Automatic Derivation of Logic Programs by Transformation*

Alberto Pettorossi[1] and Maurizio Proietti[2]
(1) DISP, University of Roma Tor Vergata, Roma, Italy
(2) IASI-CNR, Roma, Italy
{adp,proietti}@iasi.rm.cnr.it
http://www.iasi.rm.cnr.it~{adp,proietti}

Abstract

We present the program transformation methodology for the automatic development of logic programs based on the *rules + strategies* approach. We consider both definite programs and normal programs and we present the basic transformation rules and strategies which are described in the literature. To illustrate the power of the program transformation approach we also give some examples of program development. Finally, we show how to use program transformations for proving properties of predicates and synthesizing programs from logical specifications.

Contents

*Course notes for ESSLLI 2000, European Summer School on Logic, Language, and Information

1 Introduction

The process of deriving programs from specifications is often divided into three steps: (i) the derivation of formal specifications from the informal ones, (ii) the validation of the formal specifications, and (iii) the derivation of executable programs from the formal specifications.

Each step of this derivation process can be supported by the use of elegant and well-understood notions of mathematical logic. In particular, from informal specifications given as sentences in a natural language, one can derive formal specifications as formulas of a first-order logical theory [72]. One may then validate the derived formal specifications by checking whether or not they are consistent and satisfy some suitable properties. Finally, as we will illustrate in this course, from formal specifications one may obtain executable, efficient programs by using techniques for logic program transformation. This is, indeed, one of the reasons that makes logic programming very attractive for program construction. During this final transformation step from specifications to programs, efficiency is improved by avoiding multiple visits of data structures, or by replacing complex forms of recursion in favour of tail recursion, or by reducing nondeterminism of procedures.

The basic ideas of the program transformation methodology have been introduced in the early seventies for validating various techniques such as those which remove recursion in favour of iteration [45, 151]. However, the program transformation methodology has been formalized some years later [29] and since then, program transformation has been extensively used for the development of functional and logic programs. For these programs we can perform manipulations using relatively simple techniques, such as equational reasoning and logical deduction [35, 85].

We will focus our attention on the transformation of logic programs into equivalent, more efficient ones. We will also consider the problem of transforming specifications into executable logic programs. This problem is often addressed within the area of *program synthesis* [59]. We will not consider the techniques which improve program performances by transforming logic programs into lower level languages by translating, for instance, the programs into WAM code and then optimizing that code.

1.1 The Rules + Strategies Approach

We will mainly be concerned with the so-called unfold/fold program transformations based on the *rules + strategies* approach. The basic idea of this approach can be pictorially represented as in Fig. 1. From the initial program P_0 we want to obtain a final program P_n with the same semantic value, that is, we want that $Sem(P_0) = Sem(P_n)$ for some given semantic function 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 $0 \leq i < n$, $Sem(P_i) = Sem(P_{i+1})$, where P_{i+1} is obtained from P_i by applying a *transformation rule*. In this case each transformation step leading from program P_i to program P_{i+1}, for $0 \leq i < n$, is said to be *totally correct*.

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* in the sense that for $0 \leq i < n$, $Sem(P_i) \supseteq Sem(P_{i+1})$.

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 given a sequence $P_0, \ldots, P_n$ of programs there exists a cost function C which measures the computational complexity of the programs, such that $C(P_0) \geq C(P_n)$. In logic programming there are many notions of efficiency which can be used. They are related either to the size of the

$$\begin{tikzcd}
P_0 \arrow[r] \arrow[drr, "Sem"'] & P_1 \arrow[r] \arrow[dr, "Sem"] & \ldots \arrow[r] & P_n \arrow[dl, "Sem"] \\
 & & V &
\end{tikzcd}$$

Figure 1: Program Transformation: from program P_0 we derive program P_n in various steps, preserving the semantic value $V = Sem(P_0) = \ldots = Sem(P_n)$.

SLD-refutations [103] or to suitable properties of the machine model which is assumed for the execution of programs.

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.

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

There is a third important objective of program transformation which is sometimes given less attention: it is the formal description of the program transformation process itself as a tree of programs, called *derivation tree*, rooted in the given initial program. That tree is the collection of several transformation sequences starting from the same initial program and glued together along their common prefixes. Derivation trees allow us to construct and manipulate program derivations. In particular, the operation of constructing a new program derivation starting from a program which has been derived during a previous derivation, corresponds to the addition of a new branch to a derivation tree.

Derivation trees also form the basis for a language in which we can express transformation strategies and we can reason about program constructions. In that language which may be viewed as a metalanguage for program transformation, we can describe the application of a transformation rule or a transformation strategy. This makes the program transformation approach a powerful programming methodology in the same way as metalevel reasoning makes mechanical theorem proving very effective. Metalevel reasoning may suggest us the transformation rules or strategies which should be applied during program transformation. Thus, for instance, the construction of the transformation sequences starting from similar initial programs may be done by applying similar transformation strategies.

Finally, the program transformation methodology should be supported by automatic (or semiautomatic) systems which may guide the programmer by suggesting the transformation rules to be applied according to some suitable strategies. Transformation systems may also incorporate in the form of new strategies the knowledge gained by the fact that some transformation sequences produce very efficient final programs.

1.2 The Schemata Approach

Together with the *rules + strategies* approach to program transformation, in the literature one also finds the so-called *schemata* approach. This 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, a *schema*) S is an abstraction via a substitution θ, of a program P, where some terms, goals, and clauses are replaced by metavariables, 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 values of the metavariables the corresponding instances P_1 and P_2 are equivalent programs w.r.t. *Sem*. The transformation from a schema S_1 to a schema S_2 is *correct* w.r.t. *Sem* iff S_1 and S_2 are equivalent w.r.t. *Sem*. Correctness of schema transformations has been studied within various formalisms (see, for instance, [87, 115, 151] for flowchart programs and recursive schemata).

Usually, we are interested in the correct transformation from a schema S_1 to a schema S_2 iff each instance of S_2 is more efficient of the corresponding instance of S_1.

Given an initial program P_1, the schemata approach to program transformation 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.

Some methodologies for developing logic programs using program schemata have been proposed by several authors (see, for instance, [48, 68, 71, 93, 108]) and some examples of logic program schema transformations can be found in [22, 23, 136]. The schema transformations presented in these papers deal with recursion removal and the reduction of nondeterminism in generate-and-test programs.

The schemata approach has been used also for deriving efficient functional programs (see, for instance, [10, 11]). In this case programs are denoted by applicative expressions and it is possible to improve their efficiency by replacing some subexpressions by more efficient, semantically equivalent new subexpressions. These equivalences are stated as theorems in suitable theories.

The schemata approach has been followed also in [138] where it is shown how to derive divide-and-conquer algorithms starting from a divide-and-conquer schema expressed in terms of abstract functions. The derived algorithms are correct if they are obtained by substituting some concrete functions for the abstract functions and these concrete functions satisfy some suitable logical formulas.

An advantage of the schemata approach over the rules + strategies approach is that the application of a schema transformation requires little time, because it is simply the application of a substitution. However, the process of choosing 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 and verifying suitable application conditions which are usually attached to each schemata transformation. On the other hand, the schemata approach has some drawbacks because it requires more space for storing the catalogue itself, and when the program to be transformed is not an instance of any schema in the catalogue, then no action can be performed.

1.3 An Introductory Example

Let us now present 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 use two other transformation rules, called *definition rule* and *goal replacement rule*, and we will use two transformation strategies, called *tupling* *strategy* and *generalization strategy*. We, indeed, need strategies for driving the application of the transformation rules and improving efficiency, because there are rules such as folding and unfolding which are inverses of each other, and thus, we may construct useless transformation sequences in which the final program is equal to the initial program.

Let us consider the following logic program P_0 for reversing a list:

1. $rev([\,],[\,]) \leftarrow$
2. $rev([H|T],R) \leftarrow rev(T,T1),\ append(T1,[H],R)$
3. $append([\,],Y,Y) \leftarrow$
4. $append([H|X],Y,[H|Z]) \leftarrow append(X,Y,Z)$

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.

From the list T, the atom $rev(T,T1)$ constructs the list $T1$ which is then traversed by $append(T1,[H],R)$. We may avoid this intermediate list by applying the tupling strategy which suggests the introduction of the following clause for the new predicate *revacc* (this name derives form the fact that *revacc* is a version of the reverse predicate with the extra accumulator argument Acc):

5. $revacc(T,Acc,R) \leftarrow rev(T,T1),\ append(T1,Acc,R)$

Actually, clause 5 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 2 the argument $[H]$ has been generalized to the variable Acc. In Section 3, 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 5 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 A, whose predicate occurs in P_0, belongs to the least Herbrand model of P_0 iff A belongs to the least Herbrand model of P_1.

In order to avoid the double occurrence of the list $T1$ in the body of clause 2, we now fold that clause using clause 5, that is, we replace '$rev(T,T1),\ append(T1,[H],R)$' which is an instance of the body of clause 5, by the corresponding instance '$revacc(T,[H],R)$' of the head of clause 5. Thus, we get:

2.f $rev([H|T],R) \leftarrow revacc(T,[H],R)$

This folding step is the inverse of the step of unfolding clause 2.f w.r.t. $revacc(T,[H],R)$. Unfortunately, if we use the program made out of clauses 1, 2.f, 3, 4, and 5, we do not avoid the construction of the list $T1$, because *revacc* is defined in terms of the two predicates *rev* and *append*. As we will show, a gain in efficiency is possible if we derive a definition of *revacc* in terms of *revacc* itself. This recursive definition of *revacc* can be obtained as follows.

We first unfold clause 5 w.r.t. $rev(T,T1)$, that is, we derive the following two resolvents of clause 5 using clauses 1 and 2, respectively:

6. $revacc([\,],Acc,Acc) \leftarrow$
7. $revacc([H|T],Acc,R) \leftarrow rev(T,T1),\ append(T1,[H],R1),\ append(R1,Acc,R)$

Now, in order to get a recursive definition of the predicate *revacc* where no intermediate lists are constructed, we would like to fold the entire body of clause 7 using clause 5, and in order to do so we need to have one occurrence only of the atom *append*. We can do so by applying the goal replacement rule (see Section 2) which allows us to replace the goal '$append(T1,[H],R1)$,

$append(R1, Acc, R)$' by the equivalent goal '$append(T1, [H|Acc], R)$', because in the least Herbrand model of P_0 the following holds:

$\forall T1, H, Acc, R.\ (\exists R1. append(T1, [H], R1), append(R1, Acc, R)) \leftrightarrow append(T1, [H|Acc], R)$

Thus, we get the following clause:

8. $revacc([H|T], Acc, R) \leftarrow rev(T, T1),\ append(T1, [H|Acc], R)$

We can fold clause 8 using clause 5, and we get:

8.f $revacc([H|T], Acc, R) \leftarrow revacc(T, [H|Acc], R)$

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

1. $rev([\ ], [\ ]) \leftarrow$
2.f $rev([H|T], R) \leftarrow revacc(T, [H], R)$
3. $append([\ ], Y, Y) \leftarrow$
4. $append([H|X], Y, [H|Z]) \leftarrow append(X, Y, Z)$
6. $revacc([\ ], Acc, Acc) \leftarrow$
8.f $revacc([H|T], Acc, R) \leftarrow revacc(T, [H|Acc], R)$

In this final program no intermediate lists are constructed, 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 *rev* and *append*. Notice that if we are interested in the computation of the predicate *rev* only, we can discard clauses 3 and 4 in the final program.

The crucial step in the above program transformation which improves the program performance, is the introduction of clause 5 defining the new predicate *revacc*. In the literature that step is referred to as a *eureka step* and the predicate *revacc* is also called a *eureka predicate*. In Section 3, we will consider the problem of inventing the eureka predicates in a semiautomatic way and we will see that eureka steps can often be performed on the basis of syntactical properties of the programs to be transformed by applying suitable transformation strategies.

2 Transformation Rules for Definite Logic Programs

In this section we first recall some basic notions regarding the syntax and the semantics of *definite logic programs*, we then introduce the rules which we use for transforming programs, and we finally present some results which ensure that these rules preserve the least Herbrand model semantics.

2.1 Syntax of Definite Logic Programs

The syntax of definite logic programs (simply called *programs*, when no confusion arises) is defined starting from the following infinite, pairwise disjoint sets:
(i) *variables*: $X, X_1, X_2, \ldots,$
(ii) *function symbols* (with arity): $f, f_1, f_2, \ldots,$ and
(iii) *predicate symbols* (with arity): $p, p_1, p_2, \ldots,$ including *true* (of arity 0), *false* (of arity 0), and $=$ (of arity 2).

Terms: $t, t_1, t_2, \ldots,$ *atoms*: $A, A_1, A_2, \ldots,$ and *goals*: $G, G_1, G_2, \ldots,$ have the following syntax:

$t ::= X \mid f(t_1, \ldots, t_n)$
$A ::= p(t_1, \ldots, t_m)$
$G ::= A \mid G_1, G_2$

The binary operator ',' which we have used in the definition of goals and we will also use below for programs and formulas, may be written as '$\wedge$'. This operator denotes *conjunction* and it is assumed to be associative with neutral element *true*. Thus, for instance, a goal G is the same as $(true, G)$ and $(G, true)$. We extend the binary operator ',' to an n-ary operator, and the conjunction of the goals G_i, for $i = 1, \ldots, n$, will also be written as $G_1, \ldots, G_n$. Similarly, we will consider n-ary conjunctions of programs and formulas. For $n = 0$ the n-ary conjunction is *true* and it is also called the *empty conjunction*.

Clauses $C, C_1, C_2, \ldots$ have the following syntax:

$C ::= p(t_1, \ldots, t_m) \leftarrow G$

where p is a predicate different from *true*, *false*, and $=$. $p(t_1, \ldots, t_m)$ is called the *head* of clause C and it is denoted by $hd(C)$. The goal G is called the *body* of C and it is denoted by $bd(C)$. A clause of the form: $A \leftarrow true$ is called a *unit clause*, and it is also written as: $A \leftarrow$.

Programs $P, P_1, P_2, \ldots$ have the following syntax:

$P ::= true \mid C \mid P_1, P_2$

First order formulas $\varphi, \varphi_1, \varphi_2, \ldots$ have the following syntax:

$\varphi ::= A \mid \varphi_1, \varphi_2 \mid \neg\varphi \mid \forall X.\varphi$

We also use the symbols $\vee, \rightarrow, \leftarrow, \leftrightarrow$, and $\exists$, which are defined as usual in terms of conjunction (','), negation ($\neg$), and universal quantification ($\forall$). Thus, goals, clauses, and programs are first order formulas. We also make use of the familiar notions of *free* and *bound* variables of a formula. A formula is said to be *closed* iff it has no free variables. $\vdash$ denotes the usual *consequence* relation.

Terms, atoms, goals, clauses, programs, and formulas are collectively called *expressions*, ranged over by $e, e_1, e_2, \ldots$. We also use the following terminology and notation. By $vars(e)$ we denote the set of variables occurring in expression e. By $freevars(e)$ we denote the set of free variables occurring in expression e.

By a *variable renaming* we mean a bijective mapping from the set of variables onto itself. We allow for the silent application of a variable renaming to a clause. Variable renaming preserves the least Herbrand model semantics (see below). Given a clause C, a *renamed apart* clause C' is any clause obtained from C by applying a variable renaming, so that the variables of C' do not appear in any of the expressions considered so far. In this sense we say that the variables of C' are *new* variables.

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

We will say that a predicate p *depends on* a predicate q in program P iff either there exists in P a clause of the form: $p(\ldots) \leftarrow G$ such that q occurs in the goal G 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 goal G depends on a clause of the form: $p(\ldots) \leftarrow G_1$ in a program P iff either p occurs in G or there exists a predicate occurring in G which depends on p.

By a *predicate renaming* we mean a bijective mapping from the set of predicate symbols of the language 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 also be adopted for the predicate renaming in goals and clauses.

For the notions of *substitution*, *instance*, *most general unifier* (*mgu*, for short), *ground expression*, *ground substitution*, and for other notions not defined here, we refer to [103]. The most general unifier of two terms t_1 and t_2 is denoted by $mgu(t_1, t_2)$. Similar notation is adopted for the mgu of two atoms.

2.2 The Least Herbrand Model

The declarative semantics of a program is its least Herbrand model which is defined as we now indicate (see also [103]).

The *Herbrand base* is the set $\mathcal{B}$ of all ground atoms with predicate different from *true*, *false*, and $=$. An *Herbrand interpretation* is a subset of $\mathcal{B}$.

Given an Herbrand interpretation I, a closed first order formula φ *is true in* I, or I is a *model* of φ, written $I \models \varphi$, iff one of the following holds:

(i) φ is the atom *true*,

(ii) φ is a ground atom of the form $t = t$ for some ground term t,

(iii) φ is a ground atom belonging to I,

(iv) φ is a conjunction (φ_1, φ_2) and the closed formulas φ_1 and φ_2 are both true in I,

(v) φ is $\neg\varphi_1$ and it is not the case that φ_1 is true in I,

(vi) φ is $\forall X.\varphi_1$ and for all ground terms t, $\varphi_1\{X/t\}$ is true in I.

We say that φ *is false in* I iff it is not the case that φ is true in I. An Herbrand interpretation I is an *Herbrand model* of program P iff $I \models \forall X_1, \ldots, X_n. P$, where $\{X_1, \ldots, X_n\} = vars(P)$. We have the following important property.

Theorem 1 *For any program P there exists an Herbrand model of P which is least w.r.t. set inclusion.*

The least Herbrand model of program P is denoted by $M(P)$.

2.3 The Clark Completion Semantics

No negative conclusion can be inferred from a program P by means of the usual logical consequence relation, that is, no atom A exists in the Herbrand base such that $P \vdash \neg A$. It is often useful for knowledge representation purposes, to take into consideration the so-called *Clark completion* of a program P [103], denoted $comp(P)$, which is a first order formula such that: (i) $comp(P) \vdash A$ iff $P \vdash A$ and (ii) if $comp(P) \vdash \neg A$ then it is not the case that $P \vdash A$ (denoted $P \not\vdash A$).

The Clark completion of a program P is constructed as follows.

(1) For each clause of P:

(1.1) Replace every term in the arguments of the head in favour of *new* variables and add suitable equalities in the body, so that all clauses in P with the same predicate symbol in the head, have identical heads.

For instance, the conjunction $(p(t,u) \leftarrow G_1),\ (p(r,f(V)) \leftarrow G_2)$ of two clauses is transformed into $(p(X,Y) \leftarrow X{=}t,\ Y{=}u,\ G_1),\quad (p(X,Y) \leftarrow X{=}r,\ Y{=}f(V),\ G_1)$.

(1.2) Existentially quantify the body w.r.t. the variables occurring in the body and not in the head.

For instance, if $vars(t,u,G_1) = \{W,Z\}$, then $p(X,Y) \leftarrow X=t, Y=u, G_1$ is transformed into $p(X,Y) \leftarrow \exists W,Z.(X{=}t, Y{=}u, G_1)$.

(2) Fuse all clauses with equal head into one implication.
For instance, the conjunction $(p(X,Y) \leftarrow Body_1),\ (p(X,Y) \leftarrow Body_2)$ of clauses is transformed into the implication $p(X,Y) \leftarrow (Body_1 \vee Body_2)$.

(3) Universally quantify every implication derived at Step (2) w.r.t. the variables occurring in its conclusion and replace $\leftarrow$ by $\leftrightarrow$.
For instance, $p(X,Y) \leftarrow (Body_1 \vee Body_2)$ is transformed into $\forall X,Y.\ (p(X,Y) \leftrightarrow (Body_1 \vee Body_2))$.

(4) Introduce a formula of the form: $\forall X_1,\ldots,X_n.\neg r(X_1,\ldots,X_n)$ for each n-ary predicate symbol r occurring in the body of a clause of P, and not in any head of P.

(5) Introduce the conjunction of the following formulas of the first order predicate calculus, which define the so called *Clark Equality Theory*, or CET, for short.

(5.1) For each function symbol f,

$\forall X_1,\ldots,X_n,Y_1,\ldots,Y_n.\ (X_1 = Y_1,\ldots,X_n = Y_n) \leftrightarrow f(X_1,\ldots,X_n) = f(Y_1,\ldots,Y_n)$.

(5.2) For each pair of distinct function symbols f and g (possibly with different arities),

$\forall X_1,\ldots,X_n,Y_1,\ldots,Y_m.\ \neg(f(X_1,\ldots,X_n) = g(Y_1,\ldots,Y_m))$.

(5.3) For each predicate symbol p,

$\forall X_1,\ldots,X_n,Y_1,\ldots,Y_n.\ (X_1 = Y_1,\ldots,X_n = Y_n) \rightarrow (p(X_1,\ldots,X_n) \leftrightarrow p(Y_1,\ldots,Y_n))$.

(5.4) $\forall X.\ X = X$.

(5.5) For each term t such that $vars(t) = \{X,X_1,\ldots,X_n\}$ and t is not X,

$\forall X,X_1,\ldots,X_n.\ \neg(t = X)$.

The completion $comp(P)$ of P is the conjunction of:
- the formulas derived at Step (3), denoted by $\mathit{iff}(P)$,
- the formulas introduced at Step (4), and
- the formulas introduced at Step (5), that is, CET.

For any program P, the following properties hold:

(i) CET is a complete, decidable theory for the set of closed first order formulas whose predicate symbols are *true*, *false*, and $=$ only

(ii) CET $\vdash \exists X_1, \ldots, X_n. (t_1 = t_2)$ iff t_1 and t_2 are unifiable, where $\{X_1, \ldots, X_n\} = vars(t_1, t_2)$

(iii) $M(P) \models comp(P)$

(iv) for any atom A in the Herbrand base, $A \in M(P)$ iff $comp(P) \vdash A$

(v) for any atom A in the Herbrand base, if $comp(P) \vdash \neg A$ then $A \notin M(P)$ and, thus, $P \not\vdash A$.

2.4 Operational Semantics

For a given goal G, a logic program P may compute an *answer* which is either a substitution for the variables in the goal G, or the atom *false*. To model this behaviour, we define our operational semantics as a ternary relation $P\colon G \mapsto \alpha$, where α is either a substitution or *false*. This relation is defined inductively by the following rules. (For an introduction to the theory of inductively defined operational semantics the reader may refer to [154].) In these rules, (i) the composition of two substitutions μ and α [103] is denoted by $\mu\alpha$, (ii) ε denotes the *empty substitution* (that is, the identity w.r.t. composition of substitutions), and (iii) $\sigma|V$ denotes the *restriction* of the substitution σ to the set of variables V, that is, the set of bindings X/t in σ such that X belongs to V.

(*true*) $$\frac{}{P\colon true \mapsto \varepsilon}$$

(*false*) $$\frac{}{P\colon G_1, false, G_2 \mapsto false}$$

(*eq*1) $$\frac{P\colon (G_1, G_2)\mu \mapsto \alpha}{P\colon (G_1, t_1 = t_2, G_2) \mapsto \mu\alpha}$$

where t_1 and t_2 are unifiable, $\mu = mgu(t_1, t_2)$, and α is a substitution (i.e., α is different from *false*).

(*eq*2) $$\frac{}{P\colon (G_1, t_1 = t_2, G_2) \mapsto false}$$ where t_1 and t_2 are not unifiable.

(*res*) $$\frac{P\colon (G_1, bd(C), G_2)\mu \mapsto \alpha}{P\colon (G_1, A, G_2) \;\mapsto\; (\mu\alpha)|vars(G_1, A, G_2)}$$

where C is a clause in P such that A is unifiable with $hd(C)$, $\mu = mgu(A, hd(C))$, and α is a substitution (i.e., α is different from *false*).

$$(\mathit{ff}) \qquad \frac{P\colon (G_1, bd(C_1), G_2)\mu_1 \mapsto \mathit{false} \quad \ldots \quad P\colon (G_1, bd(C_k), G_2)\mu_k \mapsto \mathit{false}}{P\colon (G_1, A, G_2) \mapsto \mathit{false}}$$

where $C_1, \ldots, C_k$ are all clauses in P whose head is unifiable with the atom A and, for $i = 1, \ldots, k$, $\mu_i = mgu(A, hd(C_i))$.

In the rules (*res*) and (*ff*) we assume that every clause of P is renamed apart.

We say that, for a goal G, the program P *existentially terminates*, written $P \downarrow G$, iff there exists α, which is either a substitution or *false*, such that $P\colon G \mapsto \alpha$.

We have the following relationships between the least Herbrand model, the Clark completion, and the operational semantics.

Theorem 2 *For any program P and atom A in the Herbrand base,*
(i) $P \vdash A$ *iff* $A \in M(P)$ *iff* $Comp(P) \vdash A$ *iff* $P\colon A \mapsto \varepsilon$,
(ii) if $Comp(P) \vdash \neg A$ *then* $A \notin M(P)$,
(iii) $Comp(P) \vdash \neg A$ *iff* $P\colon A \mapsto \mathit{false}$.

2.5 Transformation Rules

Our transformation rules have the following form: given two conjunctions γ and γ_1 of clauses, we derive a new conjunction γ_2 of clauses by applying a transformation rule to γ_1 using γ.

As already mentioned, the process of program transformation can be viewed as the construction of a sequence of programs, called *transformation sequence*. A transformation sequence $P_0, \ldots, P_n$ is constructed from a given initial program P_0 by applications of transformation rules in $\{\mathrm{R1}, \ldots, \mathrm{R8}\}$ as follows. For $k = 0, \ldots, n-1$, we denote by $Defs_k$ the conjunction of all clauses which are derived by the definition introduction rule (see rule R1 below) during the construction of $P_0, \ldots, P_k$. Obviously, $Defs_0$ is the empty conjunction *true*. Program P_{k+1} is derived by: (i) selecting a (possibly empty) conjunction γ_1 of clauses in P_k, (ii) selecting a (possibly empty) conjunction γ of clauses in $P_0 \wedge Defs_k$, (iii) deriving a new conjunction γ_2 of clauses by applying a transformation rule to γ_1 using γ, and (iv) replacing γ_1 by γ_2 in P_k.

The transformation rules, with the only exception of Definition Introduction, establish in a natural way a *derivability* relation between clauses, denoted by $\Rightarrow$, such that, if we obtain γ_2 by applying a transformation rule to γ_1 using γ, then every clause in γ_2 is derivable from a clause in γ_1. We will also use the reflexive-transitive closure of $\Rightarrow$, written $\Rightarrow^*$.

Now, we assume that we have constructed from the initial program P_0 the sequence $P_0, \ldots, P_k$ of programs and we construct the program P_{k+1} which is the next one in the sequence, by applying one of the following rules R1–R8, collectively called *unfold/fold transformations*.

R1. Definition Introduction. We introduce the following n (≥ 1) clauses, called *definitions*:

$$\begin{cases} newp(X_1, \ldots, X_h) \leftarrow Body_1 \\ \quad \ldots \\ newp(X_1, \ldots, X_h) \leftarrow Body_n \end{cases}$$

where: (i) *newp* is a new predicate symbol, that is, it does not occur in $P_0 \wedge Defs_k$, (ii) for $1 \leq i \leq h$, the variable X_i occurs in $Body_j$ for some j such that $1 \leq j \leq n$, and (iii) for $1 \leq j \leq n$, all predicates occurring $Body_j$ also occur in the initial program P_0. Notice that for $1 \leq j \leq n$, a variable occurring in $Body_j$ need not be in $\{X_1, \ldots, X_h\}$.
By definition we derive the new program P_{k+1} and the new conjunction $Defs_{k+1}$ of definitions by adding the above n clauses to P_k and $Defs_k$, respectively.

R2. Unfolding. Let C be a renamed apart clause in program P_k of the form: $H \leftarrow G_1, A, G_2$, where A is an atom whose predicate is not in the set $\{true, false, =\}$, and G_1 and G_2 are (possibly empty) goals. Let $E_1, \ldots, E_n$, with $n \geq 0$, be the clauses of $P_0 \wedge Defs_k$ such that for $i = 1, \ldots, n$, A is unifiable with the head of E_i with mgu θ_i. For $i = 1, \ldots, n$, let C_i be the clause $(H \leftarrow G_1, bd(E_i), G_2)\theta_i$. Then by unfolding C w.r.t. A using $E_1, \ldots, E_n$ we derive clauses $C_1, \ldots, C_n$ and, for $i = 1, \ldots, n$, we have that: $C \Rightarrow C_i$. We derive program P_{k+1} by replacing C in P_k by $C_1, \ldots, C_n$.

