Transformation Systems and Nondeclarative Properties

Annalisa Bossi[1] and Nicoletta Cocco[1] and Sandro Etalle[2]

[1]Dipartimento di Informatica, Università di Venezia
via Torino 155, 30172 Venezia, Italy
{bossi,cocco}@dsi.unive.it

[2] University of Twente and CWI, Amsterdam
Department of Computer Science
PO Box 217, 7500 AE Enschede, The Netherlands
etalle@cs.utwente.nl

Abstract

Program transformation systems are applied both in program synthesis and in program optimization. For logic programs the "logic" component makes transformations very natural and easy to be studied formally. But, when we move to Prolog programs, the "control" component cannot be ignored. In particular we need to cope with termination properties which are essential for ensuring the reachability of solutions for a given query.

We give an overview of the main proposals in the field of transformation systems for logic programs and we emphasize how they cope with those properties of logic programs which are not strictly declarative. We focus in particular on how the transformation can affect the termination of a program.

1 Introduction

Virtuous programming methodology, which consists in focusing on correctness of programs at first and on their efficiency only afterwards, fits particularly well with the logic programming paradigm, as stated by the famous motto: *Algorithm = Logic + Control* [Kow79]. This encourages the application of *transformation systems* to logic programs both for synthesizing a correct program from a logic specification [Dev90] and for optimizing it [TS84].

The main requirements for a practical transformation systems are on one hand to guarantee the preservation of interesting program properties and on the other hand to be supported by an automatic or semi-automatic tool. The most important program properties are characterized by the "Logic" component, namely declarative semantics describing the intended results of computations. But nondeclarative properties, the "Control" component, namely the actual behaviour of the interpreter, are extremely relevant.

The logic programming paradigm, in its pure form, knows *two sources of nondeterminism*:

ND1 The choice of the atom in the query (selection rule),

ND2 The choice of the clause to resolve it.

The Prolog interpreter however gives up the first one by employing a fixed leftmost selection rule, and the second one with a fixed top-down selection method. This latter nondeterminism is recovered (thought not entirely, for the possibility on nontermination) by the use of backtracking. Between pure logic programming and pure Prolog, we find the well-studied paradigm of "logic programming + leftmost selection rule", which is of theoretical relevance.

The influence of (ND1) and (ND2) varies according to the kind of *observable behaviour*, *observable* for short, we focus on. For instance, if the observable is the *Success Set* of the program, then, by the well-known result of the independence from the selection rule, (ND2) can compensate for the absence of (ND1). For other observables this does not apply: for instance, if one observes the *Finite Failure Set* of the program, then (ND1) is of influence, while (ND2) is not. The same holds for *(universal) termination.*

A program transformation system is characterized by a set of *basic transformation operations* and a *strategy* which combines them for a given aim. The transformation operations are generally constrained by *applicability conditions* which ensure their *correctness*, that is, the preservation of the observables of interest. These applicability conditions should balance between the need to capture the majority of cases and the need to allow for a simple verification, if possible they should be purely syntactical in order to be automatically verified by the system. Similarly, the strategy should strike a balance between being powerful and being automatizable, thus reducing interactions with the user to the minimum.

When transformation techniques, formulated for logic programs in general, are applied to real programs, Prolog's choices wrt (ND1) and (ND2) become relevant for preserving the observables one is interested in. Useful observables usually represent the results of computation, which for Prolog programs can be characterized by the *Computed Answer Substitutions* [FLMP93] and the *Finite Failure Set* (when negation is used). But since Prolog replaces (ND2) by backtracking, also termination properties are essential, as they guarantee the effective reachability of solutions.

In this paper we intend to give an overview of some of the transformation systems which have been proposed for logic programs and we look at how they influence the observables of a program. We shall look especially at nondeclarative properties, and at termination in particular. In fact, transformation systems preserving termination are suitable to deal with pure Prolog programs. We think that the field is mature enough to allow for a comparison and a classification of such systems by considering the basic transformation operations, the preserved termination property, the purpose and the level of automatization.

The paper is organized as follows. In Section 2 we give some notation on general logic programs and briefly recall the major termination properties. In Section 3 we define the simplest unfold/fold transformation system, which is common to the majority of transformation systems. We illustrate the properties of both the basic operations, unfold and fold, and we discuss the problems related to preserving termination and the proposed solutions. We also discuss the need of reordering literals in clause bodies during transformations and define a switch operation. In Section 4 we introduce the powerful replacement operation, discuss its properties and the proposals for preserving termination. A short conclusion follows in Section 5.

