_n$ are partially correct (or totally correct) transformation sequences, also their 'concatenation' $P_0, \ldots, P_k, P_{k+1}, \ldots, P_n$ is partially correct (or totally correct). In what follows, by 'correctness' we will mean 'total correctness'.

The following lemma establishes a correctness result for the definition introduction and definition elimination rules assuming that **SEM** is relevant. In the subsequent sections we will give some more correctness results which hold with different assumptions on **SEM**.

Lemma 4.0.2 (Relevance). *The rules of definition introduction and definition elimination are totally correct w.r.t. any relevant semantics.*

4.1 Reversible Transformations

We present here a simple method to prove that a transformation sequence constructed by applying partially correct rules is totally correct. This method is based on the notion of *reversible* transformation sequence.

Definition 4.1.1 (Reversible Transformations). A transformation sequence $P_0, P_1, \ldots, P_{n-1}, P_n$ constructed by using a set $\mathbf{R}$ of rules is said to be *reversible* 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 rules in $\mathbf{R}$.

A transformation step which allows us to derive P_{k+1} from P_k is said to be *reversible* iff P_k, P_{k+1} is a reversible transformation sequence. In particular, if P_k, P_{k+1} has been constructed by using a rule $\mathrm{R_a}$ and P_{k+1}, P_k can be constructed by using a rule $\mathrm{R_b}$, then we say that a transformation step using $\mathrm{R_a}$ is *reversible* by a transformation step using $\mathrm{R_b}$.

Notice that, in the above definition of reversible transformations the construction of the transformation sequence $P_n, Q_1, \ldots, Q_k, P_0$ is required to be independent of the construction of the transformation sequence $P_0, P_1, \ldots, P_{n-1}, P_n$. This independence condition is essential because, in general, we can derive a new program by using clauses occurring in a program which precedes the last one in the transformation sequence at hand. Thus, it may be the case that there exists a transformation sequence $P_0, P_1, \ldots, P_{n-1}, P_n, R_1, \ldots, R_h, P_0$, for $h \geq 0$, but there is no transformation sequence $P_n, Q_1, \ldots, Q_k, P_0$. In this case the transformation sequence $P_0, P_1, \ldots, P_{n-1}, P_n$ is not reversible.

In particular, there are folding steps which are not reversible by unfolding steps, because the clauses to be used for unfolding are not available.

Example 4.1.2. Let us consider the following program:

$P_0: \qquad p \leftarrow q \qquad q \leftarrow q$

By folding the first clause using itself, we get the program

$P_1: \qquad p \leftarrow p \qquad q \leftarrow q$

This folding step is not reversible by any sequence of unfolding steps, because starting from program P_1, it is impossible to get program P_0 by applying the unfolding rule only.

The importance of the reversibility property is given by the following result, whose proof is straightforward.

Lemma 4.1.3 (Reversibility). *Let* **SEM** *be a semantics function and* **R** *a set of transformation rules.*

If a transformation step using a rule $\mathrm{R_a}$ *in* **R** *is reversible by a totally correct transformation step using a rule* $\mathrm{R_b}$ *in* **R**, *then the transformation step using* $\mathrm{R_a}$ *is totally correct.*

If the rules in **R** *are partially correct w.r.t.* **SEM**, *then any reversible transformation sequence using rules in* **R** *is totally correct w.r.t.* **SEM**.

Notice, however, that in general it is hard to check whether or not a transformation sequence is reversible.

We now consider instances of the folding rule and the goal replacement rule which always produce reversible transformation sequences.

Rule R2.1 In-Situ Folding. A folding step is called an *in-situ folding* iff, with reference to rule R2, we have that

- $P_k = P_j$ (that is, the clauses used for folding are taken from the last program, not from a previous program, in the transformation sequence at hand) and
- $\{C_1, \ldots, C_n\} \cap \{D_1, \ldots, D_n\} = \{\}$ (that is, no clause among $C_1, \ldots, C_n$ is used to fold $C_1, \ldots, C_n$).

Any in-situ folding step which derives a clause C from $C_1, \ldots, C_n$ in P_k using clauses $D_1, \ldots, D_n$ in P_k itself, is reversible by unfolding C using

$D_1, \ldots, D_n$. Indeed, clauses $D_1, \ldots, D_n$ occur in P_{k+1} and by unfolding C using $D_1, \ldots, D_n$ we get $C_1, \ldots, C_n$. Thus, P_{k+1}, P_k is a transformation sequence which can be produced by an unfolding step.

A folding rule similar to in-situ folding has been considered by Maher [1990; 1993] in the more general context of logic programs *with constraints*. Other instances of the in-situ folding rule have been proposed in [Maher, 1987; Gardner and Shepherdson, 1991].

We will see in Section 4.4 that the reversibility property of in-situ folding allows us to establish in a straightforward way some total correctness results for this rule. However, in-situ folding has limited power, in the sense that, as we will see in Section 5, most transformation strategies for improving program efficiency make use of folding steps which are not in-situ foldings.

Rule R5.2 Persistent Goal Replacement. Let C be a clause in P_k and goal G_1 be equivalent to goal G_2 w.r.t. a semantics **SEM** and program P_k. The goal replacement of G_1 by G_2 in C is said to be *persistent* iff G_1 and G_2 are equivalent w.r.t. **SEM** and the derived program P_{k+1}.

Any persistent goal replacement step which replaces G_1 by G_2 is reversible by a goal replacement step which performs the inverse replacement of G_2 by G_1. Thus, if the goal replacement rule is partially correct w.r.t. **SEM**, then any persistent goal replacement step is totally correct w.r.t. **SEM**. In the following definition we introduce a variant of the goal replacement rule which is reversible if one considers relevant semantics.

Rule R5.3 Independent Goal Replacement. Let C be a clause in a program P and goal G_1 be equivalent to goal G_2 w.r.t. a semantics **SEM** and program P. The replacement of G_1 by G_2 in C is said to be *independent* iff C belongs to neither $P_{rel}(G_1)$ nor $P_{rel}(G_2)$ (that is, neither G_1 nor G_2 depends on C).

Lemma 4.1.4 (Reversibility of Independent Goal Replacement). *Let* **SEM** *be a relevant semantics and goal* G_1 *be equivalent to goal* G_2 *w.r.t.* **SEM** *and program* P. *Any independent goal replacement of* G_1 *by* G_2 *in a clause of* P *is reversible by performing an independent goal replacement step.*

Proof. Let Q be the program obtained from P by replacing the goal G_1 by the goal G_2 in a clause C of P such that neither G_1 nor G_2 depends on C. We first show that this independent goal replacement step is persistent by proving that G_1 and G_2 are equivalent w.r.t. **SEM** and program Q, that is, $\mathbf{SEM}[Q, \leftarrow G_1] = \mathbf{SEM}[Q, \leftarrow G_2]$. Indeed, we have that

$\mathbf{SEM}[Q, \leftarrow G_1]$	(by relevance of **SEM**)
$= \mathbf{SEM}[Q_{rel}(G_1), \leftarrow G_1]$	(since $C \notin P_{rel}(G_1)$)
$= \mathbf{SEM}[P_{rel}(G_1), \leftarrow G_1]$	(by relevance of **SEM**)
$= \mathbf{SEM}[P, \leftarrow G_1]$	(since G_1 is equivalent to G_2 w.r.t.

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ **SEM** and P)

$= \mathbf{SEM}[P, \leftarrow G_2]$ (by relevance of **SEM**)

$= \mathbf{SEM}[P_{rel}(G_2), \leftarrow G_2]$ (since $C \notin P_{rel}(G_2)$)

$= \mathbf{SEM}[Q_{rel}(G_2), \leftarrow G_2]$ (by relevance of **SEM**)

$= \mathbf{SEM}[Q, \leftarrow G_2]$.

Since any independent goal replacement step is persistent, it is also reversible by performing the inverse independent goal replacement of G_2 by G_1 in the program Q. ∎

4.2 A Derived Goal Replacement Rule

In this section we show that the clausal goal replacement rule can be derived from the following transformation rules: unfolding, in-situ folding, non-recursive definition introduction, definition elimination, and goal replacement, whenever the rules of non-recursive definition introduction and in-situ folding are totally correct w.r.t. the semantics **SEM**. A different way of viewing the goal replacement rule as a sequence of transformation rules can be found in [Bossi *et al.*, 1992a].

Let P_k be the program $C_1, \ldots, C_n$ and let us consider clause C_i of the form $H \leftarrow L, G_1, R$. Suppose that the goal G_1 is equivalent to the goal G_2 w.r.t. **SEM**, P_k, and $\{X_1, \ldots, X_m\} = vars(H, L, R) \cap vars(G_1, G_2)$.

By applying the clausal goal replacement rule R5.1 (page 18) we may replace G_1 by G_2 in C_i, thereby deriving the new clause D of the form $H \leftarrow L, G_2, R$ and the new program $P_{k+1} = C_1, \ldots, C_{i-1}, D, C_{i+1}, \ldots, C_n$.

The same program P_{k+1} can be derived by the sequence of the following five transformation steps.

Step 1. By the non-recursive definition rule we introduce the clauses

D_1. $newp_1(X_1, \ldots, X_m) \leftarrow G_1$

D_2. $newp_2(X_1, \ldots, X_m) \leftarrow G_2$

Then we get the program $C_1, \ldots, C_n, D_1, D_2$.

Step 2. Since $vars(H, L, R) \cap vars(G_1) \subseteq \{X_1, \ldots, X_m\}$, condition 4 of the folding rule is fulfilled. Also the other conditions 1, 2, and 3 are fulfilled. By in-situ folding of clause C_i using D_1 we derive the clause

F. $H \leftarrow L, newp_1(X_1, \ldots, X_m), R$

Thus, we get the new program

$Q = C_1, \ldots, C_{i-1}, F, C_{i+1}, \ldots, C_n, D_1, D_2$.

Step 3. The goals $newp_1(X_1, \ldots, X_m)$ and $newp_2(X_1, \ldots, X_m)$ are equivalent w.r.t. **SEM** and Q. Indeed, we have

$\mathbf{SEM}[(C_1, \ldots, C_{i-1}, C_i, C_{i+1}, \ldots, C_n, D_1), \leftarrow newp_1(X_1, \ldots, X_m)] =$

(by the correctness of non-recursive definition introduction)

$= \mathbf{SEM}[(C_1, \ldots, C_{i-1}, C_i, C_{i+1}, \ldots, C_n, D_1, D_2), \leftarrow newp_1(X_1, \ldots, X_m)]$

(by the correctness of in-situ folding)

$= \mathbf{SEM}[(C_1,\ldots,C_{i-1},F,C_{i+1},\ldots,C_n,D_1,D_2), \leftarrow newp_1(X_1,\ldots,X_m)]$

and

$\mathbf{SEM}[(C_1,\ldots,C_{i-1},C_i,C_{i+1},\ldots,C_n,D_1), \leftarrow newp_1(X_1,\ldots,X_m)] =$

(since G_1 and G_2 are equivalent w.r.t. **SEM**, P_k, and $\{X_1,\ldots,X_m\}$)

$= \mathbf{SEM}[(C_1,\ldots,C_{i-1},C_i,C_{i+1},\ldots,C_n,D_2), \leftarrow newp_2(X_1,\ldots,X_m)]$

(by the correctness of non-recursive definition introduction)

$= \mathbf{SEM}[(C_1,\ldots,C_{i-1},C_i,C_{i+1},\ldots,C_n,D_1,D_2), \leftarrow newp_2(X_1,\ldots,X_m)]$

(by the correctness of in-situ folding)

$= \mathbf{SEM}[(C_1,\ldots,C_{i-1},F,C_{i+1},\ldots,C_n,D_1,D_2), \leftarrow newp_2(X_1,\ldots,X_m)]$.

Thus, by applying the goal replacement rule R5, from clause F in program Q we derive the new clause

$M.\ H \leftarrow L, newp_2(X_1,\ldots,X_m), R$

and we get the new program $C_1,\ldots,C_{i-1},M,C_{i+1},\ldots,C_n,D_1,D_2$.

Step 4. By unfolding M w.r.t. $newp_2(X_1,\ldots,X_m)$ we get

$D.\ H \leftarrow L, G_2, R$

and the program $C_1,\ldots,C_{i-1},D,C_{i+1},\ldots,C_n,D_1,D_2$.

Step 5. Finally, by definition elimination we discard clauses D_1 and D_2, and we get exactly the program P_{k+1}, as desired.

We have also the converse result which we will show below, that is, if by goal replacement from program P_k we get program P_{k+1} and we assume that the non-recursive definition introduction rule and the independent goal replacement rule are totally correct w.r.t. **SEM**, then we may apply the clausal goal replacement rule for deriving program P_{k+1} from program P_k.

Indeed, suppose that the goals G_1 and G_2 are equivalent w.r.t. **SEM** and $P_k = C_1,\ldots,C_n$. Also, consider the clauses

$D_1.\ newp_1(X_1,\ldots,X_m) \leftarrow G_1$
$D_2.\ newp_2(X_1,\ldots,X_m) \leftarrow G_2$

where $\{X_1,\ldots,X_m\}$ is any set of variables.

By the correctness of the non-recursive definition introduction rule we have that

$\mathbf{SEM}[(C_1,\ldots,C_n),\ \leftarrow G_1] =$

$= \mathbf{SEM}[(C_1,\ldots,C_n,\ newp_1(X_1,\ldots,X_m) \leftarrow G_1),\ \leftarrow G_1]$ and

$\mathbf{SEM}[(C_1,\ldots,C_n),\ \leftarrow G_2] =$

$= \mathbf{SEM}[(C_1,\ldots,C_n,\ newp_1(X_1,\ldots,X_m) \leftarrow G_1),\ \leftarrow G_2]$.

Since the goals G_1 and G_2 are equivalent w.r.t. **SEM** and $(C_1, \ldots, C_n)$, that is, $\mathbf{SEM}[(C_1, \ldots, C_n),\ \leftarrow G_1] = \mathbf{SEM}[(C_1, \ldots, C_n),\ \leftarrow G_2]$, we have that

$$\mathbf{SEM}[(C_1, \ldots, C_n,\ newp_1(X_1, \ldots, X_m) \leftarrow G_1),\ \leftarrow G_1] =$$
$$= \mathbf{SEM}[(C_1, \ldots, C_n,\ newp_1(X_1, \ldots, X_m) \leftarrow G_1),\ \leftarrow G_2],$$

that is, G_1 and G_2 are equivalent w.r.t. **SEM** and the program $(C_1, \ldots, C_n, newp_1(X_1, \ldots, X_m) \leftarrow G_1)$.

As a consequence, we may apply the independent goal replacement rule and from program $(C_1, \ldots, C_n,\ newp_1(X_1, \ldots, X_m) \leftarrow G_1)$ we derive program $(C_1, \ldots, C_n,\ newp_1(X_1, \ldots, X_m) \leftarrow G_2)$.

Thus,

$$\mathbf{SEM}[(C_1, \ldots, C_n, newp_1(X_1, \ldots, X_m) \leftarrow G_1), \leftarrow newp_1(X_1, \ldots, X_m)] =$$

(by the correctness of the independent goal replacement rule)

$$= \mathbf{SEM}[(C_1, \ldots, C_n, newp_1(X_1, \ldots, X_m) \leftarrow G_2), \leftarrow newp_1(X_1, \ldots, X_m)]$$

(since **SEM** is preserved by predicate renaming)

$$= \mathbf{SEM}[(C_1, \ldots, C_n, newp_2(X_1, \ldots, X_m) \leftarrow G_2), \leftarrow newp_2(X_1, \ldots, X_m)].$$

We conclude that: i) G_1 and G_2 are equivalent w.r.t. **SEM**, P_k, and any set of variables (see Definition 3.3.9, page 17), and ii) the replacement of G_1 by G_2 in any clause of P_k may be viewed as an application of the clausal goal replacement rule.

To sum up the results presented in this section we have that, if the rules of non-recursive definition introduction, in-situ folding, and independent goal replacement are correct w.r.t. **SEM**, then the goal replacement and clausal goal replacement are equivalent rules, in the sense that program Q can be derived from program P by using rules R1, R2, R3, R4, goal replacement (rule R5), and R6 iff Q can be derived from P by using rules R1, R2, R3, R4, clausal goal replacement (rule R5.1), and R6.

We will see that the non-recursive definition introduction, in-situ folding, and independent goal replacement rules are correct w.r.t. all semantics we will consider, and thus, when describing the correctness results w.r.t. those semantics, we can make no distinction between the goal replacement rule R5 and the clausal goal replacement rule R5.1.

4.3 The Unfold/Fold Proof Method

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 $G_1 \equiv_V G_2$ is valid w.r.t. a semantics **SEM** and a program P_k, then it is also valid w.r.t. **SEM** and Q for every program Q derived from P_k by using totally correct transformation rules.

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

Section 7). As an alternative approach, one can use a simple method based on unfold/fold transformations which we call *unfold/fold proof method*. This proof method was introduced by Kott for recursive equation programs [Kott, 1982] and its application to logic programs is described in [Boulanger and Bruynooghe, 1993; Proietti and Pettorossi, 1994a; Proietti and Pettorossi, 1994b].

The unfold/fold proof method can be described as follows. Given a program P, a semantics function **SEM**, and a replacement law $G_1 \equiv_V G_2$, with $V = \{X_1, \ldots, X_m\}$, we consider the clauses

D_1. $newp_1(X_1, \ldots, X_m) \leftarrow G_1$
D_2. $newp_2(X_1, \ldots, X_m) \leftarrow G_2$

and the programs $R_0 = C_1, \ldots, C_n, D_1$ and $S_0 = C_1, \ldots, C_n, D_2$.

We then construct two correct transformation sequences $R_0, \ldots, R_u$ and $S_0, \ldots, S_v$, such that R_u and S_v are equal modulo predicate renaming.

The validity of $G_1 \equiv_V G_2$ follows from the total correctness of the transformation sequences, and the assumption that **SEM** is preserved by predicate renaming.

Example 4.3.1. Consider again the program $P_k = C, A_1, A_2$ of the *Sublist* Example 3.3.8 (page 17). Suppose that we want to apply the clausal goal replacement rule to replace the goal $G_1 = (append(V, X, I), append(I, Z, Y))$ by the goal $G_2 = (append(X, Z, J),\ append(V, J, Y))$ in the body of the clause

C. $sublist(N, X, Y) \leftarrow length(X, N), append(V, X, I), append(I, Z, Y)$

We need to show the validity of the replacement law

$$append(V, X, I), append(I, Z, Y) \equiv_{\{X,Y\}} append(X, Z, J), append(V, J, Y)$$

where the equivalence w.r.t. the set $\{X, Y\}$ is justified by the fact that $vars(sublist(N, X, Y), length(X, N)) \cap vars(G_1, G_2) = \{X, Y\}$.