Notice that an application of the unfolding rule to clause C in P_k amounts to the deletion of C from P_k if $n=0$. Sometimes in the literature this particular case is treated as an extra rule.

R3. Folding. Let $C_1, \ldots, C_n$ be clauses in P_k. Let $D_1, \ldots, D_n$ be clauses in $Defs_k$ of the form: $newp(X_1, \ldots, X_h) \leftarrow Body_1, \ldots, newp(X_1, \ldots, X_h) \leftarrow Body_n$, and let these clauses be the definition of $newp$ in $Defs_k$. Suppose that there exist two goals G_1 and G_2, and a substitution θ such that for $i = 1, \ldots, n$, the following two conditions hold:
(1) C_i is a variant of the clause $H \leftarrow G_1, Body_i\theta, G_2$, and
(2) for every variable X in D_i and not in $\{X_1, \ldots, X_h\}$, we have that: (2.i) $X\theta$ is a variable which does not occur in (H, G_1, G_2), and (2.ii) $X\theta$ does not occur in $Y\theta$, for any variable Y occurring in $Body_i$ and different from X.
By folding $C_1, \ldots, C_n$ using $D_1, \ldots, D_n$ we derive the clause E: $H \leftarrow G_1, newp(X_1, \ldots, X_h)\theta, G_2$ and, for $i=1, \ldots, n$, we have that: $C_i \Rightarrow E$. We derive program P_{k+1} by replacing $C_1, \ldots, C_n$ in P_k by E.

For instance, the clauses C_1: $p(X) \leftarrow q(t(X), Y), r(Y)$ and C_2: $p(X) \leftarrow s(X), r(Y)$ can be folded (via the substitution $\theta = \{U/X, V/Y\}$) using the clauses D_1: $a(U, V) \leftarrow q(t(U), V)$ and D_2: $a(U, V) \leftarrow s(U)$, and we replace C_1 and C_2 by the clause C: $p(X) \leftarrow a(X, Y), r(Y)$.

R4. Goal Replacement. Let C be a renamed apart clause in P_k of the form $H \leftarrow G_1, Q, G_2$, where Q, G_1, and G_2 are (possibly empty) goals. Suppose that, for some goal R,

$$M(P_0 \wedge Defs_k) \models \forall X_1 \ldots X_u.\ (\exists Y_1 \ldots Y_v.\ Q \leftrightarrow \exists Z_1 \ldots Z_w. R)$$

where:
(i) $\{Y_1, \ldots, Y_v\} = vars(Q) - vars(H, G_1, G_2)$,
(ii) $\{Z_1, \ldots, Z_w\} = vars(R) - vars(H, G_1, G_2)$, and
(iii) $\{X_1, \ldots, X_u\} = vars(Q, R) - \{Y_1, \ldots, Y_v, Z_1, \ldots, Z_w\}$.
Suppose also that all predicates different from *true*, *false*, and $=$ which occur in R, also occur in $P_0 \wedge Defs_k$.
By goal replacement from C we derive the clause E: $H \leftarrow G_1, R, G_2$, and we have that: $C \Rightarrow E$. From program P_k we derive the new program P_{k+1} by replacing C by E.

R5. Clause Removal. Let C be a clause in P_k of the form: $H \leftarrow G_1, false, G_2$. We derive the new program P_{k+1} by removing C from P_k.

R6. Rearrangement. We derive the new program P_{k+1} from P_k by rearranging the order of the clauses and the order of the goals in the bodies of the clauses.

We say that a clause of the form $(H \leftarrow Body, G)\theta$ is *subsumed* by the clause $H \leftarrow Body$. We also say that a goal $G\theta$ is subsumed by the goal G.

R7. Subsumption. Let C be a clause in P_k.
(i) From clause C we derive a clause D which is subsumed by C and we have that: $C \Rightarrow D$. We derive P_{k+1} by removing from, or adding to, P_k clause D.

(ii) From clause C we derive a clause D by removing from, or adding to, the body of C a goal which is subsumed by another goal in the body C, and we have that: $C \Rightarrow D$. We derive P_{k+1} by replacing C by D.

R8. Equality Introduction and Elimination. Let C be a clause of the form $(H \leftarrow Body)\{X/t\}$, such that the variable X does not occur in t and let D be the clause: $H \leftarrow X = t, Body$. By *equality introduction* we derive clause D from clause C, and we have that: $C \Rightarrow D$. By *equality elimination* we derive clause C from clause D, and we have that: $D \Rightarrow C$.
If C occurs in P_k then we derive the new program P_{k+1} by replacing C by D. If D occurs in P_k then we derive the new program P_{k+1} by replacing D by C.

2.6 Correctness Results for Definite Programs

In this section we present some sufficient conditions for the correctness of the transformation rules w.r.t. the least Herbrand model semantics. A transformation sequence $P_0, \ldots, P_n$ is said to be *partially correct* iff $M(P_0 \wedge Defs_n) \supseteq M(P_n)$. $P_0, \ldots, P_n$ is said to be *totally correct*, or *correct*, for short, iff $M(P_0 \wedge Defs_n) = M(P_n)$.

All transformation sequences are partially correct. This result is based on the following property which establishes an invariant preserved by the transformation rules.

Lemma 1 *Let $P_0, \ldots, P_n$ be a transformation sequence. Then, for $k = 0, \ldots, n$, $M(P_0 \wedge Defs_k) \models P_k$.*

Theorem 3 (Partial correctness of a transformation sequence) *Let $P_0, \ldots, P_n$ be a transformation sequence constructed by using the transformation rules* R1–R8. *Then $M(P_0 \wedge Defs_n) \supseteq M(P_n)$.*

In general a transformation sequence is not totally correct, as shown by the following example.

Example 1 Let us consider the program P_0: $p \leftarrow q \quad \wedge \quad q \leftarrow$. Since $M(P_0) \models p \leftrightarrow q$, by goal replacement we get P_1: $p \leftarrow p \quad \wedge \quad q \leftarrow$. We have that $M(P_0) = \{p, q\} \neq M(P_1) = \{q\}$. □

In order to construct totally correct transformation sequences we need to restrict the use of the transformation rules. We will consider restrictions based on program *termination* (Section 2.6.1) and the *non-ascending* property (Section 2.6.2).

2.6.1 Transformations Which Preserve Termination

We say that a transformation sequence $P_0, \ldots, P_n$ is *termination preserving* iff for all ground atoms A, if $(P_0 \wedge Defs_n) \downarrow A$ then $P_n \downarrow A$.

The proof of Theorem 4 below is based on the following invariant.

Lemma 2 *Let $P_0, \ldots, P_n$ be a transformation sequence. Then, for $k = 0, \ldots, n$, $M(P_0 \wedge Defs_k) \models comp(P_k)$.*

Theorem 4 (Total correctness of termination preserving transformation sequences) *Let $P_0, \ldots, P_n$ be a termination preserving transformation sequence constructed by using the transformation rules* R1–R8. *Then $M(P_0 \wedge Defs_n) = M(P_n)$.*

2.6.2 Non-ascending Transformations

In order to prove the total correctness result of this section, we assume that the set of predicate symbols of the language we consider, is partitioned into two disjoint sets: (i) the *basic* predicates, including *true*, *false*, and '=', and (ii) the *non-basic* predicates. We assume that in every program P a basic predicate does not depend on any non-basic predicate. We also assume that the head predicate of each clause introduced by the definition rule is non-basic. Atoms with non-basic predicates will be called non-basic atoms. Atoms and goals with basic predicates only will be called *basic atoms* and *basic goals*, respectively.

The partition into basic and non-basic predicates is needed for proving Theorem 5 below.

Let us consider the operational semantics of Section 2.4. A *proof* of $P: G \mapsto \alpha$ is a finite tree π constructed by using the rules listed in Section 2.4, such that: (i) the root of π is $P: G \mapsto \alpha$, and (ii) the leaves of π are instances of the conclusions of the rules (*true*) and (*false*). Given a proof π of $P: G \mapsto \alpha$, with $\alpha \neq \mathit{false}$, we denote by $\lambda(\pi)$ the number of applications of rule (*res*) during the construction of π using a clause with non-basic head predicate.

Given a goal G and a program P, let $\Pi(G, P) = \{\pi \mid \pi \text{ is a proof of } P: G \mapsto \alpha, \text{ with } \alpha \neq \mathit{false}\}$. We define

$$\mu(G,P) = \begin{cases} \mathit{min}\{\lambda(\pi) \mid \pi \in \Pi(G,P)\} & \text{if } \Pi(G,P) \neq \emptyset \\ \infty \quad \text{otherwise} \end{cases}$$

Definition 1 (Non-ascending goal replacement) *With reference to rule* R4*, we say that the replacement of goal Q by goal R is non-ascending iff for each ground substitution θ of the form $\{X_1/t_1, \ldots, X_u/t_u\}$, we have that: $\mu(Q\theta, P_0 \wedge \mathit{Defs}_k) \geq \mu(R\theta, P_0 \wedge \mathit{Defs}_k)$.*

An obvious way of making sure that a goal replacement is non-ascending is to assume that R is a basic goal.

Theorem 5 (Total correctness of non-ascending transformation sequences) *Let P_0, $\ldots$, P_n be a transformation sequence constructed by using the transformation rules* R1–R8 *with the following restrictions:*

(α) *the folding rule is applied for the derivation of clause E from clauses $C_1, \ldots, C_m$ in program P_k, with $0<k<n$, using clauses $D_1, \ldots, D_m$, only if*

- *either the head predicate of $C_1, \ldots, C_m$ is not a basic predicate and it occurs in P_0*
- *or for every $i \in \{1, \ldots, m\}$, there exist two clauses, say A in P_j and B in P_{j+1}, for some $j < k$, such that* (1) *B is derived from A by unfolding w.r.t. a non-basic atom and* (2) *$B \Rightarrow^* C_i$, and*

(β) *all goal replacements are non-ascending and they are applied to clauses with a non-basic head predicate.*

Then $M(P_0 \wedge \mathit{Defs}_n) = M(P_n)$.

2.7 Bibliographic Remarks

The unfold/fold transformation rules for definite logic programs have been introduced by Tamaki and Sato [144], who proved their correctness w.r.t. the least Herbrand model semantics. Later, several extensions of the original Tamaki and Sato's rules have been studied. It has also been proved that, under suitable hypotheses, the unfold/fold rules preserve several semantics of definite logic programs, such as computed answers, finite failure, pure Prolog, and left-termination (see [14, 39, 77, 92, 100, 105, 123]).

The transformational approach has also been considered in the case of various extensions of definite logic programs, such as logic programs with *negation* (see Section 5), *constraint logic programs* [8, 64, 106], and *concurrent constraint programs* [65, 148].

General methods for proving the correctness of transformation rules are studied in [133] for the case of higher order functional languages, and [126] for programming languages whose semantics is specified by inductive definitions, like that of Section 2.4.

3 Strategies for Transforming Definite Logic Programs

The transformation process should be directed by some metarules, which we call *strategies*, because, as we have seen in Section 2, 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 [58, 67, 113, 117, 118] one can find treatments of the transformation strategies both for functional and logic programs.

In order to simplify our presentation, sometimes we will not mention the use of the rearrangement rule R6 and the subsumption rule R7. Since these rules preserve the least Herbrand model semantics, we may assume that conjunction, besides being associative, is commutative and idempotent. Thus, we will feel free to use set-theoretic notations, so that a goal is also written as a set of atoms and the conjunction of two goals is also formed by making set union. For notions concerning logic programming and not defined here, the reader may refer to [103].

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

Suppose that we are given an initial program and we want to transform it for improving 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. In order to improve the performance of the initial program, 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 19). Sometimes we also need a simultaneous application of the *generalization strategy* (page 20) whereby an argument of a predicate may be promoted to a variable. 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 occurrences of *false* in the body, or we may delete subsumed atoms and clauses, thereby avoiding redundant computations.

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*.

This recursive definition can often be obtained by performing a folding step using the clause initially introduced by tupling and, possibly, generalization.

An example of this process is the derivation of the program for reversing a list we have presented in Section 1. Indeed, by analyzing the initial program P_0 at page 6, we have discovered that the evaluation of the body of the clause

3. $rev([H|T], R) \leftarrow rev(T, T1),\ append(T1, [H], R)$

is inefficient because it determines the construction of an intermediate list $T1$. Then, by applying the tupling and generalization strategies, we have introduced the clause

5. $revacc(T, Acc, R) \leftarrow rev(T, T1),\ append(T1, Acc, R)$

and we have used this clause 5 for folding clause 2. Then, we have unfolded clause 5 and this gave us the opportunity of replacing two calls of *append* by one call only, via the goal replacement rule (see clause 8). Finally, we have iterated this efficiency improvement by finding a recursive definition of *revacc*. This definition is provided by clause 8.f, obtained by folding clause 8 using clause 5.

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 3.1 below.

In [47] 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 metastrategy. 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 [18] and program synthesis [59], where tactics for inductive proofs and inductive synthesis are driven by the need of applying suitable inductive hypotheses.

3.1 Basic Strategies based on Unfolding Trees

We now describe some basic strategies which have been introduced in the literature for transforming logic programs. They are: (S1) tupling, (S2) loop absorption, and (S3) generalization. The ideas underlying these strategies come from the early days of program transformation and they were already present in [29].

The tupling strategy was formally defined in [116] 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 [122] 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 [147], compiling control [25], as well as various techniques for partial evaluation (see Section 3.3).

Finally, the generalization strategy has its origin in the theorem proving area [18], 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 section as building blocks to describe a number of more complex transformation techniques.

For a formal description of these strategies and their mechanization we now introduce the notion of *unfolding tree*. It represents the process of transforming a given clause by performing unfolding and goal replacement steps. This notion is also related to the one of *symbolic trace tree* of [25], where, however, the goal replacement rule is not taken into account.

Definition 2 (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 goal replacement from D.

In an unfolding tree we also have the usual relations of 'descendant node' (or descendant clause) and 'ancestor node' (or ancestor 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 3 (Unfolding Selection Function) *An* unfolding selection function (*or* u-selection function*, 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

$C: \quad H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$

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

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

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 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 for variables.

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

$C: \quad H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$

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 (using subsumption), to $B_1, \ldots, B_n$. We introduce a new predicate defined by the following clause A:

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

where $\{X_1, \ldots, X_k\}$ is the minimum subset of $vars(A_1, \ldots, A_m)$ which is necessary to perform a folding step of both C and D, using a single clause whose body is $A_1, \ldots, A_m$. This minimum subset is determined by Condition (2) for the applicability of the folding rule R3 (page 13). 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 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

$C: \quad H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$

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

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

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 3, page 25). 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 (using subsumption), 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 $GenA_1, \ldots, GenA_m$.

3.2 Techniques Which Use Basic Strategies

In this section we will present some techniques which improve program efficiency by making use of the strategies which we have described in the previous Section 3.1. We will present the Compiling Control technique and the Program Composition technique.

3.2.1 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 [97]. 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 [103] 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 taking into account any information derived from the program. Much work has been done in the direction of improving the control strategy of logic languages (see, for instance, [26, 110]).

We consider here a transformation technique, called *compiling control* [25], 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 [50, 134] as well as lazy evaluation and coroutining [111]. 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 computation rule of Prolog. 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 become available.

A similar idea has also been investigated in the area of functional programming, within the so-called *filter promotion* strategy [9, 46]. 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 [23, 136].

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. the least Herbrand model semantics and behaves using the left-to-right computation rule as P_1 does using the rule C [25].

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 some 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 generalization strategy S3 (page 20).

Without loss of generality, we assume that every query in Q is of the form $\leftarrow q(\ldots)$ and in P_1 there exists exactly 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 function 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 [43] for a formalization of the techniques of *abstract interpretation*, and to [48] for a method based on abstract interpretation for generating the tree T in a semiautomatic way.

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

Example 2 (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 function 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

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

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

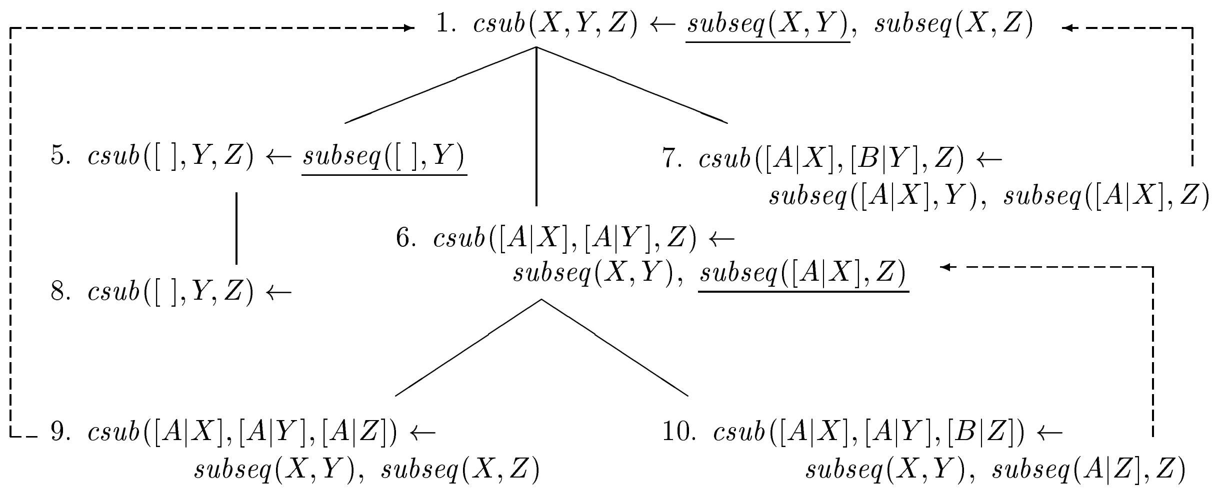

Figure 2: An unfolding tree for $\langle Csub$, clause $1\rangle$ using the computation rule U_C. We have underlined the atoms selected for unfolding.

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

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

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

and by folding we get

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

The final program is made out of the following clauses:

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

The correctness of the above transformation can easily be proved by applying Theorem 5 with the assumption that *newcsub*, *csub*, and *subseq* are non-basic predicates. In particular, the folding step which generates clause 6.f from clause 6 using clause 11, satisfies the conditions of that theorem, because clause 6 has been derived from clause 1 by unfolding w.r.t. the atom $subseq(X,Y)$. Similar conditions ensure the correctness of the other 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 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

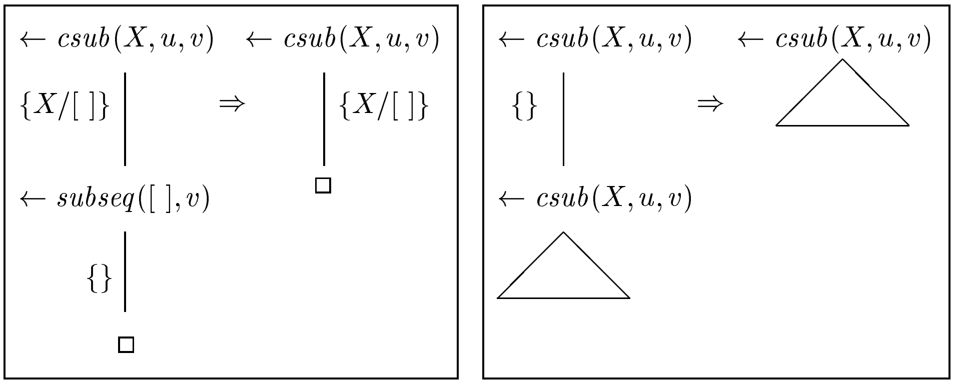

Figure 3: Tree rewritings for the SLD-tree T_1.

performing on the derived tree the rewritings shown in Fig. 3 for any unbound variable X and ground lists u and v. □

3.2.2 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 3.2.1, 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 *finite differencing* [112], *composition* or *deforestation* [66, 150], and *tupling* [116]. (See also [67, 113, 118] for surveys.)

For logic programs two main methods have been considered: *loop fusion* [52] and *elimination of unnecessary variables* [125]. 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 [125] 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 the elimination of unnecessary variables are often similar to those of loop fusion.

In the following example we recast the loop fusion and the elimination of unnecessary variables in terms of the strategies presented in Section 3.1.

Example 3 (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

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

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:

1.f $deletemax(Xs,Ys) \leftarrow gen(Xs,M,M,Ys)$
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 $gen(Xs,N,X,Ys)$ with Ys being an unbound variable.

This can be done by the application of the so called *clause fusion* [54, 58]. This technique can be mimicked by applying our transformation rules as follows.

We first perform two equality introduction steps followed by a goal rearrangement step and we transform clauses 12 and 13 into:

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, 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

1.f $deletemax(Xs, Ys) \leftarrow gen(Xs, M, M, Ys)$
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. $aux(Q, X, [X|Ys], Ys) \leftarrow Q \neq X$

□

3.3 Program Specialization

The goal of *program specialization* [88] is the automatic adaptation of a generic program to a specific context of use. We will present here two techniques for program specialization: (1) *partial evaluation* [88, 44] and (2) an enhancement of partial evaluation for reducing nondeterminism while specializing logic programs [120]. The structure of the two techniques is similar, but the second one uses more powerful transformation rules and strategies.

3.3.1 Partial Evaluation

Partial evaluation is a well known technique for program specialization which, from a program and its *static input* (that is, a subset of the input data which are known at compile time), derives a new, more efficient program by pre-computing the parts of the output which depend on the static input. Partial evaluation has been applied in several areas of computer science, and it has been applied also to logic programs [74, 101, 104], where it is also called *partial deduction*. In our approach, partial evaluation can be viewed as a transformation strategy for constructing a transformation sequence, say $P_0, \ldots, P_n$, where P_0 is the program to be specialized, P_n is the specialized program, and every program of this sequence is obtained from the previous one by applying a transformation rule (see Section 2).

As it is shown in [121, 132], partial evaluation of logic programs can be viewed as a rule-based program transformation technique using the definition introduction, unfolding, and folding rules of Section 2 with the following two restrictions: (1) each new predicate introduced by the definition rule is defined by precisely one non-recursive clause whose body consists of precisely one atom (in this sense, according to the terminology of [80], partial evaluation of logic programs is said to be *monogenetic*), and (2) the folding rule uses only clauses introduced by the definition rule. In this paper we call *atomic definition* and *atomic folding* the definition and folding rules, respectively, with restrictions (1) and (2).

The use of these restricted transformation rules eases the design of fully automated partial evaluation techniques, but it may limit the program improvements which can be achieved during program specialization.

In this section we will use the following notation. *Atoms* and *Clauses* are the sets of all atoms and all clauses, respectively. *Clauses** is the set of all finite sequences of clauses and $\mathcal{P}(\mathit{Clauses})$ is the powerset of *Clauses*.

Our partial evaluation strategy iterates two phases: (1) an *Unfold* phase, during which we derive conjunctions of clauses by the unfolding rule, and (2) a *Define-Fold* phase, during which we introduce new atomic definitions and we fold the clauses derived during the previous *Unfold* phase.

The partial evaluation strategy requires the following two functions:

(1) An *unfolding selection function* $\mathit{Select} : \mathit{Clauses}^* \times \mathit{Clauses} \longrightarrow \mathit{Atoms} \cup \{\mathit{halt}\}$, such that for any sequence $C_1, \ldots, C_n$ of clauses and for any clause C, $\mathit{Select}((C_1, \ldots, C_n), C)$ either is an

atom in the body of C or it is the special value *halt*. (This notion is slightly different from the one of Definition 3 where the unfolding selection function depends on the unfolding tree.)

During the *Unfold* phase the *Select* function is used to construct sequences of clauses as follows: given a sequence $C_1, \ldots, C_n, C_{n+1}$ of clauses such that, for $i = 1, \ldots, n$, C_{i+1} is derived by unfolding C_i, if $\mathit{Select}((C_1, \ldots, C_n), C) \neq \mathit{halt}$, then we append to $C_1, \ldots, C_n, C_{n+1}$ the clause C_{n+2} derived by unfolding C_{n+1} w.r.t. the atom $\mathit{Select}((C_1, \ldots, C_n), C_{n+1})$, otherwise we halt the construction. This explains why in the partial evaluation strategy below we have called $\mathit{Ancestors}(C)$ the first argument of *Select*.

(2) A *generalization function* $\mathit{Gen} : \mathcal{P}(\mathit{Clauses}) \times \mathit{Atoms} \longrightarrow \mathit{Clauses}$, such that for any set *Defs* of atomic definitions and for any atom A, $\mathit{Gen}(\mathit{Defs}, A)$ is a clause of the form $g(X_1, \ldots, X_h) \leftarrow \mathit{GenA}$, where (i) $\{X_1, \ldots, X_h\} = \mathit{vars}(\mathit{GenA})$, (ii) A is an instance of GenA, and (iii) either $\mathit{Gen}(\mathit{Defs}, A)$ belongs to *Defs* or g is a new predicate, that is, it occurs neither in P nor in *Defs*.

The generalization function *Gen* is used during the *Define-Fold* phase of the partial evaluation strategy as follows. When we want to fold a clause C w.r.t. an atom A in its body, we consider the set *Defs* of all atomic definitions introduced so far during the transformation sequence and we apply the folding rule using $\mathit{Gen}(\mathit{Defs}, A)$. This application of the folding rule is indeed possible because, by construction, A is an instance of the body of $\mathit{Gen}(\mathit{Defs}, A)$.

In the Partial Evaluation Strategy presented below, we assume that the only basic predicates are *true*, *false*, and $=$.

Partial Evaluation Strategy.
Input: A program P and an atom $p(t_1, \ldots, t_h)$ w.r.t. which we want to specialize P.
Output: A program P_{pe} and an atom $p_{pe}(X_1, \ldots, X_r)$, with $\{X_1, \ldots, X_r\} = \mathit{vars}(p(t_1, \ldots, t_h))$, such that for every ground substitution $\theta = \{X_1/u_1, \ldots, X_r/u_r\}$, $M(P) \models p(t_1, \ldots, t_h)\theta$ iff $M(P_{pe}) \models p_{pe}(X_1, \ldots, X_r)\theta$.

Initialize: Let S be the clause $p_{pe}(X_1, \ldots, X_r) \leftarrow p(t_1, \ldots, t_h)$ and let $\mathit{Ancestors}(S)$ be the empty sequence of clauses.
$\mathit{Defs} := \{S\};\ \ P_{pe} := \emptyset;\ \ \mathit{Cls} := \{S\};$

while $\mathit{Cls} \neq \emptyset$ **do**

(1) *Unfold*:

while there exists a clause $C \in \mathit{Cls}$ with $\mathit{Select}(\mathit{Ancestors}(C), C) \neq \mathit{halt}$ **do**
Let $\mathit{Unf}(C) = \{E \mid E$ is derived by unfolding C w.r.t. $\mathit{Select}(\mathit{Ancestors}(C), C)$ using $P\}$.
$\mathit{Cls} := (\mathit{Cls} - \{C\}) \cup \mathit{Unf}(C);$
for each $E \in \mathit{Unf}(C)$ let $\mathit{Ancestors}(E)$ be the sequence $\mathit{Ancestors}(C)$ followed by C
end-while ;

(2) *Define-Fold*:

$\mathit{NewDefs} := \emptyset$;

while there exists a clause $C \in \mathit{Cls}$ and a non-basic atom $A \in \mathit{bd}(C)$ which has not been introduced by folding **do**
Let G be the definition clause $\mathit{Gen}(\mathit{Defs}, A)$ and F be the clause derived by folding C w.r.t. A using G.
$\mathit{Cls} := (\mathit{Cls} - \{C\}) \cup \{F\};$
if $G \notin \mathit{Defs}$ **then** $(\mathit{Defs} := \mathit{Defs} \cup \{G\};\ \ \mathit{NewDefs} := \mathit{NewDefs} \cup \{G\})$

end-while ;

$P_{pe} := P_{pe} \cup Cls$; $Cls := NewDefs$

end-while □

A given unfolding selection function *Select* is said to be *progressive* iff for the empty sequence of clauses, denoted by (), and for any clause C with at least one non-basic atom in the body, we have that $Select((), C) \neq halt$.