2 Preliminaries

2.1 General Programs and LDNF-Resolution

Let $\mathcal{P}$ be a finite set of *predicate symbols* (or *relations*). An *atom* is an object of the form $p(t_1, \dots, t_n)$ where $p \in \mathcal{P}$ is an n-ary predicate symbol and $t_1, \dots, t_n \in \mathcal{T}$. A *literal* is either an atom A (a *positive literal*) or the negation of an atom $\neg A$ (a *negative literal*). A *general query* is a possibly empty finite sequence of literals $L_1, \dots, L_n$ ($n \geq 0$). Following the convention adopted by Apt in [Apt97], we use bold characters (e.g. $\mathbf{B}$) to indicate sequences of objects, typically $\mathbf{B}$ indicates a sequence of literals, $B_1, \dots, B_n$, $\mathbf{t}$ indicates a sequence of terms, $t_1, \dots, t_n$, and $\mathbf{x}$ denotes a sequence of variables, $x_1, \dots, x_n$. A *general clause* is a construct of the form $H \leftarrow \mathbf{B}$ where H is an atom (the *head*) and $\mathbf{B}$ is a general query (the *body*). When $\mathbf{B}$ is empty, $H \leftarrow \mathbf{B}$ is written $H \leftarrow$ and is called a *unit clause.* A *general program* is a finite set of general clauses.

Apart from this, we use the standard notation of Lloyd [Llo87] and Apt [Apt97]. In particular, given a syntactic construct E (so for example, a term, a literal or a set of equations) we denote by $Var(E)$ the set of the variables appearing in E. Given a substitution $\theta = \{x_1/t_1, ..., x_n/t_n\}$ we denote by $Dom(\theta)$ the set of variables $\{x_1, \dots, x_n\}$, and by $Ran(\theta)$ the set of variables appearing in $\{t_1, \dots, t_n\}$. Finally, we define $Var(\theta) = Dom(\theta) \cup Ran(\theta)$.

A substitution θ is called *grounding* if $Ran(\theta)$ is empty, and it is called a *renaming* if it is a permutation of the variables in $Dom(\theta)$. By $Pred(E)$ we denote the set of predicate symbols occurring in the expression E.

We use a notation introduced in [AP93], and we say that a predicate p *is defined in the program* P iff there is a clause in P that uses p in its head.

Definition 2.1 Let P, Q be programs, which define different predicates, and p, q relations in $Pred(P)$.

(i) We say that p *refers to* q *in* P if there is a clause in P that uses p in its head and q in its body.

(ii) We say that p *depends on* q *in* P, and write $p \sqsupseteq q$, if (p, q) is in the reflexive, transitive closure of the relation *refers to.*

(iii) We say that P *extends* Q, $P \sqsupseteq Q$, if there is no $q \in Pred(Q)$ which refers (in Q) to a predicate p defined in P.

(iv) Let B be an atom, by $P|_B$ we denote the set of clauses of P that define the predicates which the predicate of B depends on. Similarly by $P|_p$ we denote the set of clauses of P that define a predicate p and all the predicates which it depends on. □

We consider SLDNF-resolution with Prolog selection rule, that is the leftmost selection rule. As usual, we call this form of resolution *LDNF-resolution.*

Following Apt and Pedreschi's approach in studying the termination of general programs [AP93], we view the LDNF-resolution as a top-down interpreter which, given a general program P and a general query Q, attempts to build a search tree for $P \cup \{Q\}$ by constructing its branches in parallel. The branches in this tree are called *LDNF-derivations* of $P \cup \{Q\}$ and the tree itself is called *LDNF-tree* of $P \cup \{Q\}$. Negative literals are resolved using the negation as failure rule which calls for the construction of a *subsidiary LDNF-tree.* If during this subsidiary construction the interpreter diverges, the (main) LDNF-derivation is considered to be infinite. An LDNF-derivation is finite also if during its construction the interpreter encounters a query with the first literal being negative and non-ground. In such a case we say that the LDNF-derivation *flounders*. An LDNF-tree is called *non-floundering* if none of its derivations flounders.

By termination of a general program we actually mean termination of the underlying interpreter. Hence in order to ensure termination of a query Q in a program P, we require that all LDNF-derivations of $P \cup \{Q\}$ are finite.

We use the following abbreviations for a program P: M_P for the least Herbrand model of P, and $comp(P)$ for Clark's completion of P [Cla78].

2.2 Termination Properties

We recall in this section some important termination properties for logic programs which have been studied in the literature. We refer all definitions to general programs.

Definition 2.2 (Terminating Program) A program P is called *terminating* iff all SLDNF-derivations of P starting in any ground query are finite.

This is a very strong termination property since it must hold for any selection rule. If we consider only the leftmost selection rule of Prolog, namely LDNF-resolution, the following property of left termination is more appropriate.

Definition 2.3 (Left Terminating Program) A program P is called *left terminating* iff all LDNF-derivations of P starting in any ground query are finite.

For verifying a termination property on a program, a common technique is to find a measure on queries which, under certain conditions, can only decrease during the computation. Such measure is based a *level mapping*, namely a map from ground literals to natural numbers. Two important classes of programs had been characterized by means of properties of level mappings: *Acyclic programs* and *Acceptable programs.*

Acyclic programs were introduced by Cavedon [Cav89] and have been further studied by Apt and Bezem [AB91]. An acyclic program is characterized by the fact that for any ground instance of any clause, the level mapping of the head is greater than the level mapping of each literal in the body.