As suggested by the unfold/fold proof method, we introduce the clauses

D_1. $newp_1(X, Y) \leftarrow append(V, X, I), append(I, Z, Y)$
D_2. $newp_2(X, Y) \leftarrow append(X, L, J), append(K, J, Y)$

We then consider the programs $R_0 = C, A_1, A_2, D_1$ and $S_0 = C,\ A_1,\ A_2,\ D_2$.

We now construct two transformation sequences starting from R_0 and S_0, respectively, as follows.

1. *Transformation sequence starting from* R_0.
   By unfolding clause D_1 in R_0 w.r.t. $append(V, X, I)$ we derive the following two clauses:
   E_1. $newp_1(X, Y) \leftarrow append(X, Z, Y)$
   E_2. $newp_1(X, Y) \leftarrow append(T, X, U), append([H|U], Z, Y)$
   and we get the program $R_1 = C, A_1, A_2, E_1, E_2$.

Then, by unfolding clause E_2 w.r.t. $append([H|U], Z, Y)$ we get

E_3. $newp_1(X, [H|V]) \leftarrow append(T, X, U), append(U, Z, V)$

and we get the program $R_2 = C, A_1, A_2, E_1, E_3$.
Finally, by folding clause E_3 using clause D_1 in R_0 we derive

E_4. $newp_1(X, [H|V]) \leftarrow newp_1(X, V)$

and we get the program $R_3 = C, A_1, A_2, E_1, E_4$.

2. *Transformation sequence starting from* S_0.
By unfolding clause D_2 in S_0 w.r.t. $append(K, J, Y)$ we derive two clauses

F_1. $newp_2(X, Y) \leftarrow append(X, L, Y)$
F_2. $newp_2(X, [H|U]) \leftarrow append(X, L, J), append(T, J, U)$

and we get the program $S_1 = C,\ A_1,\ A_2,\ F_1,\ F_2$.
By folding clause F_2 using clause D_2 in S_0 we get the clause

F_3. $newp_2(X, [H|U]) \leftarrow newp_2(X, U)$

and we derive the final program of this transformation sequence which is $S_2 = C, A_1, A_2, F_1, F_3$.

The derived programs R_3 and S_2 are equal up to the predicate renaming (that is, for a renaming ρ which maps $newp_1$ to $newp_2$ we have $\rho(R_3) = S_2$) and the validity of the given replacement law is proved w.r.t. any semantics **SEM**, provided the transformation sequences R_0, R_1, R_2, R_3 and S_0, S_1, S_2 are correct w.r.t. **SEM**. We will show below that these transformation sequences are correct w.r.t. several semantics and, in particular, the least Herbrand model semantics **LHM**.

4.4 Correctness Results for Definite Programs

Let us now consider definite programs and let us study the correctness properties of the transformation rules w.r.t. various semantics. We will first look at the correctness properties of the unfold/fold transformations w.r.t. both the least Herbrand model (Section 4.4.1) and the computed answer substitution semantics (Section 4.4.2). We will then take into account various semantics related to program termination, such as the finite failure semantics (Section 4.4.3) and the answer substitution semantics computed by the depth-first search strategy of Prolog (Section 4.4.4).

We will assume that the equivalence between goals, as well as the various instances of the goal replacement rule, refer to the semantics considered in each section.

4.4.1 Least Herbrand Model

In this section we assume that the semantics function is **LHM** (page 11) and we present several partial correctness and total correctness results based on the work in [Tamaki and Sato, 1984; Tamaki and Sato, 1986; Maher, 1987; Gardner and Shepherdson, 1991; Gergatsoulis and Katzouraki, 1994].

The total correctness w.r.t. **LHM** of the unfolding steps is a straightforward consequence of the soundness and completeness of SLD-resolution w.r.t. the least Herbrand model semantics. As a consequence, by the reversibility lemma (page 21) any in-situ folding step is totally correct w.r.t. **LHM**.

In the general case, by applying the folding rule R2 to 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, the folding rule is partially correct w.r.t. **LHM**. It is not totally correct, as is shown by the following example.

Example 4.4.1. Given the program

P_0: $\quad p \leftarrow q \qquad q \leftarrow$

by folding the first clause using itself we get

P_1: $\quad p \leftarrow p \qquad q \leftarrow$

We have that $\mathbf{LHM}[P_1, \leftarrow p] \neq \mathbf{LHM}[P_0, \leftarrow p]$, because p is true in the least Herbrand model of P_0 and it is false in the least Herbrand model of P_1.

The correctness of the definition introduction and elimination rules w.r.t. **LHM** follows from Lemma 4.0.2, since **LHM** is a relevant semantics.

Similarly to the case of the folding rule, also the goal replacement rule R5 is partially correct w.r.t. **LHM**. Indeed, by applying the goal replacement rule to 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.

It is easy to see that, in general, the goal replacement rule is not totally correct. Indeed, the folding step considered in Example 4.4.1, may also be taken as an example of goal replacement step which is not totally correct, because p is equivalent to q w.r.t. **LHM** and P_0.

However, there are some instances of the goal replacement rule which are totally correct. In particular, since **LHM** is a relevant semantics, by the partial correctness of the goal replacement rule, the reversibility lemma (page 21), and the reversibility of independent goal replacement lemma (page 22), the independent goal replacement rule is totally correct w.r.t. **LHM**.

Let us now consider the following two special cases of the goal replacement rule.

Rule R5.4 Goal Rearrangement. By the *goal rearrangement* rule we replace a goal (G, H) in the body of a clause by the goal (H, G).

Rule R5.5 Deletion of Duplicate Goals. By the *deletion of duplicate goals* rule we replace a goal (G, G) in the body of a clause by the goal G.

Steps of goal rearrangement and deletion of duplicate goals are totally correct w.r.t. **LHM**. Indeed, they are persistent goal replacement steps, because they are based on the following equivalences w.r.t. **LHM**: $(G, H) \equiv_{vars(G,H)} (H, G)$ and $(G, G) \equiv_{vars(G)} G$, which hold w.r.t. any given program.

From these total correctness results it follows that, when dealing with the least Herbrand model semantics, we may assume that bodies of clauses are sets (not sequences) of atoms.

The total correctness of the clause replacement rules is also straightforward. Thus, since the rules for clause rearrangement (rule R6.1) and deletion of duplicate clauses (rule derived from rule R6.2) are totally correct w.r.t. **LHM**, we may assume that programs are sets (not sequences) of clauses.

However, we will see that some instances of the above rules are not correct when considering the computed answer substitution semantics (see Section 4.4.2, page 32) or the pure Prolog semantics (see Section 4.4.4, page 36).

As a summary of the results mentioned so far we have the following:

Theorem 4.4.2 (First Correctness Theorem w.r.t. LHM). *Let P_0, ..., P_n be a transformation sequence of definite programs constructed by using the following transformation rules: unfolding, in-situ folding, definition introduction, definition elimination, goal rearrangement, deletion of duplicate goals, independent goal replacement, and clause replacement. Then $P_0, \ldots, P_n$ is totally correct w.r.t.* **LHM**.

We have seen that the in-situ folding rule has the advantage of being a totally correct transformation rule, but it is a weak rule because it does not allow us to derive recursive definitions. In order to overcome this limitation we now present a different and more powerful version of the folding rule, called *single-folding*.

Let us first notice that by performing a folding step and introducing recursive clauses from non-recursive clauses, some infinite computations (due to a non well-founded recursion) may replace finite computations, thereby affecting the semantics of the program and loosing total correctness.

A simple example of this undesirable introduction of infinite computations is *self-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 by $p \leftarrow p$ (see Example 4.4.1, page 28).

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 for various techniques in which total correctness is ensured by counting the number of unfolding and folding steps performed during the transformation sequence

[Kott, 1978; Kanamori and Fujita, 1986; Bossi *et al.*, 1992a].

An alternative approach is based on the verification that some termination properties are preserved by the transformation process, thereby avoiding the introduction of infinite computations [Amtoft, 1992; Bossi and Etalle, 1994b; Bossi and Cocco, 1994; Cook and Gallagher, 1994].

The following definition introduces the version of the folding rule we have promised above. This version is a special case of rule R2 for $n = 1$.

Rule R2.2 Single-Folding. Let C be a clause in program P_k and D be a clause in a program P_j, for some j, with $0 \leq j \leq k$. Suppose that there exist two goals F and G and a substitution θ such that:

1. C is a variant of $H \leftarrow F, bd(D)\theta, G$,
2. for every clause E of P_j different from D, $hd(E)$ is not unifiable with $hd(D)\theta$, and
3. for every variable X in the set $vars(D) - vars(hd(D))$, we have that
   - $X\theta$ is a variable which does not occur in (H, F, G) and
   - the variable $X\theta$ does not occur in the term $Y\theta$, for any variable Y occurring in $bd(D)$ and different from X.

By the *single-folding* rule, using clause D, from clause C we derive clause $H \leftarrow F, hd(D)\theta, G$.

This rule is called *T&S-folding* in [Pettorossi and Proietti, 1994].

We now present a correctness result analogous to the first correctness theorem w.r.t. **LHM** for a transformation sequence including single-folding, rather than in-situ folding. We essentially follow [Tamaki and Sato, 1986], but we make some simplifying assumptions. By doing so, total correctness is ensured by easily verifiable conditions on the transformation sequence.

We assume that the set of the predicate symbols occurring in the transformation sequence $P_0, \ldots, P_n$ is partitioned into three sets, called *top predicates, intermediate predicates,* and *basic predicates*, respectively, with the following restrictions:

1. a predicate introduced by the definition rule is a *top predicate*,
2. an *intermediate predicate* does not depend in P_0 on any top predicate, and
3. a *basic predicate* does not depend in P_0 on any intermediate or top predicate.

Notice that this partition process is, in general, nondeterministic. In particular, we can choose the top predicates in P_0 in various ways. Notice also that dependencies of some intermediate predicates on top predicates may be introduced by folding steps (see the second correctness theorem below).

The approach we follow here is more general than the approach described in [Kawamura and Kanamori, 1990], where only two sets of predicates are considered (the so-called *new* predicates and *old* predicates).

Let us also introduce a new goal replacement rule, called *basic goal replacement*, which is a particular case of independent goal replacement.

Rule R5.6 Basic Goal Replacement. By the *basic goal replacement* rule we replace a goal G_1 in the body of a clause C by a goal G_2 such that any predicate occurring in G_1 or G_2 is a basic predicate and the head of C has a top or an intermediate predicate.

The following theorem establishes the correctness of transformation sequences which are constructed by applying a set of transformation rules including the single-folding rule and the basic goal replacement rule.

Theorem 4.4.3 (Second Correctness Theorem w.r.t. LHM). *Let $P_0, \ldots, P_n$ be a transformation sequence of definite programs constructed by using the following transformation rules: unfolding, single-folding, definition introduction, definition elimination, goal rearrangement, deletion of duplicate goals, basic goal replacement, and clause replacement. Suppose that no single-folding step is performed after a definition elimination step. Suppose also that when we apply the single-folding rule to a clause, say C, using a clause, say D, the following conditions hold:*

- either *D belongs to P_0* or *D has been introduced by the definition rule,*
- *$hd(D)$ has a top predicate, and*
- either *$hd(C)$ has an intermediate predicate*
  or *$hd(C)$ has a top predicate and C has been derived from a clause, say E, by first unfolding E w.r.t. an atom with an intermediate predicate and then performing zero or more transformation steps on a clause derived from the unfolding of E.*

Then $P_0, \ldots, P_n$ is totally correct w.r.t. the semantics **LHM**.

The hypothesis that no single-folding step is performed after a definition elimination step is needed to prevent that single-folding is applied using a clause with a head predicate whose definition has been eliminated. This point is illustrated by the following example.

Example 4.4.4. Let us consider the transformation sequence

P_0:	$p \leftarrow q$	$p \leftarrow fail$	$q \leftarrow$	
	(by definition introduction)			
P_1:	$p \leftarrow q$	$p \leftarrow fail$	$q \leftarrow$	$newp \leftarrow q$
	(by definition elimination)			
P_2:	$p \leftarrow q$	$p \leftarrow fail$	$q \leftarrow$	
	(by single-folding)			
P_3:	$p \leftarrow newp$	$p \leftarrow fail$	$q \leftarrow$	

We may assume that *newp* is a top predicate and p is an intermediate predicate. However, the transformation sequence is not correct w.r.t.

LHM, because the query $\leftarrow p$ succeeds in the initial program, while it fails in the final one.

The second correctness theorem w.r.t. **LHM** can be extended to other variants of the folding rule as described in [Gergatsoulis and Katzouraki, 1994].

4.4.2 Computed Answer Substitutions

We now consider a semantics function based on the notion of *computed answer substitutions* [Lloyd, 1987; Apt, 1990], which captures the procedural behavior of definite programs more accurately than the least Herbrand model semantics.

The computed answer substitution semantics can be defined as a function

$$\mathbf{CAS}\colon \mathbf{P}^+ \times \mathbf{Q}^+ \to (\mathcal{P}(\mathbf{Subst}), \leq)$$

where $\mathbf{P}^+$ is the set of definite programs, $\mathbf{Q}^+$ is the set of definite queries, and $(\mathcal{P}(\mathbf{Subst}), \leq)$ is the powerset of the set of substitutions ordered by set inclusion. We define the semantics **CAS** as follows:

$$\mathbf{CAS}[P, Q] = \{\theta \mid \text{there exists an SLD-refutation of } Q \text{ in } P \text{ with computed answer substitution } \theta\}.$$

CAS is a relevant semantics.

By the soundness and completeness of SLD-resolution w.r.t. **LHM**, we have that the equivalence of two programs w.r.t. **CAS** implies their equivalence w.r.t. **LHM**. However, the converse is not true. For instance, consider the following two programs:

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

We have that $\mathbf{LHM}[P_1, \leftarrow p(X)] = \mathbf{LHM}[P_2, \leftarrow p(X)] = \mathbf{Subst}$. However, we have that $\mathbf{CAS}[P_1, \leftarrow p(X)] = \{\{\}\}$, whereas $\mathbf{CAS}[P_2, \leftarrow p(X)] = \{\{\}, \{X/a\}\}$, where $\{\}$ is the identity substitution.

As a consequence, not all rules which are correct w.r.t. **LHM** are correct also w.r.t. **CAS**. In particular, the deletion of duplicate goals and the deletion of subsumed clauses do not preserve the **CAS** semantics, as is shown by the following examples.

Example 4.4.5. Let us consider the program

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

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

P_2: $\quad p(X) \leftarrow q(X) \qquad q(t(Y, a)) \leftarrow \qquad q(t(a, Z)) \leftarrow$

The substitution $\{X/t(a, a)\}$ belongs to $\mathbf{CAS}[P_1, \leftarrow p(X)]$ and not to $\mathbf{CAS}[P_2, \leftarrow p(X)]$.

Example 4.4.6. Let us consider the program

P: $\quad p(X) \leftarrow \quad p(a) \leftarrow$

The clause $p(a) \leftarrow$ is subsumed by $p(X) \leftarrow$. However, if we delete $p(a) \leftarrow$ from the program P the **CAS** semantics is not preserved, because $\{X/a\}$ is no longer a computed answer substitution for the query $\leftarrow p(X)$.

There are particular cases, however, in which the deletion of duplicate goals and the deletion of subsumed clauses are correct w.r.t. **CAS**. The following definitions introduce two such cases.

Rule R5.7 Deletion of Duplicate Ground Goals. By the rule of *deletion of ground duplicate goals* we replace a *ground* goal (G, G) in the body of a clause by the goal G.

This rule is an instance of the persistent goal replacement rule (see rule R5.2, page 22).

Rule R5.8 Deletion of Duplicate Clauses. By the rule of *deletion of duplicate clauses* we replace all occurrences of a clause C in a program by a single occurrence of C.

Several researchers have addressed the problem of proving the correctness of the transformation rules w.r.t. **CAS** [Kawamura and Kanamori, 1990; Bossi *et al.*, 1992a; Bossi and Cocco, 1993]. We now present for the **CAS** semantics two theorems which correspond to the first and second correctness theorems w.r.t. the **LHM** semantics. As already mentioned, in these theorems the various instances of the goal replacement rule refer to the equivalence of goals w.r.t. **CAS**.

Theorem 4.4.7 (First Correctness Theorem w.r.t. CAS). *Let P_0, $\ldots$, P_n be a transformation sequence of definite programs constructed by using the following transformation rules: unfolding, in-situ folding, definition introduction, definition elimination, goal rearrangement, deletion of ground duplicate goals, independent goal replacement, clause rearrangement, deletion of duplicate clauses, deletion of clauses with finitely failed body, and generalization + equality introduction. Then $P_0, \ldots, P_n$ is totally correct w.r.t.* **CAS**.

Theorem 4.4.8 (Second Correctness Theorem w.r.t. CAS). *Let P_0, $\ldots, P_n$ be a transformation sequence of definite programs constructed by using the following transformation rules: unfolding, single-folding, definition introduction, definition elimination, goal rearrangement, deletion of ground duplicate goals, basic goal replacement, clause rearrangement, deletion of duplicate clauses, deletion of clauses with finitely failed body, and generalization + equality introduction. Suppose that no single-folding step is performed after a definition elimination step. Suppose also that when we apply the single-folding rule to a clause, say C, using a clause, say D, the following conditions hold:*

- either *D belongs to P_0* or *D has been introduced by the definition rule,*
- *$hd(D)$ has a top predicate, and*
- either *$hd(C)$ has an intermediate predicate* or *$hd(C)$ has a top predicate and C has been derived from a clause, say E, by first unfolding E w.r.t. an atom with an intermediate predicate and then performing zero or more transformation steps on a clause derived from the unfolding of E.*

Then $P_0, \ldots, P_n$ is totally correct w.r.t. the semantics **CAS**.

4.4.3 Finite Failure

In order to reason about the preservation of finite failure during program transformation we now consider the semantics function

$$\mathbf{FF}\colon \mathbf{P}^+ \times \mathbf{Q}^+ \to (\mathcal{P}(\mathbf{Subst}), \leq)$$

such that

$$\mathbf{FF}[P, Q] = \{\theta \mid \text{there exists a finitely failed SLD-tree for } Q\theta \text{ in } P\}.$$

The reader may verify that **FF** is a relevant semantics.

Work on the correctness of transformation rules w.r.t. **FF** has been presented in [Maher, 1987; Seki, 1991; Gardner and Shepherdson, 1991; Cook and Gallagher, 1994]. Similarly to the case of **LHM** and **CAS**, we have the following result, where the independent goal replacement rule is defined in terms of goal equivalence w.r.t. **FF**.

Theorem 4.4.9 (First Correctness Theorem w.r.t. FF). *Let $P_0, \ldots, P_n$ be a transformation sequence of definite programs constructed by using the following transformation rules: unfolding, in-situ folding, definition introduction, definition elimination, goal rearrangement, deletion of duplicate goals, independent goal replacement, and clause replacement. Then $P_0, \ldots, P_n$ is totally correct w.r.t.* **FF**.