We have the following correctness result which derives from the correctness of the unfold/fold transformation rules w.r.t. the least Herbrand model semantics (see Theorem 5 in Section 2).

Theorem 6 (Correctness of Partial Evaluation) *Let Select be a progressive unfolding selection function. Given a definite program P and an atom* $p(t_1, \ldots, t_h)$, *if the Partial Evaluation Strategy using Select terminates with output program* P_{pe} *and atom* $p_{pe}(X_1, \ldots, X_r)$, *then for every ground substitution* $\theta = \{X_1/u_1, \ldots, X_r/u_r\}$,

$M(P) \models p(t_1, \ldots, t_h)\theta$ *iff* $M(P_{pe}) \models p_{pe}(X_1, \ldots, X_r)\theta$.

An unfolding selection function *Select* is *halting* iff for any infinite sequence $C_1, C_2, \ldots$ of clauses, there exists n such that $Select((C_1, C_2, \ldots, C_n), C_{n+1}) = halt$.

Given an infinite sequence $A_1, A_2, \ldots$ of atoms, its *image* under the generalization function *Gen*, is the sequence of clauses defined as follows:

$G_1 = \{newp(X_1, \ldots, X_n) \leftarrow A_1\}$, where $\{X_1, \ldots, X_n\} = vars(A_1)$

$G_{i+1} = G_i \cup \{Gen(G_i,\ A_{i+1})\}$ for $i \geq 1$.

We say that *Gen* is *stabilizing* iff for any infinite sequence $A_1, A_2, \ldots$ of atoms whose image under *Gen* is $G_1, G_2, \ldots$, there exists $n > 0$ such that $G_k = G_n$ for all $k \geq n$.

Theorem 7 (Termination of Partial Evaluation) *If Select is a halting unfolding selection function and Gen is a stabilizing generalization function, then for any input program P and atomic goal* $p(t_1, \ldots, t_h)$, *the Partial Evaluation Strategy using Select and Gen terminates.*

Example 4 (String Matching) Now we present an example of partial evaluation of a string matching program. Given a program for searching a pattern in a string and a specific pattern P, we want to derive a specialized program for searching the pattern P in any given string. A general, deterministic program *Match* for matching strings in $\{a, b\}^*$ is the following one. It is a variant of the ones presented in [73, 74].

$match(P, S) \leftarrow match1(P, S, P, S)$

$match1([\,], X, Y, Z) \leftarrow$

$match1([C|Ps], [C|Ss], P, S) \leftarrow match1(Ps, Ss, P, S)$

$match1([a|Ps], [b|Ss], P, [C|S]) \leftarrow match1(P, S, P, S)$

$match1([b|Ps], [a|Ss], P, [C|S]) \leftarrow match1(P, S, P, S)$

Suppose that we want to specialize this program *Match* w.r.t. the goal $match([a, a, b], S)$, that is, we want to derive a program which, given a string S, tells us whether or not the pattern $[a, a, b]$ occurs in S.

We apply our partial evaluation strategy with the following selection and generalization functions.

(1) The unfolding selection function $DetU : Clauses^* \times Clauses \longrightarrow Atoms \cup \{halt\}$ is defined as follows:

(i) $DetU((), C) = A$ if A is the leftmost non-basic atom in the body of clause C,

(ii) $DetU((C_1, C_2, \ldots, C_n), C) = A$ if $n \geq 1$ and A is the leftmost non-basic atom in the body of C such that A is unifiable with at most one clause head in the program to be partially evaluated, and

(iii) $DetU((C_1, C_2, \ldots, C_n), C) = \mathit{halt}$ if there exists no non-basic atom in the body of C which is unifiable with at most one clause head in the program to be partially evaluated.

(2) The generalization function $\mathit{Variant} : \mathcal{P}(\mathit{Clauses}) \times \mathit{Atoms} \longrightarrow \mathit{Clauses}$ is defined as follows:

(i) $\mathit{Variant}(\mathit{Defs}, A)$ is a clause C such that $bd(C)$ is a variant of A and $C \in \mathit{Defs}$, if any such a clause C exists, and

(ii) $\mathit{Variant}(\mathit{Defs}, A)$ is the clause $\mathit{newp}(X_1, \ldots, X_h) \leftarrow A$, where newp is a new predicate symbol and $\{X_1, \ldots, X_h\} = \mathit{vars}(A)$, otherwise.

The function $DetU$ corresponds to the *determinate unfolding rule* considered in [74]. In general we have that $DetU$ is not halting and *Variant* is not stabilizing. Nevertheless, in our example, as the reader may verify, the partial evaluation strategy using $DetU$ and *Variant* terminates and it generates the following specialized program Match_{pe}:

1. $\mathit{match}_{pe}(A) \leftarrow \mathit{new}1(A)$
2. $\mathit{new}1([a|A]) \leftarrow \mathit{new}2(A)$
3. $\mathit{new}1([b|A]) \leftarrow \mathit{new}1(A)$
4. $\mathit{new}2([a|A]) \leftarrow \mathit{new}4(A)$
5. $\mathit{new}2([b|A]) \leftarrow \mathit{new}1(A)$
6. $\mathit{new}4([b|A]) \leftarrow$
7. $\mathit{new}4([a|A]) \leftarrow \mathit{new}4(A)$

The program Match_{pe} essentially corresponds to a deterministic finite automaton in the sense that: (i) each predicate corresponds to a state, (ii) each clause, except for clause 1 and 6, corresponds to a transition from the state corresponding to the predicate of the head to the state corresponding to the predicate of the body, (iii) each transition is labelled by the symbol (either a or b) occurring in the head of the corresponding clause, (iv) clause 1 enforces that $\mathit{new}1$ is the initial state in the hypothesis that the intended queries are of the of the form $\mathit{match}_{pe}(w)$, for a ground list w in $\{a, b\}^*$, and (v) clause 6 corresponds to a transition to a final state where all words of $\{a, b\}^*$ are accepted. □

The example we have just presented shows that by partial evaluation we can derive a deterministic finite automaton from a general string matching program. This fact allows us to establish an analogy with the Knuth-Morris-Pratt (KMP) string matching algorithm [94], which matches a pattern against a string by generating a deterministic finite automaton depending on the given pattern. For this reason in [140] partial evaluation of logic programs (referred to as partial deduction), is said to pass the KMP test.

3.3.2 Reducing Nondeterminism while Specializing Programs

We argue that the ability of partial evaluation of logic programs to pass the KMP test is related to the fact that the initial string matching program *Match* is a rather sophisticated program.

Indeed, let us consider the case where the initial program is the following naive string matching program *NaiveMatch*:

N1. $naivematch(P, S) \leftarrow append(L, P, X), append(X, R, S)$
N2. $append([\,], Y, Y) \leftarrow$
N3. $append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z)$

For given pattern P and string S, *NaiveMatch* checks that P occurs in S by nondeterministically looking for strings L and R such that the concatenation of L, P, and R is S. Now suppose that we want to specialize *NaiveMatch* w.r.t. the goal $naivematch([a, a, b], S)$. By applying the Partial Evaluation strategy with unfolding selection function *DetU* and generalization function *Variant*, we derive the following program $NaiveMatch_{pe}$:

4. $naivematch_{pe}([a, a, b|S]) \leftarrow$
5. $naivematch_{pe}([C|S]) \leftarrow new1(X, Y),\ new2(Y, Z, S)$
6. $new1([\,], [a, a, b]) \leftarrow$
7. $new1([C|X], [C|Y]) \leftarrow new1(X, Y)$
8. $new2([\,], Y, Y) \leftarrow$
9. $new2([A|X], Y, [A|Z]) \leftarrow new2(X, Y, Z)$

The program $NaiveMatch_{pe}$ does *not* correspond to a deterministic finite automaton. In fact it is a nondeterministic program, and when a mismatch occurs, the search for the pattern $[a, a, b]$ within the string S is always restarted after a right shift of exactly one character from where the pattern was assumed to occur in the previous mismatch. Thus, we say that partial evaluation does *not* pass the KMP test w.r.t. *NaiveMatch*. This failure is due to the following facts:

- partial evaluation fails to eliminate intermediate data structures (see the intermediate list Y in clause 5), because it is not able to combine separate recursive computations into a unique computation,
- partial evaluation fails to reduce nondeterminism, because it does not combine partial evaluations of several alternative computations, and
- partial evaluation does not provide any support for reasoning by cases, and there is no way of introducing disequalities to take into account unification failures.

In the rest of this section, we show that these limitations on the power of partial evaluation can be overcome by the use of the definition introduction rule R1 and the folding rule R3 without the restrictions mentioned in Section 3.3.1. Moreover, in order to reduce the nondeterminism in programs, we make use of an extra rule, called *case split* rule. Case split derives two mutually exclusive instances of a given clause by introducing negative tests as follows.

R9. Case Split. Let C be a clause in P_k of the form: $H \leftarrow Body$, X be a variable occurring in C, and t be any term. By case split of C w.r.t. $\{X/t\}$ we derive the following two clauses: C_1: $(H \leftarrow Body)\{X/t\}$ and C_2: $H \leftarrow X \neq t, Body$. We derive P_{k+1} by replacing C by C_1, C_2.

For reasons of simplicity we will consider the disequality $\neq$ as a new basic predicate symbol, instead of the negation of the equality predicate. Thus, we will remain in the framework of definite logic programs and we can use all results presented in Section 2.

The declarative semantics of our language extended with the disequality predicate can be defined similarly to Section 2.2. In particular, for the notion of model, we stipulate that given an Herbrand interpretation I, a ground goal of the form $t_1 \neq t_2$ is true in I, that is, $I \models t_1 \neq t_2$, iff t_1 is syntactically different from t_2.

The operational semantics of $\neq$ is as follows:

$$(neq1) \qquad \frac{P\colon t_1 = t_2 \mapsto \mathit{false} \qquad P\colon (G_1, G_2) \mapsto \alpha}{P\colon (G_1, t_1 \neq t_2, G_2) \mapsto \alpha} \qquad \text{where } \alpha \text{ is different from } \mathit{false}$$

$$(neq2) \qquad \frac{}{P\colon (G_1, t \neq t, G_2) \mapsto \mathit{false}}$$

The reader may verify that, for each ground substitution θ for the variables of t_1 and t_2: (i) if $P\colon t_1 \neq t_2 \mapsto \mathit{true}$ then $M(P) \models (t_1 \neq t_2)\theta$, and (ii) if $P\colon t_1 \neq t_2 \mapsto \mathit{false}$ then $M(P) \models (t_1 = t_2)\theta$.

We stipulate that the unfolding of a clause w.r.t. an atom of the form $t_1 \neq t_2$ is not allowed.

Our transformation strategy for reducing nondeterminism described below, achieves its objective by deriving *mutually exclusive* clauses in the following sense. Two clauses of the form:

$H_1 \leftarrow d_1, G_1$
$H_2 \leftarrow d_2, G_2$

are said to be mutually exclusive iff either (i) H_1 and H_2 are not unifiable or (ii) H_1 and H_2 are unifiable via an mgu θ and either (ii.1) $P\colon d_1\theta \mapsto \mathit{false}$ or (ii.2) $P\colon d_2\theta \mapsto \mathit{false}$.

Our strategy takes as parameters three subsidiary strategies for guiding the applications of the: (i) unfolding and subsumption rules, (ii) case split, equality introduction/elimination, and goal replacement rules, and (iii) definition and folding rules. More details on these subsidiary strategies can be found in [120]. We will show them in action in Example 5 below.

In our strategy for specializing programs and reducing nondeterminism we assume that the only basic predicates are *true*, *false*, $=$, and $\neq$. We extend the notion of *unit clause* to be a clause without non-basic atoms in the body. Obviously, a unit clause in the usual sense (see Section 2) is a unit clause in this sense.

Suppose that we want to specialize a program P w.r.t. an atom $p(t_1, \ldots, t_h)$. Our strategy will be described as an iterative procedure that at each iteration manipulates three sets of clauses: (1) *Defs*, the set of clauses introduced by the definition rule, (2) P_s, the clauses of the specialized program to be derived, and (3) *Cls*, the set of clauses to be transformed during each iteration.

Initially, *Cls* consists of the clause S: $p_s(X_1, \ldots, X_r) \leftarrow p(t_1, \ldots, t_h)$, with $\{X_1, \ldots, X_r\} = vars(p(t_1, \ldots, t_h))$. From *Cls* we derive a new set of clauses $\mathit{UnitCls} \cup \mathit{FoldedCls}$ by applying the transformation rules according to the subsidiary strategies, and this new set is added to P_s. By construction, the clauses in *UnitCls* are unit clauses and the clauses in *FoldedCls* are mutually exclusive clauses. For the derivation of $\mathit{UnitCls} \cup \mathit{FoldedCls}$ we also introduce new predicates by the definition rule. At the end of the iteration, the set *NewDefs* of clauses that define the newly introduced predicates is added to *Defs*, and *Cls* is updated to *NewDefs*. The transformation strategy terminates when $\mathit{Cls} = \emptyset$, that is, no new predicate is introduced by the definition rule during an iteration.

A Strategy for Reducing Nondeterminism

Input: A program P, an atom $p(t_1, \ldots, t_h)$ w.r.t. which we want to specialize P.
Output: A specialized program P_s and an atom $p_s(X_1, \ldots, X_r)$, with $\{X_1, \ldots, X_r\} = vars(p(t_1, \ldots, t_h))$.

Initialize: Let S be the clause $p_s(X_1, \ldots, X_r) \leftarrow p(t_1, \ldots, t_h)$.
$Defs := \{S\}$; $P_s := \emptyset$; $Cls := \{S\}$;

while $Cls \neq \emptyset$ **do**

Unfold-Subsume:

By applications of the unfolding and subsumption rules, from *Cls* we derive the new set *UnfoldedCls* of clauses. We assume that the unfolding rule is applied to each clause in *Cls*.

Partition:

Let *UnitCls* be the unit clauses occurring in *UnfoldedCls* and *NonunitCls* be the set of the nonunit clauses in *UnfoldedCls*.

From *NonunitCls*, by applying the following rules: case split, equality introduction and elimination rules, and goal replacement restricted to '$=$' and '$\neq$', we derive a set of clauses which can be partitioned into disjoint subsets called *packets*. This partition enjoys the following properties: (i) any two clauses in different packets are mutually exclusive, and (ii) any two clauses in the same packet have same head and same conjunction of disequalities in the body.

Define-Fold:

By applications of the definition rule, we introduce a (possibly empty) set *NewDefs* of definition clauses. These definition clauses are such that we can apply the folding rule as follows.

Let us consider a packet of clauses of the form:

$$\begin{cases} H \leftarrow \; d, \; G_1 \\ \quad \ldots \\ H \leftarrow \; d, \; G_m \end{cases}$$

where d is the conjunction of the basic atoms, including the disequalities introduced by case split, which are common to all clauses in the packet. We apply the folding rule using clauses in $Defs \cup NewDefs$ so that we derive a single clause of the form:

$$H \leftarrow d, newp(\ldots)$$

where *newp* is a predicate defined in $Defs \cup NewDefs$.

Let *FoldedCls* be the set of all clauses each of which is derived from a packet by folding.

$Defs := Defs \cup NewDefs$; $P_s := P_s \cup UnitCls \cup FoldedCls$; $Cls := NewDefs$

end-while □

Suppose that our strategy for reducing nondeterminism terminates, and let P_s be the derived specialized program. Then, all nonunit clauses in P_s are mutually exclusive, because of the transformation steps indicated in the *Partition* and *Define-Fold* phases of our strategy,

Let us also consider the set *Defs* of definition clauses derived at the end of the strategy. By the correctness of the transformation rules (see Theorem 5), we have that $M(P \cup Defs) = M(P \cup P_s)$. Moreover, for every ground substitution $\theta = \{X_1/u_1, \ldots, X_r/u_r\}$ we have that:

$$M(P) \models p(t_1, \ldots, t_h)\theta \quad \text{iff} \quad M(P \cup Defs) \models p_s(X_1, \ldots, X_r)\theta$$

because p_s is defined in $P \cup Defs$ by the unique clause $p_s(X_1, \ldots, X_r) \leftarrow p(t_1, \ldots, t_h)$, and

$$M(P \cup P_s) \models p_s(X_1, \ldots, X_r)\theta \quad \text{iff} \quad M(P_s) \models p_s(X_1, \ldots, X_r)\theta$$

because in the program $P \cup P_s$, the predicate p_s does not depend on predicates defined by P.

Thus, we have the following correctness result.

Theorem 8 *Given a definite program P and an atom $p(t_1, \ldots, t_h)$, if the Strategy for Reducing Nondeterminism terminates with output program P_s and atom $p_s(X_1, \ldots, X_r)$, then for every ground substitution $\theta = \{X_1/u_1, \ldots, X_r/u_r\}$,*

$$M(P) \models p(t_1, \ldots, t_h)\theta \text{ iff } M(P_s) \models p_s(X_1, \ldots, X_r)\theta.$$

We now show that our strategy for reducing nondeterminism is able to derive from the *NaiveMatch* program a program corresponding to a deterministic finite automaton.

Example 5 (Deriving a Deterministic Finite Automaton from Naive String Matching) We present the various iterations of our strategy when the input program is the *NaiveMatch* program given at the beginning of this section and the input atom is $naivematch([a,a,b],S)$.

By definition we introduce the clause:

1. $naivematch_s(S) \leftarrow naivematch([a,a,b],S)$

Thus, initially we have $Defs = \{1\}$, $P_s = \emptyset$, and $Cls = \{1\}$.

First iteration. By unfolding clause 1 using clause N1 we get:

2. $naivematch_s(S) \leftarrow append(Left,[a,a,b],X),\ append(X,Right,S)$

For folding clause 2 we introduce the following definition clause:

3. $new1(S) \leftarrow append(Left,[a,a,b],X),\ append(X,Right,S)$

By folding clause 2 using clause 3 we get:

4. $naivematch_s(S) \leftarrow new1(S)$

Clause 4 is a clause to be added to the specialized program. Then we perform the following assignments: $Defs := Defs \cup \{3\}$; $P_s := P_s \cup \{4\}$; $Cls := \{3\}$. In the following iteration we deal with the new definition clause 3.

Second iteration. By unfolding clause 3 we get:

5. $new1([a|S]) \leftarrow append([a,b],Right,S)$
6. $new1([C|S]) \leftarrow append(Left,[a,a,b],X),\ append(X,Right,S)$

The heads of clauses 5 and 6 are unifiable by the substitution $\{C/a\}$. Thus, by case split of clause 6 w.r.t. $\{C/a\}$ we get:

6.1 $new1([a|S]) \leftarrow append(Left,[a,a,b],X),\ append(X,Right,S)$
6.2 $new1([C|S]) \leftarrow C \neq a,\ append(Left,[a,a,b],X),\ append(X,Right,S)$

For folding clauses (5, 6.1) which belong to the same packet because they have equal heads, we introduce the following definition clauses:

7. $new2(S) \leftarrow append([a,b],Right,S)$
8. $new2(S) \leftarrow append(Left,[a,a,b],X),\ append(X,Right,S)$

By folding clauses (5, 6.1) using clauses (7, 8), and by folding clause 6.2 using clause 3, we get:

9. $new1([a|S]) \leftarrow new2(S)$
10. $new1([C|S]) \leftarrow C \neq a,\ new1(S)$

Clauses 9 and 10 are mutually exclusive clauses and they are added to the specialized program. Then we perform the following assignments: $\mathit{Defs} := \mathit{Defs} \cup \{7, 8\}$; $P_s := P_s \cup \{9, 10\}$; $\mathit{Cls} := \{7, 8\}$. In the following iteration we deal with the new definition clauses 7 and 8.

Third iteration. By unfolding clauses 7 and 8 we get:

11. $\mathit{new}2([a|S]) \leftarrow \mathit{append}([b], \mathit{Right}, S)$
12. $\mathit{new}2([a|S]) \leftarrow \mathit{append}([a, b], \mathit{Right}, S)$
13. $\mathit{new}2([C|S]) \leftarrow \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$

By case split of clause 13 w.r.t. the substitution $\{C/a\}$ we get:

13.1 $\mathit{new}2([a|S]) \leftarrow \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$
13.2 $\mathit{new}2([C|S]) \leftarrow C \neq a,\ \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$

For folding clauses (11, 12, 13.1) which belong to the same packet because they have equal heads, we introduce the following definition clauses:

14. $\mathit{new}3(S) \leftarrow \mathit{append}([b], \mathit{Right}, S)$
15. $\mathit{new}3(S) \leftarrow \mathit{append}([a, b], \mathit{Right}, S)$
16. $\mathit{new}3(S) \leftarrow \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$

By folding clauses (11, 12, 13.1) using clauses (14, 15, 16), and by folding clause 13.2 using clause 3, we get:

17. $\mathit{new}2([a|S]) \leftarrow \mathit{new}3(S)$
18. $\mathit{new}2([C|S]) \leftarrow C \neq a,\ \mathit{new}1(S)$

Clauses 17 and 18 are mutually exclusive clauses and they are added to the specialized program. We perform the following assignments: $\mathit{Defs} := \mathit{Defs} \cup \{14, 15, 16\}$; $P_s := P_s \cup \{17, 18\}$; $\mathit{Cls} := \{14, 15, 16\}$. In the following iteration we deal with the new definition clauses 14, 15, and 16.

Fourth iteration. By unfolding clauses 14, 15, and 16 we get:

19. $\mathit{new}3([b|S]) \leftarrow \mathit{append}([\,], \mathit{Right}, S)$
20. $\mathit{new}3([a|S]) \leftarrow \mathit{append}([b], \mathit{Right}, S)$
21. $\mathit{new}3([a|S]) \leftarrow \mathit{append}([a, b], \mathit{Right}, S)$
22. $\mathit{new}3([C|S]) \leftarrow \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$

By a case split step on clause 22 w.r.t. $\{C/b\}$ and then by a second case split step w.r.t. $\{C/a\}$ we get:

22.1 $\mathit{new}3([b|S]) \leftarrow \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$
22.2 $\mathit{new}3([a|S]) \leftarrow b \neq a,\ \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$
22.3 $\mathit{new}3([C|S]) \leftarrow C \neq a, C \neq b, \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$

Since $a \neq b$ is *true*, by goal replacement in the body of (22.2) we get:

23. $\mathit{new}3([a|S]) \leftarrow\ \mathit{append}(\mathit{Left}, [a, a, b], X),\ \mathit{append}(X, \mathit{Right}, S)$

For folding clauses (19, 22.1) we introduce the following definition clauses:

24. $\mathit{new}4(S) \leftarrow \mathit{append}([\,], \mathit{Right}, S)$
25. $\mathit{new}4(S) \leftarrow \mathit{append}(\mathit{Left}, [a, a, b], X), \mathit{append}(X, \mathit{Right}, S)$

By folding clauses (19, 22.1) using clauses (24, 25), by folding clauses (20, 21, 23) using clauses (14, 15, 16), and by folding clause 22.3 using clause 3, we get:

26. $new3([b|S]) \leftarrow new4(S)$
27. $new3([a|S]) \leftarrow new3(S)$
28. $new3([C|S]) \leftarrow C \neq b,\ C \neq a,\ new1(S)$

Clauses 26, 27, and 28 are pairwise mutually exclusive clauses and they are added to the specialized program. We perform the following assignments: $\mathit{Defs} := \mathit{Defs} \cup \{24, 25\}$; $P_s := P_s \cup \{26, 27, 28\}$; $\mathit{Cls} := \{24, 25\}$. In the following iteration we deal with the new definition clauses 24 and 25.

Fifth iteration. By unfolding clauses 24 and 25 we get:

29. $new4(S) \leftarrow$
30. $new4([a|S]) \leftarrow append([a, b], Right, S)$
31. $new4([C|S]) \leftarrow append(Left, [a, a, b], X),\ append(X, Right, S)$

We discard clauses 30 and 31 because they are subsumed by clause 29. Since $\mathit{NonunitCls} = \emptyset$ no folding steps are to be performed. *Defs* need not be updated because *NewDefs* is empty. We also have: $P_s := P_s \cup \{29\}$; $\mathit{Cls} := \emptyset$.
After this fifth iteration we end the application of our strategy for reducing nondeterminism, because we did not introduce any extra definition clause for performing folding steps, and thus, $\mathit{Cls} = \emptyset$.

Our final program P_s is:

4. $naivematch_s(S) \leftarrow new1(S)$
9. $new1([a|S]) \leftarrow new2(S)$
10. $new1([C|S]) \leftarrow C \neq a, new1(S)$
17. $new2([a|S]) \leftarrow new3(S)$
18. $new2([C|S]) \leftarrow C \neq a, new1(S)$
26. $new3([b|S]) \leftarrow new4(S)$
27. $new3([a|S]) \leftarrow new3(S)$
28. $new3([C|S]) \leftarrow C \neq b, C \neq a, new1(S)$
29. $new4(S) \leftarrow$

P_s corresponds to a finite automaton which is deterministic, and in fact, clauses 9, 10, 17, 18, 26, 27, 28, and 29 correspond to the minimal finite automaton. However, in general, our strategy does not ensure the minimality of the derived finite automaton. We leave this problem for future research. Notice also that due to the introduction of disequations, our program P_s may be used also when the alphabet is larger than $\{a, b\}$. □

3.4 Overview of Other Techniques and Bibliographic Remarks

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.

Recursion Removal

Recursion is the main control structure for logic and functional 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 [115, 151]. 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 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 goal.

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 [24] 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. the least Herbrand model semantics. However, goal rearrangements may increase the amount of nondeterminism, thus making useless the efficiency improvements due to tail-recursion optimization. Moreover, goal rearrangements do not preserve Prolog operational semantics, 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 two methods.

The first method consists in transforming *almost-tail-recursive* clauses into tail-recursive ones by using the unfold/fold rules [6, 51, 52].

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 basic predicates, such as the ones for computing concatenation of lists and arithmetic operations, like addition or multiplication. The transformation techniques presented in [6, 51, 52] use the generalization strategy and some replacement laws which are valid for the basic 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 [5] for functional programs.

The second method is based on schema transformations [12, 22, 23], where some almost-tail recursive program schemata are shown to be equivalent to tail-recursive ones.

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' [155]. However, it is sometimes difficult to identify the data structures which allow very efficient algorithms before actually writing the programs. Moreover, complex data structures make it hard to prove program correctness.

Program transformation has shown its strength as a methodology for deriving appropriate data structures in a *dynamic* way (see Chapter 8 of [113]), that is, while looking for the recursive definition of predicates. For instance, in the example for reversing a list which we have presented in Section 1.3, the new argument *Acc* is the appropriate list to be introduced for achieving a linear algorithm. One may show that *Acc* can be introduced by applying the generalization strategy.

Another example of a change of data representations is given in Section 4.3, where we show how to transform logic programs which use lists into equivalent logic programs which use difference-lists (see Example 7 below).

Conjunctive Partial Deduction

Conjunctive partial deduction has been proposed in [49] as an extension of partial evaluation, whereby conjunctions of atoms, rather than single atoms (as done in [104]), are taken into consideration. The techniques presented in [49] recast in the style of Lloyd and Shepherdson's [104] those for the elimination of intermediate data structures described in [125]. However, these techniques neither can combine partial evaluations of alternative branches of the SLD-trees, nor can introduce disequalities. Thus, our definition and folding rules which allow us to specialize programs w.r.t. *disjunctions of conjunctions of goals*, generalize conjunctive partial deduction.

Supercompilation

Supercompilation is a technique introduced and studied in [79, 147] for the semiautomatic improvement of programs by taking into consideration the various computation paths. However, those paths are considered in isolation and no relationship among them is exploited. Since our definition and folding rules allow us to factorize common computations in different branches of the SLD-trees, our approach provides an extension of the basic supercompilation techniques.

4 Theorem Proving and Program Synthesis via Unfold/Fold Transformations

In this section we illustrate how unfold/fold transformations can be used both for proving properties of predicates defined by means of logic programs and also for the synthesis of logic programs.