We can relate acyclic and terminating programs: If P is an acyclic program then P is terminating. Moreover if P is a definite program, then P is terminating iff P is acyclic. When negation is allowed in clause bodies, there are programs which are terminating but not acyclic. This is caused by the presence of floundering derivations, since non-ground negative literals are not selected and some infinite branches of the search tree cannot be explored [AB91]. Note that if a program is terminating or acyclic, it is also left terminating.

The concept of *acceptable program* generalizes the one of acyclic program and had been introduced by Apt and Pedreschi in [AP90, AP93] to characterize left terminating programs.

In the two previous definitions, all ground queries were requested to terminate. Vasak and Potter in [VP86] introduced two different termination properties which refer to a specific query **Q** in a program: Universal termination and existential termination.

Definition 2.4 (Universal and Existential Termination)

- *A query* $\mathbf{Q}$ *is universally terminating in* P iff all LDNF-derivations for $\mathbf{Q}$ in P are finite.
- *A query* $\mathbf{Q}$ *is existentially terminating in* P iff there exists at least one LDNF-derivation for $\mathbf{Q}$ in P which is finite.

Note that if every ground query universally terminates in a program P, then P is left terminating. Conversely, if P is left terminating then every ground query is universally terminating in it.

In order to characterize programs where every query is universally terminating, we introduced also the following very strong termination property.

Definition 2.5 (Always Left Terminating Program) A program P is called *always left terminating* iff all LDNF-derivations of P starting in any query are finite. $\square$

In an always left terminating program no computation can diverge. These programs are generally defined by clauses which are not recursive or by built-ins and used to perform some checks.

Note that if a program is always left terminating, then it is also left terminating. Hence the class of left terminating programs includes the ones of terminating, acyclic and always left terminating programs.

Another interesting class of queries we will refer to, and which contains all the ground ones, is the class of *well-moded* queries. Modes are extensively used in the literature on logic programs, usually they indicate how the arguments of a relation should be used. A *mode* is a function that labels as *input* (+) or *output* (-) the positions of each predicate in order to indicate how the arguments of a predicate should be used. Most predicates have a natural moding, which reflects their intended use. For example, the natural moding for the usual program `append`, when used for concatenating two lists, is `append(+,+,-)`. When talking about moded programs, we assume that each predicate symbol has a unique mode associated to it; multiple modes may be obtained by simply renaming the predicates. If Q is a query, we denote by $In(Q)$ (resp. $Out(Q)$) the set of terms filling in the input (resp. output) positions of predicates in Q.

The concept of *well-moded program* is essentially due to Dembinski and Maluszynski [DM88]. Intuitively a program is well-moded when the modes of literals in each clause, which reflect the dataflow taking place in it, are consistent with the left-to-right selection rule. The definition of well-moded program and query has been given for definite programs but it can be extended to general programs, as in [BCER01]. *Moded level mappings* are introduced in [EBC99], they do not take into account output terms, but only input terms. The relevance of well-moding and moded level mappings for termination is studied for definite programs in [EBC99]. We give here an extended definition for general programs.

Definition 2.6 (Well-Terminating Program) A program is called *well-terminating* iff all its LDNF-derivations starting in any well-moded query are finite.

Notice that a well-terminating program is also left terminating.

3 A Simple Transformation System: Unfold and Fold

In their seminal papers [TS84, ST84], Sato and Tamaki adapted to definite logic programs the ideas on program transformations firstly introduced by Burstall and Darlington for functional programs [BD77]. They defined *the basic unfold/fold transformation system*, based on the operations of *new definition, unfold and fold.* Then they made it more powerful with *replacement* and further with *clause addition* and *clause deletion.* They studied the system wrt to a declarative semantics given by the least Herbrand model. This was the starting point for a number of studies on transformations preserving properties of logic programs, both definite and general, which can be expressed by a declarative semantics, such as: Success Set, Computed Answer Substitutions and Finite Failures Set, see for example [Mah87, KK90, Sek91, GS91, BCE92, AD93, BC93] just to quote some of these efforts.

We start by considering the basic operation: Unfold. From now on, standardization apart is always assumed.

3.1 Unfold

Unfold is the fundamental operation for partial evaluation [LS91] and it consists in applying a resolution step to an atom in a clause body, by using all possible resolving clauses.

Definition 3.1 (Unfold) Let $cl : H \leftarrow \mathbf{J}, L, \mathbf{K}.$ be a clause of a program P, L a positive literal and $\{A_1 \leftarrow \mathbf{B}_1., \dots, A_n \leftarrow \mathbf{B}_n.\}$ the set of clauses of P whose heads unify with L, by mgu's $\{\theta_1, \dots, \theta_n\}$.