However, the use of the rules listed in the second correctness theorems w.r.t. **LHM** and **CAS** (pages 31 and 33), may affect **FF**. In particular, if we allow folding steps which are not in-situ foldings, we may transform a finitely failing program into an infinitely failing program, as shown by the following example.

Example 4.4.10. Let us consider the transformation sequence, where p is a top predicate and q and r are intermediate predicates:

P_0: $\quad p(X) \leftarrow q(X), r(X) \qquad q(a) \leftarrow \qquad r(b) \leftarrow r(b)$

(by unfolding the first clause w.r.t. $r(X)$)

P_1: $\quad p(b) \leftarrow q(b), r(b) \qquad q(a) \leftarrow \qquad r(b) \leftarrow r(b)$

(by applying single-folding to the first clause)

P_2: $p(b) \leftarrow p(b)$ $\quad q(a) \leftarrow$ $\quad r(b) \leftarrow r(b)$

This transformation sequence satisfies the conditions stated in the second correctness theorem w.r.t. both **LHM** and **CAS**, but P_0 finitely fails for the query $\leftarrow p(b)$, while P_2 does not.

As we have shown in the above Example 4.4.10, if we allow folding steps which are not in-situ foldings, it may be the case that the derived transformation sequence is not correct w.r.t. **FF**.

The fact that such folding steps are not totally correct w.r.t. **FF** is related to the notion of fair SLD-derivation [Lloyd, 1987].

A possibly infinite SLD-derivation is *fair* iff it is either failed or for every occurrence of an atom A in the SLD-derivation, that occurrence of A or an instance (possibly via the identity substitution) of A which is derived from that occurrence, is selected for SLD-resolution within a finite number of steps.

Fairness of SLD-derivations is a sufficient condition for the completeness of SLD-resolution w.r.t. **FF**.

Let us consider a program P_1 and a query Q, and let us apply a folding step which replaces a goal B by an atom H, thereby obtaining a program, say P_2. We may view every SLD-derivation δ_2 of Q in the derived program P_2 as 'simulating' an SLD-derivation δ_1 of Q in P_1. Indeed, the simulated SLD-derivation δ_1 can be obtained by replacing in δ_2 the instances of H introduced by folding steps, with the corresponding instances of B.

By applying a folding step (which is not an in-situ folding) to a clause in P_1, we may derive a program P_2 such that a fair SLD-derivation for Q using P_2 simulates an unfair SLD-derivation for Q using P_1, as shown by the following example.

Example 4.4.11. Let us consider the program P_2 of Example 4.4.10 and the infinite sequence of queries

$\leftarrow p(b) \qquad \leftarrow p(b) \qquad \leftarrow p(b) \qquad \ldots$

which constitutes a fair SLD-derivation for the program P_2 and the query $\leftarrow p(b)$. The folding step which produces P_2 from P_1 replaces the goal $(q(b), r(b))$ by the goal $p(b)$. The above SLD-derivation can be viewed as a simulation of the following SLD-derivation for P_1:

$\leftarrow p(b) \qquad \leftarrow q(b), r(b) \qquad \leftarrow q(b), r(b) \qquad \ldots$

which is unfair, because it has been obtained by always selecting the atom $r(b)$ for performing an SLD-resolution step.

The Theorem 4.4.12 below is the analogous for the **FF** semantics of the second correctness theorems w.r.t. **LHM** and **CAS**. Its proof is based on the fact that an unfair SLD-derivation of a given program cannot be simulated by a fair SLD-derivation of a transformed program, 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 4.4.10 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.

Theorem 4.4.12 (Second Correctness Theorem w.r.t. FF). *Let P_0, $\ldots, P_n$ be a transformation sequence of definite programs constructed by using the following transformation rules: unfolding, single-folding, definition introduction, definition elimination, goal rearrangement, deletion of duplicate goals, basic goal replacement, and clause replacement. Suppose that no single-folding step is performed after a definition elimination step. Suppose also that when we apply the single-folding rule to a clause, say C, using a clause, say D, the following conditions hold:*

- either *D belongs to P_0* or *D has been introduced by the definition rule,*
- *$hd(D)$ has a top predicate, and*
- either *$hd(C)$ has an intermediate predicate*
  or *$hd(C)$ has a top predicate and* each *atom of $bd(C)$ w.r.t. which the single-folding step is performed, has been derived in a previous transformation step by first unfolding a clause, say E, w.r.t. an atom with an intermediate predicate and then performing zero or more transformation steps on a clause derived from the unfolding of E.*

Then $P_0, \ldots, P_n$ is totally correct w.r.t. the semantics **FF**.

4.4.4 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 performed in a given goal. In this SLD-tree, the nodes which are 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, in Prolog we have that the SLD-tree is an ordered tree, and it 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, by using Prolog control strategy SLD-resolution is not complete.

We will define a semantics function **Prolog** by taking into consideration the 'generation order' of the answer substitutions, their 'multiplicity', and their 'computability in finite time'. 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 brother nodes in T determines a left-to-right ordering of branches and leaves.

If T is finite then $\mathbf{Prolog}[P, Q]$ is the sequence of the computed answer substitutions 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, each substitution being associated with a leaf of T. The sequence F is obtained by visiting from left to right the non-failed leaves which are at the end of branches to the left of the leftmost infinite branch. There are two cases: either F is infinite, in which case $\mathbf{Prolog}[P,Q]$ is F or F is finite, in which case $\mathbf{Prolog}[P,Q]$ is F followed by the symbol $\perp$, called the *undefined substitution*. All substitutions different from $\perp$ are said to be *defined*.

Thus, our semantics for Prolog is a function

$$\mathbf{Prolog}\colon \mathbf{P}^{+} \times \mathbf{Q}^{+} \to (\mathbf{SubstSeq}, \leq)$$

where $\mathbf{P}^{+}$ and $\mathbf{Q}^{+}$ are the sets of definite programs and definite queries, respectively. $(\mathbf{SubstSeq}, \leq)$ is the set 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 and Mycroft, 1984; Debray and Mishra, 1988; Deville, 1990; Baudinet, 1992].

The sequence consisting of the substitutions $\theta_1, \theta_2, \ldots$ is denoted by $\langle \theta_1, \theta_2, \ldots \rangle$, 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: $\langle \perp \rangle @ S = \langle \perp \rangle$, for any sequence S.

Example 4.4.13. 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

$$\mathbf{Prolog}[P_1, \leftarrow p(X)] = \langle \{X/a\}, \{X/b\}, \{X/a\} \rangle$$
$$\mathbf{Prolog}[P_2, \leftarrow p(X)] = \langle \{X/a\}, \{X/a\}, \ldots \rangle$$
$$\mathbf{Prolog}[P_3, \leftarrow p(X)] = \langle \{X/a\}, \perp \rangle.$$

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 $S_1 = S_3 @ \langle \perp \rangle$ and $S_2 = S_3 @ S_4$, for some S_3 and S_4 in **SubstSeq**. For instance, we have that: (i) for all substitutions η_1 and η_2 with $\eta_1 \neq \perp$ and $\eta_2 \neq \perp$, $\langle \eta_1, \perp \rangle \leq \langle \eta_1, \eta_2 \rangle$, and the sequences $\langle \eta_1 \rangle$ and $\langle \eta_1, \eta_2 \rangle$ are not comparable w.r.t. $\leq$, and (ii) for any (possibly empty) sequence S, $\langle \perp \rangle \leq S$.

Unfortunately, most transformation rules presented in the previous sections are not even partially correct w.r.t. **Prolog**. Indeed, it is easy to see that a clause rearrangement may affect the 'generation order' or the 'computability in finite time' of the answer substitutions, and the deletion of a duplicate clause may affect their multiplicity.

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

Example 4.4.14. By unfolding w.r.t. $r(Y)$ the first clause of the following program:

$$
\begin{array}{llllll}
P_0: & p(X,Y) \leftarrow q(X), r(Y) & & & \\
 & q(a) \leftarrow & q(b) \leftarrow & r(a) \leftarrow & r(b) \leftarrow
\end{array}
$$

we get

$$
\begin{array}{llllll}
P_1: & p(X,a) \leftarrow q(X) & p(X,b) \leftarrow q(X) & & \\
 & q(a) \leftarrow & q(b) \leftarrow & r(a) \leftarrow & r(b) \leftarrow
\end{array}
$$

The order of the computed answer substitutions is not preserved by this unfolding step. Indeed, we have

$$
\begin{aligned}
&\mathbf{Prolog}[P_0, \leftarrow p(X,Y)] = \\
&\quad = \langle \{X/a, Y/a\}, \{X/a, Y/b\}, \{X/b, Y/a\}, \{X/b, Y/b\} \rangle \text{ and} \\
&\mathbf{Prolog}[P_1, \leftarrow p(X,Y)] = \\
&\quad = \langle \{X/a, Y/a\}, \{X/b, Y/a\}, \{X/a, Y/b\}, \{X/b, Y/b\} \rangle.
\end{aligned}
$$

Example 4.4.15. By unfolding w.r.t. r the first clause of the following program:

$$
P_0: \quad p \leftarrow q, r \qquad q \leftarrow \qquad q \leftarrow q \qquad r \leftarrow \mathit{fail} \qquad r \leftarrow
$$

we get

$$
P_1: \quad p \leftarrow q, \mathit{fail} \qquad p \leftarrow q \qquad q \leftarrow \qquad q \leftarrow q \qquad r \leftarrow \mathit{fail} \qquad r \leftarrow
$$

We have that $\mathbf{Prolog}[P_0, \leftarrow p] \neq \mathbf{Prolog}[P_1, \leftarrow p]$, because

$$
\mathbf{Prolog}[P_0, \leftarrow p] = \langle \{\}, \{\}, \ldots \rangle \text{ and } \mathbf{Prolog}[P_1, \leftarrow p] = \langle \bot \rangle.
$$

Example 4.4.16. By unfolding w.r.t. $r(X)$ the first clause of the program

$$
P_0: \quad p \leftarrow q(X), r(X) \qquad q(a) \leftarrow q(a) \qquad r(b) \leftarrow
$$

we get

$$
P_1: \quad p \leftarrow q(b) \qquad q(a) \leftarrow q(a) \qquad r(b) \leftarrow
$$

We have that $\mathbf{Prolog}[P_0, \leftarrow p] \neq \mathbf{Prolog}[P_1, \leftarrow p]$, because

$$
\mathbf{Prolog}[P_0, \leftarrow p] = \langle \bot \rangle \text{ and } \mathbf{Prolog}[P_1, \leftarrow p] = \langle \rangle.
$$

We also have that the use of the folding rule does not necessarily preserve the **Prolog** semantics. In order to overcome this inconvenience, several researchers have proposed restricted versions of the unfolding and folding rules [Proietti and Pettorossi, 1991; Sahlin, 1993]. The following two instances of the unfolding rule can be shown to be totally correct w.r.t. **Prolog**.

Rule R1.1 Leftmost Unfolding. The unfolding of a clause C w.r.t. the leftmost atom of its body is said to be a *leftmost unfolding* of C.

Rule R1.2 Single Non-left-propagating Unfolding. The unfolding of a clause $H \leftarrow F, A, G$ w.r.t. the atom A is said to be a *single non-left-propagating* unfolding 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 $(H \leftarrow F)\theta$.

This rule is called *deterministic-non-left-propagating unfolding* in [Pettorossi and Proietti, 1994].

If a folding step is both a single-folding and an in-situ folding, called here *single in-situ folding*, then it is reversible by an application of the single non-left-propagating unfolding rule. By the reversibility lemma 4.1.3 (page 21), each single in-situ folding step is totally correct w.r.t. **Prolog**.

Since **Prolog** is relevant, the definition introduction and definition elimination rules are totally correct w.r.t. **Prolog**.

We also have that the goal replacement rule is partially correct, and by the reversibility lemma and the reversibility of independent goal replacement lemma (page 22) the independent goal replacement is totally correct w.r.t. **Prolog**.

Thus, we can state the following two results which are the analogous of the First and the second correctness theorems w.r.t. **LHM**, **CAS**, and **FF**. In the following Theorems 4.4.17 and 4.4.18 the instances of the goal replacement rule are defined in terms of the notion of goal equivalence w.r.t. **Prolog**.

Their proofs are based on the fact that an application of the leftmost unfolding rule can be viewed as 'a step forward in the computation' using the left-to-right computation rule.

Theorem 4.4.17 (First Correctness Theorem w.r.t. Prolog). *Let $P_0, \ldots, P_n$ be a transformation sequence of definite programs constructed by using the transformation rules: leftmost unfolding, single non-left-propagating unfolding, single in-situ folding, definition introduction, definition elimination, independent goal replacement, and generalization + equality introduction. Then $P_0, \ldots, P_n$ is totally correct w.r.t.* **Prolog**.

Theorem 4.4.18 (Second Correctness Theorem w.r.t. Prolog). *Let $P_0, \ldots, P_n$ be a transformation sequence of definite programs constructed* *by using the following transformation rules: leftmost unfolding, single non-left-propagating unfolding, single-folding, definition introduction, definition elimination, basic goal replacement, and generalization + equality introduction. Suppose that no single-folding step is performed after a definition elimination step. Suppose also that when we apply the single-folding rule to a clause, say C, using a clause, say D, the following conditions hold:*

- either *D belongs to P_0* or *D has been introduced by the definition rule,*
- *$hd(D)$ has a top predicate, and*
- either *$hd(C)$ has an intermediate predicate*
  or *$hd(C)$ has a top predicate and C has been derived from a clause, say E, by first performing a leftmost unfolding step w.r.t. an atom in $bd(E)$ with an intermediate predicate and then performing zero or more transformation steps on a clause derived from the unfolding of E.*

Then $P_0, \ldots, P_n$ is totally correct w.r.t. the semantics **Prolog**.

The following example shows that in the above Theorem 4.4.18 we cannot replace 'leftmost unfolding step' by 'single non-left-propagating unfolding step'.

Example 4.4.19. Let us consider the following initial program:

P_0: $\quad p \leftarrow q, r \qquad q \leftarrow \mathit{fail} \qquad r \leftarrow r, q$

We assume that p is a top predicate and q, r, and fail are intermediate predicates. By single non-left-propagating unfolding of $p \leftarrow q, r$ w.r.t. r, we get the program

P_1: $\quad p \leftarrow q, r, q \qquad q \leftarrow \mathit{fail} \qquad r \leftarrow r, q$

If we now fold the first clause of P_1 using the first clause of P_0, we get

P_2: $\quad p \leftarrow p, q \qquad q \leftarrow \mathit{fail} \qquad r \leftarrow r, q$

P_2 is not equivalent to P_0 w.r.t. **Prolog**, because

$$\mathbf{Prolog}[P_0, \leftarrow p] = \langle\rangle \text{ and } \mathbf{Prolog}[P_2, \leftarrow p] = \langle\perp\rangle.$$

In this chapter we have considered only the case of pure Prolog, where the SLD-resolution steps have no side-effects. Properties which are preserved by unfold/fold rules when transforming Prolog programs with side-effects, including cuts, are described in [Deville, 1990; Sahlin, 1993; Prestwich, 1993b; Leuschel, 1994a].

4.5 Correctness Results 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 transformation rules which preserve the various semantics 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 computed answer substitution semantics and the finite failure semantics [Gardner and Shepherdson, 1991; Seki, 1991], the Clark's completion [Gardner and Shepherdson, 1991], the Fitting's and Kunen's three-valued extensions of Clark's completion [Fitting, 1985; Kunen, 1987; Bossi *et al.*, 1992b; Sato, 1992; Bossi and Etalle, 1994a], the perfect model semantics [Przymusinsky, 1987; Maher, 1993; Seki, 1991], the stable model semantics [Gelfond and Lifschitz, 1988; Maher, 1990; Seki, 1990], and the well-founded model semantics [Van Gelder *et al.*, 1989; Maher, 1990; Seki, 1990; Seki, 1993].

A uniform approach for proving the correctness of the unfold/fold transformation rules w.r.t. various non-monotonic semantics of logic programs (including stable model, well-founded model, and perfect model semantics) has been proposed by Aravindan and Dung [1995], who showed that the unfolding and some variants of folding transformations preserve the so-called *semantic kernel* of a normal logic program.

We will report here only on the results concerning the following three semantics [Lloyd, 1987]: i) computed answer substitutions, ii) finite failure, and iii) Clark's completion.

The computed answer substitution semantics for normal programs is a function

$$\mathbf{CASNF}\colon \mathbf{P} \times \mathbf{Q} \to (\mathcal{P}(\mathbf{Subst}), \leq)$$

where $\mathbf{P}$ is the set of normal programs, $\mathbf{Q}$ is the set of normal queries, and $(\mathcal{P}(\mathbf{Subst}), \leq)$ is the powerset of the set of substitutions ordered by set inclusion. (The suffix '**NF**' stands for 'negation as failure'.) We define **CASNF** as follows:

$$\mathbf{CASNF}[P,Q] = \{\theta \mid \text{there exists an SLDNF-refutation of } Q \text{ in } P \\ \text{with computed answer substitution } \theta\}.$$

CASNF is a relevant semantics.

For the correctness of a transformation sequence w.r.t. **CASNF** there are results which are analogous to the ones for **CAS**. In particular, the statement of the first correctness theorem w.r.t. **CAS** (page 33) is valid also for **CASNF** if we replace 'definite programs' with 'normal programs' and **CAS** with **CASNF**.

However, the first correctness theorem w.r.t. **CAS** (and the new version for **CASNF**) does not ensure the correctness of transformation sequences which also include folding steps different from in-situ foldings.

If we want to perform transformation steps which are not applications of the in-situ folding rule, and still ensure their correctness, we may use Theorem 4.5.1 below, which combines the second correctness theorems w.r.t. **CAS** and **FF**.

Following [Seki, 1991] in the hypotheses of Theorem 4.5.1 we assume that the programs are *stratified*.

We recall that a program is stratified iff for every program clause $p(\ldots) \leftarrow B$ and for every negative literal $\neg q(\ldots)$ in B, we have that q does not depend on p.

Theorem 4.5.1 (Second Correctness Theorem w.r.t. CASNF). *Let $P_0, \ldots, P_n$ be a transformation sequence of stratified normal programs constructed by using the following transformation rules: unfolding, single-folding, definition introduction, definition elimination, goal rearrangement, deletion of ground duplicate goals, basic goal replacement, clause rearrangement, deletion of duplicate clauses, deletion of clauses with finitely failed body, and generalization + equality introduction. Suppose that no single-folding step is performed after a definition elimination step. Suppose also that when we apply the single-folding rule to a clause, say C, using a clause, say D, the following conditions hold:*

- either *D belongs to P_0* or *D has been introduced by the definition rule,*
- *$hd(D)$ has a top predicate, and*
- either *$hd(C)$ has an intermediate predicate*
  or *$hd(C)$ has a top predicate and* each *atom of $bd(C)$ w.r.t. which the single-folding step is performed, has been derived in a previous transformation step by unfolding a clause, say E, w.r.t. an atom with an intermediate predicate and then performing zero or more transformation steps on a clause derived from the unfolding of E.*

Then $P_0, \ldots, P_n$ is totally correct w.r.t. the semantics **CASNF**.