4.1 The Unfold/Fold Proof Method

As we have seen in Section 2, during program development by transformation, program efficiency may be improved by performing some goal replacement steps. In order to apply the goal replacement steps, we need to prove some equivalences which are required to hold in the least Herbrand model of the given initial programs. Such equivalences are formulas of the form: $\forall X. (\exists Y. F(X,Y) \leftrightarrow \exists Z. G(X,Z))$. Here and in what follows: (i) X, Y, and Z may stand either for single variables or for disjoint *vectors* of variables, and (ii) $F(X,Y)$ is a goal whose variables are among those of (X,Y). Analogously for $G(X,Z)$. In general, a goal, say H, is also denoted by $H(X_1,\ldots,X_n)$ to express that its variables are among those of $(X_1,\ldots,X_n)$.

We assume that equivalence formulas of the form indicated above are stored as theorems in a knowledge base. Addition of new theorems to the knowledge base can be done either *off-line* or *on-line*.

In the off-line mode new theorems are added independently of the program derivation activity, while in the on-line mode they are added during program derivation as follows. Suppose that in the process of deriving a new program, in order to perform a goal replacement step we need to check the validity of a given property. Then we may interrupt the program derivation process and look for a proof of that property. If the desired proof is constructed, then the property is added as a new theorem to the knowledge base, and the goal replacement step based on that theorem, is performed. Otherwise, if the desired proof cannot be constructed, we may consider alternative properties or we may resume the program derivation process by investigating alternative derivation steps.

Now, in the case of the on-line mode, it is useful to be able to look for a proof of the property of interest by using the same techniques that are used for program derivation. In our case they are the ones based on transformation rules. Thus, we may construct systems where program derivations and proofs of properties may be freely interleaved. For that purpose people have studied theorem prover techniques which are based on rewriting rules, and among those techniques we want to present the *unfold/fold proof method* [95, 124].

In what follows we present this method and we show how to prove that a given equivalence formula holds in the least Herbrand model of a given program using the unfold/fold proof method (see also [119]). For simplicity, here we consider the following transformation rules only: (R1) Definition Introduction, (R2) Unfolding, (R3) Folding, (R4) Goal Replacement, and (R8) Equality Introduction and Elimination.

Definition 4 (Unfold/Fold proof) *Let P be a program and Equiv be the formula $\forall X. (\exists Y. F(X,Y) \leftrightarrow \exists Z. G(X,Z))$, where X, Y, and Z are pairwise disjoint vectors of variables and $F(X,Y)$ and $G(X,Z)$ are two goals. An unfold/fold proof of Equiv using P consists of two totally correct transformation sequences $T_1 : P, P \wedge C_1, \ldots, P \wedge S_1$ and $T_2 : P, P \wedge C_2, \ldots, P \wedge S_2$* (see Fig. 4) *such that:*

(i) *the sequences T_1 and T_2 are constructed by first adding, using rule* R1, *to program P the following definition clauses C_1 and C_2, respectively:*

$$
\begin{tikzcd}
P \wedge \boxed{C_1 : new1(X) \leftarrow F(X,Y)} \arrow[d, "(\mathrm{R2}+\mathrm{R3}+\mathrm{R4}+\mathrm{R8})^*"'] & & \boxed{C_2 : new2(X) \leftarrow G(X,Y)} \wedge P \arrow[d, "(\mathrm{R2}+\mathrm{R3}+\mathrm{R4}+\mathrm{R8})^*"] \\
P \wedge \boxed{S_1} & \overset{\{new1/new2\}}{=} & \boxed{S_2} \wedge P
\end{tikzcd}
$$

Figure 4: Unfold/Fold proof of $\forall X.\, (\exists Y.\, F(X,Y) \leftrightarrow \exists Z.\, G(X,Z))$ using program P.

$C_1:\ new1(X) \leftarrow F(X,Y)$
$C_2:\ new2(X) \leftarrow G(X,Z)$

and then applying a sequence of transformation rules, each of which is taken from the set $\{\mathrm{R2}, \mathrm{R3}, \mathrm{R4}, \mathrm{R8}\}$*,*

(ii) *for* $i = 1, 2$*, and for each new clause* D *derived during the construction of* T_i *we have that* $C_i \Rightarrow^* D$*, and*

(iii) S_2 *can be obtained from* S_1 *by substituting the predicate symbol* $new2$ *for* $new1$*.*

Notice that, by the folding rule R3 in T_1 every folding step is performed using clause C_1 only, and analogously, in T_2 every folding step is performed using C_2 only.

Theorem 9 (Soundness of the unfold/fold proof method) *If there exists an unfold/fold proof of the formula* $\forall X.\, (\exists Y.\, F(X,Y) \leftrightarrow \exists Z.\, G(X,Z))$ *using* P*, then we have that:*
$M(P) \models \forall X.\, (\exists Y.\, F(X,Y) \leftrightarrow \exists Z.\, G(X,Z))$.

In the following example we apply the unfold/fold proof method for showing the functionality of the Fibonacci predicate. Recall that a predicate $p(X,Y)$ is said to be *functional* w.r.t. X in a program P iff we have that:
$M(P) \models \forall X,Y,Z.\, ((p(X,Y), p(X,Z)) \leftrightarrow (p(X,Y), Y\!=\!Z))$.

Example 6 (Functionality of Fibonacci) Let us consider the following program *Fib* for the computation of the Fibonacci numbers:

$fib(0, s(0)) \leftarrow$
$fib(s(0), s(0)) \leftarrow$
$fib(s(s(N)), F2) \leftarrow fib(s(N), F1),\ fib(N, F),\ plus(F1, F, F2)$
$plus(0, Y, Y) \leftarrow$
$plus(s(X), Y, s(Z)) \leftarrow plus(X, Y, Z)$

where the predicate *fib* is assumed to be non-basic and the predicate *plus* is assumed to be basic. $plus(X,Y,Z)$ is also assumed be functional w.r.t. X and Y, that is, the following formula holds in $M(Fib)$:
$\forall X,Y,Z1,Z2.\, ((plus(X,Y,Z1),\ plus(X,Y,Z2)) \leftrightarrow (plus(X,Y,Z1),\ Z1\!=\!Z2))$.
We want to prove that the predicate $fib(N,F)$ is functional w.r.t. N, that is, the following equivalence formula holds in $M(Fib)$:
EquivF: $\forall N,F1,F2.\, ((fib(N,F1),\ fib(N,F2)) \leftrightarrow (fib(N,F1),\ F1\!=\!F2))$.
We apply the unfold/fold proof method and we introduce the following two clauses:

C_1: $new1(N, F1, F2) \leftarrow fib(N, F1),\ fib(N, F2)$
C_2: $new2(N, F1, F2) \leftarrow fib(N, F1),\ F1 = F2$

Clauses C_1 and C_2 can be transformed into the two sets of clauses S_1 and S_2, respectively, as shown in Fig. 5 by means of two totally correct transformation sequences. This property follows from Theorem 5 of Section 2.6.2 because *plus* is a basic predicate. Since S_1 and S_2 are equal modulo predicate renaming, we have that *EquivF* holds in $M(Fib)$.

Notice that also the functionality of *plus*, which is used as a lemma in the unfold/fold proof shown in Fig. 5, can be proved by using the unfold/fold method. For that proof we need to assume that *plus* is a non-basic predicate. We leave that proof to the reader. □

The reader should notice that the unfold/fold proof method cannot be used for proving that a formula φ is a logical consequence of a program P (i.e., φ is true in *all* models of P) because in general the unfold/fold rules do not preserve all models of P but only the least Herbrand model.

In what follows we will study the correctness of the unfold/fold proof method w.r.t. the goal replacement rule (Section 4.2), we will address the problem of finding the unfold/fold proofs in a mechanical way, and we will propose a strategy (Section 4.3) which when successful, allows us to automatically construct those proofs and also to synthesize programs from specifications (recall that the existence of unfold/fold proofs is, in general, undecidable). We will also illustrate an application of this method for avoiding unnecessary nondeterminism (Section 4.4).

4.2 Unfold/Fold Proofs and Goal Replacement

We have seen that proofs of equivalence formulas are needed to apply the goal replacement rule. However, these proofs are *not* sufficient for ensuring the total correctness of a transformation sequence when it includes a goal replacement step, as Example 1 in Section 2 shows.

As stated by Theorem 5, a sufficient condition for the total correctness of a transformation sequence is that goal replacements are performed only if they are non-ascending. In what follows we provide a sufficient condition for ensuring that a goal replacement is non-ascending. Our condition relies on the construction of a suitable unfold/fold proof of the equivalence formula which justifies the goal replacement.

Definition 5 (Non-ascending unfold/fold proof) *Let P be a program and Equiv be the formula $\forall X. (\exists Y. F(X,Y) \leftrightarrow \exists Z. G(X,Z))$, where X, Y, and Z are pairwise disjoint vectors of variables and $F(X,Y)$ and $G(X,Z)$ are two goals. A* non-ascending *unfold/fold proof of Equiv consists of two transformation sequences $T_1:\ P, P \wedge C_1, \ldots, P \wedge S_1$ and $T_2:\ P, P \wedge C_2, \ldots, P \wedge S_2$ satisfying the properties at Points* (i)–(iii) *of Definition* 4 *and also satisfying the following properties:*

- (iv) *Each transformation sequence includes at least one unfolding step w.r.t. a non-basic atom.*
- (v) *In each transformation sequence if a folding step is not the last one, then it is followed by folding steps only.* (*Recall that, by our folding rule* R3*, in T_1 every folding step is performed using clause C_1 only, and analogously, in T_2 every folding step is performed using C_2 only.*)
- (vi) *In each transformation sequence each application of the goal replacement rule which replaces goal G_1 by goal G_2 is restricted to one of the following two cases:* (1) *in G_1 and G_2 there are basic predicates only;* (2) *G_1 is of the form* $(H1, H2)$ *and G_2 is of the form* $(H2, H1)$ (*that is, the goal replacement consists in rearranging the order of the goals*). (*In these two*

$Fib \wedge$ $\boxed{C_1: \; new1(N,F1,F2) \leftarrow fib(N,F1),\, fib(N,F2)}$

`unfold w.r.t.` $fib(N,F1)$ `and` $fib(N,F2)$

$new1(0,s(0),s(0)) \leftarrow$
$new1(s(0),s(0),s(0)) \leftarrow$
$new1(s(s(N)),F1,F2) \leftarrow$
$\quad \underline{fib(s(N),F11)},\, fib(N,F12),$
$\quad \underline{plus(F11,F12,F1)},$
$\quad \underline{fib(s(N),F12)},\, fib(N,F22),$
$\quad \underline{plus(F21,F22,F2)}$

`fold (twice)`
`using` C_1

$Fib \wedge$ S_1:
$new1(0,s(0),s(0)) \leftarrow$
$new1(s(0),s(0),s(0)) \leftarrow$
$new1(s(s(N)),F1,F2) \leftarrow$
$\quad \underline{new1(s(N),F11,F21)},$
$\quad new1(N,F12,F22),$
$\quad plus(F11,F12,F1),$
$\quad plus(F21,F22,F2)$

$\{new1/new2\}$
$=$

$\boxed{C_2: \; new2(N,F1,F2) \leftarrow fib(N,F1),\, F1=F2}$ $\wedge\ Fib$

`unfold w.r.t.` $fib(N,F1)$

$new2(0,s(0),s(0)) \leftarrow$
$new2(s(0),s(0),s(0)) \leftarrow$
$new2(s(s(N)),F1,F2) \leftarrow$
$\quad fib(s(N),F11),\, fib(N,F12),$
$\quad \underline{plus(F11,F12,F1),\, F1=F2}$

`goal replacement`
`(functionality of` $plus$`)`

...
$new2(s(s(N)),F1,F2) \leftarrow$
$\quad fib(s(N),F11),\, fib(N,F12),$
$\quad \underline{plus(F11,F12,F1),\, plus(F11,F12,F2)}$

`equality introduction`
`(twice)`

...
$new2(s(s(N)),F1,F2) \leftarrow$
$\quad \underline{fib(s(N),F11)},\, fib(N,F12),$
$\quad \underline{plus(F11,F12,F1)},\, plus(\underline{F21},\underline{F22},F2),$
$\quad \underline{F11=F21},\, F12=F22$

`fold (twice)`
`using` C_2

S_2: $\wedge\ Fib$
$new2(0,s(0),s(0)) \leftarrow$
$new2(s(0),s(0),s(0)) \leftarrow$
$new2(s(s(N)),F1,F2) \leftarrow$
$\quad \underline{new2(s(N),F11,F21)},$
$\quad new2(N,F12,F22),$
$\quad plus(F11,F12,F1),$
$\quad plus(F21,F22,F2)$

Figure 5: Unfold/Fold proof of the functionality of the Fibonacci predicate *fib* assuming the functionality of the predicate *plus*.

cases the equivalence formulas which justify the goal replacements hold in the least Herbrand model of every program.)

(vii) *For each sequence of clauses $R_1: C_1 \Rightarrow \ldots \Rightarrow L$, where L is a clause in S_1 there exists a sequence of clauses $R_2: C_2 \Rightarrow \ldots \Rightarrow M$, where M is a clause in S_2 such that:* (1) *M can be obtained from L by replacing every occurrence of $new1$ by $new2$, and* (2) *the number of clauses obtained by unfolding steps and occurring in R_1, is not less than the number of clauses obtained by unfolding steps and occurring in R_2.*

An example of a non-ascending unfold/fold proof is given by the proof of the functionality of the Fibonacci predicate in Example 6. By the following theorem we have that goal replacement steps justified by non-ascending unfold/fold proofs (see Definition 5) are non-ascending (see Definition 1) and thus, by Theorem 5, they are totally correct w.r.t. the least Herbrand model semantics.

Theorem 10 *Let P_0 be a program and let $F(X,Y)$ and $G(X,Z)$ be goals. If there exists a non-ascending unfold/fold proof of $\forall X. (\exists Y. F(X,Y) \leftrightarrow \exists Z. G(X,Z))$ using P_0, then for each transformation sequence $P_0, \ldots, P_k$ the replacement of $F(X,Y)$ by $G(X,Z)$ in the body of a clause in P_k is a non-ascending goal replacement.*

4.3 A Method for Program Synthesis from Implicit Definitions

We now present a method for the synthesis of programs from implicit definitions which is based on the construction of unfold/fold proofs, and we also see an example of its use. This example is the synthesis of a program which implements queues by using difference-lists, instead of lists. The synthesized program is essentially the one given (but not synthesized) in [143, page 297].

We present our method for program synthesis from implicit definitions with reference to a given program P and a closed formula of the form:

$EquivDef$: $\quad \forall X. (\exists Y. F(X,Y) \leftrightarrow \exists Z. (H(X,Z), newp(X,Z)))$

where (i) the predicates occurring in the goals $F(X,Y)$ and $H(X,Z)$ are defined in P, (ii) $newp$ does not occur in P, and (iii) X, Y, and Z are pairwise disjoint vectors of variables. The formula $EquivDef$ is said to be an *implicit definition* of $newp$.

We look for a set of clauses, say *Eureka*, which is the (explicit) definition of the predicate $newp$, such that $M(P \wedge Eureka) \models EquivDef$. The reader who is familiar with the abduction theory, may realize that the task of generating the set *Eureka* can be viewed as an instance of an *abduction problem* [90], where *EquivDef* is the *observed formula* and *Eureka* is a set of *abductive explanations* to be added to P for justifying the formula *EquivDef*.

Our synthesis method consists of the following five phases (see also Fig. 6).

Phase (1). We introduce the following two clauses:

$C_1: \; new1(X) \leftarrow F(X,Y)$
$C_2: \; new2(X) \leftarrow H(X,Z), \; newp(X,Z)$

where $new1$ and $new2$ are predicate symbols not occurring in P.

Phase (2). We construct a totally correct transformation sequence $P, \; P \wedge C_1, \ldots, \; P \wedge S_1$ by first adding (using rule R1) clause C_1 to program P and then applying a sequence of transformation rules, each of which is taken from the set $\{R2, R3, R4, R8\}$. We assume

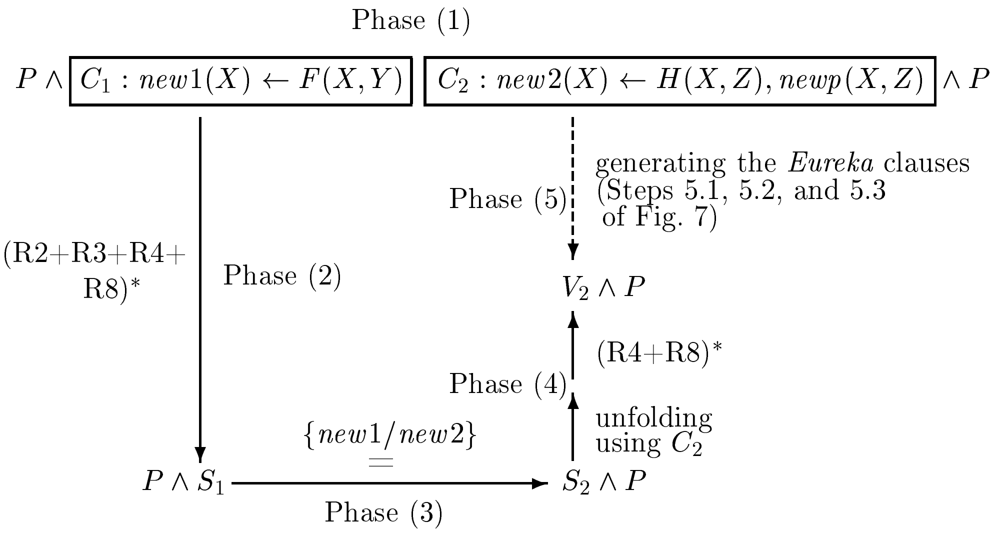

Figure 6: Five phases of our synthesis method from implicit definitions using unfold/fold proofs.

that each clause in S_1 is derived, in zero or more steps, from clause C_1, that is, for each clause D not in P, derived during the construction of the transformation sequence, we have that $C_1 \Rightarrow^* D$.

Phase (3). We get a set S_2 of clauses from the set S_1 by replacing every occurrence of the predicate symbol *new*1 by *new*2.

Phase (4). We eliminate the occurrences of *new*2 from the bodies of the clauses in S_2 by performing some unfolding steps using clause C_2. We then apply zero or more times the rules R4, and R8, thereby getting a new set of clauses, say V_2.

Phase (5). We generate a set *Eureka* of clauses which allows us to construct a transformation sequence from $P \wedge \mathit{Eureka} \wedge C_2$ to $P \wedge \mathit{Eureka} \wedge V_2$ by applying rules R2, R4, and R8. This final phase is further detailed below (see also Fig. 7).

During Phases (4) and (5) we restrict the application of rule R4 to the cases indicated at Point (vi) of Definition 5, and during Phase (5) we perform at least one unfolding step.

One can show that at the end of Phase (5) there are two transformation sequences $T_1 : P \wedge \mathit{Eureka}, P \wedge \mathit{Eureka} \wedge C_1, \ldots, P \wedge \mathit{Eureka} \wedge S_1$ and $T_2 : P \wedge \mathit{Eureka}, P \wedge \mathit{Eureka} \wedge C_2, \ldots, P \wedge \mathit{Eureka} \wedge S_2$ which are totally correct. Thus, they constitute an unfold/fold proof of the equivalence formula *EquivDef* using $P \wedge \mathit{Eureka}$ and the following theorem follows from Theorem 9.

Theorem 11 (Soundness of the synthesis method) *Let P be a program, $F(X,Y)$ and $H(X,Z)$ be goals and newp be a predicate symbol not occurring in P. Let Eureka be the set of clauses defining newp derived by the synthesis method from implicit definitions starting from the formula $\forall X. (\exists Y. F(X,Y) \leftrightarrow \exists Z. (H(X,Z), \mathit{newp}(X,Z)))$. Then $M(P \wedge \mathit{Eureka}) \models \forall X. (\exists Y. F(X,Y) \leftrightarrow \exists Z. (H(X,Z), \mathit{newp}(X,Z)))$.*

Now we illustrate the five phase synthesis method from implicit definitions that we have described above, by looking at the synthesis of a program which implements queues by using difference-lists, instead of lists (see Example 7 below).

Difference-lists are data structures which are sometimes used, instead of lists, for manipulating sequences of elements. The advantage of using difference-lists is that the concatenation Z of two sequences X and Y represented as difference-lists, can be performed in constant time, while it takes linear time (w.r.t. the length of X) if we use the standard predicate $append(X,Y,Z)$ which concatenates the lists X and Y and constructs the list Z. A difference-list can be thought of as a pair of lists, denoted by $L\backslash R$, such that there exists a third list Y for which $append(Y,R,L)$ holds [36]. In that case we say that Y is *represented* by the difference-list $L\backslash R$.

Programs that use lists are often simpler to write and understand than the equivalent ones which make use of difference-lists. Thus, one may be interested in providing techniques for transforming in an automatic way programs which use lists, into programs which use difference-lists. Several such techniques have been proposed in the literature [82, 156]. We will show that by applying our synthesis method we can automatically perform the transformation which introduces difference-lists. Our method is very general and it can also be used to perform other changes of data representations.

In general the problem of transforming programs which use difference-lists, instead of lists, can be formulated as follows. Let $p(X,Y)$ be a predicate defined in a program P and let Y be a list. We want to synthesize a new predicate, say $diff\_p(X,L\backslash R)$, where $L\backslash R$ is a difference-list, together with an additional set of clauses, say *Eureka*, defining $diff\_p$. We also want $diff\_p(X,L\backslash R)$ to be equivalent to $p(X,Y)$, being $L\backslash R$ a difference-list representing Y. Thus, our program transformation problem reduces to the problem of synthesizing a set *Eureka* of clauses such that

$$M(P \wedge Eureka) \models \forall X,Y.\, (p(X,Y) \leftrightarrow \exists L,R.\, (append(Y,R,L), diff\_p(X,L\backslash R)))$$

where, as already mentioned, $append(Y,R,L)$ holds iff $L\backslash R$ is a difference-list representing Y.

Example 7 (Implementing Queues by Difference-lists) Let us consider the following program *Queue* defining a predicate $queue(S)$ which holds iff S is a sequence of *enqueue* and *dequeue* operations represented as a list of terms of the form $enqueue(X)$ and $dequeue(X)$, respectively.

$$\begin{array}{l}
queue(S) \leftarrow q(S,[\,])\\
q([enqueue(X)|Xs],Q) \leftarrow append(Q,[X],Q1),\ q(Xs,Q1)\\
q([dequeue(X)|Xs],Q) \leftarrow append([X],Q1,Q),\ q(Xs,Q1)\\
q([\,],Q) \leftarrow Q=[\,]
\end{array}$$

Queues of elements are represented as lists. The second argument of the predicate q is a queue which is initially empty (represented as $[\,]$), and it is updated according to the sequence of $enqueue(X)$ and $dequeue(X)$ operations specified by the value of the first argument of q. The $enqueue(X)$ and $dequeue(X)$ operations are implemented by means of list concatenations using $append(Q,[X],Q1)$ and $append([X],Q1,Q)$, respectively, that is, elements enter a queue from the 'right end' and exit a queue from the 'left end'. Since the evaluation of $append(Q,[X],Q1)$ is expensive, we would like to represent the lists Q and $Q1$, which occur in the second argument of q, as difference-lists. Thus, we look for a predicate $diff\_q(S,L\backslash R)$ defined by a set *Eureka* of clauses such that:

EquivQ: $M(Queue \wedge Eureka) \models \forall S,Q.\, (q(S,Q) \leftrightarrow \exists L,R.\, (append(Q,R,L),\ diff\_q(S,L\backslash R)))$

The predicates *append*, '$=$', and '$\neq$' are considered to be basic predicates (by this hypothesis we may apply Theorem 11 and we are guaranteed the soundness of our synthesis method). The synthesis of *Eureka* can be performed by routine application of our five phase synthesis method as follows.

Phase (1). We introduce two clauses:

$$\begin{array}{ll} C_1: & new1(S,Q) \leftarrow q(S,Q) \\ C_2: & new2(S,Q) \leftarrow append(Q,R,L),\ \mathit{diff\_q}(S, L\backslash R) \end{array}$$

Phase (2). We unfold clause C_2 w.r.t. $q(S,Q)$. We then perform some folding steps and we get the following set of clauses:

$$\begin{array}{ll} S_1: & new1([enqueue(X)|Xs],Q) \leftarrow append(Q,[X],Q1),\ new1(Xs,Q1) \\ & new1([dequeue(X)|Xs],Q) \leftarrow append([X],Q1,Q),\ new1(Xs,Q1) \\ & new1([\,],Q) \leftarrow Q=[\,] \end{array}$$

As one may expect by looking at clause C_1, the clauses for $new1$ are equal to those for q where each occurrence of q has been replaced by $new1$.

Phase (3). By replacing the predicate symbol $new1$ by $new2$ we get the following set of clauses:

$$\begin{array}{ll} S_2: & new2([enqueue(X)|Xs],Q) \leftarrow append(Q,[X],Q1),\ new2(Xs,Q1) \\ & new2([dequeue(X)|Xs],Q) \leftarrow append([X],Q1,Q),\ new2(Xs,Q1) \\ & new2([\,],Q) \leftarrow Q=[\,] \end{array}$$

Phase (4). By unfolding using clause C_2, by performing goal replacement steps which are justified by properties of the basic predicates (see below), and finally, by applying the equality introduction rule, we get the set V_2 consisting of the following three clauses:

$$\begin{array}{ll} D_1: & new2([enqueue(X)|Xs],Q) \leftarrow append(Q,R,L),\ R=[X|R1],\ \mathit{diff\_q}(Xs, L\backslash R1) \\ D_2: & new2([dequeue(X)|Xs],Q) \leftarrow append(Q,R,L),\ L=[X|L1],\ R\neq[X|L1], \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{diff\_q}(Xs, L1\backslash R) \\ D_3: & new2([\,],Q) \leftarrow append(Q,R,L),\ R=L \end{array}$$

We have used the following three simple properties of *append* for deriving D_1, D_2, and D_3, respectively:

$$\forall Q,X,R1,L.\ (\exists Q1.\ (append(Q,[X],Q1),\ append(Q1,R1,L)) \leftrightarrow append(Q,[X|R1],L))$$
$$\forall Q,X,R,L1.\ (\exists Q1.\ (append([X],Q1,Q),\ append(Q1,R,L1)) \leftrightarrow (append(Q,R,[X|L1]),\ R\neq[X|L1]))$$
$$\forall Q.\ (Q=[\,] \leftrightarrow \exists L,R.\ (append(Q,L,R),\ R=L))$$

Phase (5). The generation of the set *Eureka* of clauses which allows us to derive the program $Queue \wedge Eureka \wedge V_2$ from the program $Queue \wedge Eureka \wedge C_2$ is shown in Example 8 at page 48. □

Now, in order to complete the presentation of our synthesis method we need to specify Phase (5). This phase is the most complex phase of all, and in general, it cannot be constructive, that is, no algorithm exists for the generation of the set *Eureka* in all cases. We now present a strategy which is successful for the synthesis of our program for implementing queues using difference-lists (see Example 8 below) and also in many other cases.

Our strategy for Phase (5) consists of the following three steps (see Fig. 7).

Step 5.1. (*Instantiation*) Let us assume that V_2 is the set $\{D_i \mid i = 1,\dots,n\}$, where for $i = 1,\dots,n$, the clause D_i is of the form: $new2(t_i) \leftarrow Body_i$.