- *unfolding L in cl* consists of substituting cl with $\{cl'_1, \dots, cl'_n\}$, where, for each i, $cl'_i = (H \leftarrow \mathbf{J}, \mathbf{B}_i, \mathbf{K})\theta_i$.

L is the *unfolded atom*, $\{A_1 \leftarrow \mathbf{B}_1., \dots, A_n \leftarrow \mathbf{B}_n.\}$ are called the *unfolding clauses* and $\{cl'_1, \dots, cl'_n\}$ are the unfolded clauses (*unfoldings* for short). □

There are a few slightly different versions of this operation. A more powerful definition of unfold for definite programs has been given in [Gal91] where the unfold operation is based on partial evaluation, namely it consists in building a finite (incomplete) SLD-tree for the body of a clause $H \leftarrow \mathbf{B}.$ and in getting the resultants $H\alpha_i \leftarrow \mathbf{G}_i.$ as unfoldings. With such a "multi-step" unfold it is possible to obtain unfoldings which are not obtainable through a series of one step unfoldings.

Note that we define the unfold only for positive literals (atoms). In some proposals also a negative literal can be unfolded: For example in [GS91], if $L = \neg A$ with A ground, and A has a finitely failing SLDNF-tree, then unfolding L is done by deleting L from the clause cl. On the other hand if A has a successful derivation, then the same operation yields the removal of cl. In [AD94] another unfold for negative literals is defined in the context of a well-founded partial evaluation: Any negative literal $\neg p(\mathbf{t})$ in the program is replaced by an atom $notp(\mathbf{t})$, where the new predicate $notp$ is defined as the negation of the completed definition of p.

3.1.1 Declarative Properties

Thanks to its correspondence to a resolution step, the unfold operation in Definition 3.1, is safe wrt basically all the declarative semantics available for logic programs:

The least Herbrand model, as shown already in [TS84], the Success Set and the Computed Answer Substitutions semantics and this was shown by [KK90]. When used alone, unfold preserves also the Finite Failure Set, while in combination with other transformation operations, such as fold, this is no more preserved [Sek91].

3.1.2 Fixing the Selection Rule

Let us discuss now what happens when we are interested in preserving more procedural properties as termination ones. In this Section we analyze what happens when we give up on (ND1) by fixing the selection rule.

Let us start by considering termination. *For definite programs*, in [BC94] we proved that *unfold preserves universal termination of a query* and as a consequence it preserves also left termination and well-termination. Unfortunately this reasoning does not carry over to general logic programs, due to the possibility that they terminate by floundering. Consider:

```
p ←not(trigger(X)), q(X), p.
trigger(a).
q(b).
```

This program is left terminating. Notice however that the query `p` terminates by floundering. Now, if we unfold `q(X)` in the first clause, we obtain a program containing the clause `p ←not(trigger(b)), p`. In this program the query `p` does not left terminate any longer.

This has to do with the fact that Definition 3.1 of *unfold allows for a left-propagation of the bindings*: The atoms on the left of the unfolded atom might be instantiated during the unfolding (i.e. **J** becomes instantiated). In the above example this happened to `not(trigger(X))`. This implies that after the unfold operation some "calls" might be more instantiated than before the unfold.

On the other hand, *for general programs* we can say that *the unfold operation maintains acyclicity* [BE94] *and acceptability* [BCE96b]: If P is acyclic (resp. acceptable wrt a certain level mapping $|\ |$ and a model M) and P' is obtained from P by the application of an unfold operation, then P' is acyclic as well (resp. acceptable wrt $|\ |$ and M as well).

Another consequence of the already mentioned left-propagation of the bindings is that *unfold can " increase termination"*, namely the transformed program can terminate with a failure also for queries which are non-terminating in the original program. Consider the trivial program:

```
c1:   q ←p.
c2:   p ←p, r.
```

If we unfold r in $c2$ we obtain the one-line program `q ←p.` for which all queries terminate since they finitely fail. This can happen also when the unfolded atom is not finitely failing. Consider the non-left terminating program:

```
q ←p(X), r(X).
p(s(X)) ←p(X).
r(a).
```

In this program the query `q` does not left terminate, however, by unfolding `r(X)`, we obtain a left terminating program.

3.1.3 Pure Prolog

We now see what happens when we substitute (ND2) by a fixed clause selection augmented with backtracking. In this setting the order of the clauses in a program becomes relevant, and we intend to consider more procedural observables. We notice the following: *Unfold can change the order of the computed answer substitutions.* Consider:

```
c1:  p(X) ← q(X), r(X).
c2:  q(a).
c3:  q(b).
c4:  r(b).
c4:  r(a).
```

The query `p(X)`, with a Prolog interpreter, has the sequence of computed answer substitutions $\{\mathtt{X} = \mathtt{a}, \mathtt{X} = \mathtt{b}\}$. If we unfold `r(X)` in `c1` we obtain:

```
c1':  p(X) ← q(b).
c2':  p(X) ← q(a).
        ⋮
```

Now the query `p(X)` returns first the answer `X=b` and then the one `X=a`.