The finite failure semantics for normal programs, denoted **FF** as in the case of definite programs, has the same domain and codomain of **CASNF**. We define **FF** as follows:

$$\mathbf{FF}[P, Q] = \{\theta \mid \text{there exists a finitely failed SLDNF-tree for } Q\theta \text{ in } P\}.$$

For normal programs we may state some correctness results which are analogous to those stated in the case of definite programs. Indeed, the first and second correctness theorems w.r.t. **FF** (pages 34 and 36) continue to hold if we replace 'definite programs' with 'stratified normal programs' and we assume **FF** to be defined with reference to the set of normal programs and normal queries, instead of the set of definite programs and definite queries.

Now we consider the Clark's completion semantics.

Let **P** and **Q** be the sets of all normal programs and normal queries, respectively. For any program $P \in \mathbf{P}$, let $Comp(P)$ be the set of first order formulas, called the *completion* of P, and constructed as indicated in [Lloyd, 1987], except that $Comp(P)$ also contains a formula $\forall X \neg p(X)$ for every predicate p in the language **L** not occurring in P.

The Clark's completion semantics is defined by the function

$$\mathbf{COMP}: \mathbf{P} \times \mathbf{Q} \rightarrow (\mathcal{P}(\mathbf{Subst}), \leq)$$

such that

$$\mathbf{COMP}[P, \leftarrow G] = \{\theta \mid Comp(P) \models \forall(G\theta)\}$$

where $\forall C$ denotes the universal closure of a conjunction C of literals, and similarly to the case of **LHM**, we identify any program P and any goal G with the corresponding logical formulas.

As already mentioned, **COMP** is not a relevant semantics. Thus, we cannot use the relevance lemma (page 20), and indeed, the definition introduction rule and the definition elimination rule are *not* totally correct w.r.t. **COMP**. To see this, let us consider the case where we add to a program P_1 whose completion is consistent, a new clause of form $newp(X) \leftarrow \neg newp(X)$. We get a new program, say P_2, whose completion contains the formula $newp(X) \leftrightarrow \neg newp(X)$ and it is inconsistent. Thus, $\mathbf{COMP}[P_1, Q] \neq \mathbf{COMP}[P_2, Q]$ for $Q = \leftarrow newp(X)$.

However, it can be shown that if the definition introductions are only non-recursive definition introductions then any step of non-recursive definition introduction or definition elimination is totally correct w.r.t. **COMP**.

The partial correctness w.r.t. **COMP** of the unfolding rule R1 and the folding rule R2 can easily be established, as illustrated by the following example.

Example 4.5.2. Let us consider the program

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

whose completion is

$$\begin{array}{llllll} Comp(P_0): & p \leftrightarrow q \wedge \neg r & q \leftrightarrow (s \wedge t) \vee (s \wedge u) & & & \\ & v \leftrightarrow t \vee u & s & u & \neg r & \neg t \end{array}$$

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

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

whose completion is

$$\begin{array}{llllll} Comp(P_1): & p \leftrightarrow (s \wedge t \wedge \neg r) \vee (s \wedge u \wedge \neg r) & q \leftrightarrow (s \wedge t) \vee (s \wedge u) & & & \\ & v \leftrightarrow t \vee u & s & u & \neg r & \neg t \end{array}$$

$Comp(P_1)$ 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

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

$$v \leftarrow t \qquad v \leftarrow u \qquad s \leftarrow \qquad u \leftarrow$$

whose completion is

$$\begin{array}{lllll} Comp(P_2): & p \leftrightarrow s \wedge v \wedge \neg r & & q \leftrightarrow (s \wedge t) \vee (s \wedge u) & \\ & v \leftrightarrow t \vee u \quad s & u & \neg r & \neg t \end{array}$$

$Comp(P_2)$ can be obtained from $Comp(P_1)$ 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 substituting v for $t \vee u$. Since $v \leftrightarrow t \vee u$ holds in $Comp(P_1)$, we have that $Comp(P_2)$ is a logical consequence of $Comp(P_1)$.

In general, given a transformation sequence $P_0, \ldots, P_k$, if a program P_{k+1} can be obtained from a program P_k by an unfolding step using P_j, with $0 \leq j \leq k$, and both $Comp(P_j)$ and $Comp(P_k)$ are logical consequences of $Comp(P_0)$, then $Comp(P_{k+1})$ can be obtained from $Comp(P_k)$ by one or more replacements of a formula F by a formula G such that $F \leftrightarrow G$ is a logical consequence of $Comp(P_j)$. Thus, also $Comp(P_{k+1})$ is a logical consequence of $Comp(P_0)$.

A similar fact holds if P_{k+1} can be obtained from P_k by applying the folding rule, or the goal replacement rule, or the clause replacement rules. Thus, we have the following result.

Theorem 4.5.3 (Partial Correctness w.r.t. COMP). *Let $P_0, \ldots, P_n$ be a transformation sequence of normal programs constructed by using the following rules: unfolding, folding, non-recursive definition introduction, definition elimination, goal replacement, and clause replacement. If no folding step is performed after a definition elimination step, then $P_0, \ldots, P_n$ is partially correct w.r.t. the semantics* **COMP**.

Unfortunately, the unfolding rule is not totally correct w.r.t. **COMP** as shown by the following example adapted from [Maher, 1987].

Example 4.5.4. 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)

$$\begin{array}{ll} Comp(P_0): & \forall X(p(X) \leftrightarrow q(X) \vee \neg q(succ(X))) \\ & \forall X(q(X) \leftrightarrow q(succ(X))) \end{array}$$

together with the axioms of Clark's equality theory [Lloyd, 1987; Apt, 1990]. By unfolding the last clause of P_0 we get

$$\begin{array}{ll} P_1: & p(X) \leftarrow q(X) \qquad p(X) \leftarrow \neg q(succ(X)) \\ & q(X) \leftarrow q(succ(succ(X))) \end{array}$$

whose completion is (equivalent to)

$$Comp(P_1): \quad \forall X(p(X) \leftrightarrow q(X) \vee \neg q(succ(X)))$$

$$\forall X(q(X) \leftrightarrow q(succ(succ(X))))$$

together with the axioms of Clark's equality theory.

We have that $\forall X p(X)$ is a logical consequence of $Comp(P_0)$. On the other hand, $\forall X p(X)$ is not a logical consequence of $Comp(P_1)$. Indeed, let us consider the interpretation I whose domain 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 $Comp(P_1)$, whereas 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. **COMP**, as indicated in the following definition.

Rule R1.3 In-Situ Unfolding. The unfolding of a clause C in a program P_k w.r.t. an atom A using a program P_j is said to be an *in-situ unfolding* iff $P_j = P_k$, and $hd(C)$ is not unifiable with A.

If program P_{k+1} is derived from program P_k by performing an in-situ unfolding step, then the transformation sequence P_k, P_{k+1} is reversible by in-situ folding. Thus, by the reversibility lemma 4.1.3 (page 21) and the partial correctness theorem w.r.t. **COMP** (page 44), we have that every in-situ unfolding is totally correct w.r.t. **COMP**.

By similar arguments we can show the total correctness of any in-situ folding and independent goal replacement step, because, as the reader may verify, the independent goal replacement rule is reversible even though **COMP** is not a relevant semantics (and, thus, the reversibility of independent goal replacement lemma cannot be applied).

It is also straightforward to show the total correctness of any clause replacement step. Thus, we have the following result.

Theorem 4.5.5 (First Correctness Theorem w.r.t. COMP). *Let $P_0, \ldots, P_n$ be a transformation sequence of normal programs constructed by using the transformation rules: in-situ unfolding, in-situ folding, non-recursive definition introduction, definition elimination, independent goal replacement, and clause replacement. Then $P_0, \ldots, P_n$ is totally correct w.r.t.* **COMP**.

We end this section by showing, through the following example, that the hypotheses of the second correctness theorem w.r.t. **CASNF** are not sufficient to ensure the correctness of folding w.r.t. **COMP**.

Example 4.5.6. Let us consider the following transformation sequence:

P_0: $p \leftarrow q \quad q \leftarrow q \quad r \leftarrow p \quad r \leftarrow \neg q$
(by in-situ unfolding of $p \leftarrow q$)
P_1: $p \leftarrow q \quad q \leftarrow q \quad r \leftarrow p \quad r \leftarrow \neg q$
(by single-folding of $p \leftarrow q$)
P_2: $p \leftarrow p \quad q \leftarrow q \quad r \leftarrow p \quad r \leftarrow \neg q$

where we assume that p and r are top predicates and q is an intermediate one.

By the second correctness theorem w.r.t. **CASNF** we have that P_0 and P_2 (and also P_1) are equivalent w.r.t. **CASNF**. Let us now consider the completions of P_0 and P_2, respectively:

$Comp(P_0)$:	$p \leftrightarrow q$	$q \leftrightarrow q$	$r \leftrightarrow p \vee \neg q$
$Comp(P_2)$:	$p \leftrightarrow p$	$q \leftrightarrow q$	$r \leftrightarrow p \vee \neg q$

We have that r is a logical consequence of $Comp(P_0)$. On the contrary, r is not a logical consequence of $Comp(P_2)$. Indeed, the interpretation where p is false, q is true, and r is false, is a model of $Comp(P_2)$, but not of r. Thus, P_0 and P_2 are not equivalent w.r.t. **COMP**.

It should be noted that in the above Example 4.5.6, 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.

For the case where unfolding and folding are not in-situ, the reader may find various correctness results w.r.t. the above mentioned semantics in [Seki, 1990; Seki, 1991; Sato, 1992; Seki, 1993; Aravindan and Dung, 1995; Bossi and Etalle, 1994a].

5 Strategies for Transforming Logic Programs

The transformation process should be directed by some metarules, which we call *strategies*, because, as we have seen in Section 3, the transformation rules have inverses, and thus, they allow the final program of a transformation sequence to be equal to the initial program. 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 crucial problems of the transformation methodology, that is, the use of the definition rule for the introduction of the so-called *eureka predicates*.

In [Feather, 1987; Partsch, 1990; Deville, 1990; Pettorossi and Proietti, 1994; Pettorossi and Proietti, 1996] one can find a treatment of the transformation strategies both for functional and logic programs.

For reasons of simplicity, when we describe the various transformation strategies we consider only the case of definite programs with the least Herbrand model semantics **LHM**.

We assume that the following rules are available: unfolding (rule R1, page 12), in-situ folding (rule R2.1, page 21), single-folding (rule R2.2, page 30), definition introduction (rule R3, page 15), definition elimination (rule R4, page 16), independent goal replacement (rule R5.3, page 22), goal rearrangement (rule R5.4, page 28), deletion of duplicate goals (rule R5.5, page 28), basic goal replacement (rule R5.6, page 31), and clause replacement (rule R6, page 18).

The correctness of those rules w.r.t. **LHM** is ensured by the first and the second correctness theorems w.r.t. **LHM** (pages 29 and 31).

In order to simplify our presentation, sometimes we will not mention the use of goal rearrangement and deletion of duplicate goals.

As already pointed out, from the correctness of the goal rearrangement, deletion of duplicate goals, clause rearrangement (rule R6.1, page 18), and deletion of duplicate clauses (rule derived from rule R6.2, page 18) it follows that the concatenation of sequences of literals and the concatenation of sequences of clauses are associative, commutative, and idempotent. Therefore, when dealing with goals or programs, we will feel free to use set-theoretic notations, such as $\{\ldots\}$ and $\cup$, instead of sequence-theoretic notations.

Before giving the technical details concerning the transformation strategies we would like to present, we now informally explain the main ideas which justify their use.

Suppose that, 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, say C, is inefficient, because it generates some redundant computations. For example, by analyzing the initial program P_0 given in the Palindrome example of Section 2 (page 6), we may discover that the evaluation of the body of the clause

3. $pal([H|T]) \leftarrow append(Y, [H], T), pal(Y)$

is inefficient because it determines multiple traversals of the list Y.

In order to improve the performance of 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 can be formalized as an application of the so-called *tupling strategy* (page 50). Sometimes we also need a simultaneous application of the *generalization strategy* (page 51). Then we fold clause C w.r.t. the goal $A_1, \ldots, A_n$ by using clause N, and we 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, thereby avoiding failures at run-time, and we may delete duplicate atoms, thereby avoiding repeated computations.

Looking again at the Palindrome example of Section 2, we see that by applying the tupling and generalization strategies, we have introduced the clause

6. $newp(L, T) \leftarrow append(Y, L, T), pal(Y)$

and we have used this clause 6 for folding clause 3. Then we have unfolded clause 6 w.r.t. the atoms $pal(\ldots)$ and $append(\ldots)$ and we have derived the clauses

10. $newp(L, L) \leftarrow$
11. $newp(L, [X|L]) \leftarrow$
13. $newp(L, [H|U]) \leftarrow append(R, [H|L], U), pal(R)$

These clauses for *newp*, together with clauses 1, 2, and 3f for *pal* and clauses 4 and 5 for *append*, avoid multiple traversals of the input list, but as the reader may verify, only when that list has at most three elements.

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 list will be avoided for lists of any length.

This recursive definition can often be obtained by performing a folding step using the clause initially introduced by tupling. In our palindrome example that clause is clause 6. By folding clause 13 using clause 6, we get

13f. $newp(L, [H|U]) \leftarrow newp([H|L], U)$

This recursive clause, together with clauses 10 and 11, indeed provides a recursive definition of *newp*, and it avoids multiple traversals of any input list.

In some unfortunate cases we may be unable to perform the desired folding steps for deriving the recursive definition of the predicates introduced by the initial applications of the tupling and generalization strategies. 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 strategy* and the already mentioned *generalization strategy*, both described in Section 5.1 below.

In [Darlington, 1981] the expression 'forced folding' is introduced to refer to the need of performing the folding steps for improving program efficiency. This *need for folding* 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 appropriate strategy to apply at each step of the derivation.

The need for folding in program transformation is related to similar ideas in the field of automated theorem proving [Boyer and Moore, 1975] and program synthesis [Deville and Lau, 1994], where tactics for inductive proofs and inductive synthesis are driven by the need of applying suitable inductive hypotheses.

5.1 Basic 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 and Darlington, 1977].

The tupling strategy was formally defined in [Pettorossi, 1977] 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 and Pettorossi, 1990] for indicating a strategy which derives a new predicate definition when a goal is recurrently generated during program transformation. This strategy is present in various forms in a number of different transformation techniques, such as the above mentioned tupling, supercompilation [Turchin, 1986], compiling control [Bruynooghe *et al.*, 1989], as well as various techniques for partial evaluation (see Section 6).

Finally, the generalization strategy has its origin in the automated theorem proving context [Boyer and Moore, 1975], where it is used to generate a new generalized conjecture allowing the application of an inductive hypothesis.

The tupling, loop absorption, and generalization strategies are used in this chapter as building blocks to describe a 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 transforming a given clause by performing unfolding and basic goal replacement steps. This notion is also related to the one of *symbolic trace tree* of [Bruynooghe *et al.*, 1989], where, however, the basic goal replacement rule is not taken into account.

Definition 5.1.1 (Unfolding Tree). Let P be a program and C a clause. An *unfolding tree* for $\langle P, C\rangle$ is a (finite or infinite) labeled tree such that

- the root is labeled by the clause C,
- if M is a node labeled by a clause D then
  either M has no sons,
  or M has $n(\geq 1)$ sons labeled 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 labeled by a clause obtained by basic 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 $\langle P, C\rangle$ is nondeterministic. 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 a node should have a son-node constructed by unfolding, we need to choose the atom w.r.t. which that unfolding step should be performed. Those choices can be realized by using a function defined as follows.

Definition 5.1.2 (Unfolding Selection Rule). 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 we should 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.

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

S1. Tupling Strategy. Let us consider a clause C of the form

$$H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$$

with $m \geq 1$ and $n \geq 0$. We introduce a new predicate *newp* defined by a clause T of the form

$$newp(X_1, \ldots, X_k) \leftarrow A_1, \ldots, A_m$$

where the arguments $X_1, \ldots, X_k$ are the elements of $vars(A_1, \ldots, A_m) \cap vars(H, B_1, \ldots, B_n)$. We then look for the recursive definition of the eureka predicate *newp* by performing some unfolding and basic goal replacement steps followed by suitable folding steps using clause T. We finally fold clause C w.r.t. the atoms $A_1, \ldots, A_m$ using clause T.

The tupling strategy is often applied when $A_1, \ldots, A_m$ 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 $A_1, \ldots, A_m$. By tupling we can also avoid multiple visits of data structures and the construction of intermediate bindings.

S2. Loop Absorption Strategy. Suppose that a non-root clause C in an unfolding tree has the form

$$H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$$

with $m \geq 1$ and $n \geq 0$, and the body of a descendant D of C contains (as a subsequence of atoms) the instance $(A_1, \ldots, A_m)\theta$ of $A_1, \ldots, A_m$ via 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 goal rearrangement and deletion of duplicate goals, to $B_1, \ldots, B_n$. We introduce a new predicate defined by the following clause A:

$$newp(X_1, \ldots, X_k) \leftarrow A_1, \ldots, A_m$$

where $\{X_1, \ldots, X_k\}$ is the minimum subset of $vars(A_1, \ldots, A_m)$ which is necessary to perform a single-folding step on C and a single-folding step on

D, both using a clause whose body is $A_1, \ldots, A_m$. This minimum subset is determined by condition 3 for the applicability of the single-folding rule (page 30). We fold clause C using clause A and we then look for the recursive definition of the eureka predicate *newp*. This recursive definition can be found starting from clause A by first performing the unfolding steps and the basic goal replacement steps corresponding to the ones which lead from clause C to clause D, and then folding using clause A again.

S3. Generalization Strategy. Let us consider a clause C of the form

$$H \leftarrow A_1, \ldots, A_m,\, B_1, \ldots, B_n$$

with $m \geq 1$ and $n \geq 0$. We introduce a new predicate *genp* defined by a clause G of the form

$$genp(X_1, \ldots, X_k) \leftarrow GenA_1, \ldots, GenA_m$$

where $(GenA_1, \ldots, GenA_m)\,\theta = (A_1, \ldots, A_m)$, for some 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 obtained by matching clause C against one of its descendants, say D, in the unfolding tree generated during program transformation (see Example 5.2.2, page 57). In particular, we 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 by applying no transformation rule, except for goal rearrangement and deletion of duplicate goals, to $B_1, \ldots, B_n$,
2. for $i = 1, \ldots, m$, E_i has the same predicate symbol of A_i,
3. $E_1, \ldots, E_m$ is not an instance of $A_1, \ldots, A_m$,
4. 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
5. $\{X_1, \ldots, X_k\}$ 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 $GenA_1, \ldots, GenA_m$.