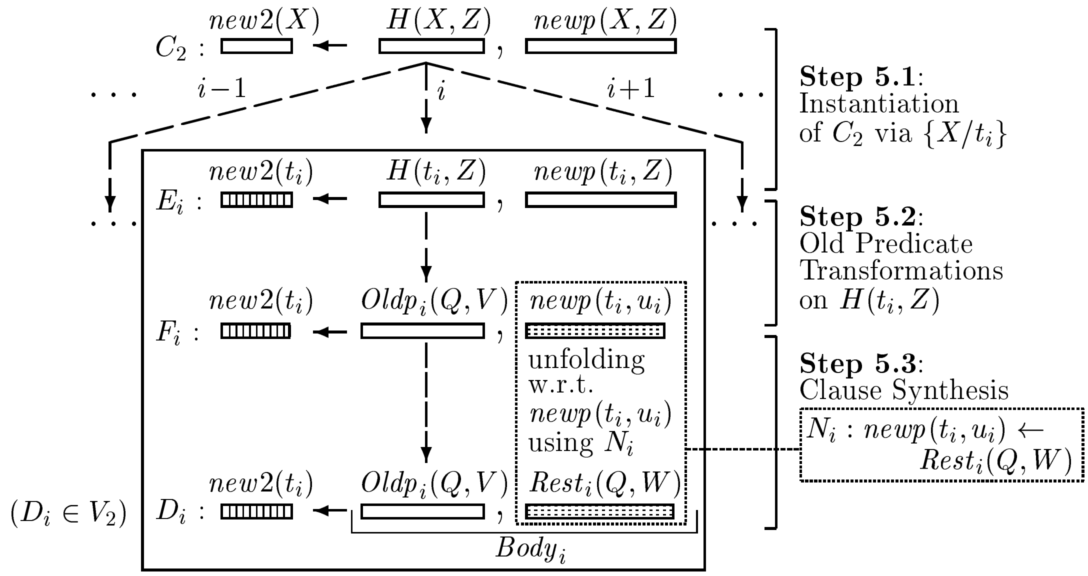

Figure 7: Steps 5.1, 5.2, and 5.3 of Phase (5) of our synthesis method.

In this step we construct a *multiset* $S_E = \{E_i \mid i = 1, \ldots, n\}$ of instances of clause C_2 such that the heads of the clauses in S_E are equal (modulo variable renaming) to the heads of the clauses in V_2.

This construction is performed by applying to C_2, for $i = 1, \ldots, n$, the substitution $\{X/t_i\}$ (possibly with identity bindings), thereby obtaining clauses $E_1, \ldots, E_n$, such that for $i = 1, \ldots, n$:

E_i: $\;new2(t_i) \leftarrow H(t_i, Z),\; newp(t_i, Z)$

where we assume that the following two constraints are satisfied: (1) the variables affected by non-identity bindings occur in $newp(X, Z)$, and (2) $vars(t_i) \cap vars(Z) = \emptyset$.

This first step is motivated by the fact that in order to derive the clauses of V_2 from C_2, we should eventually derive clauses whose heads are equal to the heads of the clauses in V_2. This instantiation may provide a guidance for further transformation steps. Notice that instantiation is not among our transformation rules. However, it may be viewed as an unfolding step using the clauses in the set *Eureka*.

Step 5.2. (*Old Predicate Transformations*) For $i = 1, \ldots, n$, starting from clause E_i, we repeatedly apply transformation rules taken from {R2, R4, R8} (with the restrictions mentioned at Point (vi) of Definition 5) with the restriction that the effect on the atom with predicate *newp* can only be an instantiation. Moreover, unfolding steps are allowed only if they produce exactly one clause, whose head is equal (modulo variable renaming) to the one of E_i. Notice that by complying with these restrictions, the body of each derived clause contains exactly one occurrence of the predicate *newp*. We stop this transformation process which started from E_i, when we get to a clause of the form:

F_i: $\;new2(t_i) \leftarrow Oldp_i(Q, V),\; newp(t_i, u_i)$

where: (i) Q is the vector of the variables occurring in $newp(t_i, u_i)$, (ii) V is the vector of the variables of F_i not occurring in Q, and (iii) D_i is equal to a clause of the form: $new2(t_i) \leftarrow Oldp_i(Q, V),\; Rest_i(Q, W)$, where W is a vector of variables which do not belong to $vars(Q, V)$. By suitable variable renamings, we may also assume that $vars(Z) \cap vars(Q, V, W) = \emptyset$.

Step 5.3. (*Clause Synthesis*) For each clause F_i derived at Step 5.2 we consider the clause:

N_i: $\mathit{newp}(t_i, u_i) \leftarrow \mathit{Rest}_i(Q, W)$

and we take *Eureka* to be the set $\{N_i \mid i = 1, \ldots, n\}$.

Now it is possible to show that there exists a transformation sequence from $P \wedge \{N_i \mid i = 1, \ldots, n\} \wedge C_2$ to $P \wedge \{N_i \mid i = 1, \ldots, n\} \wedge V_2$.

In the following example we continue Example 7 by illustrating the above Steps 5.1, 5.2, and 5.3 for implementing queues by difference-lists.

Example 8 (Implementing Queues by Difference-lists, Continued) During Phase (5) the set *Eureka* can be synthesized according to the following three steps:

Step 5.1 (*Instantiation*) By instantiation and variable renaming from C_2 we get:

E_1: $\mathit{new2}([\mathit{enqueue}(X)|Xs], Q) \leftarrow \mathit{append}(Q, R, L),\ \mathit{diff\_q}([\mathit{enqueue}(X)|Xs], L\backslash R)$
E_2: $\mathit{new2}([\mathit{dequeue}(X)|Xs], Q) \leftarrow \mathit{append}(Q, R, L),\ \mathit{diff\_q}([\mathit{dequeue}(X)|Xs], L\backslash R)$
E_3: $\mathit{new2}([\,], Q) \leftarrow \mathit{append}(Q, R, L),\ \mathit{diff\_q}([\,], L\backslash R)$

Step 5.2 (*Old Predicate Transformations*) No transformation is applied because, with reference to Fig. 7, we have that for $i = 1, 2, 3$, F_i is E_i, and

$\mathit{Oldp}_1(X, Xs, L, R, Q)$ is $\mathit{append}(Q, R, L)$
$\mathit{Rest}_1(X, Xs, L, R, R1)$ is $(R\!=\![X|R1],\ \mathit{diff\_q}(Xs, L\backslash R1))$
$\mathit{Oldp}_2(X, Xs, L, R, Q)$ is $\mathit{append}(Q, R, L)$
$\mathit{Rest}_2(X, Xs, L, R, L1)$ is $(L\!=\![X|L1],\ R\!\neq\![X|L1],\ \mathit{diff\_q}(Xs, L1\backslash R))$
$\mathit{Oldp}_3(L, R, Q)$ is $\mathit{append}(Q, R, L)$
$\mathit{Rest}_3(L, R)$ is $R\!=\!L$.

Step 5.3 (*Clause Synthesis*) The set *Eureka* consists of the clauses:

$\mathit{diff\_q}([\mathit{enqueue}(X)|Xs], L\backslash R) \leftarrow R\!=\![X|R1],\ \mathit{diff\_q}(Xs, L\backslash R1)$
$\mathit{diff\_q}([\mathit{dequeue}(X)|Xs], L\backslash R) \leftarrow L\!=\![X|L1],\ R\!\neq\![X|L1],\ \mathit{diff\_q}(Xs, L1\backslash R)$
$\mathit{diff\_q}([\,], L\backslash R) \leftarrow R\!=\!L$

By simplification of the equalities, we get:

$\mathit{diff\_q}([\mathit{enqueue}(X)|Xs], L\backslash[X|R1]) \leftarrow \mathit{diff\_q}(Xs, L\backslash R1)$
$\mathit{diff\_q}([\mathit{dequeue}(X)|Xs], [X|L1]\backslash R) \leftarrow R\!\neq\![X|L1],\ \mathit{diff\_q}(Xs, L1\backslash R)$
$\mathit{diff\_q}([\,], L\backslash L) \leftarrow$

Notice that, by *EquivQ* of Example 7, for $Q\!=\![\,]$ the following property holds:

$$M(\mathit{Queue} \wedge \mathit{Eureka}) \models \forall S.\ (q(S, [\,]) \leftrightarrow \exists L.\ \mathit{diff\_q}(S, L\backslash L))$$

Thus, we can express the predicate *queue* in terms of *diff_q* as follows (see the first clause of program *Queue* in Example 7:

$$\mathit{queue}(S) \leftarrow \mathit{diff\_q}(S, L\backslash L)$$

This clause, together with the clauses of the set *Eureka* (see Step 5.3), gives us a program analogous to the one presented in [143, page 297], except for the inequality occurring in the body of one of our final clauses. As the reader may verify, that inequality is necessary for establishing *EquivQ*. □

4.4 Reducing Nondeterminism via Synthesis from Implicit Definitions

In this section we present, through an example, a transformation technique for improving efficiency by avoiding unnecessary nondeterminism. For some steps of our technique we make use of the method of program synthesis from implicit definitions described in Section 4.3.

A well known technique for avoiding nondeterminism is based on *clause fusion* [54]. It consists in replacing the two clauses $H \leftarrow I, F$ and $H \leftarrow I, G$ by the clauses:

$H \leftarrow I, B \qquad B \leftarrow F \qquad B \leftarrow G$

where B is an atom with a new predicate symbol, say *newp*. Thus, clause fusion can be viewed as a definition step, for introducing the predicate *newp*, followed by a folding step. By this transformation we factorize the goal I which is common to the two initial clauses for H and we avoid the repeated evaluation of this atom in case of backtracking.

However, computations which are common to several clauses are not always apparent in the syntactic structure of the clauses. In fact, we may have clauses of the form: $H \leftarrow Body1$ and $H \leftarrow Body2$, where $Body1$ and $Body2$ do not syntactically include any common subgoal and yet during their evaluation, they produce redundant computations. Indeed, this is the case when there exist three goals, say I, F, and G, such that the following equivalence formulas hold in the least Herbrand model of the program at hand:

*Equiv*1: $\forall X_1 . (\exists Y_1 . Body1 \leftrightarrow \exists Z_1 . (I, F))$

*Equiv*2: $\forall X_2 . (\exists Y_2 . Body2 \leftrightarrow \exists Z_2 . (I, G))$

In *Equiv*1, X_1 denotes the vector of the variables in $vars(H) \cap vars(Body1)$, Y_1 denotes the vector of variables occurring in $Body1$ and not occurring in X_1, and Z_1 denotes the vector of variables occurring in (I, F) and not in X_1. Similarly in *Equiv*2, X_2 denotes the vector of the variables in $vars(H) \cap vars(Body2)$, Y_2 denotes the vector of variables occurring in $Body2$ and not occurring in X_2, and Z_2 denotes the vector of variables occurring in (I, G) and not in X_2.

In this case, in order to avoid unnecessary nondeterminism, we may replace $Body1$ by (I, F) and $Body2$ by (I, G) and then we apply clause fusion as described above. For these transformations to be totally correct transformations it is required that these two goal replacements are non-ascending (see Theorem 5).

We do not want to study in detail here algorithmic methods for finding such goals I, F, and G. We only present a technique which will see in action in Example 9 below. The first step of this technique is a preliminary analysis of the resolution steps starting from the goal H, and this analysis may suggest us a suitable choice for the goal I. Then, in order to construct the goals F and G such that *Equiv*1 and *Equiv*2 hold for the chosen goal I, we may apply our synthesis method from implicit definitions. For instance, in order to construct F we may introduce a new predicate, say f, which is implicitly defined by the fact that the following equivalence formula holds in the least Herbrand model of the program at hand:

*Equiv*3: $\forall X_1 . (\exists Y_1 . Body1 \leftrightarrow \exists Z_1 . (I, f(X_1, Z_1)))$

where Z_1 is the vector of the variables occurring in I and not in X_1.

If we are able to construct a set *EurekaF* of clauses defining f, by using our synthesis method, then we may stipulate that the goal F is $f(X_1, Z_1)$. Analogously, we can apply our synthesis method for constructing the goal G, thereby deriving the set *EurekaG* of clauses which define a new predicate, say g.

Our technique will then continue by transforming the sets *EurekaF* and *EurekaG* of clauses defining f and g, because they, in turn, may still contain some unnecessary nondeterminism.

The reader should notice that the addition to the program at hand of the clauses *EurekaF* and *EurekaG* may not be possible by rule R1 which does not allow for the introduction of recursive definitions. However, we can use Theorem 5 for showing the total correctness of a suitable transformation sequence constructed by applying our technique for avoiding nondeterminism, because given the initial program $P_0^+ = P_0 \wedge \mathit{EurekaF} \wedge \mathit{EurekaG}$, Theorem 5 ensures that the transformation sequence from program $P_0^+ \wedge (H \leftarrow \mathit{Body}1) \wedge (H \leftarrow \mathit{Body}2)$ to the new program $P_0^+ \wedge (H \leftarrow I,\ F) \wedge (H \leftarrow I,\ G)$ is totally correct if it is based on non-ascending goal replacements.

In the following example we indeed verify that the goal replacements we perform are non-ascending by constructing non-ascending unfold/fold proofs and by using Theorem 10.

Example 9 (Reachability in a graph) Let us consider the following program P_0 which defines the reachability relation in a directed graph:

1. $\mathit{reach}(X) \leftarrow \mathit{initial}(X)$
2. $\mathit{reach}(X) \leftarrow \mathit{reach}(Y),\ \mathit{edge}(Y, X)$

together with some suitable clauses defining the *initial* and *edge* predicates, which we do not show here. All predicates in P_0 are assumed to be non-basic.

Program P_0 may perform some redundant computations when evaluating the goal $\mathit{reach}(X)$, where X is an unbound variable, according to the standard left-to-right, depth-first Prolog strategy. Indeed, the computed answer substitutions for the goal $\mathit{reach}(X)$ are obtained by evaluating goals of the form $(\mathit{initial}(X_0),\ \mathit{edge}(X_0, X_1),\ \mathit{edge}(X_1, X_2),\ \ldots,\ \mathit{edge}(X_n, X))$ for increasing values of n. However, the partial results obtained during the evaluation of 'shorter' goals, are not taken into account for the evaluation of 'longer' goals, and equal subgoals may be repeatedly evaluated along different branches of the SLD-tree.

In order to discover redundant computations of this kind, we may symbolically generate and examine the set of the SLD-derivations starting from a given goal. In particular, by constructing a finite upper portion of the SLD-tree starting from the goal of interest, we may find equal subgoals which have to be evaluated along distinct branches of that SLD-tree, and thus, they produce redundant computations.

In our case, in order to avoid unnecessary computations and reduce nondeterminism, we may apply the technique we have described above. It consists in looking for some goals I, F, and G, such that clauses 1 and 2 can be rewritten as:

$1'$. $\mathit{reach}(X) \leftarrow I,\ F$
$2'$. $\mathit{reach}(X) \leftarrow I,\ G$

We start off by considering the upper portion of the SLD-tree with root goal $\mathit{reach}(X)$ depicted in Fig. 8.

By analyzing that SLD-tree, we discover that for all successful SLD-derivations starting from $\mathit{reach}(X)$ we will get to a goal of the form $\mathit{initial}(Z), E$ where E is a goal whose definition may depend on the SLD-derivation. In particular, (i) the SLD-derivation which uses clause 1 in the first step, contains an occurrence of the goal $\mathit{initial}(Z)$ with $Z = X$ (see goal $N0$), and (ii) all successful SLD-derivations which use clause 2 in the first step, contain a goal of the form: $(\mathit{initial}(Z),\ G(Z, X))$ where Z is a variable distinct from X and $G(Z, X)$ is a goal whose definition depends on the SLD-derivation. Property of Point (ii) derives from the fact that a leftmost subgoal of the goal $N2$ is an instance of the goal in M (see the dashed arrow in Fig. 8).

As a result of this analysis, we may conclude that: (i) the body of clause 1 is equivalent to the goal $(\mathit{initial}(Z), Z = X)$ and (ii) the body of clause 2 is equivalent to a goal of the form $(\mathit{initial}(Z),\ G(Z, X))$ for a suitable goal $G(Z, X)$.

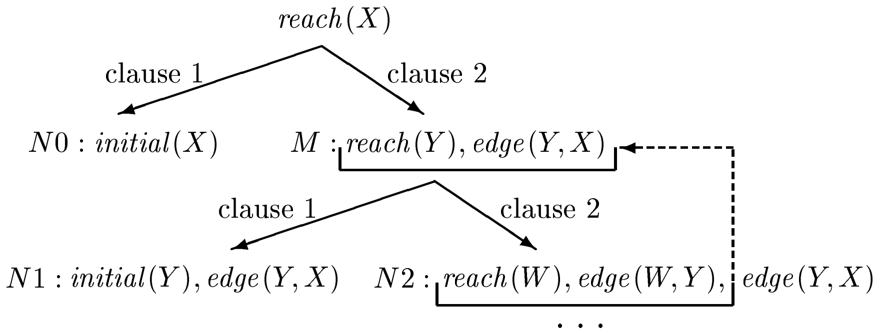

Figure 8: An upper portion of the SLD-tree for $reach(X)$.

Thus, in clauses $1'$ and $2'$ we can choose the goal I to be $initial(Z)$, the goal F to be $Z = X$, and the goal G to be $g(Z,X)$, where $g(Z,X)$ is a new predicate implicitly defined by:

*Equiv*4: $\quad M(P_0 \wedge EurekaG) \models \forall X.\, (\exists Y.\, (reach(Y),\ edge(Y,X)) \leftrightarrow \exists Z.\, (initial(Z),\ g(Z,X)))$

The suitable set *EurekaG* of clauses defining $g(Z,X)$ can be generated by using, as we will indicate below, our five phase synthesis method.

When we have the set *EurekaG*, the transformation continues by adding this set of clauses to P_0 and replacing clauses 1 and 2 (by using the equality introduction and goal replacement rules, respectively) by the following two clauses:

1*. $\quad reach(X) \leftarrow initial(Z),\ Z = X$
2*. $\quad reach(X) \leftarrow initial(Z),\ g(Z,X)$

Here are the five phase synthesis method for constructing the set *EurekaG*.

Phase (1). We introduce two clauses:

$C_1 : \quad new1(X) \leftarrow\ reach(Y),\ edge(Y,X)$
$C_2 : \quad new2(X) \leftarrow initial(Z),\ g(Z,X)$

Phase (2). By unfolding clause C_1 w.r.t. $reach(Y)$ we get the set R_1 consisting of the following two clauses:

$R_1 : \quad new1(X) \leftarrow initial(Y),\ edge(Y,X)$
$\qquad\ \ new1(X) \leftarrow initial(Y1),\ edge(Y1,Y),\ edge(Y,X)$

and then by folding we derive the set S_1 of clauses:

$S_1 : \quad new1(X) \leftarrow initial(Y),\ edge(Y,X)$
$\qquad\ \ new1(X) \leftarrow new1(Y),\ edge(Y,X)$

Phase (3). By replacing in S_1 the predicate symbol $new1$ by $new2$ we get the following set S_2 of clauses:

$S_2 : \quad new2(X) \leftarrow initial(Y),\ edge(Y,X)$
$\qquad\ \ new2(X) \leftarrow new2(Y),\ edge(Y,X)$

Phase (4). By unfolding the second clause in S_2 using clause C_2 and by variable renaming, we get the set V_2 consisting of the two clauses:

$D_1 : \quad new2(X) \leftarrow initial(Z),\ edge(Z,X)$
$D_2 : \quad new2(X) \leftarrow initial(Z),\ g(Z,Y),\ edge(Y,X)$

Phase (5). The set $EurekaG$ can now be synthesized according to the following three steps.

Step 5.1. (*Instantiation*) By instantiation from C_2 we get two copies of the same clause:

E_1: $new2(X) \leftarrow initial(Z),\ g(Z,X)$
E_2: $new2(X) \leftarrow initial(Z),\ g(Z,X)$

Step 5.2. (*Old Predicate Transformations*) We do not apply any transformation rule to E_1 or E_2 because the conditions stated in Step 5.2 of Section 4.3 for stopping the transformation process are already satisfied. Indeed, with reference to Fig. 7 we have that:

F_1 is E_1, F_2 is E_2, and
$Oldp_1(Z,X)$ is $initial(Z)$, $Rest_1(Z,X)$ is $edge(Z,X)$,
$Oldp_2(Z,X)$ is $initial(Z)$, $Rest_2(Z,Y,X)$ is $(g(Z,Y),\ edge(Y,X))$.

Step 5.3. (*Clause Synthesis*) The set $EurekaG$ consists of the following clauses:

3. $g(Z,X) \leftarrow edge(Z,X)$
4. $g(Z,X) \leftarrow g(Z,Y), edge(Y,X)$

Thus, the derived program version P_1 is:

1*. $reach(X) \leftarrow initial(Z),\ Z=X$
2*. $reach(X) \leftarrow initial(Z),\ g(Z,X)$
3. $g(Z,X) \leftarrow edge(Z,X)$
4. $g(Z,X) \leftarrow g(Z,Y),\ edge(Y,X)$

together with the clauses defining the predicates *initial* and *edge*. By Theorem 5 we have that $M(P_0 \wedge EurekaG) = M(P_1 \wedge EurekaG)$ because it is the case that the replacement of the body of clause 2 by $(initial(Z),\ g(Z,X))$ is a non-ascending goal replacement. This property can be established by providing a non-ascending unfold/fold proof of $Equiv4$ using $P_0 \wedge EurekaG$ and by applying Theorem 10. We now show the two transformation sequences T_1 and T_2 which constitute that proof and then we show that the proof is non-ascending.

The initial program of T_1 is $P_0 \wedge EurekaG$. By definition introduction we derive $P_0 \wedge EurekaG \wedge C_1$, then by unfolding C_1 w.r.t. $reach(Y)$ we get $P_0 \wedge EurekaG \wedge R_1$, and finally, by folding we get $P_0 \wedge EurekaG \wedge S_1$.

The initial program of T_2 is $P_0 \wedge EurekaG$. By definition introduction we derive $P_0 \wedge EurekaG \wedge C_2$, then by unfolding C_2 w.r.t. $g(Z,X)$ we get $P_0 \wedge EurekaG \wedge D_1 \wedge D_2$, and finally, by folding clause D_2 using C_2 we get $P_0 \wedge EurekaG \wedge S_2$.

The transformation sequences T_1 and T_2 constitute a non-ascending proof because: (i) every sequence of clauses $C_1 \Rightarrow \ldots \Rightarrow E_1$ where E_1 is a clause in S_1, contains precisely one step of unfolding w.r.t. a non-basic atom, and similarly, (ii) every sequence of clauses $C_2 \Rightarrow \ldots \Rightarrow E_2$ where E_2 is a clause in S_2, contains precisely one step of unfolding w.r.t. a non-basic atom (for the notion of a non-ascending proof given in Definition 5 at page 41 the number of applications of the other transformation rules is not significant).

Now we may continue our derivation from program P_1 and we may apply the clause fusion technique. Thus, we replace clauses 1* and 2* by the following three clauses (by performing a definition introduction and a folding step):

5. $reach(X) \leftarrow initial(Z),\ b(Z,X)$
6. $b(Z,X) \leftarrow Z=X$
7. $b(Z,X) \leftarrow g(Z,X)$

The current program version, call it P_2, consists of clauses 3, 4, 5, 6, and 7, together with the clauses for *edge* and *initial*. Now we continue our derivation by transforming the clauses for the predicate g. We consider an upper portion of the SLD-tree with root-goal $g(Z,X)$ and we perform an analysis similar to the one described above for $reach(X)$. By this analysis we get that the body of clause 4 is equivalent to a goal of the form $(edge(Z,V),\ Q(V,X))$. Thus, we may apply our synthesis method by introducing the new predicate q implicitly defined by the equivalence formula:

*Equiv*5: $M(P_2 \wedge EurekaQ) \models \forall X, Z.\, (\exists Y.\, (g(Z,Y),\ edge(Y,X) \leftrightarrow \exists V.\, (edge(Z,V),\ q(V,X)))$

where *EurekaQ* is the set of clauses which should be synthesized and thus, it provides the definition of the predicate q.

By one more application of our five phase method which we do not present here, we derive the following *EurekaQ* clauses for the predicate q:

8. $q(Z,X) \leftarrow edge(Z,X)$
9. $q(Z,X) \leftarrow q(Z,Y),\ edge(Y,X)$

Thus, the clauses defining q are equal to those defining g and we may replace q by g in *Equiv*5, and we get:

*Equiv*6: $M(P_2) \models \forall X, Z.\, (\exists Y.\, (g(Z,Y),\ edge(Y,X) \leftrightarrow \exists V.\, (edge(Z,V),\ g(V,X)))$

By a goal replacement justified by *Equiv*6, from clause 4 we derive:

10. $g(Z,X) \leftarrow edge(Z,V),\ g(V,X)$

We may easily verify that also this goal replacement step is non-ascending (by constructing a non-ascending unfold/fold proof of *Equiv*6) and therefore $M(P_2) = M(P_3)$, where P_3 is the program obtained from P_2 by replacing clause 4 by clause 10. Now we may apply the clause fusion technique to clauses 3 and 10. In order to do so, we apply rule R8 to clause 3 and we get:

3′. $g(Z,X) \leftarrow edge(Z,V),\ V = X$

and then we fold clauses 3′ and 10 by using clauses 6 and 7. We thus derive:

11. $g(Z,X) \leftarrow edge(Z,V),\ b(V,X)$

The current program version P_4 is:

5. $reach(X) \leftarrow initial(Z),\ b(Z,X)$
6. $b(Z,X) \leftarrow Z = X$
7. $b(Z,X) \leftarrow g(Z,X)$
11. $g(Z,X) \leftarrow edge(Z,V),\ b(V,X)$

together with the clauses 8 and 9 and the clauses for *edge* and *initial*. By some final transformation steps by which (i) we unfold the equality in the body of clause 6 (by rule R8), (ii) we unfold clause 7 w.r.t. the predicate g, and (iii) we discard clauses 8 and 9 because the predicate q is not needed, we get the following final program P_5 (apart from the clauses for *edge* and *initial*):

5. $reach(X) \leftarrow initial(Z),\ b(Z,X)$
12. $b(X,X) \leftarrow$
13. $b(Z,X) \leftarrow edge(Z,V),\ b(V,X)$

This program is right-recursive and it computes the set of reachable vertices from the given initial ones in a *forward-chaining* fashion. This means that the evaluation of the goal $reach(X)$ is done by program P_5 in a more deterministic way w.r.t. the initial program P_0 which, instead, is left-recursive and evaluates the goal $reach(X)$ in a *backward-chaining* fashion. □

A similar transformation of left-recursive programs into right-recursive programs was presented in [23] where, however, the transformation of P_0 into P_5 is presented in one 'big step' as a schema-based transformation, which is validated by an ad-hoc inductive proof.

4.5 Bibliographic Remarks

The use of unfold/fold rules for the verification of program properties has been often suggested since the early days of program transformation [29]. In this Section 4 we have formalized a method based on unfold/fold transformations, called the unfold/fold proof method, which can be used for proving properties of logic programs w.r.t. the least Herbrand model semantics. Since the unfold/fold transformations may be designed to preserve many different semantics (see, for instance, [117]), one may extend our method to prove properties w.r.t. those semantics as well.

We have provided some conditions which ensure that, when a property is used as a lemma to perform program transformations, these transformations are indeed totally correct, that is, they preserve the least Herbrand model semantics. These conditions rely on the existence of a *non-ascending* unfold/fold proof of the property of interest, and thus, since they refer to finite objects, they have a more constructive nature w.r.t. other techniques, such as those based on *consistency with weight tuple measures* [145] or *non-increasingness* [15], which rely on the verification of properties of possibly infinite sets of SLD-derivations.

We have also presented a method for synthesizing programs from unfold/fold proofs of program properties. Although our method makes use of unfold/fold transformations, it falls into the category of *synthesis methods* because the initial specification is not in Horn clause form (that is, it is not a logic program), but it is assumed to be a more general formula of the form:

$$\forall X. (\exists Y. F(X,Y) \leftrightarrow \exists Z. (H(X,Z),\ newp(X,Z))) \tag{1}$$

where $F(X,Y)$ and $H(X,Z)$ are conjunctions of atoms and *newp* is the predicate for which we would like to synthesize a program. Specifications of the form (1) can be considered to be an implicit definition of the new predicate *newp*, and thus, we say that our proposed technique is a *synthesis method from implicit definitions.*

A very large number of synthesis methods have been proposed in the literature. All these methods may vary because of: (i) the form of the initial specification, (ii) the rules used for deriving programs from specifications, and (iii) the language used for the synthesized programs (see, for instance, [28, 59, 69, 85] for references in the case of logic programs).