Even in the absence of non-logical predicates, this apparently minor fact can have annoying consequences. In particular, unfold can deteriorate the performances of a program. Consider:

```
p ← heavy(X), q(X).
heavy(X) ← lots_of_calculations, X is b.
q(a).
q(b)
```

By unfolding `q(X)`, we obtain.

```
p ← heavy(a).
p ← heavy(b).
heavy(X) ← lots_of_calculations, X is b.
```

In the second program, the query `p` generates two calls to `heavy`, while in the first one `heavy` was called only once.

Thus *the left-propagation of bindings can deteriorate efficiency and/or spoil termination.* Partial evaluation systems have different ways for dealing with this. Gallagher, [Gal91] for the SP system introduces the concept of *determinate unfolding.* Roughly speaking, an unfolding is determinate if it returns no more than one clause (more precisely, no more than one *live* clause, where dead clauses are those that contain an immediately failing atom). This guarantees that the amount of nondeterminism is not increased, which would be harmful for the efficiency. Determinate unfolding is also used in the ECCE partial evaluation system [LMS98].

In [PP91] Pettorossi and Proietti propose two restrictive definitions of unfold for definite programs: *Unfold of the leftmost atom*, which is clearly not harmful wrt any semantics (as it trivially cannot cause any left-propagation), and *deterministic non-left-propagating unfold.*

Definition 3.2 (Deterministic Non-Left-Propagating Unfold) The unfolding of a clause $cl : H \leftarrow \mathbf{J}, A, \mathbf{K}.$ wrt the atom A is *deterministic non-left-propagating* iff

1. there exists exactly one clause whose head is unifiable with A via an mgu θ;
2. $(H \leftarrow \mathbf{J})\theta$ is a variant of $H \leftarrow \mathbf{J}$.

They proved that both such restrictive unfold operations preserve the "sequence of answer substitution semantics" (a semantics for Prolog programs, defined in [JM84, Bau89], which takes into account also the order of the computed answer substitutions, together with their multiplicity). This guarantees that if the initial program is left terminating, then the resulting program is left terminating as well.

Also *partial evaluation systems for real Prolog generally forbid left propagation of the variable bindings.* Mixtus [Sah93], for instance exploits disjunction for unfolding Prolog programs.

Definition 3.3 (Unfold in Mixtus) Let $cl : H \leftarrow \mathbf{J}, L, \mathbf{K}.$ be a clause of a program P, L a positive literal and $\{A_1 \leftarrow \mathbf{B}_1., \ldots, A_n \leftarrow \mathbf{B}_n.\}$ the set of clauses of P whose heads unify with L, by mgu's $\{\theta_1, \ldots, \theta_n\}$.

- *unfolding L in cl* consists of substituting cl with
  $cl' : H \leftarrow \mathbf{J}, (\bigvee_{i=1}^{n} (L = A_i, \mathbf{B}_i)), \mathbf{K}.$

This unfold guarantees that no left-propagation is performed, which in turn ensures that the system is correct also in the presence of extra-logical predicates.

A conceptually similar approach is used in the PADDY system [Pre92] for partial evaluation, but the disjunction is obtained by means of the introduction of a new definition.

3.1.4 Prolog

When extra-logical features are involved, unfold can create further problems. One of them is the possible *loss of computed answer substitutions*, as shown in the following examples. Let us consider the program

```
c1:   p(X)  ← q(X).
c2:   p(a).
c3:   q(b)  ← !.
c4:   q(c).
```

for the query `p(X)` we get the computed answer substitutions `X = b, X = a`.
By unfolding `q(X)` in `c1` we get

```
d1:   p(b)  ← !.
d2:   p(c).
c2:   p(a).
c3:   q(b)  ← !.
c4:   q(c).
```

Now, for the query `p(X)`, we obtain only `X = b`. Similarly let us consider

```
c1:   p(X)  ← var(X), q(X).
c2:   q(b).
c3:   q(c).
```

for the query `p(X)` we obtain the computed answer substitutions $\mathtt{X} = \mathtt{b}, \mathtt{X} = \mathtt{c}$. But if we unfold `q(X)` in `c1`, for the same query `p(X)` we now have a failure.

These problems are deeply connected with the left-propagation of bindings, and are automatically avoided if one employs a definition of unfold such as Definition 3.2 or 3.3.

Unfolding in Prolog has been studied in the context of partial evaluation systems also in [LS88, BR89, Pre93]. They proposed either to restrict unfold on extra-logical features or to transform the program before unfolding, in order to eliminate such extra-logical features or at least to have them only in a standard form.

3.2 Fold

Fold is possibly the transformation operation for which we find the most different definitions. In order to approach it, we first define the concept of transformation sequence.