5.2 Techniques Which Use Basic Strategies

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

5.2.1 Compiling Control

One of the advantages of logic programming over conventional imperative programming languages is that by writing a logic program one may easily separate the 'logic' part of an algorithm from the 'control' part [Kowalski, 1979]. 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 naive Prolog strategy for controlling SLD-resolution (see Section 4.4.4) does not always give us the desired level of efficiency, because the search space generated by the nondeterministic evaluation of a program is explored without using any information about the program. Much work has been done in the direction of improving the control strategy of logic languages (see, for instance, [Bruynooghe and Pereira, 1984; Naish, 1985]).

We consider here a transformation technique, called *compiling control* [Bruynooghe *et al.*, 1989], which follows a different approach. Instead of enhancing the naive Prolog evaluator using a clever (and often more complex) control strategy, we transform the given program so that the derived program behaves using the naive evaluator as the given program behaves using an enhanced evaluator.

The main advantage of the compiling control approach is that we can use relatively simple evaluators which have small and efficient compilers.

The compiling control technique can also be used to 'compile' bottom-up and mixed evaluation strategies [De Schreye *et al.*, 1991; Sato and Tamaki, 1988] as well as lazy evaluation and coroutining [Narain, 1986]. Here we will only show the use of compiling control 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 the compiling control technique 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, 1978; Bird, 1984]. Some other transformation techniques for improving generate-and-test logic programs which are closely related to the compiling control technique and the filter promotion strategy, can be found in [Seki and Furukawa, 1987; Brough and Hogger, 1991; Träff and Prestwich, 1992].

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. **LHM** and behaves using the left-to-right computation rule as P_1 does using the rule C [Bruynooghe *et al.*, 1989; De Schreye and Bruynooghe, 1989].

By 'equal behavior' 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 (thus, we disregard the goals in the nodes),

ii) we replace zero or more 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, and

iii) we replace zero or more subtrees of T_1 whose roots have an outgoing arc only, and this arc is labeled by the identity substitution, as follows: every subtree is replaced by the subtree below the arc outgoing from its root.

We can formulate basic forms of compiling control in terms of the program transformation methodology as we now indicate. Given a program P_1, a set Q of queries, and a computation rule C, the program P_2 obtained by the compiling control technique, can be derived by first constructing a suitable unfolding tree, say T, using the unfolding rule only, and then applying the loop absorption strategy. Some more complex forms of compiling control require the use of generalization strategies possibly more powerful than the strategy S3 (page 51).

Without loss of generality, we assume that every query in Q is of the form $\leftarrow q(\ldots)$ and in P_1 there exists only one clause, say R, whose head predicate is q. (Indeed, we can use the definition rule to comply with this condition.) The root clause of T is R and the nodes of T are generated by using a 'suitable' u-selection rule which simulates the evaluation of an 'abstract query' representing the whole set Q, by using the computation rule C. We will not give here the formal notions of 'simulation' and 'abstraction' which may be used to effectively construct the unfolding tree T from the given P_1, Q, and C. We refer to [Cousot and Cousot, 1977] for a formalization of the techniques of *abstract interpretation*, and to [De Schreye and Bruynooghe, 1989] for a method based on abstract interpretation for generating the tree T in a semi-automatic way.

We now give an example of application of the compiling control technique by using the tupling and the loop absorption strategies.

Example 5.2.1. [Common Subsequences] Let sequences be represented as lists of items. We assume that for any given sequence X and Y, $subseq(X,Y)$ 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 verifying that X is a subsequence of Z.

1. $csub(X,Y,Z) \leftarrow subseq(X,Y), subseq(X,Z)$
2. $subseq([\,],X) \leftarrow$
3. $subseq([A|X],[A|Y]) \leftarrow subseq(X,Y)$

4. $subseq([A|X],[B|Y]) \leftarrow subseq([A|X],Y)$

where for any sequence X, Y, and Z, $csub$(X,Y,Z) holds iff X is a subsequence of Y and X is also a subsequence of Z.

Let Q be the set of queries $\{\leftarrow csub(X,s,t) \mid s$ and t are ground lists and X is an unbound variable$\}$ and the computation rule C be the following one:

if the goal is '$subseq(w,x), subseq(y,z)$' and w is a proper subterm of y
then C selects for resolution the atom $subseq(y,z)$
else C selects for resolution the leftmost atom in the goal.

We may expect that the evaluation of a query in Q using the computation rule C, is more efficient than the evaluation of that query using the standard left-to-right Prolog computation rule, because the second occurrence of $subseq(\ldots)$ in a goal of the form '$subseq(\ldots), subseq(\ldots)$' is selected as soon as it gets suitably instantiated by the evaluation of the first occurrence. Thus, using the computation rule C, it may be the case that the evaluation of a goal of the form '$subseq(\ldots), subseq(\ldots)$' fails even if the first occurrence of $subseq(\ldots)$ has not been completely evaluated.

We now construct an unfolding tree T for $\langle Csub$, clause 1$\rangle$ by using the following u-selection rule U_C which simulates the computation rule C:

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 U_C selects for unfolding the atom $subseq(y,z)$
else U_C selects for unfolding the leftmost atom in the body.

Clause 1 is the only clause whose head unifies with $csub(X,s,t)$.

A finite portion of T is depicted in Fig. 2, where a dashed arrow from clause M to clause N means that the body of M is an instance of the body of N.

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

11. $newcsub(A,X,Y,Z) \leftarrow subseq(X,Y), subseq([A|X],Z)$

and we fold clause 6, whereby obtaining

6f. $csub([A|X],[A|Y],Z) \leftarrow newcsub(A,X,Y,Z)$

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. $csub([A|X],[B|Y],Z) \leftarrow csub([A|X],Y,Z)$

We now have to look for the recursive definition of the predicate *newcsub*. Starting from clause 11, we perform the unfolding step corresponding to the one which leads from clause 6 to clauses 9 and 10. We get the clauses

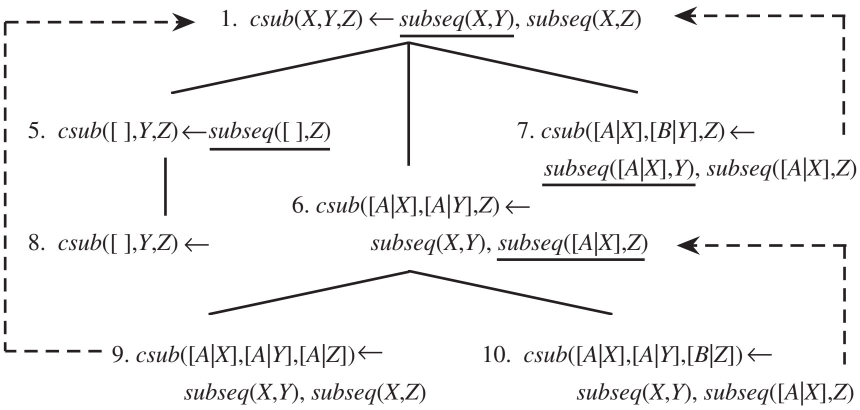

Fig. 2. An unfolding tree for $\langle Csub$, clause 1$\rangle$ using the computation rule U_C. We have underlined the atoms selected for unfolding.

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

and by folding we get

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

The final program is made out of the following clauses:

8. $csub([\,], Y, Z) \leftarrow$
6f. $csub([A|X], [A|Y], Z) \leftarrow newcsub(A, X, Y, Z)$
7f. $csub([A|X], [B|Y], Z) \leftarrow csub([A|X], Y, Z)$
12f. $newcsub(A, X, Y, [A|Z]) \leftarrow csub(X, Y, Z)$
13f. $newcsub(A, X, Y, [B|Z]) \leftarrow newcsub(A, X, Y, Z)$

The correctness of the above transformation can easily be proved by applying the second correctness theorem w.r.t. **LHM** with the assumption that *newcsub* and *csub* are top predicates and *subseq* is an intermediate predicate. In particular, the single-folding step which generates clause 6f from clause 6 using clause 11, satisfies the conditions of that theorem, because: (i) clause 11 has been introduced by the definition rule, (ii) the head of clause 11 has a top predicate, and (iii) clause 6 has been derived from clause 1 by unfolding w.r.t. the intermediate atom $subseq(X, Y)$. Similar conditions ensure the correctness of the other single-folding steps.

Let us now compare the SLD-tree, say T_1, for *Csub*, a query of form $\leftarrow csub(X, s, t)$ in Q, and the computation rule C, with the SLD-tree, say

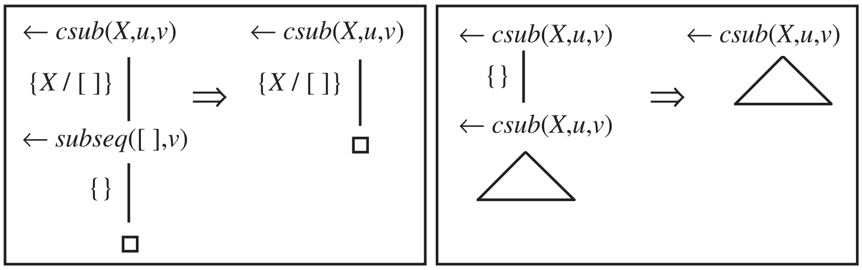

Fig. 3. Tree rewritings for the SLD-tree T_1.

T_2, for the final program, the query $\leftarrow csub(X, s, t)$, and the left-to-right computation rule.

As the reader may verify, the tree T_2 can be obtained from the tree T_1 by first replacing every query of the form '$\leftarrow subseq(x, b), subseq(x, c)$' by the query '$\leftarrow csub(x, b, c)$' and every query of the form '$\leftarrow subseq(x, b)$, $subseq([a|x], c)$' by the query '$\leftarrow newcsub(a, x, b, c)$', and then by performing on the derived tree the rewritings shown in Fig. 3 for any unbound variable X and ground lists u and v.

5.2.2 Composing Programs

A popular style of programming, which can be called *compositional*, consists in decomposing a given goal in easier subgoals, then writing program modules which solve these subgoals, and finally, composing the various program modules 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.

However, 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, the memorization, and the destruction of that binding are 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, for instance, 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 subtasks in which a given task 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 5.2.1, two approaches have been followed:

1. the improvement of the evaluator by using, for instance, garbage collection, memoing, 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, standard evaluator.

In the imperative and functional cases, various transformation methods have been proposed, such as, for instance: *finite differencing* [Paige and Koenig, 1982], *composition* or *deforestation* [Feather, 1982; Wadler, 1990], and *tupling* [Pettorossi, 1977]. (See also [Feather, 1987; Partsch, 1990; Pettorossi and Proietti, 1996] for surveys.)

For logic programs two main methods have been considered: *loop fusion* [Debray, 1988] and *unnecessary variable elimination* [Proietti and Pettorossi, 1995]. The aim of loop fusion is to transform a program which computes a predicate defined by the composition of two independent recursive predicates, into a program where the computations corresponding to these two predicates are performed by one predicate only. Using loop fusion one may avoid the multiple traversal of data structures and the construction of intermediate data structures.

The method presented in [Proietti and Pettorossi, 1995] 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 5.1.

Example 5.2.2. [Maximal Number Deletion] Suppose that we are given a list Xs of positive numbers. We want to delete from Xs every occurrence of its maximal number, say M. This can be done by first computing the value of M, and then visiting again Xs for deleting each occurrence of M. A program which realizes this algorithm is as follows:

1. $deletemax(Xs, Ys) \leftarrow maximal(Xs, M), delete(M, Xs, Ys)$
2. $maximal([\,], 0) \leftarrow$
3. $maximal([X|Xs], M) \leftarrow maximal(Xs, N), max(N, X, M)$
4. $delete(M, [\,], [\,]) \leftarrow$
5. $delete(M, [M|Xs], Ys) \leftarrow delete(M, Xs, Ys)$
6. $delete(M, [X|Xs], [X|Ys]) \leftarrow M \neq X, delete(M, Xs, Ys)$

where, for any positive number A, B, and M, $max(A, B, M)$ holds iff M is the maximum of A and B.

We would like to derive a program which traverses the list Xs once only. This could be done by applying the loop fusion method and obtaining a new program where the computations corresponding to *maximal* and *delete* are performed by one predicate only. A similar result can be achieved by eliminating the shared variables whose bindings are lists, and in particular, the variable Xs in clause 1.

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

7. $deletemax([\,], [\,]) \leftarrow$
8. $deletemax([M|Xs], Ys) \leftarrow maximal(Xs, N), max(N, M, M),$
   $delete(M, Xs, Ys)$
9. $deletemax([X|Xs], [X|Ys]) \leftarrow maximal(Xs, N), max(N, X, M),$
   $M \neq X, delete(M, Xs, Ys)$

As suggested by the tupling strategy, we may now look for a fold of the goal '$maximal(Xs, N)$, $delete(M, Xs, Ys)$' using clause 1. Unfortunately, no matching is possible because this goal is not an instance of '$maximal(Xs, M)$, $delete(M, Xs, Ys)$'. Thus, we apply the generalization strategy and we introduce the following clause:

10. $gen(Xs, P, Q, Ys) \leftarrow maximal(Xs, P), delete(Q, Xs, Ys)$

whose body is the most specific generalization of the following two goals: '$maximal(Xs, M), delete(M, Xs, Ys)$', which is the body of clause 1, and '$maximal(Xs, N)$, $delete(M, Xs, Ys)$'. By folding clause 1 using clause 10, we get

1f. $deletemax(Xs, Ys) \leftarrow gen(Xs, M, M, Ys)$

We are now left with the problem of finding the recursive definition of the predicate *gen* introduced in clause 10. This is an easy task, because we can perform the unfolding steps corresponding to those leading from clause 1 to clauses 7, 8, and 9, and then we can use clause 10 for folding. After those steps we get the following program:

1f. $deletemax(Xs, Ys) \leftarrow gen(Xs, M, M, Ys)$
11. $gen([\,], 0, Q, [\,]) \leftarrow$
12. $gen([X|Xs], P, X, Ys) \leftarrow gen(Xs, N, X, Ys), max(N, X, P)$
13. $gen([X|Xs], P, Q, [X|Ys]) \leftarrow gen(Xs, N, Q, Ys), Q \neq X,$
$max(N, X, P)$

This program performs the desired list transformation in one visit. Indeed, let us consider a query of the form: $\leftarrow deletemax(l, Ys)$, where l is a ground list and Ys is an unbound variable. During the evaluation of that query, while visiting of the input list l, the predicate $gen(l, P, Q, Ys)$ both computes the maximal number P and deletes all elements of l which are equal to P.

Notice also that no shared variable whose binding is a list, occurs in the clauses defining *deletemax* and *gen*. Thus, we have been successful in eliminating the unnecessary variables at the expense of increasing nondeterminism (see clauses 12 and 13).

Now in order to avoid nondeterminism, we may continue our program derivation by looking for a program in which one avoids the evaluation of the goal $gen(Xs, N, Q, Ys)$ in clause 13, when the body of clause 12 fails after the evaluation of $gen(Xs, N, X, Ys)$.

This can be done by the application of the so called *clause fusion* [Debray and Warren, 1988; Deville, 1990]. This technique can be mimicked by applying our transformation rules as follows.

We first perform two generalization + equality introduction steps followed by a goal rearrangement step and we get

14. $gen([X|Xs], P, Q, Zs) \leftarrow gen(Xs, N, Q, Ys), max(N, X, P),$
$Q = X, Zs = Ys$
15. $gen([X|Xs], P, Q, Zs) \leftarrow gen(Xs, N, Q, Ys), max(N, X, P),$
$Q \neq X, Zs = [X|Ys]$

We then introduce the following definition:

16. $aux(Q, X, Zs, Ys) \leftarrow Q = X, Zs = Ys$
17. $aux(Q, X, Zs, Ys) \leftarrow Q \neq X, Zs = [X|Ys]$

and we fold clauses 14 and 15 by applying rule R2.1, thereby getting

18. $gen([X|Xs], P, Q, Zs) \leftarrow gen(Xs, N, Q, Ys), max(N, X, P),$
$aux(Q, X, Zs, Ys)$

We can then simplify clauses 16 and 17 by unfolding, and we obtain

19. $aux(X, X, Ys, Ys) \leftarrow$

20. $aux(Q, X, [X|Ys], Ys) \leftarrow Q \neq X$

Thus, the final program is made out of the clauses

1f. $deletemax(Xs, Ys) \leftarrow gen(Xs, M, M, Ys)$

11. $gen([\,], 0, Q, [\,]) \leftarrow$

18. $gen([X|Xs], P, Q, Zs) \leftarrow gen(Xs, N, Q, Ys), max(N, X, P),$
$aux(Q, X, Zs, Ys)$

19. $aux(X, X, Ys, Ys) \leftarrow$

20. $aux(Q, X, [X|Ys], Ys) \leftarrow Q \neq X$

5.2.3 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, 1976].

However, it is sometimes difficult to identify the data structures which allow very efficient algorithms before actually writing the programs. Moreover, complex data structures makes it harder to prove program correctness.

Program transformation has been proposed as a methodology for providing appropriate data structures in a *dynamic* way (see Chapter 8 of [Partsch, 1990]): 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 $\langle L, R \rangle$ of lists, denoted by $L \backslash R$, such that there exists a third list X for which the concatenation of X and R is L [Clark and Tärnlund, 1977]. In that case we say that the list X is *represented* by the difference-list $L \backslash R$. 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 methods for transforming programs which use lists into programs which use difference-lists have been proposed in the literature [Hansson and Tärnlund, 1982; Brough and Hogger, 1987; Zhang and Grant, 1988; Marriot and Søndergaard, 1993; Proietti and Pettorossi, 1993].

The problem of deriving programs which manipulate 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 $\mathit{diff\text{-}p}(X, L\backslash R)$ which holds iff $p(X, Y)$ holds and Y is represented by the difference-list $L\backslash R$.

Let us assume that the concatenation of lists is defined in P by means of a predicate $\mathit{append}(X, Y, Z)$ which for any given list X, Y, and Z, holds iff the concatenation of X and Y is Z. Then, the desired transformation can often be achieved by applying the definition rule and introducing the following clause for the predicate *diff-p* [Zhang and Grant, 1988]:

$D.\ \mathit{diff\text{-}p}(X, L\backslash R) \leftarrow p(X, Y), \mathit{append}(Y, R, L)$

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 $P \cup \{D\}$, $\mathit{diff\text{-}p}(X, Y\backslash[\,])$ holds iff $p(X, Y)$ holds. Thus, in our transformed program the clauses for the predicate p can be replaced by the clause

$E.\ p(X, Y) \leftarrow \mathit{diff\text{-}p}(X, Y\backslash[\,])$

We leave it to the reader to check that this replacement of clauses can be performed by a sequence of in-situ folding, unfolding, and independent goal replacement steps, which are correct by the first correctness theorem w.r.t. **LHM** (page 29).