Our synthesis method is related to the methods for logic program synthesis (see, for instance, [85]) where the initial specification is an equivalence formula of the first order predicate calculus and one is allowed to use derivation rules similar to the unfold/fold rules. These methods, called *deductive synthesis* methods in the survey paper [59], allow for initial specifications of the form:

$$\forall X. (spec(X) \leftrightarrow newp(X))$$

where *newp* is the predicate for which we want to synthesize a program and *spec* is any formula of the first order predicate calculus. Thus, no implicit definitions like those provided by formulas of the form (1) above are allowed.

Our synthesis method can also be viewed as a technique for the extraction of a program from an unfold/fold proof. Thus, the basic idea of our method is also related to the *proofs-as-programs* approach [7, 28, 69, 107] whereby the constructive proof of a property can be used for synthesizing a program which satisfies that property. However, between our approach and the proofs-as-programs approach, there are many differences. Among them we recall the differences due to: (i) the derivation rules considered (in particular, *constructive type theory* is used in [7, 28], untyped first-order logic is used in [107], and *extended execution* is used in [69]), (ii) the languages in which the synthesized programs are written (indeed, the authors of [7, 107] consider applicative languages), and (iii) the form of the specifications. With reference to this last difference, one should notice that the synthesis methods based on the proofs-as-programs approach are used for synthesizing programs from specifications of the form:

$$\forall X.\, \exists Y.\, spec(X,Y)$$

Thus, in the case of functional programming, this means that a synthesized program corresponds to a total function f such that $\forall X.\, spec(X, f(X))$, or equivalently, $\forall X.\ \forall Y.\ (Y = f(X) \rightarrow spec(X,Y))$. This specification is less general than the implicit definitions considered in this section. The same holds in the case of logic programming.

The synthesis method we propose also extends the standard techniques which are currently available in the framework of unfold/fold program transformation. An informal argument to support this claim can be given as follows.

By using the unfold/fold rules as defined in [144], a new predicate, say *newp*, can be introduced in terms of already available predicates only in an explicit way, in the sense that one may add to the current program P a clause C of the form:

$$newp(X) \leftarrow F(X,Y)$$

where F is a conjunction of atoms whose predicates occur in P. Thus, in the least Herbrand model of $P \wedge C$ the new predicate *newp* is specified by the formula:

$$\forall X.\, (\exists Y.\, F(X,Y) \leftrightarrow newp(X))$$

which is a less general formula than the ones we have considered in this section.

Implicit definitions are also considered in [91], where some modifications of the unfolding and folding rules are introduced to deal with generalized definitions of the form:

$$(H(X),\ newp(X)) \leftarrow F(X,Y)$$

for some goal $H(X)$ and $F(X,Y)$. In our method we do not need to introduce any modified rule. Moreover, in [91] the form of allowed derivations is very restricted, while in our case, the unfold/fold proofs may be of any general form.

The reader may also verify that the difference-list transformation example we have presented cannot be derived in a natural way by using the unfold/fold transformations of [144]. Some modified versions of the rules should be used instead, like, for instance, the *inverse definition* and the *data structure mapping* introduced by [156].

Our example on the avoidance of nondeterminism shows that the unfold/fold transformation technique enhanced with our synthesis method, is able to derive programs for which other methods require the off-line proof of some insightful lemmas (like the schema-based equivalence in [23]). By using our synthesis method, in fact, we produce equivalences which may be used as lemmas during the program derivation itself. Thus, the synthesis method we propose may also be useful to enhance other unfold/fold-based techniques for avoiding nondeterminism which do not use lemmas (like, for instance, [120]).

Finally, we would like to point out that our synthesis method can be used for the program specialization as indicated in [119], and the resulting program specialization method is strictly more general than the usual partial evaluation methods [104]. Indeed, we are able to specialize our initial program w.r.t. a set of input values which can be described by *any* predicate, while in [104] the set of input values can only be a set of instances of a given tuple of terms.

More formally, by using our method one can solve program specialization problems specified by the formula:

$$\forall X.\, ((input(X),\, p(X)) \leftrightarrow (input(X),\, spec\_p(X)))$$

where $input(X)$ is any predicate and $spec\_p(X)$ is the specialized version of $p(X)$ which satisfies $input(X)$ for each X. The methods based on [104] can only solve problems specified by:

$$\forall Y.\, (p(t(Y)) \leftrightarrow spec\_p(t(Y)))$$

which can be viewed as a special case of the above specification when $X = t(Y)$ and $input(t(Y))$ is true.

A final remark concerns the mechanization of our synthesis method. As it is the case for general purpose synthesis and transformation techniques, suitable strategies need to be devised for dealing with particular classes of program specifications and ensuring the derivation of efficient programs. If one uses our approach, there are various strategies, such as the ones described in [117], which are available, and one can indeed apply them for guiding the application of the unfold/fold transformation rules.

5 Rules and Strategies for Logic Programs with Negation

In this section we present the 'rules + strategies' approach to program transformation in the case of logic programs with stratified negation which in this section we simply call *programs*, when no confusion arises. We know that definite programs are Turing complete in the sense that every computable function can be written as a definite program, thus, in principle, it is not necessary to deal with the case of logic programs with negation. Nonetheless, we would like to deal with programs with negation because these programs can be very useful in practice to express in a direct way the problems at hand or the properties of interest, as the examples below will demonstrate.

5.1 Syntax and Semantics of Stratified Logic Programs

In this section we recall some basic definitions concerning *locally stratified logic programs* and *perfect models*. For notions not defined here the reader may refer to [4, 103, 127].

The formulas and programs we consider are constructed by using a fixed first order language $\mathcal{L}$. Programs are conjunctions of clauses of the form: $H \leftarrow L_1, \ldots, L_n$, where H is an atom different from *true*, *false*, $=$, and $\neq$, and for every $i = 1, \ldots, n$, L_i is a literal, that is, an atom (i.e., *positive* literal) or a negated atom (i.e., *negative* literal). A goal is a conjunction of literals. The empty conjunction is *true* and the empty disjunction is *false*. In this section we assume that conjunction is associative, commutative, and its neutral element is *true*. Thus, the order of literals in a goal is immaterial. A clause of the form $H \leftarrow true$ is said to be a *unit* clause and it is also written as $H \leftarrow$. A clause (or a program) is said to be *definite* iff no negated atom occurs in it. The predicate symbol of the head of a clause C is called the *head predicate* of C.

In this Section 5 variables are denoted by lower case letters. Given a clause C, a variable in $bd(C)$ is said to be *existential* iff it belongs to $vars(bd(C)) - vars(hd(C))$. Given a formula φ we denote by $freevars(\varphi)$ the set of all free variables of φ. A clause C is said to be *ground* iff no variable occurs in it. A literal is said to be *propositional* iff its predicate symbol is nullary, that is, it has arity 0. A goal (or a clause, or a program) is propositional iff all its literals are propositional. A formula is said to be *function-free* iff no function symbols occur in it.

The *definition* of a predicate p in a program P, denoted by $Def(p, P)$, is the conjunction of all clauses of P whose head predicate is p. We say that *p is defined in P* iff $Def(p, P)$ is not empty. The *extended definition* of a predicate p in a program P, denoted by $Def^*(p, P)$, is the conjunction of the definition of p and the definitions of all the predicates on which p depends in P.

The set of the *useless* predicates of a program P is the maximal set U of predicates of P such that a predicate p is in U iff the body of each clause of $Def(p, P)$ has a positive literal whose predicate is in U. For instance, p and q are useless and r is not useless in the program $(p \leftarrow q, r) \wedge (q \leftarrow p) \wedge (r \leftarrow)$.

By $ground(P)$ we denote the conjunction of all clauses in $\mathcal{L}$ which are ground instances of clauses of P, and by $B_\mathcal{L}$ we denote the *Herbrand Base* of $\mathcal{L}$, that is, the set of all ground atoms in $\mathcal{L}$ whose predicate is different from *true*, *false*, $=$, and $\neq$. A stratification σ is a total function from $B_\mathcal{L} \cup \{true, false, =, \neq\}$ to the set W of countable ordinals so that $\sigma(true) = \sigma(false) = \sigma(=) = \sigma(\neq) = 0$. Given a ground literal L which is the atom A or the negated atom $\neg A$, we say that L is in *stratum* α iff $\sigma(A) = \alpha$.

A ground clause $H \leftarrow L_1, \ldots, L_n$ is locally stratified w.r.t. a stratification σ iff for every $i = 1, \ldots, n$, if L_i is an atom then $\sigma(H) \geq \sigma(L_i)$, and if L_i is a negated atom then $\sigma(H) > \sigma(L_i)$.

We say that the program P is *locally stratified* iff there exists a stratification σ such that every clause in *ground*(P) is locally stratified w.r.t. σ.

Let P_α be the conjunction of the clauses in *ground*(P) whose head is in the stratum α. We may assume without loss of generality, that the stratum of every ground literal is greater than 0, so that P_0 may be assumed to be the empty conjunction of clauses. The *perfect model* $M(P)$ of a program P which is locally stratified w.r.t. a stratification σ, is the subset $\bigcup_{\alpha \in W} M_\alpha$ of $B_{\mathcal{L}}$, where for every ordinal α in W, the set M_α is defined as follows:
(1) M_0 is the empty set,
(2) for every successor ordinal $\alpha+1$, $M_{\alpha+1}$ is the least Herbrand model of the definite logic program derived from $P_{\alpha+1}$ as follows: (i) every literal L in stratum τ ($\leq \alpha$) in the body of a clause in $P_{\alpha+1}$ is deleted iff $M_\tau \models L$, and (ii) every clause C in $P_{\alpha+1}$ is deleted iff in $bd(C)$ there exists a literal L in stratum τ ($\leq \alpha$) such that $M_\tau \not\models L$, and
(3) for every limit ordinal λ, M_λ is $\bigcup_{\alpha<\lambda} M_\alpha$.

Note that in this section $M(P)$ denotes the perfect model of the program P, not its least Herbrand model.

5.2 Transformation Rules for Stratified Logic Programs

In this section we present the rules for transforming logic programs and we provide a sufficient condition which ensures that perfect models are preserved during program transformation. Our set of transformation rules is an enhancement over the previous proposal by Seki [135] who considers the Definition, Positive Unfolding and Folding rules only.

Similarly to the case of definite programs (see Section 2), for the application of the transformation rules we divide the predicate symbols of the language into two categories: *basic* predicates and *non-basic* predicates. Atoms, literals, and goals which have occurrences of basic predicates only, are called *basic atoms*, *basic literals*, and *basic goals*, respectively. We assume that *true*, *false*, $=$, and $\neq$ are basic predicates, and also that each basic atom is in a strictly smaller stratum w.r.t. any non-basic atom.

Our partition of the set of predicates into basic or non-basic predicates is arbitrary and it is required only for proving the Correctness Theorem 12 below. The actual partition one may want to consider in practice, depends on the given initial program P_0. For instance, if one is interested in proving properties of a program on the domain of integer numbers, one may assume that addition, multiplication, and ordering relations are defined by basic predicates.

For $0 \leq k \leq n$, we also consider the conjunction $Defs_k$ of the clauses introduced by the definition rule (see rule R1) during the transformation sequence $P_0, \ldots, P_k$.

R1. Definition Rule. We get program P_{k+1} by adding to program P_k a clause C of the form: $newp(x_1, \ldots, x_m) \leftarrow L_1, \ldots, L_n$, with $n > 0$, such that:
(i) the predicate *newp* is a non-basic predicate which does not occur in $P_0 \wedge Defs_k$,
(ii) $x_1, \ldots, x_m$ are distinct variables occurring in $L_1, \ldots, L_n$,
(iii) all predicates occurring in $L_1, \ldots, L_n$ also occur in P_0, and
(iv) at least one literal among $L_1, \ldots, L_n$ is a non-basic positive literal.

R2. Positive Unfolding Rule. Let C be a renamed apart clause in P_k of the form $H \leftarrow G_1, A, G_2$, where A is an atom whose predicate is different from *true*, *false*, $=$, and $\neq$, and G_1 and G_2 are (possibly empty) goals. Suppose that:

1. $D_1, \ldots, D_m$, with $m \geq 0$, are all clauses of program P_k, such that A is unifiable with $hd(D_1)$, $\ldots$, $hd(D_m)$, with most general unifiers $\theta_1, \ldots, \theta_m$, respectively, and

2. C_i is the clause $(H \leftarrow G_1, bd(D_i), G_2)\theta_i$, for $i = 1, \ldots, m$.

By *unfolding C w.r.t. A* we derive from program P_k the new program P_{k+1} by replacing C by $C_1, \ldots, C_m$.

In particular, if $m = 0$, that is, if we unfold a clause C in program P_k w.r.t. an atom which is not unifiable with the head of any clause in P_k, then we derive from program P_k the new program P_{k+1} by deleting clause C.

R3. Negative Unfolding Rule. Let C be a renamed apart clause in P_k of the form $H \leftarrow G_1, \neg A, G_2$, where A is an atom whose predicate is different from *true*, *false*, $=$, and $\neq$, and G_1 and G_2 are (possibly empty) goals. Let $D_1, \ldots, D_m$, with $m \geq 0$, be all clauses of program P_k, such that A is unifiable with $hd(D_1), \ldots, hd(D_m)$, with most general unifiers $\theta_1, \ldots, \theta_m$, respectively. Assume that:

1. $A = hd(D_1)\theta_1 = \cdots = hd(D_m)\theta_m$, that is, for $i = 1, \ldots, m$, A is an instance of $hd(D_i)$,
2. for $i = 1, \ldots, m$, D_i has no existential variables, and
3. from G_1, $\neg(bd(D_1)\theta_1 \vee \ldots \vee bd(D_m)\theta_m)$, G_2 we get an equivalent disjunction $Q_1 \vee \ldots \vee Q_r$ of goals, with $r \geq 0$, by first pushing $\neg$ inside and then pushing $\vee$ outside.

By *unfolding C w.r.t. $\neg A$* we derive from program P_k the new program P_{k+1} by replacing C by $C_1, \ldots, C_r$, where C_i is the clause $H \leftarrow Q_i$, for $i = 1, \ldots, r$.

In particular: (i) if $m = 0$, that is, if we unfold a clause C w.r.t. a negative literal $\neg A$ such that A is not unifiable with the head of any clause in P_k, then we get the new program P_{k+1} by deleting $\neg A$ from the body of clause C, and (ii) if for some $i \in 1, \ldots, m$, $bd(D_i) = true$, that is, if we unfold a clause C w.r.t. a negative literal $\neg A$ such that A is an instance of the head of a unit clause in P_k, then we derive from program P_k the new program P_{k+1} by deleting clause C.

R4. Folding Rule. Let D be a renamed apart definition in $Defs_k$ and C be a clause in P_k of the form $H \leftarrow G_1, B, G_2$, where B, G_1, and G_2 are (possibly empty) goals. Suppose that for some substitution θ: (i) $B = bd(D)\theta$, and (ii) for every variable x in the set $vars(D) - vars(hd(D))$, we have that $x\theta$ is a variable which occurs neither in $\{H, G_1, G_2\}$ nor in the term $y\theta$, for any variable y occurring in $bd(D)$ and different from x.
By *folding* clause C w.r.t. B using clause D we derive clause $E : H \leftarrow G_1, hd(D)\theta, G_2$ and we get from program P_k the new program P_{k+1} by replacing C by E.

R5. Tautology Rule. We get the new program P_{k+1} by replacing in P_k a conjunction of clauses by the corresponding equivalent conjunction of clauses, according to the following equivalences, where G and R denote (possibly empty) goals, and H and A denote atoms:

(1) $(H \leftarrow A, \neg A, G) \quad \leftrightarrow \quad true$
(2) $(H \leftarrow H, G) \quad \leftrightarrow \quad true$
(3) $(H \leftarrow G,\ \ H \leftarrow G, R) \quad \leftrightarrow \quad (H \leftarrow G)$
(4) $(H \leftarrow A, G, R,\ \ H \leftarrow \neg A, G) \quad \leftrightarrow \quad (H \leftarrow G, R,\ \ H \leftarrow \neg A, G)$

R6. Clause Deletion Rule. We get the new program P_{k+1} by removing from P_k the definitions of the useless predicates of P_k.

R7. Basic Goal Replacement Rule. Let C be a renamed apart clause in P_k of form $H \leftarrow G_1, Q, G_2$, where Q, G_1, and G_2 are (possibly empty) goals. Suppose that, for some goal R,

$M(P_0) \models \forall x_1 \ldots x_u. (\exists y_1 \ldots y_v. Q \leftrightarrow \exists z_1 \ldots z_w. R)$
where: (i) $\{y_1, \ldots, y_v\} = vars(Q) - vars(H, G_1, G_2)$, (ii) $\{z_1, \ldots, z_w\} = vars(R) - vars(H, G_1, G_2)$, and (iii) $\{x_1, \ldots, x_u\} = vars(Q, R) - \{y_1, \ldots, y_v, z_1, \ldots, z_w\}$. Suppose also that Q and R are basic goals and H is a non-basic atom. Then from program P_k we derive the new program P_{k+1} by replacing C by the clause $H \leftarrow G_1, R, G_2$.

Theorem 12 (Correctness of the Transformation Rules) *Let $P_0, \ldots, P_n$ be a transformation sequence such that the following holds:* if *for some k with $1 \leq k < n$, we have applied rule* R4 *for folding clause C in P_k using clause D in $Defs_k$,* then *there exists i, with $0 \leq i < k$ such that D occurs in P_i and P_{i+1} is derived from P_i by positive unfolding of D w.r.t. a non-basic atom. Then we have that $M(P_0 \wedge Defs_n) = M(P_n)$.*

Notice that the statement obtained from Theorem 12 by replacing 'positive unfolding' by 'negative unfolding' is not a theorem as shown by the following example. Let P_0 be the program $(p \leftarrow \neg q(x)) \wedge (q(x) \leftarrow q(x)) \wedge (q(x) \leftarrow r)$. By negative unfolding w.r.t. $\neg q(x)$ we get program P_1: $(p \leftarrow \neg q(x), \neg r) \wedge (q(x) \leftarrow q(x)) \wedge (q(x) \leftarrow r)$. Then by folding we get program P_2: $(p \leftarrow p, \neg r) \wedge (q(x) \leftarrow q(x)) \wedge (q(x) \leftarrow r)$. We have that $M(P_0) \models p$, while $M(P_2) \models \neg p$.

5.3 Verification of Properties with respect to Perfect Models

In this section we show how program transformation rules and strategies may be used for proving the satisfiability of first order formulas in the perfect models. In particular, we propose a technique for showing that a closed first order formula φ holds in the perfect model $M(P)$ of a logic program P with locally stratified negation. This property is denoted by $M(P) \models \varphi$.

Our proof method for showing that $M(P) \models \varphi$ holds, consists of two steps:

Step 1: we use a variant of the Lloyd-Topor transformation [103] for transforming a statement of the form: $f \leftarrow \varphi$ where f is a new predicate symbol, into a conjunction $F(f, \varphi)$ of clauses, possibly with negated atoms in their bodies, such that $P \wedge F(f, \varphi)$ is locally stratified and $M(P) \models \varphi$ iff $M(P \wedge F(f, \varphi)) \models f$, and

Step 2: we show that $M(P \wedge F(f, \varphi)) \models f$ holds by applying transformation rules which preserve perfect models, and deriving from $P \wedge F(f, \varphi)$ a new program of the form: $Q \wedge (f \leftarrow)$.

Since f is false in the perfect model of any program which has no clause with head f, our method can also be used to show that $M(P) \not\models \varphi$, (that is, $M(P) \models \neg\varphi$), by deriving from $P \wedge F(f, \varphi)$ a new program R, such that f does not occur in the head of any clause in R.

We illustrate our transformational proof method using the following example where we prove a property of a given transition system.

Example 10 (Semaphore) Consider the following program P, where x and y are variables, and $s \ldots s0$ with n occurrences of the successor function s, denotes the natural number n:

1. $down(sx) \leftarrow \neg down(x)$
2. $up(0, 0) \leftarrow$
3. $up(sx, 0) \leftarrow down(x)$
4. $up(sx, sy) \leftarrow up(sx, y),\ x > y$

Program P describes a semaphore which as time progresses, alternates between the states *up* and *down*. When the semaphore goes *up* for the n-th time, it stays *up* for a period of $2n$ time-units, and when it goes *down*, it stays *down* for one time-unit only (see the following table).

up :	0,0		2,0	2,1		4,0	4,1	4,2	4,3		6,0	...	6,5		8,0	...
$down$:		1			3					5				7		
$time$:	0	1	2	3	4	5	6	7	8	9	10	...	15	16	17	...

We want to prove the following Property (A) of the semaphore:

(A) $\forall x, y.\, (x > y,\ up(x,y) \rightarrow up(ssx, 0))$

which states that if the semaphore is *up* then it will be *up* again in the future. We start from the statement: $f \leftarrow (\forall x, y.\, (x > y,\ up(x,y)) \rightarrow up(ssx,0))$. By applying our LT transformation (see Section 5.3.1) which is a variant of the Lloyd-Topor transformation, we get the clauses:

5. $f \leftarrow \neg g$
6. $g \leftarrow x > y,\ up(x,y),\ \neg up(ssx,0)$

This concludes Step 1 of our proof method.

Step 2 is realized by applying the following transformation rules: definition introduction, positive and negative unfolding, folding, tautology, clause deletion, and basic goal replacement. These rules are extensions of the familiar rules presented in [135, 144] and, under the conditions of Section 5.2 below, they preserve perfect models.

We start off by unfolding clause 6 w.r.t. $up(x,y)$. We get three clauses: (i) one of them can be deleted because of the atom $0 > 0$ in its body, (ii) a second one: $g \leftarrow down(x),\ \neg up(sssx,0)$ can also be deleted, because by unfolding $\neg up(sssx,0)$ using clause 3 we get: $g \leftarrow down(x),\ \neg down(ssx)$, and eventually, after some unfolding and basic goal replacement steps, we get: $g \leftarrow down(x),\ \neg down(x)$, and (iii) the third clause produces, after a basic goal replacement step based on the equivalence $(sx > sy,\ x > y) \leftrightarrow x > y$, the following clause:

7. $g \leftarrow x > y,\ up(sx,y),\ \neg up(sssx,0)$

By the definition and folding rules, from clause 7 we get:

8. $h \leftarrow x > y,\ up(sx,y),\ \neg up(sssx,0)$

7.f $g \leftarrow h$

By unfolding clause 8 w.r.t. $up(sx,y)$ and by performing the same steps which led us from clause 6 to clause 7, we get:

9. $h \leftarrow x > sy,\ up(sx,y),\ \neg up(sssx,0)$

From this clause, by applying the basic goal replacement rule based on the equivalence $plus(y, s0, sy) \leftrightarrow true$, and then by applying the definition and folding rules, we get:

10. $k(n) \leftarrow x > z,\ plus(y,n,z),\ up(sx,y),\ \neg up(sssx,0)$

9.f $h \leftarrow k(s0)$

Notice that clause 10 has been derived by: (i) comparing clause 8 and clause 9, and (ii) generalizing their mismatching terms y and sy to the term $s \ldots s0$ with $n\,(\geq 0)$ occurrences of s, that is, generalizing y and sy to the variable z such that $plus(y,n,z)$. This generalization avoids the introduction of an unbounded number of new definitions, and it makes the transformation process to halt. Indeed, by unfolding clause 10 w.r.t. $up(sx,y)$ and $\neg up(sssx,0)$ and by performing basic goal replacement steps involving properties of $>$ and the equivalence $plus(sy,n,z) \leftrightarrow plus(y,sn,z)$, we get:

11. $k(n) \leftarrow x > z,\ plus(y,sn,z),\ up(sx,y),\ \neg up(sssx,0)$

which we can be folded using clause 10 and we get:

11.f $k(n) \leftarrow k(sn)$.

Thus, we get the program

$$Q: \; f \leftarrow \neg g, \;\; g \leftarrow h, \;\; h \leftarrow k(s0), \;\; k(n) \leftarrow k(sn)$$

Now, since k is a useless predicate being defined by the recursive clause $k(n) \leftarrow k(sn)$ only, without base case, we may apply the clause deletion rule (see rule R6 in Section 5.2) and we get the program

$$R: \; f \leftarrow \neg g, \;\; g \leftarrow h, \;\; h \leftarrow k(s0)$$

By unfolding clause $h \leftarrow k(s0)$ w.r.t. $k(s0)$ we delete that clause, because the definition of the predicate k in program R is empty. Then, by unfolding $g \leftarrow h$ w.r.t. h we also delete that clause, because the definition of h has been deleted in the previous unfolding step. Thus, the derived program is the clause $f \leftarrow \neg g$ only. Finally, by unfolding $f \leftarrow \neg g$ w.r.t. $\neg g$ we get $f \leftarrow$, because the definition of g in the derived program is empty. This completes our transformational proof of Property (A) we wanted to show. □

Notice that our proof of Property (A) is based on the crucial step of deleting the clause $k(n) \leftarrow k(sn)$, and this step is correct because in the perfect model of program R there are no atoms with predicate k.

The tight correspondence between program transformation and theorem proving can be exploited to turn program transformation strategies into proof strategies. In particular, at Step 2 of our proof method, in order to direct the transformation rules, we may use the so called UFS strategy which is an enhancement of a strategy introduced in the case of definite logic programs for deriving programs without existential variables [125]. When our UFS strategy terminates, it derives from $P \wedge F(f, \varphi)$ *either* (i) a new program of the form $Q \wedge (f \leftarrow)$, in which case $M(P) \models \varphi$, *or* (ii) a program where no clause has head predicate f, in which case $M(P) \not\models \varphi$. We will show that the UFS strategy indeed terminates for some classes of properties and some classes of logic programs with locally stratified negation, and thus, it can be used for deciding whether or not $M(P) \models \varphi$ holds when φ and P belong to those classes.

The plan of the rest of this section is as follows. In Section 5.3.1 we present Step 1 of our proof method by introducing a variant of the Lloyd-Topor transformation technique and we show that it preserves the perfect model semantics. (Notice that the original Lloyd and Topor's correctness result w.r.t. the completion semantics [103] is often not suitable for proving properties of perfect models.) In Section 5.3.2 we present our UFS strategy for locally stratified programs, and in Section 5.3.3 we define some classes of properties and programs for which a deterministic version of the UFS strategy terminates. We show that these classes are non trivial by proving that both the equivalence problem of tree automata [62] and the decision problem for the clausal fragment of the weak monadic second order theory with n successors [128] can be reduced to the problem of deciding whether or not $M(P) \models \varphi$ holds for suitable programs and formulas in those classes. Finally, in Section 5.4 we compare our transformational proof method to other theorem proving methods, such as, (i) methods based on the *Clark completion* [34, 103], (ii) methods based on *resolution* augmented by the *negation as (finite or infinite) failure rule* [4, 127], (iii) methods based on *induction*, and (iv) the *proof by consistency* method [86, 89, 109].

5.3.1 Translating First Order Formulas into Logic Programs

In this section we present a method that given a locally stratified program P and a closed first order formula φ in $\mathcal{L}$, introduces a new predicate f and constructs a locally stratified program $P \wedge F(f, \varphi)$ such that $M(P) \models \varphi$ iff $M(P \wedge F(f, \varphi)) \models f$. This method is based upon a variant of a transformation proposed by Lloyd and Topor [103].

In order to construct the locally stratified program $F(f,\varphi)$ for a given closed first order formula φ, we need to consider a class of formulas, called *statements* [103], of the form $A \leftarrow \beta$ where A is an atom and β, called the *body* of the statement, is a (possibly open) first order logic formula. We write $C[\gamma]$ to denote a first order formula where the subformula γ occurs as a conjunct 'at top level', that is, $C[\gamma] = \rho_1 \wedge \ldots \wedge \rho_r \wedge \gamma \wedge \sigma_1 \wedge \ldots \wedge \sigma_s$ for some first order formulas $\rho_1, \ldots, \rho_r, \sigma_1, \ldots, \sigma_s$, and some $r \geq 0$ and $s \geq 0$. When we say that the formula $C[\gamma]$ is transformed into the formula $C[\delta]$, we mean that $C[\delta]$ is obtained from $C[\gamma]$ by replacing the top level conjunct γ by the new top level conjunct δ.

Given any conjunction of statements the following LT transformation, similar to the one proposed by Lloyd and Topor and reported in [103], terminates and it produces a conjunction of clauses which is a locally stratified program.