Definition 3.4 (Transformation Sequence) A *transformation sequence* is a sequence of programs $P_0, \ldots, P_n$, $n \geq 0$, such that each program P_{i+1}, $0 \leq i < n$, is obtained from P_i by applying a basic transformation operation to a clause of P_i. $\Box$

The simplest transformation systems (and then sequences) include only three basic operations which allow for a reasonable set of transformations: *new definition, unfold and fold.* In the literature we can find many different definitions for these operations. Here we are forced to choose one and we try to choose it in the most general way, giving a short description of other proposals. Actually, in [TS84] the *new definition* operation is not explicitly considered as a transformation operation; rather, all new definitions are assumed to be present at the beginning of the transformation. Here we follow the same syntax and we assume that every transformation process starts from an *initial program* which already contains new definitions expressed as Prolog clauses.

Definition 3.5 (Initial Program) We call a program P_0 an *initial program* if it can be partitioned into two programs P_{new} and P_{old} such that the following conditions are satisfied:

(I1) $P_{new} \sqsupseteq P_{old}$;

(I2) P_{new} is not recursive.

P_{new} contains the *new definitions*, that is the completed definitions of the predicates defined in P_{new} which are called *new* predicates. The predicates defined in P_{old} are instead the *old* ones. Similarly we say that a literal is a *new* (resp. *old*) literal iff its predicate symbol is.

We now give the most "classical" definition of fold, equivalent to the one by Tamaki and Sato [TS84]: Fold is the inverse of unfold when one single unfold is possible, and it consists in substituting an atom A for an equivalent conjunction of literals $\mathbf{B}$ in the body of a clause cl. The transformation sequence and the fold operation are defined in terms of each other.

Definition 3.6 (Fold) Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence and P_0 an initial program, $cl : H \leftarrow \mathbf{J}, \mathbf{B}, \mathbf{K}.$ be a clause in P_i, and $d : D \leftarrow \mathbf{B}'.$ be a clause in P_{new}. *Folding* $\mathbf{B}$ *in* cl *via* τ consists of replacing cl by $cl' : H \leftarrow \mathbf{J}, D\tau, \mathbf{K}.$, provided that τ is a substitution such that $Dom(\tau) = Var(d)$ and such that the following conditions hold:

(F1) d is the only clause in P_{new} whose head is unifiable with $D\tau$;

(F2) If we unfold $D\tau$ in cl' using d as unfolding clause, then the result of the operation is a variant of cl;

(C1) Either cl defines an *old* predicate, or at least one atom of cl is the result of a previous unfolding.

Notice that the clause used for folding, d, does not necessarily belong to the program P_i in which the folding is performed. The following example is inspired by one in [Sek93].

Example 3.7 Consider the initial program

```
c1:   path(X,X,[X]).
c2:   path(X,Z,[X|Xs]) ← arc(X,Y), path(Y,Z,Xs).

c3:   goodlist([]).
c4:   goodlist([X|Xs]) ← good(X), goodlist(Xs).

c5:   goodpath(X,Z,Xs) ← path(X,Z,Xs), goodlist(Xs).
```

$\mathtt{P_{new}} = \{\mathtt{c5}\}$, thus `goodpath` is the only new predicate. The query `goodpath(X,Z,Xs)` can be employed for finding a path `Xs` starting in the node `X` and ending in the node `Z` which contains exclusively “good” nodes. As it is now, `goodpath` works on a “generate and test” basis: First it produces a whole path, and then it checks whether it contains only “good” nodes or not. Of course this strategy is quite naive: Checking if the node is “good” or not *while* generating the path would noticeably increase the performances of the program. We can obtain such an improvement via an unfold/fold transformation. By unfolding `path(X, Z, Xs)` in the body of `c5`, we obtain

```
c6:   goodpath(X,X,[X]) ← goodlist([X]).
c7:   goodpath(X,Z,[X|Xs]) ← arc(X,Y), path(Y,Z,Xs),
      goodlist([X|Xs]).
```

In the above clauses we can unfold `goodlist([X])` and `goodlist([X|Xs])`. The resulting clauses, after further unfolding `goodlist([])` in the clause obtained from `c6`, are

```
c8:   goodpath(X,X,[X]) ← good(X).
c9:   goodpath(X,Z,[X|Xs]) ← arc(X,Y), path(Y,Z,Xs), good(X),
      goodlist(Xs).
```

Let $\mathtt{P_2} = \{\mathtt{c1}, \mathtt{c2}, \mathtt{c3}, \mathtt{c4}, \mathtt{c8}, \mathtt{c9}\}$. Now we have reached a crucial step in the transformation: According to Definition 3.6 we can *fold* `path(Y, Z, Xs)`, `goodlist(Xs)` in `c9`. The result is the following recursive clause:

```
c10:  goodpath(X,Z,[X|Xs]) ← arc(X,Y), good(X), goodpath(Y,Z,Xs).
```

Let $\mathtt{P_3} = \{\mathtt{c1}, \mathtt{c2}, \mathtt{c3}, \mathtt{c4}, \mathtt{c8}, \mathtt{c10}\}$. Notice that this definition is now directly recursive and it checks the “goodness” of the path while generating the path itself. □

A different definition of fold is given in [Mah87] and in [GS91]: Both cl, the folded clause, and d, the clause used for folding, are in P_i and they must be different clauses (hence conditions **F1** and **F2** must hold in P_i, while **C1** can be dropped).