Example 5.2.3. [List Reversal Using Difference-lists] Let us consider the following program for reversing a list:

1. $\mathit{reverse}([\,],[\,]) \leftarrow$
2. $\mathit{reverse}([H|T], R) \leftarrow \mathit{reverse}(T, V), \mathit{append}(V, [H], R)$
3. $\mathit{append}([\,], L, L) \leftarrow$
4. $\mathit{append}([H|T], L, [H|S]) \leftarrow \mathit{append}(T, L, S)$

Given a ground list l of length n and the query $\leftarrow \mathit{reverse}(l, R)$, where R is an unbound variable, this program requires $O(n^2)$ SLD-resolution steps. Indeed, for the evaluation of $\leftarrow \mathit{reverse}(l, R)$, 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 definition rule and introducing the clause

5. $\textit{diff-rev}(X, L\backslash R) \leftarrow \textit{reverse}(X,Y), \textit{append}(Y,R,L)$

corresponding to clause D above.

The recursive definition of *diff-rev* can easily be derived as follows. We unfold clause 5 w.r.t. *reverse*(X,Y) and we get

6. $\textit{diff-rev}([\,], L\backslash R) \leftarrow \textit{append}([\,],R,L)$
7. $\textit{diff-rev}([H|T], L\backslash R) \leftarrow \textit{reverse}(T,V), \textit{append}(V,[H],Y),$ $\textit{append}(Y,R,L)$

By unfolding, clause 6 is replaced by

8. $\textit{diff-rev}([\,], R\backslash R) \leftarrow$

By using the unfold/fold proof method described in Section 4.3 we can prove the validity of the replacement law

$F.$ $\textit{append}(V,[H],Y), \textit{append}(Y,R,L) \equiv_{\{V,H,R,L\}} \textit{append}(V,[H|R],L)$

w.r.t. **LHM** and the current program made out of clauses 1, 2, 3, 4, 7, and 8.

Thus, we apply the goal replacement rule to clause 7 and we get

9. $\textit{diff-rev}([H|T], L\backslash R) \leftarrow \textit{reverse}(T,V), \textit{append}(V,[H|R],L)$

We now fold clause 9 using clause 5 and we get

10. $\textit{diff-rev}([H|T], L\backslash R) \leftarrow \textit{diff-rev}(T, L\backslash[H|R])$

which, together with clause 8, provides the desired recursive definition of *diff-rev*.

The correctness of the transformation steps described above is ensured by the second correctness theorem w.r.t. **LHM** with the assumption that *diff-rev* is a top predicate, *reverse* is an intermediate predicate, and *append* is a basic predicate. Thus, in particular, the replacement performed to derive clause 9 is a basic goal replacement step, and the folding step which generates clause 10 from clause 9 using clause 5 is a single-folding step satisfying the conditions of that second correctness theorem, because: (i) clause 5 has been introduced by the definition rule, (ii) the heads of clauses 5 and 9 have top predicates, and (iii) clause 9 has been derived by first unfolding clause 5 w.r.t. the atom $\textit{reverse}(X,Y)$ with intermediate predicate and then performing a basic goal replacement step.

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

11. $reverse(X,Y) \leftarrow diff\text{-}rev(X, Y\backslash[\,])$

The derived program is as follows:

11. $reverse(X,Y) \leftarrow diff\text{-}rev(X, Y\backslash[\,])$
8. $diff\text{-}rev([\,], R\backslash R) \leftarrow$
10. $diff\text{-}rev([H|T], L\backslash R) \leftarrow diff\text{-}rev(T, L\backslash[H|R])$

It 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 is the introduction of the clause of the form

$D.\ \ diff\text{-}p(X, L\backslash R) \leftarrow p(X,Y), append(Y,R,L)$

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, 1984]. Indeed, as indicated in Example 5.2.3, the argument R of $diff\text{-}rev(X, L\backslash R)$ 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 5.1.

Example 5.2.4. [Inventing Difference-lists by the Generalization Strategy] Let us consider again the initial program of Example 5.2.3 (page 61). 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 V) and introducing the eureka predicate *new-rev* by the following clause:

$N.\ \ new\text{-}rev(T,H,R) \leftarrow reverse(T,V), append(V,[H],R)$

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 '$reverse(\ldots), append(\ldots)$' 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 is depicted in Fig. 4.

Let us now consider clause N_4 in the unfolding tree of Fig. 4. 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 $[K,H]$ of the call of *append* in clause

$N.\ \textit{new-rev}(T,H,R) \leftarrow \underline{\textit{reverse}(T,V)},$
$\qquad \textit{append}(V,[H],R)$

$N_1.\ \textit{new-rev}([\ ],H,R) \leftarrow \underline{\textit{append}([\ ],[H],R)}$

$N_2.\ \textit{new-rev}([K|A],H,R) \leftarrow \textit{reverse}(A,B), \textit{append}(B,[K],V), \textit{append}(V,[H],R)$

$N_3.\ \textit{new-rev}([\ ],H,[H]) \leftarrow$

$N_4.\ \textit{new-rev}([K|A],H,R) \leftarrow \textit{reverse}(A,B), \textit{append}(B,[K,H],R)$

Fig. 4. An unfolding tree for the reverse program. We have underlined the atoms selected for unfolding.

N_4 is not an instance of $[H]$ in clause N. Since N_4 is a descendant of N, we are in a situation where we can apply the generalization strategy. By doing so we introduce the new eureka predicate *gen-rev* defined by the following clause:

$G.\ \textit{gen-rev}(U,X,Y,R) \leftarrow \textit{reverse}(U,B), \textit{append}(B,[X|Y],R)$

where the body of G is the most specific generalization of the body of N and the body of N_4.

The recursive definition of *gen-rev* can be found by performing the transformation steps which correspond to those leading from N to N_4 in the unfolding tree. We get the following clauses:

$\textit{gen-rev}([\ ],X,Y,[X|Y]) \leftarrow$
$\textit{gen-rev}([H|T],X,Y,R) \leftarrow \textit{gen-rev}(T,H,[X|Y],R)$

We can then fold clause 2 using G and we get

2f. $\textit{reverse}([H|T],R) \leftarrow \textit{gen-rev}(T,H,[\ ],R)$

The final program is as follows:

1. $\textit{reverse}([\ ],[\ ]) \leftarrow$
2f. $\textit{reverse}([H|T],R) \leftarrow \textit{gen-rev}(T,H,[\ ],R)$
$\textit{gen-rev}([\ ],X,Y,[X|Y]) \leftarrow$
$\textit{gen-rev}([H|T],X,Y,R) \leftarrow \textit{gen-rev}(T,H,[X|Y],R).$

It has a computational behavior similar to the program derived in Example 5.2.3 (page 61). In particular, the third argument of *gen-rev* is used as an accumulator.

5.3 Overview of Other Techniques

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

5.3.1 Schema-based Transformations

A common feature of the strategies we have described in Section 5.2 is that they are made out of sequences of transformation rules which are not specified in advance; on the contrary, they depend on the structure of the program at hand during the transformation process.

The schema-based approach to program transformation is complementary to the 'rules + strategies' approach and it consists in providing a catalogue of predefined transformations of *program schemata*.

A program schema (or simply, *schema*) S is an abstraction via a substitution θ, of a program P, where some terms, goals, and clauses are replaced by meta-variables, which once instantiated using θ, give us back the program P.

If a program 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 programs 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 transformation from a schema S_1 to a schema S_2 if each instance of S_2 is more efficient of the corresponding instance of S_1.

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 via a substitution θ of P_1, then we choose a transformation from the schema S_1 to a schema S_2 in a given catalogue of correct schema transformations, and finally, we instantiate S_2 using θ to get the transformed program P_2.

The issue of proving the equivalence of program schemata has been addressed within various formalisms, such as flowchart programs, recursive schemata, etc. (see, for instance, [Paterson and Hewitt, 1970; Walker and Strong, 1972; Huet and Lang, 1978]). Some methodologies for developing logic programs using program schemata are proposed by several authors (see, for instance, [Deville and Burnay, 1989; Kirschenbaum *et al.*, 1989; Fuchs and Fromherz, 1992; Flener and Deville, 1993; Marakakis and Gallagher, 1994]) and some examples of logic program schema transformations can be found in [Brough and Hogger, 1987; Seki and Furukawa, 1987; Brough and Hogger, 1991]. The schema transformations presented in these papers are useful for *recursion removal* (see Section 5.3.2 below) and for reducing nondeterminism in generate-and-test programs (see Section 5.2.1, page 51).

An advantage of the schema-based approach over the strategy-based approach is that the application of a schema transformation requires little time, because it is simply the application of a substitution. However, the choice of a suitable schema transformation in the catalogue of the available transformations may be time consuming, because it requires the time for computing the matching substitution. On the other hand, one of the drawbacks of the schema-based approach is the space requirement for storing the catalogue itself. One more drawback is 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.

5.3.2 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 w.r.t. time and space. In the case of imperative programs some program transformation techniques that remove recursion in favor of iteration have been studied, for instance, in [Paterson and Hewitt, 1970; Walker and Strong, 1972].

In logic programming languages, where no iterative constructs are available, recursion removal can be understood as a technique for deriving 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 definite goal. (For simplicity reasons when dealing with recursion removal, 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 favor 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* or *last-call optimization* (see, [Bruynooghe, 1982] for a detailed description and the applicability conditions in the case of Prolog implementations).

Tail-recursion 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. **LHM** (see Section 4.4.1). However, goal rearrangements can increase the amount of nondeterminism, thus making useless the efficiency improvements due to tail-recursion optimization. Moreover, goal rearrangements do not preserve Prolog semantics (see Section 4.4.4), and tail-recursion optimization is usually applied to Prolog programs.

Many researchers have proposed more complex transformation strategies for obtaining tail-recursive programs without increasing the nondeterminism. We would like to mention the following three methods.

The first method consists in transforming *almost-tail-recursive* clauses into tail-recursive ones [Debray, 1985; Azibi, 1987; Debray, 1988] by using the 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, like addition or multiplication of integers. The transformation techniques presented in [Debray, 1985; Azibi, 1987; Debray, 1988] use the generalization strategy and some replacement laws which are valid for the primitive predicates, such as the associativity of list concatenation, the associativity and the commutativity of addition, and the distributivity of multiplication over addition. Those techniques are closely related to the ones considered by [Arsac and Kodratoff, 1982] for functional programs.

The second method is based on schema transformations [Bloch, 1984; Brough and Hogger, 1987; Brough and Hogger, 1991], where some almost-tail recursive program schemata are shown to be equivalent to tail-recursive ones.

The third method consists in transforming a given program into a *binary program*, that is, a program whose clauses have only one atom in their bodies [Tarau and Boyer, 1990]. This transformation method is applicable to all programs and it is in the style of the continuation-based transformations for functional programs [Wand, 1980]. The transformation works by adding to each predicate an extra argument which encodes the next goal to be evaluated. This extra argument represents the so-called *continuation*.

For instance, the program

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

is transformed into the program

$$p \leftarrow r(true)$$
$$r(G) \leftarrow G$$
$$r(G) \leftarrow r((q, G))$$

This transformation in itself does not improve efficiency. However, it allows us to use compilers based on a specialized version of the Warren Abstract Machine [Warren, 1983], and to perform further efficiency improving transformations [Demoen, 1993; Neumerkel, 1993].

5.3.3 Annotations and Memoing

In the previous sections 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 4.4.4), 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, 1982].

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 Prolog program fragment:

$$p(X) \leftarrow C, BodyA$$
$$p(X) \leftarrow not(C), BodyB$$

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

$$p(X) \leftarrow C, !, BodyA$$
$$p(X) \leftarrow BodyB$$

The derived code is more efficient than the initial one and behaves like an if-then-else statement.

Prolog program transformations based on the insertion of cuts are reported in [Sawamura and Takeshima, 1985; Debray and Warren, 1989; Deville, 1990].

Other techniques which introduce annotations for the evaluator are related to the automatic generation of *delay declarations* [Naish, 1985; Wiggins, 1992], which procrastinate calls to predicates until they are suitably instantiated.

A final form of annotation technique which has been used for improving program efficiency is the so-called *memoing* [Michie, 1968]. 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, 1992] or by using the 'assert' predicate for the run-time updating of programs [Sterling and Shapiro, 1994].

6 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 [Rogers, 1967], has been extensively applied in the field of imperative and functional languages [Futamura, 1971; Ershov, 1977; Bjørner *et al.*, 1988; Jones *et al.*, 1993] and first used in logic programming by [Komorowski, 1982] (see also [Venken, 1984; Gallagher, 1986; Safra and Shapiro, 1986; Takeuchi, 1986; Takeuchi and Furukawa, 1986; Ershov *et al.*, 1988] 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 program because, by using the partially known input, it is possible to perform at compile time some run-time computations.

Partial evaluation can be viewed as a particular case of *program specialization* [Scherlis, 1981], 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 values of the input to the program.

No much work has been done in the area of logic program specialization, apart from the particular case of partial deduction. However, some results are reported in [Bossi *et al.*, 1990] and in various papers by Gallagher and others [Gallagher *et al.*, 1988; Gallagher and Bruynooghe, 1991; de Waal and Gallagher, 1992]. In the latter papers the use of the abstract interpretation methodology plays a crucial role. Using this methodology one can 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, that is, during the so-called *preprocessing* phase and *postprocessing* phase. During the preprocessing phase, by 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 and Bondorf, 1993] and the determinacy analysis performed by Mixtus [Sahlin, 1993].

During the postprocessing phase, abstract interpretations may be used for improving the program obtained by the specialization process, as indicated, for instance, in [Gallagher, 1993] where it is shown how one can get rid of the so-called *useless clauses*.

The idea of partial evaluation of logic programs can be presented as follows [Lloyd and Shepherdson, 1991]. Let us consider a normal program P and a query $\leftarrow A$, where A is an atom. We construct a finite portion of an 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 that tree. The notion of unfolding strategy is analogous to the one of u-selection rule (page 50), but it applies to queries, instead of clauses. The design of unfolding strategies which eventually terminate, thereby producing a *finite* SLDNF-tree, can be done within general frameworks like the ones described in [Bruynooghe *et al.*, 1992; Bol, 1993].

We then construct the set of clauses $\{A\theta_i \leftarrow G_i \mid i = 1, \ldots, n\}$, called *resultants*, obtained by collecting from each non-failed leaf of the SLDNF-tree, the query $\leftarrow G_i$ 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 as follows. Let A be of the form $p(\ldots)$. We first replace the clauses of P which constitute the definition of the predicate symbol p by the set of resultants $\{A\theta_i \leftarrow G_i \mid i = 1, \ldots, n\}$, and then we throw away the definitions of the predicates, different from p, on which p does not depend after the replacement.

Example 6.0.1. Let us consider the following program P:

$p([\,], Y) \leftarrow$
$p([H|T], Y) \leftarrow q(T, Y)$
$q(T, Y) \leftarrow Y = b$
$q(T, Y) \leftarrow p(T, Y)$

and the atom $A = p(X, a)$. Let us use the unfolding strategy U which performs unfolding steps starting from the query $\leftarrow p(X, a)$ until each leaf of the SLDNF-tree is either a success or a failure or it is an atom with predicate p. We get the tree depicted in Fig. 5.

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

$p([\,], a) \leftarrow$
$p([H|T], a) \leftarrow p(T, a)$

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 above resultants.

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

The notion of partial evaluation of a program w.r.t. an atom can be extended to the notion of partial evaluation w.r.t. a set S of atoms by considering the union of the sets of resultants relative to the atoms in S.

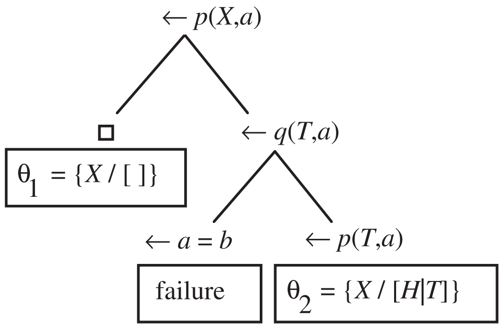

Fig. 5. An SLDNF-tree for $P \cup \{\leftarrow p(X, a)\}$ using U.

We now introduce a *correctness* notion for partial evaluation which refers to the semantics **CASNF** and **FF** considered in Section 4. Analogous notions may be given with reference to other semantics.

Definition 6.0.2 (Correctness of Partial Evaluation). Let P be a program, Q be a query, and S be a set of atoms. A partial evaluation P_S of P w.r.t. S is *correct w.r.t.* Q iff we have that

- $\mathbf{CASNF}[P, Q] = \mathbf{CASNF}[P_S, Q]$, and
- $\mathbf{FF}[P, Q] = \mathbf{FF}[P_S, Q]$.

Theorem 6.0.5 below establishes a criterion for the correctness of partial evaluation. First we need the following definitions, where the notion of instance is relative to a substitution which may be the identity substitution.

Definition 6.0.3. Let R be a program or a query. Given a set S of atoms, we say that R is *S-closed* iff every atom in R with predicate symbol occurring in S is an instance of an atom in S.

Definition 6.0.4. Given a set S of atoms, we say that S is *independent* iff no two atoms in S have a common instance.

Theorem 6.0.5. [Lloyd and Shepherdson, 1991]. *Given a program P, a query Q, and a set S of atoms, let us consider a partial evaluation P_S of P w.r.t. S. If S is independent, and both P and Q are S-closed, then P_S is correct w.r.t. every instance of Q.*

In Example 6.0.1, the correctness w.r.t. every instance of the query $\leftarrow p(X, a)$ of the partial evaluation P_A of the program P, follows from Theorem 6.0.5. Indeed, for the singleton $\{p(X, a)\}$ the independence property trivially holds, and 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 6.0.5, as it is shown by the following two examples.

Example 6.0.6. 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 $p(a) \leftarrow p(b)$. Let A be $\{p(a)\}$. Thus, a partial evaluation 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 (that is, the whole program P) by the resultant $p(a) \leftarrow p(b)$. P_A is not $\{p(a)\}$-*closed* and we have that $\mathbf{CASNF}[P_A, \leftarrow p(a)] = \{\}$, whereas $\mathbf{CASNF}[P, \leftarrow p(a)] = \{\{\}\}$.

Example 6.0.7. Let us consider the following program P:

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

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

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

The program P_S is S-*closed* and $\mathbf{CASNF}[P_S, \leftarrow p] = \{\{\}\}$, whereas $\mathbf{CASNF}[P, \leftarrow p] = \{\}$, because the unique SLDNF-derivation of $P \cup \{\leftarrow p\}$ flounders.

Lloyd and Shepherdson's theorem suggests the following methodology for performing correct partial evaluations. Given a program P and a query Q, we look for an independent set S of atoms and a partial evaluation P_S of P w.r.t. S such that both P_S and Q are S-closed.