The LT transformation from conjunctions of statements to locally stratified programs.
Given a conjunction of statements, perform the following transformations.
(A) Eliminate from the body of every statement all occurrences of logical constants, connectives, and quantifiers other than $true, \neg, \wedge$, and $\exists$. For every statement st, rename the bound variables of st so that none of them occurs in $freevars(st)$ and all of them are distinct.

(B) Apply *as long as possible* the following rules:

(B.1)	$A \leftarrow C[\neg true]$	is deleted	
(B.2)	$A \leftarrow C[\neg\neg\varphi]$	is transformed into	$A \leftarrow C[\varphi]$
(B.3)	$A \leftarrow C[\neg(\varphi \wedge \psi)]$	is transformed into	$A \leftarrow C[\neg newp(y_1, \ldots, y_k)] \;\wedge$ $newp(y_1, \ldots, y_k) \leftarrow \varphi \wedge \psi$

where $newp$ is a new predicate symbol and $\{y_1, \ldots, y_k\} = freevars(\varphi \wedge \psi)$. We assume that $\varphi \neq true$ and $\psi \neq true$.

(B.4)	$A \leftarrow C[\neg\exists x.\, \varphi]$	is transformed into	$A \leftarrow C[\neg newp(y_1, \ldots, y_k)] \;\wedge$ $newp(y_1, \ldots, y_k) \leftarrow \varphi$

where $newp$ is a new predicate symbol and $\{y_1, \ldots, y_k\} = freevars(\exists x.\, \varphi)$.

(B.5)	$A \leftarrow C[\exists x.\, \varphi]$	is transformed into	$A \leftarrow C[\varphi]$ $\square$

Given a locally stratified program P and a closed first order formula φ, we denote by $F(f,\varphi)$ the conjunction of the clauses derived by applying the LT transformation to the statement $f \leftarrow \varphi$, where f is a new predicate symbol occurring neither in φ nor in P. We assume that the new predicates introduced during the construction of $F(f,\varphi)$ do not occur in P.

The reader may verify that in the Semaphore Example in Section 5.3, clauses 5 and 6 have been derived by applying the LT transformation starting from the following statement:

$$f \;\leftarrow\; \forall x, y.\, (x > y,\ up(x, y) \rightarrow up(ssx, 0))$$

The following result states that the LT transformation is correct w.r.t. the perfect model semantics, and thus, Step 1 of our proof method is sound. Recall that, instead, the Lloyd-Topor transformation as described in [103], is known to be correct w.r.t. the completion semantics only.

Theorem 13 (Correctness of LT Transformation w.r.t. Perfect Models) *Let P be a locally stratified program, φ be a closed first order formula, and f be a predicate symbol occurring neither in φ nor in P. If $F(f,\varphi)$ is obtained from $f \leftarrow \varphi$ by the* LT *transformation, then (i) $P \wedge F(f,\varphi)$ is a locally stratified program, and (ii) $M(P) \models \varphi$ iff $M(P \wedge F(f,\varphi)) \models f$.*

Proof. (i) Since f is a new predicate symbol and rules B.1-B.5 do not introduce cycles in the dependency graph, we have that the dependency graph of $F(f,\varphi)$ has no cycles. Since no predicate in P depends on the predicates in $F(f,\varphi)$ and P is locally stratified, we have that $(P \wedge F(f,\varphi))$ is locally stratified.

(ii) We have that $M(P \wedge F(f,\varphi)) \models comp(P \wedge F(f,\varphi))$ where $comp(P)$ denotes the Clark completion of program P [103]. By construction we have that $comp(P \wedge F(f,\varphi)) \models f \leftrightarrow \varphi$ and thus, $M(P \wedge F(f,\varphi)) \models f \leftrightarrow \varphi$. To conclude the proof it remains to show that $M(P) \models \varphi$ holds iff $M(P \wedge F(f,\varphi)) \models \varphi$. This follows from the fact that no predicate defined in P and no predicate occurring in φ depends on a predicate defined in $F(f,\varphi)$. □

5.3.2 A Strategy for Making Unfold/Fold Proofs

In order to verify whether or not $M(P) \models \varphi$ holds, Step 2 of our unfold/fold proof method requires the construction of a transformation sequence from program $P \wedge F(f,\varphi)$ to a new program, say T, such that *either* (i) $Def(f,T)$ is $f \leftarrow$, and in this case we infer that $M(P) \models \varphi$, *or* (ii) T is a program where $Def(f,T)$ is the empty conjunction, and in this case we infer that $M(P) \not\models \varphi$. To construct this transformation sequence we need a strategy for guiding the application of the transformation rules, and indeed, in this section we present such a strategy, called UFS (short for Unfold/Fold proof Strategy). The UFS strategy is an extension of the strategy introduced in the case of definite logic programs for eliminating existential variables [125]. The basic idea is that by eliminating existential variables, from $P \wedge F(f,\varphi)$ we may derive a program, say S, such that $Def^*(f,S)$ is a *propositional* program. Then, we can transform S by using the clause deletion, unfolding, tautology, and basic goal replacement rules, into a program T satisfying either (i) or (ii) above.

Obviously, since in general $M(P) \models \varphi$ is an undecidable property, our UFS strategy may fail to produce a propositional program S starting from an arbitrary program $P \wedge F(f,\varphi)$. However, in the next section we will show that in some cases our strategy is guaranteed to terminate and it produces a propositional program S, and hence, the desired program T. Thus, our UFS strategy is a decision procedure for the problem of verifying whether or not $M(P) \models \varphi$ holds, when P and φ belong to suitable classes of programs and formulas, respectively. Moreover, even when our strategy derives a program which is *not* propositional, it may still be the case that we are able to infer whether or not $M(P) \models \varphi$ holds, and indeed, this happened in the Semaphore Example given in Section 5.3.

For specifying the UFS strategy, we need the following definition (see also [103]).

Definition 6 *A* level mapping *of a program P is a mapping from the set of predicate symbols occurring in P to the set of natural numbers. The* level *of predicate p is the value of p under this mapping.*

By the definition of the LT transformation there exists a level mapping of $P \wedge F(f,\varphi)$ such that: (i) the level of every predicate defined in P is 0, (ii) the level of every predicate defined in $F(f,\varphi)$ is greater than 0, (iii) for each clause $p(\ldots) \leftarrow B$ in $F(f,\varphi)$ the level of every predicate symbol in B is strictly smaller than the level of p, and (iv) the predicate f has the highest level, say K. For instance, in the Semaphore Example we may choose the level mapping m defined by the following equations: $m(down) = m(up) = m(>) = 0$, $m(g) = 1$, and $m(f) = 2$ (thus, $K = 2$).

The UFS strategy, as we will present it now, requires three substrategies: (1) UNFOLD, (2) TAUTOLOGY & REPLACE, and (3) DEFINE & FOLD. Below we will give an algorithm for each of them. These algorithms can be used for the particular classes of programs and formulas for which the UFS strategy terminates.

The Unfold/Fold Proof Strategy UFS.

Input: (i) $P \wedge F(f,\varphi)$, where P is a locally stratified program and φ is a closed first order formula, and (ii) a set *Laws* of equivalences of the form: $M(P) \models \forall x_1 \ldots x_u.\, (\exists y_1 \ldots y_v.\, Q \leftrightarrow \exists z_1 \ldots z_w.\, R)$, where Q and R are basic goals.
Output: A program T such that: (i) $M(T) \models f$ iff $M(P \wedge F(f,\varphi)) \models f$, and (ii) $Def(f,T)$ is either $f \leftarrow$ or the empty conjunction.

Let $P \wedge F(f,\varphi)$ be $P \wedge P^1 \wedge \ldots \wedge P^K$, where for $i = 1,\ldots,K$, program P^i is the conjunction of the clauses whose head predicate has level i.

$T := P$;
FOR $i = 1,\ldots,K$ DO
Let *Pos* be the conjunction of the clauses of P^i whose body has at least one non-basic positive literal, and let *Neg* be the conjunction of the clauses of P^i which are not in *Pos*.
Let *Defs* be *Pos* and *Out* be the empty conjunction.
WHILE *Pos* is not the empty conjunction DO (†)
(1) UNFOLD(T, *Pos*, U): From program $T \wedge Pos$ we derive $T \wedge U$ by a finite sequence of applications of the positive or negative unfolding rules to the clauses in *Pos*. We require that:
[*Progression*] Each clause in *Pos* is unfolded at least once using rule R2 w.r.t. a non-basic atom.
(2) TAUTOLOGY & REPLACE(T, *Laws*, U, R): From program $T \wedge U$ we derive $T \wedge R$ by a finite sequence of applications of the tautology and basic goal replacement rules to the clauses in U, using the equivalences in the set *Laws*.
(3) DEFINE & FOLD(T, R, *Defs*, *OutClauses*, *NewDefs*): From program $T \wedge R$ we derive $T \wedge OutClauses \wedge NewDefs$ by: (i) a finite sequence of applications of the definition rule by which we introduce the (possibly empty) conjunction *NewDefs* of clauses, followed by (ii) a finite sequence of applications of the folding rule to the clauses in R, using clauses occurring in $Defs \wedge NewDefs$. We assume that the definition and folding steps are such that the following conditions are satisfied:
(3.1) [*Positive definitions*] The body of each clause in *NewDefs* has at least one non-basic positive literal.
(3.2) [*No existential variables*] Each clause in *OutClauses* which has been derived by folding has no existential variables.
(3.3) [*Full folding*] Each predicate in the body of each clause in *OutClauses* which has been derived by folding, occurs in $Defs \wedge NewDefs$.
$Out := Out \wedge OutClauses$; $Pos := NewDefs$; $Defs := Defs \wedge NewDefs$
OD;
Delete from *Out* the definitions of useless predicates, thereby deriving Out'; $T := T \wedge Out' \wedge Neg$;
Initialize D to the conjunction of all definitions of nullary predicates in T;
Initialize Q to the conjunction of all definitions of non-nullary predicates in T;
WHILE D is a not a conjunction of unit clauses DO (††)
UNFOLD(Q, D, U): From program $Q \wedge D$ we derive $Q \wedge U$ by a (possibly empty) sequence of applications of the positive or negative unfolding rules to the clauses in D. (Here the Progression requirement need not be satisfied.)
TAUTOLOGY & REPLACE(Q, *Laws*, U, R); $D := R$
OD;

Unfold the clauses of Q w.r.t. their propositional literals, thereby deriving Q'; (†††)
$T := Q' \wedge D$
END FOR □

Our UFS strategy proceeds by iterating over levels (from level 1 to level K) a sequence of transformations on the program T, which is initialized to P. For $i = 1, \ldots, K$, program P^i which is the conjunction of the clauses defining the predicates with level i, is processed by the UFS strategy, and the WHILE loop (†) generates from program $T \wedge P^i$ (which is $T \wedge Neg \wedge Pos$) the new program T. The objective of the WHILE loop (†) is: (i) to ensure that positive unfolding steps w.r.t. non-basic atoms are performed before folding, and (ii) to avoid existential variables via definition and folding steps. Then useless predicates are deleted. Finally, the WHILE loop (††) performs unfolding, tautology, and basic goal replacement steps on the definitions of nullary predicates so to reduce each of them, if possible, either to the empty definition (in which case the corresponding predicate is false in the perfect model) or to a unit clause (in which case the corresponding predicate is true in the perfect model). The truth values of the propositional predicates are then propagated by the unfolding steps (†††). When the last program level K has been processed, we get for the predicate f either the empty definition or the clause $f \leftarrow$. Thus, we may establish whether or not $M(P \wedge F(f, \varphi)) \models f$ holds.

The reader may check that our Semaphore Example 10 has indeed been worked out using the UFS proof strategy.

The soundness of our proof strategy follows from the fact that the transformation rules are used in such a way that the hypothesis of the Correctness Theorem 12 of Section 5.2 holds.

Obviously, the UFS strategy may not terminate, because in general $M(P \wedge F(f, \varphi)) \models f$ is undecidable, and indeed: (i) during the execution of the WHILE loop (†) we may introduce by applying the definition rule, an infinite number of new clauses, and thus, Pos never becomes the empty conjunction, and (ii) the WHILE loop (††) may not terminate for nullary predicates which depend on non-nullary predicates.

5.3.3 Decision Procedures Using the Unfold/Fold Proof Strategy

In this section we present two classes of formulas, called *tree-typed formulas* and *tree-typed clausal formulas*, and two classes of programs, called *MR programs* and *DL programs*, for which a deterministic version of the UFS strategy, called dUFS strategy, terminates. Thus, the dUFS strategy is a decision procedure for establishing whether or not $M(P) \models \varphi$ holds, when φ and P belong to the given classes.

In this section we assume that all predicates are non-basic. The tree-typed formulas are function-free formulas whose variables range over sets of trees denoted by *tree programs* which we now define.

Definition 7 (Tree Programs) *A* tree program *is a conjunction of tree clauses. A* tree clause *is a clause of the form:* $r_0(t(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n)$, *with* $n \geq 0$, *where* t *is a function symbol and* $x_1, \ldots, x_n$ *are distinct variables. A* tree predicate *is a predicate defined by a tree program. A* tree atom *is an atom with a tree predicate.*

Definition 8 (Tree-typed Formulas) *A* tree-typed formula *over a program* P *is a first order formula* φ, *defined as follows:*

$$\varphi \ ::= \ p(x_1, \ldots, x_n) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \forall x. \, (r(x) \rightarrow \varphi) \mid \exists x. \, (r(x) \wedge \varphi)$$

where: (i) $x_1, \ldots, x_n$, *with* $n \geq 0$, *are distinct variables,* (ii) *all predicates occurring in* φ *are defined in* P, *and* (iii) $Def^*(r, P)$ *is a tree program.*

Example 11 (Even-Odd Paths) Let us consider the following *Even-Odd* program:

1. $bin(leaf)$
2. $bin(t(x, y)) \leftarrow bin(x), bin(y)$
3. $even(leaf)$
4. $even(t(x, y)) \leftarrow \neg even(x)$
5. $even(t(x, y)) \leftarrow \neg even(y)$
6. $odd(t(x, y)) \leftarrow \neg odd(x), \neg odd(y)$

Clauses 1 and 2 are tree clauses. The formula $\varphi\colon \forall x. (bin(x) \rightarrow even(x) \vee odd(x))$ is a tree-typed formula over *Even-Odd*. Informally, φ means that in every binary tree there exists a path of even length or all paths have odd length. Thus, we expect that $M(P) \models \varphi$ holds, as we will formally prove below. □

The Tree-typed LT Transformation.
In order to construct the locally stratified program $F(f, \varphi)$ for a given tree-typed formula φ, we apply the so called *tree-typed LT transformation*, which is a variant of the LT transformation (see Section 5.3.1) obtained by replacing rules (B.3) and (B.4) by the following ones:

(B.3)* $A \leftarrow C[\neg(\varphi \wedge \psi)]$ is transformed into $A \leftarrow C[\neg newp(y_1, \ldots, y_k)] \wedge$
$newp(y_1, \ldots, y_k) \leftarrow \rho \wedge \varphi \wedge \psi$

where *newp* is a new predicate symbol, $\{y_1, \ldots, y_k\} = freevars(\varphi \wedge \psi)$, and ρ is the conjunction of all tree atoms $r(y)$ occurring as conjuncts at top level in $C[\neg(\varphi \wedge \psi)]$ and such that $y \in \{y_1, \ldots, y_k\}$.

(B.4)* $A \leftarrow C[\neg \exists x. (r(x) \wedge \varphi)]$ is transformed into $A \leftarrow C[\neg newp(y_1, \ldots, y_k)] \wedge$
$newp(y_1, \ldots, y_k) \leftarrow \rho \wedge r(x) \wedge \varphi$

where *newp* is a new predicate symbol, $\{y_1, \ldots, y_k\} = freevars(\exists x. (r(x) \wedge \varphi))$, and ρ is the conjunction of all tree atoms $r(y)$ occurring as conjuncts at top level in $C[\neg \exists x. (r(x) \wedge \varphi)]$ and such that $y \in \{y_1, \ldots, y_k\}$. □

Example 12 (Even-Odd Paths, Continued 1) For the tree-typed formula $\varphi\colon \forall x. (bin(x) \rightarrow even(x) \vee odd(x))$, the corresponding $F(f, \varphi)$ obtained by tree-typed LT transformation is the conjunction of the following two clauses:

7. $f \leftarrow \neg g$
8. $g \leftarrow bin(x), \neg even(x), \neg odd(x)$ □

The tree-typed LT transformation is correct w.r.t the perfect model semantics as stated by the following theorem.

Theorem 14 (Correctness of Tree-Typed LT Transformation w.r.t. Perfect Models) *For every locally stratified logic program* P *and closed tree-typed formula* φ *over* P, *if* $F(f, \varphi)$ *is obtained from* φ *by the tree-typed* LT *transformation, then*
(i) $P \wedge F(f, \varphi)$ *is a locally stratified program and* $M(P) \models \varphi$ *iff* $M(P \wedge F(f, \varphi)) \models f$, *and*
(ii) *for each clause* C *in* $F(f, \varphi)$ *we have that:* (ii.1) C *is of the form* $newp(x_1, \ldots, x_m) \leftarrow r_1(x_1), \ldots, r_n(x_n), G$, *where* $r_1(x_1), \ldots, r_n(x_n)$ *are tree atoms,* (ii.2) $vars(C) = \{x_1, \ldots, x_n\}$, (ii.3) G *is a function-free goal, and* (ii.4) $C = Def(newp, P \wedge F(f, \varphi))$.

Definition 9 (Monadic Regular Programs) *A* monadic regular program (*or* MR program, *for short*) *is a conjunction of monadic regular clauses. A* monadic regular clause (*or* MR clause, *for short*) *is a locally stratified clause of the form:*

$$p_0(t(x_1,\ldots,x_n)) \leftarrow p_1(y_1),\ldots,p_k(y_k),\neg p_{k+1}(y_{k+1}),\ldots,\neg p_m(y_m)$$

with $n,m\geq 0$*, where* t *is a function symbol,* $x_1,\ldots,x_n$ *are distinct variables, and* $y_1,\ldots,y_m$ *are (not necessarily distinct) variables occurring in* $\{x_1,\ldots,x_n\}$*.*

Tree programs are MR programs.

We now describe the deterministic version of the UFS strategy, which we call dUFS.

The Unfold/Fold Proof Strategy dUFS.

The dUFS strategy is obtained from the UFS strategy defined in Section 5.3.2, by replacing the substrategies: (1) UNFOLD, (2) TAUTOLOGY & REPLACE, and (3) DEFINE & FOLD by the following ones, respectively:

(1d) BREADTH-FIRST UNFOLD(T, Pos, U): From program $T \wedge Pos$ we derive $T \wedge U$ by: (i) one application of positive unfolding w.r.t. each positive literal occurring in the body of a clause in Pos, followed by (ii) one application of negative unfolding w.r.t. each negative literal occurring in the body of a clause in Pos.

(2d) TAUTOLOGY(T, U, R): From program $T \wedge U$ we derive $T \wedge R$ by a sequence of applications of the tautology rule, constructed by rewriting as long as possible an instance of the left hand side of an equivalence of Rule R5 by the corresponding instance of the right hand side.

(3d) BLOCK-DEFINE & FOLD(T, R, *Defs, OutClauses*, *NewDefs*): From program $T \wedge R$ we derive program $T \wedge$ *OutClauses* $\wedge$ *NewDefs* as follows:
for each non-unit clause C in R
(i) we partition $bd(C)$ into subconjunctions, called *blocks*, such that $bd(C) = B_1 \wedge \ldots \wedge B_m$, and two literals occur in the same subconjunction B_i, for some i, $1 \leq i \leq m$, iff they share a variable,
(ii) for $i = 1,\ldots,m$, we apply the definition rule and we add to *NewDefs* a clause of the form $Newp \leftarrow B_i$, where $vars(Newp) = vars(B_i) \cap vars(hd(C))$, unless a variant clause already occurs in *Defs*, modulo the head predicate symbol, and
(iii) we fold C w.r.t. B_1, and we fold the resulting clause w.r.t. B_2, and so on, until we fold w.r.t. B_m. □

We have that:
- the BREADTH-FIRST UNFOLD substrategy fulfills the Progression requirement,
- in the TAUTOLOGY substrategy we have omitted the *Laws* argument because basic goal replacements are not performed, and
- the BLOCK-DEFINE & FOLD substrategy fulfills the conditions (3.1), (3.2), and (3.3) of the DEFINE & FOLD substrategy of Section 5.3.2. Indeed, Condition (3.1) is fulfilled, because each variable occurring in the body of a clause generated by the strategy dUFS also occurs in a tree atom, and therefore, each block has at least one non-basic positive literal. Moreover, by Points (ii) and (iii) in (3d) above, also Conditions (3.2) and (3.3) are fulfilled.

Example 13 (Even-Odd Paths, Continued 2) The proof that the formula $\forall x.\,(bin(x) \rightarrow even(x) \vee odd(x))$ holds in the perfect model of the *Even-Odd* program is as follows. We start from program P made out of clauses from 1 to 6, program P^1 made out of clause 8, and program P^2 made out of clause 7.

We start from level 1. Initially Pos = clause 8. By breadth-first unfolding, from clause 8 we get the following clauses:

9. $g \leftarrow bin(x), bin(y), even(x), even(y), odd(x)$
10. $g \leftarrow bin(x), bin(y), even(x), even(y), odd(y)$

By the tautology rule R5.3 we delete clause 10 (it is subsumed by clause 9). Then we apply the BLOCK-DEFINE & FOLD substrategy by introducing the following two definitions:

11. $h_{eo} \leftarrow bin(x), even(x), odd(x)$
12. $h_e \leftarrow bin(y), even(y)$

and by folding clauses 9 we get:

9.f $\quad g \leftarrow h_{eo}, h_e$

Now $Pos = \text{clause}\,11 \wedge \text{clause}\,12$ and we have to execute once more the body of the WHILE loop (†) of the dUFS strategy. By unfolding clauses 11 and 12 we get:

11.1 $h_{eo} \leftarrow bin(x), bin(y), \neg even(x), \neg odd(x), \neg odd(y)$
11.2 $h_{eo} \leftarrow bin(x), bin(y), \neg even(y), \neg odd(x), \neg odd(y)$
12.1 $h_e \leftarrow$
12.2 $h_e \leftarrow bin(x), bin(y), \neg even(x)$
12.3 $h_e \leftarrow bin(x), bin(y), \neg even(y)$

By the tautology rule R5.3 we delete clause 11.2 (it is subsumed by clause 11.1) and we also delete clauses 12.2 and 12.3 (they are subsumed by clause 12.1). Then we apply the BLOCK-DEFINE & FOLD substrategy by introducing the following definition:

13. $h_o \leftarrow bin(x), \neg odd(x)$

and by folding clause 11.1, we get:

11.1f $h_{eo} \leftarrow g, h_o$

Now $Pos = \text{clause}\,13$. By unfolding clause 13 we get:

13.1 $h_o \leftarrow$
13.2 $h_o \leftarrow bin(x), bin(y), \neg odd(x), \neg odd(y)$

By the tautology rule R5.3 we delete clause 13.2 (it is subsumed by clause 13.1). No application of the BLOCK-DEFINE & FOLD substrategy is required. Now *Pos* is empty, and *Out* is the conjunction of the following clauses:

9.f $\quad g \leftarrow h_{eo}, h_e$
11.1f $h_{eo} \leftarrow g, h_o$
12.1 $\quad h_e \leftarrow$
13.1 $\quad h_o \leftarrow$

We then delete clauses 9.f and 11.1f because they are the definitions of the predicates g and h_{eo} which are useless in *Out*. The WHILE loop (††) does not change D. Since $Q = Q'$, we have that T is made out of the clauses of P together with the clauses 12.1 and 13.1.
We can now start processing level 2, that is, program P^2. Since *Pos* is the empty conjunction, the body of the WHILE loop (†) is never executed. *Neg* is clause 7, and before the execution of the WHILE loop (††), program T is made out of the clauses in P together with the following clauses:

7. $\quad f \leftarrow \neg g$

12.1 $\quad h_e \leftarrow$
13.1 $\quad h_o \leftarrow$

After the WHILE loop (††) and the statement (†††) we get the new program T made out of the clauses in P and the clause:

7.1 $\quad f \leftarrow$

together with clauses 12.1 and 13.1. Now, since $M(T) \models f$, we have that $M(P) \models \varphi$. □

Obviously, the BREADTH-FIRST UNFOLD, TAUTOLOGY, and BLOCK-DEFINE & FOLD substrategies terminate. Also the dUFS proof strategy terminates for tree-typed formulas and MR programs, as stated by the following theorem.

Theorem 15 (Termination of the dUFS Proof Strategy for MR Programs) *Let P be an* MR *program and φ be a tree-typed formula over P. Then the* dUFS *proof strategy with input program $P \wedge F(f,\varphi)$ terminates with output program T. Moreover, $Def(f,T)$ is f iff $M(P) \models \varphi$ and $Def(f,T)$ is the empty conjunction iff $M(P) \not\models \varphi$.*

Although very restricted, the classes of MR programs and tree-typed formulas allow us to express some interesting properties, such as the equivalence of finite tree automata (see, for instance, [62]), as shown in the following example.

Example 14 (Equivalence of Tree Automata) The tree language recognized by a (nondeterministic top-down) finite tree automaton $T = (Q, \Sigma, \delta, q0, F)$ corresponds to a subset of the perfect model of a tree program P_T (which is also its least Herbrand model) defined as follows:
- for each state $q \in Q$ we define a unary predicate p_q,
- for each symbol $a \in \Sigma$ of arity k we define a k-ary function symbol f_a,
- for each tuple $(q, a, q1, \ldots, qk)$ in the transition relation δ, where a is a k-ary symbol with $k \geq 1$, P_T has the clause:

$p_q(f_a(x_1, \ldots, x_k)) \leftarrow p_{q1}(x_1), \ldots, p_{qk}(x_k)$

- for each state $q \in F \subseteq Q$ and 0-ary symbol b such that q is a final state for b, P_T has the unit clause:

$p_q(f_b) \leftarrow$

Then the tree language recognized by the tree automaton T with initial state $q0$, corresponds to the subset $\{t \mid M(P_T) \models p_{q0}(t)\}$. Given another tree automaton $T1$ with initial state $r0$ and represented by program P_U, we have that T and U recognize the same language iff

$$M(P_T \wedge P_U) \models \forall x.\, (p_{q0}(x) \rightarrow p_{r0}(x)) \;\wedge\; \forall x.\, (p_{r0}(x) \rightarrow p_{q0}(x))$$

Thus the equivalence of T and U can be reduced to the verification of a tree-typed formula over the tree program $P_T \wedge P_U$ (recall that each tree program is also an MR program). □

We now introduce a second class of formulas, called *tree-typed clausal formulas*, and a second class of programs, called *deterministic linear programs* (or DL *programs*, for short) for which the unfold/fold proof strategy dUFS terminates. Thus, given a tree-typed clausal formula φ, and a DL program P, we can decide whether or not $M(P) \models \varphi$ holds by using the proof strategy dUFS.

Definition 10 (Tree-typed Clausal Formulas) *Let P be any program and R be a tree program. A* tree-typed clausal formula *over a program $P \wedge R$ is a closed first order formula φ generated by the following grammar:*

$$\varphi \;::=\; \forall x.\, (r(x) \rightarrow \varphi) \mid \delta$$
$$\delta \;::=\; p(x_1, \ldots, x_n) \mid \neg p(x_1, \ldots, x_n) \mid \delta \vee \delta$$

where: (i) *$x_1, \ldots, x_n$, with $n \geq 0$, are distinct variables,* (ii) *all predicates occurring in φ are defined in $P \wedge R$,* (iii) *for each predicate p, $Def^*(p, P \wedge R)$ is a subconjunction of P,* (iv) *for each predicate r, $Def^*(r, P \wedge R)$ is a subconjunction of R, and* (v) *no predicate is defined in both P and R.*

Definition 11 (Deterministic Linear Programs) *A* linear clause *is a locally stratified clause of one of these three forms:*

$$p(t_1(x_1,\ldots,x_m),\ldots,t_n(x_u,\ldots,x_v))$$
$$p(t_1(x_1,\ldots,x_m),\ldots,t_n(x_u,\ldots,x_v)) \leftarrow q(y_1,\ldots,y_k)$$
$$p(t_1(x_1,\ldots,x_m),\ldots,t_n(x_u,\ldots,x_v)) \leftarrow \neg q(y_1,\ldots,y_k)$$

with $n \geq 0$*, where:* (i) $t_1,\ldots,t_n$ *are function symbols,* (ii) $x_1,\ldots,x_m,\ldots,x_u,\ldots,x_v$ *are distinct variables, and* (iii) $y_1,\ldots,y_k$ *are distinct variables occurring in* $\{x_1,\ldots,x_m,\ldots,x_u,\ldots,x_v\}$*.*
A deterministic linear program (*or* DL program*, for short*) *is a conjunction of linear clauses such that no two clause heads are unifiable.*

Notice that there are MR programs which are not DL programs and vice versa. We have the following result.