This fold is normally regarded as weaker than the Tamaki-Sato's one, in fact it cannot produce all the same transformed programs, but its correctness wrt the least Herbrand model, the Success Set and the Computed Answers Substitutions is easier to prove. The disadvantage of this fold operation is that it does not allow us to introduce direct recursion in a definition (as done in Example 3.7), which is generally regarded as the key aspect of the folding operation.

We should mention another definition of fold, strictly stronger than Definition 3.6. It allows for simultaneous folding of different clauses and has been proposed in [PP94a] by extending the idea of fold as the inverse of unfold to the case when multiple unfoldings are possible.

3.2.1 Declarative Properties

Of course, it is of primary importance to ensure the correctness of an unfold/fold system from a declarative point of view. For the system presented here the following properties hold in case of definite programs:

- the least Herbrand Models of the initial and final programs coincide [TS84];
- the Success Sets of the initial and final programs coincide [KK90];
- the Computed Answers Substitutions of the initial and final programs coincide [KK90].

The first unfold/fold transformation system was later generalized to general logic programs, and proved correct wrt the well-founded semantics [Sek93]. Aravindan and Dung proved in [AD93] that it preserves also the so-called *semantic kernel*, that guarantees that the transformation is correct wrt a number of semantics for programs with negation.

On the other hand, the Finite Failure Set is not preserved. Consider the following example.

Example 3.8 Let $\mathtt{P}_0$ be the program

```
c1:   p ←q, h(X).
c2:   h(s(X)) ←h(X).
```

Where $\mathtt{P_{old}} = \{\mathtt{c}_2\}$ and $\mathtt{P_{new}} = \{\mathtt{c}_1\}$. Notice that there is no definition for predicate `q`, so `p` and `q` finitely fail. By unfolding `h(X)` in `c1` we obtain a variant of `c1`:

```
c3:   p ←q, h(Y).
```

Now, we can fold `q, h(Y)` in `c3`, using clause `c1` for folding. The result is

```
c4:   p ←p
```

The Finite Failure Set has changed: `p` does not finitely fail any longer. □

This problem was addressed and fixed by Seki, who in [Sek91] provides a *modified fold* operation for stratified general programs which requires that **C1** be modified into

(C2) either *cl* defines an old atom, or all the literals in the folded part of the clause *cl*, **B**, must result from a previous unfolding.

This restriction is sufficient to guarantee that the Finite Failure Set of the initial and of the final programs are the same. Seki also introduces *a labelling of literals in clause bodies in order to keep track of the ones coming from previous unfolding and to make syntactically checkable this condition.*

Termination Properties

We discuss now what happens when we are interested in preserving more procedural properties such as termination ones.

If we consider termination wrt *all* selection rules, then the system we present here is correct: In [BE94] we proved that if the initial program is (definite and) terminating for all selection rules, then the transformed program is terminating for all selection rules as well. This result extends also to general programs by considering the concept of *acyclic program*: If the initial program is acyclic then the resulting one is acyclic as well.

However, many usual programs are not terminating wrt all selection rules, and it is clearly of interest to see what happens for instance to *left termination* when we apply a fold-unfold transformation sequence. First of all note the obvious fact that *when we fix the selection rule, the order of literals in the bodies becomes relevant.* This is in contrast to the way the transformation rules were originally defined in [TS84, Sek91]. For instance in the last transformation step of Example 3.7 we have actually *swapped* the two atoms `path(Y,Z,Xs), good(X)` before applying the fold operation. Since Definition 3.6 is given modulo reordering of the body atoms, this does not pose any problem in applying it. On the other hand, if one fixes the selection rule, such a swapping can easily introduce non-termination: For instance think about swapping the two atoms `fail, loop` in a clause body. Thus, in order to apply the fold operation to Prolog or pure Prolog programs, one has to give a definition for it which takes into account the order of the literals in clause bodies.

A first relevant result in the direction of an unfold/fold transformation system which preserves left termination was presented by Proietti and Pettorossi in [PP91]. They propose a transformation system for definite programs which is similar to [TS84] with three additional conditions: (a) no reordering of the atoms is allowed, (b) unfolding is allowed only for the leftmost atom of a clause or in the case of a deterministic non-left propagating atom, and (c) folding is allowed if **C1** is modified as:

(C3) either *cl* defines an old atom, or the leftmost atom of the folded clause is the result of a previous unfolding. [1]

They proved that this system preserves a very strong semantics, namely, the *sequence of answer substitutions semantics* (a semantics for Prolog programs, defined in [JM84, Bau89]). This guarantees also that if the initial program is left terminating, then the resulting program is left terminating as well. While this system has rather restrictive applicability conditions (e.g. the transformation of Example 3.8 is not possible within it), we believe that in order to preserve such a strong semantics it is hardly possible to do any better.