Various strategies have been proposed in the literature for computing from a given program P and a given query Q, a suitable set S with the independence and closedness properties (see, for instance, [Benkerimi and Lloyd, 1990; Bruynooghe *et al.*, 1992; Gallagher, 1991; Martens *et al.*, 1992; Gallagher, 1993]). 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, 1987; Bossi *et al.*, 1990; Sahlin, 1993; Bossi and Cocco, 1993; Prestwich, 1993a; Proietti and Pettorossi, 1993]. By using those techniques, given a program P and a query $\leftarrow G$, we introduce a new predicate *newp* defined by the clause

$D.\ \ newp(X_1, \ldots, X_n) \leftarrow G$

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

Obviously, $newp(X_1, \ldots, X_n)$ and G are equivalent goals w.r.t. the semantics **CASNF** and the program $P \cup \{D\}$, and also w.r.t. **FF** and $P \cup \{D\}$. Moreover, we have that

$$\mathbf{CASNF}[P \cup \{D\}, \leftarrow newp(X_1, \ldots, X_n)] = \mathbf{CASNF}[P, \leftarrow G], \text{ and}$$
$$\mathbf{FF}[P \cup \{D\}, \leftarrow newp(X_1, \ldots, X_n)] = \mathbf{FF}[P, \leftarrow G].$$

Thus, we may look for a partial evaluation of the program $P \cup \{D\}$ w.r.t. $newp(X_1, \ldots, X_n)$, instead of a partial evaluation of P w.r.t. G.

The partial evaluation of $P \cup \{D\}$ w.r.t. $newp(\ldots)$ can be achieved by transforming $P \cup \{D\}$ into a program P_G such that

$$\mathbf{CASNF}[P \cup \{D\}, \leftarrow newp(X_1, \ldots, X_n)] =$$
$$= \mathbf{CASNF}[P_G, \leftarrow newp(X_1, \ldots, X_n)], \text{ and}$$
$$\mathbf{FF}[P \cup \{D\}, \leftarrow newp(X_1, \ldots, X_n)] =$$
$$= \mathbf{FF}[P_G, \leftarrow newp(X_1, \ldots, X_n)].$$

These two equalities hold if, for instance, we derive P_G from $P \cup \{D\}$ by using the definition, unfolding, and folding rules according to the restrictions of the second correctness theorems w.r.t. **CASNF** (page 42) and **FF** (page 36), respectively.

Let us now briefly compare the two approaches to partial evaluation we have mentioned above, that is, the one based on Lloyd and Shepherdson's theorem and the one based on the unfold/fold rules.

In the approach based on Lloyd and Shepherdson's theorem, 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 described in Section 5. 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 approach based on the unfold/fold rules does not require to find the set S with the closedness and independence properties, but as we show in Example 6.0.8 below, we often need to introduce some auxiliary clauses by the definition rule and we also need to perform some final folding steps using those clauses.

Example 6.0.8 below also shows that in the partial evaluation approach based on the unfold/fold rules, the use of the renaming technique for structure specialization [Benkerimi and Lloyd, 1990; Gallagher and Bruynooghe, 1990; Gallagher, 1993; Benkerimi and Hill, 1993] which is often required in the first approach, is not needed. For other issues concerning the use of folding during partial evaluation the reader may refer to [Owen, 1989].

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 6.0.8. [String Matching] [Sahlin, 1991; Gallagher, 1993]. Let us consider the following program *Match* for string matching:

1. $match(P, S) \leftarrow aux(P, S, P, S)$
2. $aux([\,], X, Y, Z) \leftarrow$
3. $aux([A|Ps], [A|Ss], P, S) \leftarrow aux(Ps, Ss, P, S)$
4. $aux([A|Ps], [B|Ss], P, [C|S]) \leftarrow \neg(A = B), aux(P, S, P, S)$

where the pattern P and the string S are represented as lists, and the relation $match(P, S)$ holds iff the pattern P occurs in the string S. For instance, the pattern [a,b] occurs in the string [c,a,b], but it does not occur in the string [a,c,b].

Let us now partially evaluate the given program *Match* w.r.t. the atom $match([a, a, b], X)$. In order to do so we first introduce the following definition:

5. $newp(X) \leftarrow match([a, a, b], X)$

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, *newp* in our case. Then we construct the unfolding tree for $\langle$*Match*, clause 5$\rangle$ using the u-selection rule which

i) unfolds a clause w.r.t. any atom of the form either $match(\ldots)$ or $aux(\ldots)$, and
ii) does not unfold a clause for which we can apply the loop absorption strategy, that is, a clause in whose body there is an instance of an atom which occurs in the body of a clause in an ancestor node.

We get the tree depicted in Fig. 6. In clause 8 of Fig. 6, the atom $aux([a, a, b],\ S,\ [a, a, b],\ S)$ is an instance of the body of clause 6 via the substitution $\{X/S\}$.

Analogously, in clause 10 the atom $aux([a, a, b], [H|S], [a, a, b], [H|S])$, and in clause 12 the atom $aux([a, a, b], [a, H|S], [a, a, b], [a, H|S])$ are instances of the body of clause 6.

Thus, we can apply the loop absorption strategy and we introduce the new definition:

14. $newq(S) \leftarrow aux([a, a, b], S, [a, a, b], S)$

We fold clause 6 (see Fig. 6) using clause 14 and we get:

6f. $newp(X) \leftarrow newq(X)$

Now the unfold/fold derivation continues by looking for the recursive definition of the predicate *newq*. This can be done by constructing the

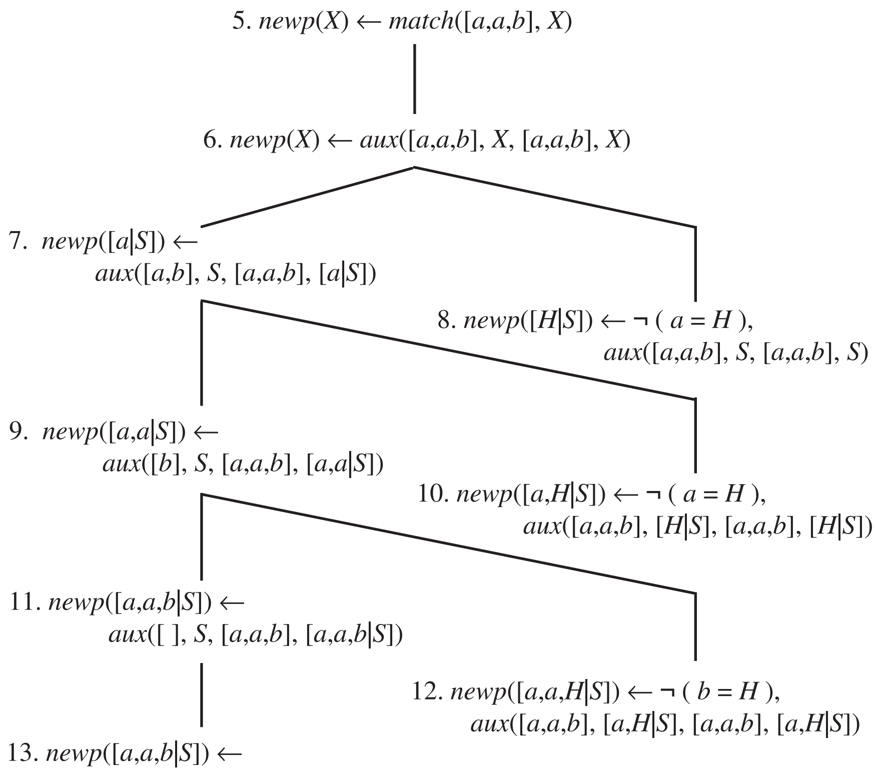

Fig. 6. An unfolding tree for $\langle Match, newp(X) \leftarrow match([a,a,b],X)\rangle$.

unfolding tree for $\langle Match$, clause 14$\rangle$. This tree is equal to the tree depicted in Fig. 6, except that clause 5 is deleted and the name *newp* is replaced by the name *newq*.

Thus, the leaves of the unfolding tree for $\langle Match$, clause 14$\rangle$ have the following clauses:

13q. $newq([a,a,b|S]) \leftarrow$
12q. $newq([a,a,H|S]) \leftarrow \neg(b = H), aux([a,a,b],[a,H|S],[a,a,b],[a,H|S])$
10q. $newq([a,H|S]) \leftarrow \neg(a = H), aux([a,a,b],[H|S],[a,a,b],[H|S])$
8q. $newq([H|S]) \leftarrow \neg(a = H), aux([a,a,b],S,[a,a,b],S)$

By folding clauses 12q, 10q, and 8q, we get the following program:

6f. $newp(X) \leftarrow newq(X)$
13q. $newq([a,a,b|S]) \leftarrow$
12qf. $newq([a,a,H|S]) \leftarrow \neg(b = H), newq([a,H|S])$
10qf. $newq([a,H|S]) \leftarrow \neg(a = H), newq([H|S])$
8qf. $newq([H|S]) \leftarrow \neg(a = H), newq(S)$

which is exactly the program produced by the Mixtus partial evaluator (see [Sahlin, 1991], page 124).

One of the most interesting 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, 1971]. For this last application it is necessary that the partial evaluator be self-applicable. This means that it should be able to partially evaluate itself. The interested reader may refer to [Jones *et al.*, 1993] for a general overview, and to [Fujita and Furukawa, 1988; Fuller and Abramsky, 1988; Mogensen and Bondorf, 1993; Gurr, 1993; Leuschel, 1994a] for more details on the problem of self-applicability 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 and integrity checking. Some results in this direction can be found in [Sakama and Itoh, 1988; Bry, 1989; Leuschel, 1994b].

7 Related Methodologies for Program Development

From what we have presented in the previous sections it is clear that the program transformation methodology for program development is very much related to various fields of Artificial Intelligence, Theoretical Computer Science, and Software Engineering. Here we want to briefly indicate some of the techniques and methods which are used in those fields and are of relevance to the transformation methodology and its applications.

Let us begin by considering some *program analysis* techniques by which the programmer can investigate various program properties. 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/or freeness of arguments, and the functionality (or determinacy) of predicates.

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 [Cousot and Cousot, 1977; Debray, 1992]. They make use of finite interpretation domains where information can be derived via 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.*, 1992].

A general framework where program transformation strategies are supported by abstract interpretation techniques, is defined in [Boulanger and

Bruynooghe, 1993]. Among the many transformation techniques which depend on program analysis techniques, we would like to mention: i) compiling control (see Section 5.2.1), where the information about the flow of computation is used for generating the unfolding tree, ii) the specialization method of [Gallagher and Bruynooghe, 1991], 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 determinacy information (see Section 5.3), and iv) various techniques implemented in the Spes system [Alexandre *et al.*, 1992; Bsaïes, 1992] in which *mode analysis* is used to mechanize several transformation strategies.

Very much related with these methodologies for the analysis of programs are the methodologies for the proof of properties of programs. They have been used for program verification, and in particular, for ensuring 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 guiding the application of suitable instances of the goal replacement rule.

Many proof techniques can be found in the literature, and in particular, in the field of Theorem Proving and Automated Deduction. For the ones which have been used for logic programs and may be adapted for program transformation we recall those in [Drabent and Małuszyński, 1988; Bossi and Cocco, 1989; Deransart, 1989].

The field of program transformation partially overlaps with the field of program synthesis (see [Deville and Lau, 1994] for a survey in the case of logic programs). 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 that specification. However, it is usually understood that program synthesis differs from program transformation because in program synthesis the specification is a somewhat implicit description of the program to be derived, and such implicit description often does not allow us to get the desired program by a sequence of simple manipulations, like those determined 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 belong to the field of logic program synthesis.

The transformational methods for developing logic programs are also closely related to methods for *logic program construction* [Sterling and Lakhotia, 1988; Deville, 1990; Sterling and Kirschenbaum, 1993], where complex programs are developed by enhancing and composing together simpler programs (see Section 5.2.2). However, the basic ideas and objectives of program construction are quite different from those of program transformation. In particular, the starting point for the above mentioned techniques for program construction is not a logic program, but a possibly incomplete and not fully formalized specification. Thus, the notion of semantics plays a minor role, in comparison with the techniques for program transformation. Moreover, the main objective of program construction is the improvement of the efficiency in software production, rather than the improvement of the efficiency of programs.

Finally, we would like to mention that the transformation and specialization techniques considered in this chapter have been partially extended to the case of concurrent logic programs [Ueda and Furukawa, 1988] and constraint logic programs [Hickey and Smith, 1991; Maher, 1993; Bensaou and Guessarian, 1994; Etalle and Gabbrielli, 1996].

Conclusions

We have looked at the theoretical foundations of the so-called 'rules + strategies' approach to logic program transformation. We have established a unified framework for presenting and comparing 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. different semantics of definite logic programs, such as: 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 have presented the rules which preserve computed answer substitutions, finite failure, and Clark's completion semantics. We have briefly mentioned the results concerning the rules which preserve other semantics for normal programs.

We have also presented a unified framework in which it is possible to describe some of the most significant techniques for guiding the application of the transformation rules with the aim of improving program efficiency. We have singled out a few basic strategies, such as tupling, loop absorption, and generalization, and we have shown that various methods for compiling control, program composition, change of data representations, and partial evaluation, 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 all programs which are equivalent to the given initial program, and in the 'complexity' sense, that is, their capability of deriving programs which are more efficient than the initial program. No conclusive results are available in either direction.

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.

Unfortunately, the transformational methodology in the practice of logic programming has gained only moderate attention in the past. However, it is recognized that the automation of transformation techniques and their integrated use is of crucial importance for building advanced software development systems.

There is a growing interest in the mechanization of transformation strategies and the production of interactive tools for implementing program transformers. Moreover, some optimizing compilers already developed, make use of various transformation techniques.

The importance of the transformation methodology will substantially increase if we extend its theory and applications also to the case of complex logic languages which manipulate constraints, and support both concurrency and object-orientation.

Acknowledgements

We thank M. Bruynooghe, J. P. Gallagher, M. Leuschel, M. Maher, and H. Seki for their helpful comments and advice on many issues concerning the transformation of logic programs. Our thanks go also to O. Aioni, P. Dell'Acqua, M. Gaspari, and M. Kalsbeek for reading of a preliminary version of this chapter.

References

[Alexandre *et al.*, 1992] F. Alexandre, K. Bsaïes, J. P. Finance, and A. Quéré. Spes: A system for logic program transformation. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning, LPAR '92*, Lecture Notes in Computer Science 624, pages 445–447, 1992.

[Amtoft, 1992] T. Amtoft. Unfold/fold transformations preserving termination properties. In *Proc. PLILP '92, Leuven, Belgium*, Lecture Notes in Computer Science 631, pages 187–201. Springer-Verlag, 1992.

[Apt, 1990] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–576. Elsevier, 1990.

[Aravindan and Dung, 1995] C. Aravindan and P. M. Dung. On the correctness of unfold/fold transformation of normal and extended logic programs. *Journal of Logic Programming*, 24(3):201–217, 1995.

[Arsac and Kodratoff, 1982] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, 1982.

[Azibi, 1987] N. Azibi. *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, 1992] M. Baudinet. Proving termination properties of Prolog programs: A semantic approach. *Journal of Logic Programming*, 14:1–29, 1992.

[Benkerimi and Hill, 1993] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, 1993.

[Benkerimi and Lloyd, 1990] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, USA*, pages 343–358. The MIT Press, 1990.

[Bensaou and Guessarian, 1994] N. Bensaou and I. Guessarian. Transforming constraint logic programs. In *11th Symp. on Theoretical Aspects of Computer Science, STACS '94*, Lecture Notes in Computer Science 775, pages 33–46. Springer-Verlag, 1994.

[Bird, 1984] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Toplas*, 6(4):487–504, 1984.

[Bjørner *et al.*, 1988] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988. IFIP TC2 Workshop on Partial and Mixed Computation, Gammel Avernæs, Denmark, 1987.

[Bloch, 1984] C. Bloch. Source-to-source transformations of logic programs. Master's thesis, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1984.

[Bol, 1993] R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16:25–46, 1993.

[Bossi and Cocco, 1989] A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings TAPSOFT '89*, Lecture Notes in Computer Science 352, pages 96–110. Springer-Verlag, 1989.

[Bossi and Cocco, 1993] A. Bossi and N. Cocco. Basic transformation operations which preserve computed answer substitutions of logic programs. *Journal of Logic Programming*, 16(1&2):47–87, 1993.

[Bossi and Cocco, 1994] A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In *Proceedings ALP '94*, LNCS 850, pages 269–286. Springer-Verlag, 1994.

[Bossi and Etalle, 1994a] A. Bossi and S. Etalle. More on unfold/fold transformations of normal programs: Preservation of Fitting's semantics. In L. Fribourg and F. Turini, editors, *Proceedings of LOPSTR'94 and META'94*, Pisa, Italy, Lecture Notes in Computer Science 883, pages 311–331, Springer-Verlag, 1994.

[Bossi and Etalle, 1994b] A. Bossi and S. Etalle. Transforming acyclic programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1081–1096, July 1994.

[Bossi *et al.*, 1990] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

[Bossi *et al.*, 1992a] A. Bossi, N. Cocco, and S. Etalle. On safe folding. In *Proceedings PLILP '92, Leuven, Belgium*, Lecture Notes in Computer Science 631, pages 172–186. Springer-Verlag, 1992.

[Bossi *et al.*, 1992b] A. Bossi, N. Cocco, and S. Etalle. Transforming normal programs by replacement. In A. Pettorossi, editor, *Proceedings 3rd International Workshop on Meta-Programming in Logic, Meta '92, Uppsala, Sweden*, Lecture Notes in Computer Science 649, pages 265–279. Springer-Verlag, 1992.

[Boulanger and Bruynooghe, 1993] D. Boulanger and M. Bruynooghe. Deriving unfold/fold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation*, 15:495–521, 1993.

[Boyer and Moore, 1975] R. S. Boyer and J. S. Moore. Proving theorems about Lisp functions. *Journal of the ACM*, 22(1):129–144, 1975.

[Brough and Hogger, 1987] D. R. Brough and C. J. Hogger. Compiling associativity into logic programs. *Journal of Logic Programming*, 4:345–359, 1987.

[Brough and Hogger, 1991] D. R. Brough and C. J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9(1):115–134, 1991.

[Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J. A. Campbell, editor, *Implementations of Prolog*, pages 253–266. Ellis Horwood, 1984.

[Bruynooghe *et al.*, 1989] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.

[Bruynooghe *et al.*, 1992] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. *New Generation Computing*, 11:47–79, 1992.

[Bruynooghe, 1982] M. Bruynooghe. The memory management of Prolog implementations. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 83–98. Academic Press, 1982.

[Bry, 1989] F. Bry. Query evaluation in recursive data bases: Bottom-up and top-down reconciled. In *Proceedings 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan*, 1989.

[Bsaïes, 1992] K. Bsaïes. Static analysis for the synthesis of eureka properties for transforming logic programs. In *Proceedings 4th UK Conference* *on Logic Programming, ALPUK '92*, Workshops in Computing, pages 41–61. Springer-Verlag, 1992.

[Burstall and Darlington, 1975] R. M. Burstall and J. Darlington. Some transformations for developing recursive programs. In *Proceedings of the International Conference on Reliable Software, Los Angeles, USA*, pages 465–472, 1975.