Theorem 16 (Termination of the dUFS Proof Strategy for DL Programs) *Let* P *be a* DL *program,* R *be a tree program, and* φ *be a tree-typed clausal formula over* $P \wedge R$*. Then the* dUFS *strategy with input program* $P \wedge F(f,\varphi)$ *terminates with output program* T*. Moreover,* $Def(f,T)$ *is* f *iff* $M(P) \models \varphi$ *and* $Def(f,T)$ *is the empty conjunction iff* $M(P) \not\models \varphi$*.*

Example 15 (Clausal wMSnS) The clausal fragment of the weak monadic second order theory of two successor functions (cwMS2S) [128] is the set of closed formulas φ such that $\mathcal{W} \models \varphi$, where:
(i) φ is a closed formula generated by the following grammar:

$$\varphi \ ::= \ \forall w.\,(word(w) \rightarrow \varphi) \mid \forall x.\,(set(x) \rightarrow \varphi) \mid \delta$$
$$\delta \ ::= \ member(w,x) \mid \neg member(w,x) \mid \delta \vee \delta$$

(ii) $\mathcal{W}$ is the structure $\mathcal{P}_{fin}(\{0,1\}^*)$ of finite sets of words over $\{0,1\}$, where: (ii.1) the successor functions s_0 and s_1 are interpreted as $s_0(w) = w0$ and $s_1(w) = w1$, respectively, (ii.2) the predicates *word* and *set* hold of all words and finite sets of words, respectively, and (ii.3) $member(w,x)$ is interpreted as membership of word w to set x.

In order to write a logic program which computes the *word*, *set*, and *member* predicates, we represent a finite set of words as a finite tree, where (1) t (of arity 3) and *leaf* (of arity 1) are the tree constructors, (2) both the internal nodes and the leaf nodes are labeled by either the constant *accept* or the constant *refuse,* and they are called *accept nodes* or *refuse nodes*, respectively, and (3) left arcs are labeled by 0 and right arcs are labeled by 1. The empty word is represented by the constant *emptyword.* The *accept paths* of a tree x are the sequences of labels from an *accept node* to the root. A word w is member of a set x iff w is an accept path of x. For instance, the set $\{0, 01\}$ is represented by the tree $t(leaf(accept), refuse, t(leaf(accept), refuse, leaf(refuse)))$. We may define the *word*, *set*, and *member* predicates by means of the following *Member* program, which is $P \wedge R$, where:

P:
$member(emptyword, leaf(n)) \leftarrow acceptlabel(n)$
$member(emptyword, t(x,n,y)) \leftarrow acceptlabel(n)$
$member(s_0(w), t(x,n,y)) \leftarrow member(w,x)$
$member(s_1(w), t(x,n,y)) \leftarrow member(w,y)$
$acceptlabel(accept) \leftarrow$

R:
$word(emptyword) \leftarrow$
$word(s_0(w)) \leftarrow word(w)$
$word(s_1(w)) \leftarrow word(w)$
$set(leaf(n)) \leftarrow label(n)$
$set(t(x,n,y)) \leftarrow set(x), label(n), set(y)$
$label(accept) \leftarrow$
$label(refuse) \leftarrow$

We have that the structure $\mathcal{W}$ corresponds to the perfect model (equal to the least Herbrand model) of *Member*, in the sense that: $\mathcal{W} \models \varphi$ iff $M(Member) \models \varphi$. Since (1) P is a DL program, (2) R is a tree program, and (3) every formula φ which is generated by the grammar of Point (i) above is a tree-typed clausal formula over $P \wedge R$, by Theorem 16 we have that by using our dUFS proof strategy we can test whether or not $M(Member) \models \varphi$ holds. Thus, dUFS is a decision procedure for cwMS2S. The extension from cwMS2S to cwMSnS (with n successors, instead of 2) can be obtained by a straightforward modification of the *Member* program. □

There are various possibilities of enhancing the classes of programs for which our dUFS proof strategy terminates. In particular, we may add constant arguments to predicates, we may add nullary predicates, and we may also combine the MR and LD programs in several ways. We do not discuss these enhancements here.

5.4 Bibliographic Remarks

The idea of using unfold/fold transformations for proving program properties goes back to [96], where it was advocated as a method for proving the equivalence of functional terms. Some limitations and completeness results for this proof method are presented in [16, 42]. Our completeness result (Theorem 15) shows that the unfold, fold, clause deletion, and tautology rules are sufficient to prove the equivalence between tree automata, while Courcelle in [42] proves a similar result by making use of the (restricted) unfold, (restricted) fold, cycle elimination, and redefinition rule. We leave for future research further investigation on the power of the various sets of program transformation rules.

In this Section 5 we have extended the techniques proposed in [119] for showing equivalences of definite programs w.r.t. least Herbrand models (see also Section 4.1). The main extensions presented here are the following: (i) we consider logic programs with locally stratified negation and perfect model semantics, (ii) we prove first order formulas, and (iii) we present an automated strategy for performing proofs. A different extension of [119] has been recently presented in [131], where the authors prove equivalences of definite programs w.r.t. least Herbrand models by using a more powerful folding rule.

Our transformational method for proving properties of the perfect model of a locally stratified logic program is related to other methods for theorem proving. Among these methods we would like to mention the following ones: (i) the method based on the *Clark completion* [34, 103], (ii) the methods based on *resolution* augmented by the *negation as* (*finite or infinite*) *failure rule* [4, 127], (iii) the method based on *partial evaluation* [21], (iv) the methods based on *induction*, and (v) the *proof by consistency* method [37, 86, 89, 109].

(i) The method based on the Clark completion amounts to prove that $M(P) \models \varphi$ by showing that $comp(P) \vdash \varphi$, where $comp(P)$ denotes the Clark completion of program P. (Equivalently, since the Lloyd-Topor transformation from $f \leftarrow \varphi$ to $F(f, \varphi)$ preserves the completion semantics, one may show that $comp(P \wedge F(f, \varphi)) \vdash f$.) This method is sound because $M(P) \models comp(P)$ and it reduces the proof of a satisfaction relation to the proof a theorem in first order predicate calculus. Notice that for some program P and formula φ, such as the ones introduced in our Semaphore and Even-Odd Paths Examples, the relation $M(P) \models \varphi$ can be proved by our unfold/fold proof method and yet $comp(P) \not\vdash \varphi$. This is due to the fact that the transformation rules presented in this section, with the only exception of the tautology rule, preserve the perfect model of a program P, which is a model of $comp(P)$, but they do *not* necessarily preserve *all* models of $comp(P)$.

(ii) Several enhancements of the resolution method have been proposed in the literature for verifying properties of logic programs with negation w.r.t. the perfect model semantics (or generalizations thereof). SLDNF-resolution [103] is based on the *negation as finite failure* rule, but it is unable to deal with infinitely failed derivations and moreover, it is not complete w.r.t. the Clark completion. We have shown that the ability of detecting and eliminating infinitely failed derivations (via the clause deletion rule) is a crucial step in our method for proving theorems.

SLS-resolution [4, 127] enhances the resolution method by using the *negation as* (finite or infinite) *failure* rule. In the absence of *floundering* (see [103] for this notion), SLS-resolution is sound and complete w.r.t. the perfect model semantics, however it is not an effective inference rule, because the set of consequences of the perfect model of a logic program is not recursively enumerable.

SLG-resolution [32] combines resolution and memoing (also called *tabling*). SLG-resolution is an effective method. It is more powerful than the SLDNF-resolution methods for dealing with infinitely failed derivations. Indeed, in [129] it has been shown that logic programming systems based on tabling are good for inductive theorem proving and they are able to verify CTL properties of finite-state transition system [61]. However, the unfold/fold proof method is more powerful than logic programming systems based on tabling, because during program transformation one has the possibility of using the folding rule. This rule is more powerful than tabling, because it allows us to tabulate a conjunction of literals, instead of one literal only. Moreover, during program transformation we may apply the basic goal replacement rule, which is equivalent to applying lemmas in the proofs by induction. Of course, the automation of the unfold/fold method needs more effort than the automation of tabling, but we believe that this difficulty can be overcome by considering specific classes of programs and properties. And indeed, in this section we have presented two such classes.

(iii) Brass and Dix [21] have proposed a query evaluation algorithm for disjunctive logic programs based on transformation rules. The rules considered in [21] include the positive unfolding and tautology rules, and they preserve several semantics including the perfect model semantics for the class of locally stratified logic programs. However, Brass and Dix do not take into consideration the folding and basic goal replacement rules, which play a crucial role in our technique.

(iv) The satisfaction relation $M(P) \models \varphi$ may also be proved by adding to $comp(P)$ a set of formulas, or *induction schemata*, that formalize an induction principle over terms of the Herbrand universe. Thus, one may use standard techniques for inductive theorem proving [19]. The main difference between this method and our unfold/fold proof method is that the latter does not require any induction schema. We leave for future research a formal comparison between the class of formulas that can be proved by induction and the class of formulas that can be proved by the unfold/fold proof method.

(v) The unfold/fold proof method is related to methods for *proof by consistency* (also called *inductionless induction* method) of equational formulas by using term rewriting systems (see [86, 89, 109] and [37] for a brief survey). This relationship is based on the ability of the unfold/fold proof method of proving inductive properties without using an explicit induction schema. However, the proofs by consistency are refutational proofs, they work by finding minimal counterexamples, and they require suitable well-founded orderings on terms, while the unfold/fold proof method does not require such term orderings.

Finally, we would like to mention that in [131] the use of unfold/fold proofs (of definite programs) is advocated for verifying parametrized concurrent systems. More generally, since finite and infinite transition systems can be represented in a straightforward way by logic programs, we can use our unfold/fold transformation techniques for proving properties of those systems and thus, performing finite or infinite model checking. The fact that transitions may be subject to negative conditions [81] is not a difficulty in our approach because, as we already mentioned, we allow logic programs with locally stratified negation. The use of logic programming techniques for infinite state model checking seems to be very promising, as shown in [55, 70, 102]. We believe that the combination of these techniques with transformational proof methods may be very fruitful.

6 Related Methodologies for Program Development and Final Discussion

From what we have presented in these lecture notes 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. Some of these techniques have been developed also in the area of functional programming, and they have been partially extended to the case of concurrent logic programs [148] and constraint logic programs [8, 64, 83, 106].

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 program 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 [1, 43, 53]. 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 [41].

A general framework where program transformation strategies are supported by abstract interpretation techniques, is defined in [17]. Among the many transformation techniques which depend on program analysis techniques, we would like to mention: (i) compiling control (see Section 3.2.1), where the information about the flow of computation is used for generating the unfolding tree, (ii) the specialization method of [75], 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 also Section 3.4), and (iv) various techniques implemented in the Spes system [3, 27] in which *mode analysis* is used to mechanize several transformation strategies.

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 [13, 56, 60].

The field of program transformation partially overlaps with the field of *program synthesis* (see [59] 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 the usual transformation rules. If we allow the use of very sophisticated laws or properties during program transformation (much more sophisticated than the property of *append* used in introductory example in Section 1.3) then it is very difficult to separate the areas of program synthesis and program transformation.

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 suitable transformation rules, but these techniques, we think, go beyond the area of traditional program transformation and belong to the field of logic program synthesis.

There are also techniques by which program synthesis may be viewed as part of program transformation. Among those techniques we would like to mention the so called *completion techniques* presented in [57]. By completion a new program can be derived from an old one by successive applications of semantics-preserving rewriting steps each of which can be viewed as a program transformation step.

Recent developments have indicated a new interesting way of considering the process of program development (or transformation) from specifications. This approach is based on *Constructive Type Theories* (see, for instance, [7, 30, 40, 76, 137]) where a program can be derived from the constructive proof of the existence of an element in a suitable domain type.

Other techniques related to the transformation strategies we have described, such as *program slicing*, *program integration*, and *program enhancement*, have been described in [99, 130].

The transformational methods for developing logic programs are also closely related to methods for *logic program construction* [58, 141, 142], where complex programs are developed by enhancing and composing together simpler programs (see Section 3.2.2). 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.

Concerning the limitations of the transformation approach, the reader should notice that, in general, transformation strategies *cannot* be complete in the sense that their ability of deriving new programs from old programs is limited by the fact that the semantics equivalence of programs is undecidable. Thus, there is no algorithm which for any given initial program explores in finite time all equivalent programs which are more efficient than the initial program.

An area of further investigation is the characterization of the power of the transformation rules and strategies w.r.t. their capability of deriving programs which are more efficient than the initial ones. No conclusive results are available in this direction.

Moreover, the transformation methodology based on the kind of rules we have presented, such as unfolding and folding, has some intrinsic complexity limitations in the sense that there exists a bound on the efficiency improvements that can be obtained via those rules. This has been shown in [157] for the case of functional programs.

A final issue is about the *mechanization* of the transformation rules and strategies for program development. 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 description of some systems which have been already implemented and are under development can be found in [2, 3, 20, 31, 33, 66, 67, 98, 114, 139, 153]. We consider that it is particularly important to construct systems which support the design and execution of transformation rules and strategies as we now indicate.

(i) *User-defined rules and strategies.* The transformation systems should allow the user to introduce and apply different sets of transformation rules, strategies, and higher-order strategies. Higher-order strategies should make it possible to combine already defined strategies and also to dynamically generate new strategies. The need for higher-order manipulations has already emerged both in partial evaluation, where the concepts of generating extension and metasystem transition play an important role [63, 78, 146], and in unfold/fold program transformation where, for instance, suitable generalization strategies may be applied after analyzing the outcome of other strategies [117]. A theoretical issue to be addressed is the capability of those rules and strategies to derive programs with certified levels of efficiency when transforming programs of some given classes.

(ii) *Language parametricity.* The transformation systems should be usable for different programming languages, that is, it should be parametric w.r.t. the syntax and semantics of the language in use, and also w.r.t. the calculus for establishing the computational complexity of the programs written in that language (for such a calculus see, for instance, [149, 84]).

(iii) *Analysis tools.* The transformation systems should allow for the use of static analysis techniques, possibly based on abstract interpretation, such as those for binding-time analysis [88], mode analysis, and type analysis. The information collected by these techniques is used to direct the application of sophisticated transformation and specialization strategies as indicated in [38, 74, 88, 120]. In order to enhance the strategies for program transformation some algorithmic forms of theorem proving may also be added to the system.

(iv) *Auxiliary transformation techniques.* The transformation systems should allow an easy application of various transformation techniques such as: continuation-based transformations [152], finite differencing [112], schema transformations (see [113] for a survey), and changes of data structures [117].

Acknowledgements

We would like to thank A. Jung, E. Ritter, and the members of the Programme Committee of the ESSLLI 2000 European Summer School on Logic, Language, and Information. We are very happy that they kindly accepted our contribution. We are also grateful to our students who through their remarks and questions helped us clarify some obscure points.

Many thanks also to M. Bruynooghe, D. De Schreye, L. Fribourg, J. P. Gallagher, M. Leuschel, M. Maher, H. Seki, and our colleagues of the IFIP Working Group 2.1 on *Algorithmic Languages and Calculi* for their helpful comments and advice on many issues concerning the transformation of logic programs. Their encouragement throughout the years of our involvement in the area of Program Transformation and Derivation is greatly appreciated.

Many of the program derivations described in these notes have been performed using the MAP transformation system. A first version of that system was written by O. Aioni. A second, more advanced version has been developed by S. Renault.

References

[1] S. Abramsky and C. Hankin (eds.). *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] K. Aerts and D. Van Besien. Autolap: A system for transforming logic programs. Technical report, Department of Electronics, University of Rome II, Rome (Italy), 1991.

[3] 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.

[4] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.

[5] 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.

[6] 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.

[7] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Toplas*, 7(1):113–136, 1985.

[8] 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.

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

[10] R. S. Bird and L. G. L. T. Meertens. Two exercises found in a book on algorithmics. In L. G. L. T. Meertens, editor, *TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Tölz, Germany*, pages 451–457. North-Holland, 1987.

[11] R. S. Bird and O. De Moor. *Algebra of Programming.* Prentice Hall, 1996.

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

[13] 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.

[14] 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.

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

[16] G. Boudol and L. Kott. Recursion induction principle revisited. *Theoretical Computer Science*, 22:135–173, 1983.

[17] 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.

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

[19] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[20] J. M. Boyle. Lisp to Fortran – program transformation applied. In P. Pepper, editor, *Proceedings Workshop on Program Transformation and Programming Environments*, Nato ASI Series F, Computer and Systems Sciences 8, pages 291–298. Springer-Verlag, 1984.

[21] S. Brass and J. Dix. Semantics of (disjunctive) logic programs based on partial evaluation. *Journal of Logic Programming*, 40:1–46, 1999.

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] A. Bundy, A. Smaill, and G. Wiggins. The synthesis of logic programs from inductive proofs. In J. W. Lloyd, editor, *Computational Logic, Symposium Proceedings, Brussels, November 1990*, pages 135–149, Berlin, 1990. Springer-Verlag.

[29] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[30] R. M. Burstall et al. The Lego project. Technical report, Edinburgh University, Edinburgh (Scotland), 1991.

[31] J. Cai and R. Paige. The RAPTS transformation system. Technical report, Computer Science Department, New York University, New York, USA, 1990.

[32] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.

[33] CIP Language Group. *The Munich Project CIP. Volume II: The Transformation System CIP-S*. Lecture Notes in Computer Science 292. Springer-Verlag, 1987.

[34] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[35] 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.

[36] 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.

[37] H. Comon and R. Nieuwenhuis. Induction = I-axiomatization + first-order consistency. *Information and Computation*, 1999. to appear.

[38] C. Consel and S.C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[39] 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.

[40] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[41] 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.

[42] B. Courcelle. Equivalences and Transformations of Regular Systems – Applications to Recursive Program Schemes and Grammars. *Theoretical Computer Science*, 42:1–122, 1986.

[43] 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.

[44] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation. International Seminar, Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[45] J. Darlington. *A Semantic Approach to Automatic Program Improvement*. PhD thesis, Department of Machine Intelligence, Edinburgh University, Edinburgh (Scotland) U.K., 1972.

[46] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[47] J. Darlington. An experimental program transformation system. *Artificial Intelligence*, 16:1–46, 1981.

[48] 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.

[49] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.

[50] 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.

[51] 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.

[52] 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.

[53] S. K. Debray, editor. *Special Issue of the Journal of Logic Programming on Abstract Interpretation*, volume 12, Nos. 2&3. Elsevier, 1992.

[54] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5:207–229, 1988.

[55] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *5th International Conference TACAS'99*, Lecture Notes in Computer Science 1579, pages 223–239. Springer-Verlag, 1999.

[56] 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.

[57] N. Dershowitz. Synthesis by completion. In *Proc. 9th International Joint Conference on Artificial Intelligence*, pages 208–214, 1985.

[58] Y. Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.

[59] Y. Deville and K.-K. Lau. Logic program synthesis. *Journal of Logic Programming*, 19, 20:321–350, 1994.

[60] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 1(1):133–155, 1988.

[61] E. A. Emerson. Temporal and modal logic. In J. van Leuveen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.

[62] J. Engelfriet. Tree Automata and Tree Grammars. DAIMI FN-10, Department of Computer Science, University of Aarhus, 8000 Aarhus, Denmark, April 1975.

[63] A. P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18(1):41–67, 1982.

[64] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

[65] S. Etalle, M. Gabbrielli, and M. C. Meo. Unfold/fold transformations of CCP programs. In D. Sangiorgi and R. de Simone, editors, *Proceedings of the International Conference on Concurrency Theory, Concur98*, Lecture Notes in Computer Science 1466, pages 348–363, 1998.

[66] M. S. Feather. A system for assisting program transformation. *ACM Toplas*, 4(1):1–20, 1982.

[67] 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.

[68] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation*, 15:775–805, 1993.

[69] L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In D. H. D. Warren and P. Szeredi, editors, *Proceedings Seventh International Conference on Logic Programming, Jerusalem, Israel, June 18-20, 1990*, pages 685–699. The MIT Press, 1990.

[70] L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *CONCUR '97*, Lecture Notes in Computer Science 1243, pages 96–107. Springer-Verlag, 1997.

[71] 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, U.K.*, pages 111–125. Springer-Verlag, 1992.

[72] N. E. Fuchs and R. Schwitter. Specifying logic programs in controlled natural language. In *Workshop on Computational Logic for Natural Language Processing, Proceedings CLNLP '95*, 1995.

[73] H. Fujita. An algorithm for partial evaluation with constraints. Technical Memorandum TM-0367, ICOT, Tokyo, Japan, 1987.

[74] 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.

[75] J. P. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 6(2):305–333, 1991.

[76] D. Galmiche. Proofs and program development in AF2. Notes of the IFIP W.G. 2.1 meeting, Louvain-la-Neuve, Belgium, 1991.

[77] 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.

[78] R. Glück. On the mechanics of metasystem hierarchies in program transformation. In M. Proietti, editor, *Proceedings of LoPSTr '95, Utrecht, The Netherlands*, Lecture Notes in Computer Science 1048, pages 234–251. Springer-Verlag, 1996.

[79] R. Glück and A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *3rd International Workshop on Static Analysis, Padova, Italy, September 1993*, Lecture Notes in Computer Science 724, pages 112–123. Springer-Verlag, 1993.

[80] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Gück, and P. Thiemann, editors, *Partial Evaluation*, Lecture Notes in Computer Science 1110, pages 137–160. Springer, 1996.

[81] J. F. Groote. Transition system specification with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[82] 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.

[83] 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.

[84] M. Hofmann. Type systems for polynomial time. Lecture Notes of 12th ESSLLI School, School of Computer Science, Birmingham University, Birmingham B15 2TT, U.K., 2000.

[85] C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.

[86] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25:239–266, 1982.

[87] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[88] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[89] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1):1–33, 1989.

[90] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1992.

[91] T. Kanamori and M. Maeji. Derivation of logic programs from implicit definition. Technical Report TR-178, ICOT, Tokyo (Japan), 1986.

[92] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformation. *Theoretical Computer Science*, 75:139–156, 1990.

[93] M. Kirschenbaum, A. Lakhotia, and L. Sterling. Skeletons and techniques for Prolog programming. TR 89-170, Case Western Reserve University, 1989.

[94] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[95] L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.

[96] L. Kott. Unfold/fold program transformation. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 411–434. Cambridge University Press, 1985.

[97] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[98] B. Krieg-Brückner, editor. *COMPASS, a COMPrehensive Algebraic approach to System Specification and development. ESPRIT Basic Research Working Group 3264, Objectives, State of the Art, References.* University of Bremen, Germany, 1989.

[99] A. Lakhotia and L. Sterling. Composing recursive logic programs with clausal join. *New Generation Computing*, 6(2 & 3):211–226, 1988.

[100] K.-K. Lau, M. Ornaghi, A. Pettorossi, and M. Proietti. Correctness of logic program transformation based on existential termination. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*. MIT Press, 1995.

[101] M. Leuschel, B. Martens, and D. de Schreye. Some achievements and prospects in partial deduction. *ACM Computing Surveys*, 30 (Electronic Section)(3es):4–es, 1998.

[102] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In *PreProceedings of LOPSTR '99, Venice, Italy*, pages 137–144. Università Ca' Foscari di Venezia, Dipartimento di Informatica, 1999.

[103] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.

[104] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[105] M. J. Maher. Correctness of a logic program transformation system. IBM Research Report RC 13496, T. J. Watson Research Center, 1987.

[106] M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

[107] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Toplas*, 2:90–121, 1980.

[108] E. Marakakis and J. P. Gallagher. Schema-based tod-down design of logic programs using abstract data types. In *Proceedings of LoPSTr '94, Pisa, Italy*, Lecture Notes in Computer Science 883, pages 138–153. Springer-Verlag, 1994.

[109] D. R. Musser. On proving inductive properties of abstract data types. In *Proceedings of the 7-th ACM Symposium on Principles of Programming Languages (POPL'80)*, pages 154–162. ACM Press, 1980.

[110] L. Naish. *Negation and Control in Prolog*. Lecture Notes in Computer Science 238. Springer-Verlag, 1985.

[111] S. Narain. A technique for doing lazy evaluation in logic. *Journal of Logic Programming*, 3(3):259–276, 1986.

[112] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[113] H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[114] H. A. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15:199–236, 1983.

[115] 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.

[116] A. Pettorossi. Transformation of programs and use of tupling strategy. In *Proceedings Informatica 77, Bled, Yugoslavia*, pages 1–6, 1977.

[117] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[118] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[119] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2&3):197–230, 1999.

[120] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *Proc. 24-th ACM Symposium on Principles of Programming Languages, Paris, France*, pages 414–427. ACM Press, 1997.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] M. Proietti and A. Pettorossi. Transforming inductive definitions. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming*, pages 486–499. MIT Press, 1999.

[127] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journ. of Automated Reasoning*, 5:167–205, 1989.

[128] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–34, July 1969.

[129] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV '97*, Lecture Notes in Computer Science 1254, pages 143–154. Springer-Verlag, 1997.

[130] T. Reps. Algebraic properties of program integration. In *Proc. European Symposium on Programming, ESOP '90*, Lecture Notes in Computer Science 432, pages 326–340. Springer-Verlag, 1990.

[131] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. Proofs by program transformation. In *PreProceedings of LOPSTR '99, Venice, Italy*, pages 57–64. Università Ca' Foscari di Venezia, Dipartimento di Informatica, 1999.

[132] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12:7–51, 1993.

[133] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Toplas*, 18(2):175–234, 1996.

[134] T. Sato and H. Tamaki. Deterministic transformation and deterministic synthesis. In *Future Generation Computers*. North-Holland, 1988.

[135] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[136] 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.

[137] M. Sintzoff, M. Weber, Ph. de Groote, and J. Cazin. Definition 1.1 of the generic development language Deva. Technical report, Unité d'Informatique, Université Catholique de Louvain, Louvain-La-Neuve, Belgium, 1989.

[138] D. R. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.

[139] D. R. Smith. Automating the design of algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development, IFIP TC2/W.G 2.1 State-of-the-Art Report*, Lecture Notes in Computer Science 755, pages 324–354. Springer-Verlag, 1993.

[140] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Fifth European Symposium on Programming Languages and Systems, ESOP '94*, Lecture Notes in Computer Science 788, pages 485–500. Springer-Verlag, 1994.

[141] L. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In J.-M. Jacquet, editor, *Constructing Logic Programs*, chapter 6, pages 127–140. Wiley, 1993.

[142] 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.

[143] L. S. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1994. Second Edition.

[144] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

[145] H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan, 1986.

[146] V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker and J van Leeuwen, editors, *Proceedings 7th Colloquium on Automata, Languages and Programming, Noordwijkerhout, the Netherlands*, Lecture Notes in Computer Science 85, pages 645–657. Springer-Verlag, 1980.

[147] V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.

[148] 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.

[149] J. S. Vitter and Ph. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leuveen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 431–524. Elsevier, 1990.

[150] P. L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[151] 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.

[152] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

[153] D. Wile. POPART: Producer of parser and related tools. Technical Report RR-82-21, ISI, 4676 Admiralty Way, Marina del Rey, California USA, 1982.

[154] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, 1993.

[155] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc., 1976.

[156] J. Zhang. An automatic d–list transformation algorithm for Prolog programs. Technical report, Department of Computer Science, University College of Swansea, U.K., 1987.

[157] H. Zhu. How powerful are folding / unfolding transformations? *Journal of Functional Programming*, 4(1):89–112, 1994.