[Burstall and Darlington, 1977] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Clark and Sickel, 1977] K. L. Clark and S. Sickel. Predicate logic: A calculus for deriving programs. In *Proceedings 5th International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, USA*, pages 419–420, 1977.

[Clark and Tärnlund, 1977] K. L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In *Proceedings Information Processing '77*, pages 939–944. North-Holland, 1977.

[Cook and Gallagher, 1994] J. Cook and J. P. Gallagher. A transformation system for definite programs based on termination analysis. In L. Fribourg and F. Turini, editors, *Proceedings of LOPSTR'94 and META'94, Pisa, Italy*, Lecture Notes in Computer Science 883, pages 51–68. Springer-Verlag, 1994.

[Cortesi *et al.*, 1992] A. Cortesi, G. Filé, and W. Winsborough. Comparison of abstract interpretations. In *Proceedings Nineteenth ICALP*, Wien, Austria, Lecture Notes in Computer Science 623, pages 521–532. Springer-Verlag, 1992.

[Cousot and Cousot, 1977] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings 4th ACM-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, 1977.

[Darlington, 1972] J. Darlington. *A Semantic Approach to Automatic Program Improvement*. PhD thesis, Department of Machine Intelligence, Edinburgh University, Edinburgh (Scotland) UK, 1972.

[Darlington, 1978] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[Darlington, 1981] J. Darlington. An experimental program transformation system. *Artificial Intelligence*, 16:1–46, 1981.

[De Schreye and Bruynooghe, 1989] D. De Schreye and M. Bruynooghe. On the transformation of logic programs with instantiation based computation rules. *Journal of Symbolic Computation*, 7:125–154, 1989.

[De Schreye *et al.*, 1991] D. De Schreye, B. Martens, G. Sablon, and M. Bruynooghe. Compiling bottom-up and mixed derivations into top-down executable logic programs. *Journal of Automated Reasoning*, 7:337–-358, 1991.

[de Waal and Gallagher, 1992] D. A. de Waal and J. P. Gallagher. Specialization of a unification algorithm. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '91, Manchester, UK*, Workshops in Computing, pages 205–221. Springer-Verlag, 1992.

[Debray and Mishra, 1988] S. K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5:61–91, 1988.

[Debray and Warren, 1988] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5:207–229, 1988.

[Debray and Warren, 1989] S. K. Debray and D. S. Warren. Functional computations in logic programs. *ACM TOPLAS*, 11(3):451–481, 1989.

[Debray, 1985] S. K. Debray. Optimizing almost-tail-recursive Prolog programs. In *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, Lecture Notes in Computer Science 201, pages 204–219. Springer-Verlag, 1985.

[Debray, 1988] S. K. Debray. Unfold/fold transformations and loop optimization of logic programs. In *Proceedings SIGPLAN 88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA*, SIGPLAN Notices, 23, (7), pages 297–307, 1988.

[Debray, 1992] S. K. Debray, editor. *Special Issue of the Journal of Logic Programming on Abstract Interpretation*, volume 12, Nos. 2&3. Elsevier, 1992.

[Demoen, 1993] B. Demoen. On the transformation of a Prolog program to a more efficient binary program. In K.-K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '92, Manchester, UK*, Workshops in Computing, pages 242–252. Springer-Verlag, 1993.

[Deransart, 1989] P. Deransart. Proof methods of declarative properties of logic programs. In *Proceedings TAPSOFT '89*, Lecture Notes in Computer Science 352, pages 207–226. Springer-Verlag, 1989.

[Deville and Burnay, 1989] Y. Deville and J. Burnay. Generalization and program schemata. In *Proceedings NACLP '89*, pages 409–425. The MIT Press, 1989.

[Deville and Lau, 1994] Y. Deville and K.-K. Lau. Logic program synthesis. *Journal of Logic Programming*, 19, 20:321–350, 1994.

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

[Dix, 1995] J. Dix. A classification theory of semantics of normal logic programs: II weak properties. *Fundamenta Informaticae*, XXII(3):257–288, 1995.

[Drabent and Małuszyński, 1988] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 1(1):133–155, 1988.

[Ershov *et al.*, 1988] A. P. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldson, and W. Scherlis, editors. *Special Issue of New Generation Computing: Workshop on Partial Evaluation and Mixed Computation*, volume 6, Nos. 2&3. Ohmsha Ltd. and Springer-Verlag, 1988.

[Ershov, 1977] A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.

[Etalle and Gabbrielli, 1996] S. Etalle and M. Gabbrielli. Modular transformations of CLP programs. *Theoretical Computer Science*, 166:101–146, 1996.

[Feather, 1982] M. S. Feather. A system for assisting program transformation. *ACM Toplas*, 4(1):1–20, 1982.

[Feather, 1987] M. S. Feather. A survey and classification of some program transformation techniques. In L. G. L. T. Meertens, editor, *Proceedings IFIP TC2 Working Conference on Program Specification and Transformation, Bad Tölz, Germany*, pages 165–195. North-Holland, 1987.

[Fitting, 1985] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Flener and Deville, 1993] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation*, 15:775–805, 1993.

[Fuchs and Fromherz, 1992] N. E. Fuchs and M. P. J. Fromherz. Schema-based transformations of logic programs. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '91, Manchester, UK*, pages 111–125. Springer-Verlag, 1992.

[Fujita and Furukawa, 1988] H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2&3):91–118, 1988.

[Fujita, 1987] H. Fujita. An algorithm for partial evaluation with constraints. Technical Memorandum TM-0367, ICOT, Tokyo, Japan, 1987.

[Fuller and Abramsky, 1988] D. A. Fuller and S. Abramsky. Mixed computation of Prolog programs. *New Generation Computing*, 6(2&3):119–141, 1988.

[Futamura, 1971] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[Gallagher and Bruynooghe, 1990] J. P. Gallagher and M. Bruynooghe.

Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic, Leuven, Belgium*, pages 229–246. Department of Computer Science, KU Leuven (Belgium), April 1990.

[Gallagher and Bruynooghe, 1991] J. P. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 6(2):305–333, 1991.

[Gallagher *et al.*, 1988] J. P. Gallagher, M. Codish, and E. Shapiro. Specialization of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6(2&3):159–186, 1988.

[Gallagher, 1986] J. P. Gallagher. Transforming programs by specializing interpreters. In *Proceedings Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 109–122, 1986.

[Gallagher, 1991] J. P. Gallagher. A system for specializing logic programs. Technical Report TR-91-32, University of Bristol, Bristol, UK, 1991.

[Gallagher, 1993] J. P. Gallagher. Tutorial on specialization of logic programs. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.

[Gardner and Shepherdson, 1991] P. A. Gardner and J. C. Shepherdson. Unfold/fold transformations of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 565–583. MIT, 1991.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1070–1080. The MIT Press, 1988.

[Gergatsoulis and Katzouraki, 1994] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Proceedings Sixth International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, Lecture Notes in Computer Science 844, pages 340–354. Springer-Verlag, 1994.

[Gurr, 1993] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, University of Bristol, Bristol, UK, 1993.

[Hansson and Tärnlund, 1982] Å. Hansson and S.-Å. Tärnlund. Program transformation by data structure mapping. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 117–122. Academic Press, 1982.

[Hickey and Smith, 1991] T. J. Hickey and D. A. Smith. Towards the partial evaluation of CLP languages. In *Proceedings ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM*

'91, New Haven, CT, USA, SIGPLAN Notices, 26, 9, pages 43–51. ACM Press, 1991.

[Hogger, 1981] C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.

[Huet and Lang, 1978] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[Jones and Mycroft, 1984] N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proceedings 1984 International Symposium on Logic Programming, Atlantic City, New Jersey, USA*, pages 289–298, 1984.

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

[Kanamori and Fujita, 1986] T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. Technical Report 179, ICOT, Tokyo, Japan, 1986.

[Kanamori and Horiuchi, 1987] T. Kanamori and K. Horiuchi. Construction of logic programs based on generalized unfold/fold rules. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 744–768. The MIT Press, 1987.

[Kawamura and Kanamori, 1990] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformation. *Theoretical Computer Science*, 75:139–156, 1990.

[Kirschenbaum *et al.*, 1989] M. Kirschenbaum, A. Lakhotia, and L. Sterling. Skeletons and techniques for Prolog programming. TR 89-170, Case Western Reserve University, 1989.

[Kleene, 1971] S. C. Kleene. *Introduction to Metamathematics.* North-Holland, 1971.

[Komorowski, 1982] H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Ninth ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, USA, pages 255–267, 1982.

[Kott, 1978] L. Kott. About transformation system: A theoretical study. In *3ème Colloque International sur la Programmation*, pages 232–247, Paris (France), 1978. Dunod.

[Kott, 1982] L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.

[Kowalski, 1979] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[Kunen, 1987] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):289–308, 1987.

[Kunen, 1989] K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7:231–246, 1989.

[Leuschel, 1994a] M. Leuschel. Partial evaluation of the real thing. In L. Fribourg and F. Turini, editors, *Proceedings of LOPSTR'94 and META'94*, Pisa, Italy, Lecture Notes in Computer Science 883, pages 122–137, Springer-Verlag, 1994.

[Leuschel, 1994b] M. Leuschel. Partial evaluation of the real thing and its application to integrity checking. Technical report, Computer Science Department, K.U. Leuven, Heverlee, Belgium,

[Lloyd and Shepherdson, 1991] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[Lloyd, 1987] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Second Edition, 1987.

[Maher, 1987] M. J. Maher. Correctness of a logic program transformation system. IBM Research Report RC 13496, T. J. Watson Research Center, 1987.

[Maher, 1990] M. J. Maher. Reasoning about stable models (and other unstable semantics). IBM research report, T. J. Watson Research Center, 1990.

[Maher, 1993] M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

[Marakakis and Gallagher, 1994] E. Marakakis and J. P. Gallagher. Schema-based tod-down design of logic programs using abstract data types. In L. Fribourg and F. Turini, editors, *Proceedings of LOPSTR'94 and META'94*, Pisa, Italy, Lecture Notes in Computer Science 883, pages 138–153, Springer-Verlag, 1994.

[Marriot and Søndergaard, 1993] K. Marriot and H. Søndergaard. Difference-list transformation for Prolog. *New Generation Computing*, 11:125–177, 1993.

[Martens *et al.*, 1992] B. Martens, D. De Schreye, and M. Bruynooghe. Sound and complete partial deduction with unfolding based on well-founded measures. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 473–480. Ohmsha Ltd., IOS Press, 1992.

[Michie, 1968] D. Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[Mogensen and Bondorf, 1993] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '92, Manchester, UK*, Workshops in Computing, pages 214–227. Springer-Verlag, 1993.

[Naish, 1985] L. Naish. *Negation and Control in Prolog.* Lecture Notes in Computer Science 238. Springer-Verlag, 1985.

[Narain, 1986] S. Narain. A technique for doing lazy evaluation in logic. *Journal of Logic Programming*, 3(3):259–276, 1986.

[Neumerkel, 1993] U. W. Neumerkel. *Specialization of Prolog Programs with Partially Static Goals and Binarization.* PhD thesis, Technical University Wien, Austria, 1993.

[Owen, 1989] S. Owen. Issues in the partial evaluation of meta-interpreters. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 319–339. The MIT Press, 1989.

[Paige and Koenig, 1982] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[Partsch, 1990] H. A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[Paterson and Hewitt, 1970] M. S. Paterson and C. E. Hewitt. Comparative schematology. In *Conference on Concurrent Systems and Parallel Computation Project MAC*, Woods Hole, Mass., USA, pages 119–127, 1970.

[Pettorossi and Proietti, 1989] A. Pettorossi and M. Proietti. Decidability results and characterization of strategies for the development of logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, Portugal, pages 539–553. The MIT Press, 1989.

[Pettorossi and Proietti, 1994] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[Pettorossi and Proietti, 1996] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[Pettorossi, 1977] A. Pettorossi. Transformation of programs and use of tupling strategy. In *Proceedings Informatica 77, Bled, Yugoslavia*, pages 1–6, 1977.

[Prestwich, 1993a] S. Prestwich. Online partial deduction of large programs. In *Proceedings ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 111–118. ACM Press, 1993.

[Prestwich, 1993b] S. Prestwich. An unfold rule for full Prolog. In K.-K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '92, Manchester, UK*, Workshops in Computing, pages 199–213. Springer-Verlag, 1993.

[Proietti and Pettorossi, 1990] M. Proietti and A. Pettorossi. Synthesis of eureka predicates for developing logic programs. In N. D. Jones, editor, *Third European Symposium on Programming, ESOP '90*, Lecture Notes in Computer Science 432, pages 306–325. Springer-Verlag, 1990.

[Proietti and Pettorossi, 1991] M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '91, Yale University, New Haven, Connecticut, USA*, pages 274–284. ACM Press, 1991.

[Proietti and Pettorossi, 1993] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming*, 16(1–2):123–161, 1993.

[Proietti and Pettorossi, 1994a] M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In Y. Deville, editor, *Logic Program Synthesis and Transformation, Proceedings of LOPSTR '93, Louvain-la-Neuve, Belgium*, Workshops in Computing, pages 141–158. Springer-Verlag, 1994.

[Proietti and Pettorossi, 1994b] M. Proietti and A. Pettorossi. Total correctness of the goal replacement rule based on unfold/fold proofs. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proceedings of the 1994 Joint Conference on Declarative Programming, GULP-PRODE '94*, Peñíscola, Spain, September 19–22, pages 203–217. Universidad Politécnica de Valencia, 1994.

[Proietti and Pettorossi, 1995] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.

[Przymusinsky, 1987] T. Przymusinsky. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1987.

[Rogers, 1967] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[Safra and Shapiro, 1986] S. Safra and E. Shapiro. Meta interpreters for real. In H. J. Kugler, editor, *Proceedings Information Processing 86*, pages 271–278. North-Holland, 1986.

[Sahlin, 1991] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, SICS, Sweden, 1991.

[Sahlin, 1993] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12:7–51, 1993.

[Sakama and Itoh, 1988] C. Sakama and H. Itoh. Partial evaluation of queries in deductive databases. *New Generation Computing*, 6(2, 3):249–258, 1988.

[Sato and Tamaki, 1988] T. Sato and H. Tamaki. Deterministic transformation and deterministic synthesis. In *Future Generation Computers.* North-Holland, 1988.

[Sato, 1992] T. Sato. An equivalence preserving first order unfold/fold transformation system. *Theoretical Computer Science*, 105:57–84, 1992.

[Sawamura and Takeshima, 1985] H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their application to Prolog optimization. In *Proceedings of the International Symposium on Logic Programming*, Boston, USA, pages 200–207. IEEE Computer Society Press, 1985.

[Scherlis, 1981] W. L. Scherlis. Program improvement by internal specialization. In *Proc. 8th ACM Symposium on Principles of Programming Languages, Williamsburgh, Va*, pages 41–49. ACM Press, 1981.

[Schwarz, 1982] J. Schwarz. Using annotations to make recursive equations behave. *IEEE Transactions on Software Engineering SE*, 8(1):21–33, 1982.

[Seki and Furukawa, 1987] H. Seki and K. Furukawa. Notes on transformation techniques for generate and test logic programs. In *Proceedings of the International Symposium on Logic Programming, San Francisco, USA*, pages 215–223. IEEE Press, 1987.

[Seki, 1990] H. Seki. A comparative study of the well-founded and the stable model semantics: Transformation's viewpoint. In *Proceedings of the Workshop on Logic Programming and Non-monotonic Logic*, pages 115–123. Cornell University, USA, 1990.

[Seki, 1991] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[Seki, 1993] H. Seki. Unfold/fold transformation of general logic programs for well-founded semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.

[Shepherdson, 1992] J. C. Shepherdson. Unfold/fold transformations of logic programs. *Mathematical Structures in Computer Science*, 2:143–157, 1992.

[Sterling and Kirschenbaum, 1993] L. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In J.-M. Jacquet, editor, *Constructing Logic Programs*, chapter 6, pages 127–140. Wiley, 1993.

[Sterling and Lakhotia, 1988] L. Sterling and A. Lakhotia. Composing Prolog meta-interpreters. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings Fifth International Conference on Logic Programming*, Seattle, WA, USA, pages 386–403. The MIT Press, 1988.

[Sterling and Shapiro, 1994] L. S. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1994. Second Edition.

[Takeuchi and Furukawa, 1986] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta-programming. In H. J. Kugler, editor, *Proceedings of Information Processing '86*, pages 279–282. North-Holland, 1986.

[Takeuchi, 1986] A. Takeuchi. Affinity between meta-interpreters and partial evaluation. In H. J. Kugler, editor, *Proceedings of Information Processing '86*, pages 279–282. North-Holland, 1986.

[Tamaki and Sato, 1984] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärlund, editor, *Proceedings Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

[Tamaki and Sato, 1986] H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan, 1986.

[Tarau and Boyer, 1990] P. Tarau and M. Boyer. Elementary logic programs. In P. Deransart and J. Małuszyński, editors, *Proceedings PLILP '90*, pages 159–173. Springer-Verlag, 1990.

[Träff and Prestwich, 1992] J. L. Träff and S. D. Prestwich. Meta-programming for reordering literals in deductive databases. In A. Pettorossi, editor, *Proceedings 3rd International Workshop on Meta-Programming in Logic, Meta '92*, Uppsala, Sweden, Lecture Notes in Computer Science 649, pages 280–293. Springer-Verlag, 1992.

[Turchin, 1986] V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.

[Ueda and Furukawa, 1988] K. Ueda and K. Furukawa. Transformation rules for GHC programs. In *Proceedings International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, Japan, pages 582–591, 1988.

[van Emden and Kowalski, 1976] M. H. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[Van Gelder *et al.*, 1989] A. Van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the ACM Sigact-Sigmod Symposium on Principles of Database Systems*, pages 221–230. ACM Press, 1989.

[Venken, 1984] R. Venken. A Prolog meta-interpretation for partial evaluation and its application to source-to-source transformation and query optimization. In T. O'Shea, editor, *Proceedings of ECAI '84*, pages 91–100. North-Holland, 1984.

[Wadler, 1990] P. L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Walker and Strong, 1972] S. A. Walker and H. R. Strong. Characterization of flowchartable recursions. In *Proceedings 4th Annual ACM Symposium on Theory of Computing*, Denver, CO, USA, 1972.

[Wand, 1980] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

[Warren, 1983] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, 1983.

[Warren, 1992] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

[Wiggins, 1992] G. A. Wiggins. Negation and control in automatically generated logic programs. In A. Pettorossi, editor, *Proceedings 3rd International Workshop on Meta-Programming in Logic, Meta '92*, Uppsala, Sweden, Lecture Notes in Computer Science 649, pages 250–264. Springer-Verlag, 1992.

[Wirth, 1976] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc., 1976.

[Zhang and Grant, 1988] J. Zhang and P. W. Grant. An automatic difference-list transformation algorithm for Prolog. In *Proceedings 1988 European Conference on Artificial Intelligence, ECAI '88*, pages 320–325. Pitman, 1988.