In our works on unfold/fold transformations systems which preserve left termination [BC94, BC97, BCE96b, BCE00] we have explored two different approaches:

- In [BC94, BC97] we have considered the preservation of universal termination for definite programs, based on the (semantic) concept of *non-increasing operation.*
- In [BCE96b, BCE00] we show that the crucial aspect in preserving left termination is the reordering of the atoms in a clause and we provide – among other

[1] These are not *exactly* the conditions proposed in [PP91], but a conservative approximation of them in terms of the concepts we have introduced so far. We do this in order to avoid introducing too much notation and to make it easier to compare the different approaches.

things – novel applicability conditions for reordering which in some cases are purely syntactic, hence of practical nature. These will be discussed in Section 3.2.2.

In [BC94] we studied the preservation of universal termination for a query with *LD-resolution*. In order to capture computed answer substitutions plus universal termination, we defined an appropriate operational semantics for definite programs and split the equivalence condition to be satisfied between the original and the transformed program wrt a query into two complementary conditions: A "completeness" condition, which ensures that successful LD-derivations for the query are preserved, and the condition of being "non-increasing". This second condition is very operational since it compares the lengths of corresponding partial LD-derivations of the query in the initial and the transformed program. Its validity ensures that a transformation cannot introduce infinite derivations. We proved that, by appropriately restricting the version with no reordering of Tamaki-Sato's system, based on new definition, unfold and fold, the whole transformation sequence is non-increasing and then it preserves also universal termination for a query. As a consequence, left termination of programs is also preserved by such a restricted transformation sequence. The restriction we introduce into the unfold/fold transformation system imposes that fold can be performed only after a "decreasing" unfold of the clause to be folded. Namely let $cl : H \leftarrow \mathbf{J}, \mathbf{B}, \mathbf{K}.$ be the clause to be folded in $\mathbf{B}$, we require instead of **C1** the condition:

(C4) either *cl* defines an old atom, or at least one reachable atom in $\mathbf{J}, \mathbf{B}$ comes from a previous unfolding,

where an atom is called *reachable* when it has no finitely failing or diverging atom to its left in *cl*. Clearly condition **C4** is not decidable in general, even if there are sufficient conditions for it, for example condition **C3**.

Another related work is [Amt92], where Amtoft gives a unified treatment of conditions for preserving termination properties in transformation systems based on unfold and fold. He sets up a model, parametrized with respect to the evaluation order, which allows one to reason about termination in an algebraic fashion. In such a model he can represent most of the previous results in the literature as a special case: He can represent the condition of folding wrt a *new* predicate in [TS84, KK90], the condition **C2** of [Sek91], namely that all the atoms in the folded part of the body have to be labelled (i.e. they must result from a previous unfolding), or the weaker one, **C3**, in [PP91] for Prolog leftmost selection rule, namely that at least the leftmost atom in the clause to be folded is labelled.

3.2.2 Folding + Switching

We have already stressed that unfold/fold transformations might at some point require a reordering of body literals in order to perform a fold operation. Such a reordering can harm the termination of a program, and – because of this – we have seen that unfold/fold systems for (pure) Prolog programs do not allow for any permutation of literals in the clause's bodies. This restriction limits sensibly the effectiveness of a transformation system; to alleviate this problem, in [BCE96b, BCE00] we have addressed the problem of introducing in the transformation system a *switch* operation for reordering the body literals and of finding suitable applicability conditions for such operation in order to guarantee persistency of left termination: If the original program is left terminating, then the transformed program is left terminating as well. We now give a brief summary of those results, starting by the obvious definition of switch.

Definition 3.9 (Switch) Let $cl : H \leftarrow \mathbf{J}, A, B, \mathbf{K}$. be a clause of a program P. *switching* A *with* B *in* cl consists of replacing cl with $cl' : H \leftarrow \mathbf{J}, B, A, \mathbf{K}$. □

The switch operation can be seen as a replacement (discussed in the next Section) which *trivially maintains all the declarative properties of a program.* On the other hand, the *switch does not preserve left termination.* For instance if we take the contrived program

```
p ←q, p.
```

we have that at the moment the program is terminating (`q` fails), however, if we swap the two atoms in the body of the clause, we get a program which is not terminating. Another typical situation is the one in which we have in the body of a clause a combination such as `...p(X,Y), q(Y,Z)...` , where the rightmost atom uses `Y` as input variable; in this case, bringing `q(Y,Z)` to the left of `p(X,Y)` can easily introduce non-termination, as `q(Y,Z)` might be called with its arguments not sufficiently instantiated. In the context of an unfold/fold transformation system, this situation is further complicated by the presence of the other operations, in particular of fold which may introduce recursion and hence non-termination. Consider the following example.

Example 3.10 Let $\mathtt{P_0}$ be the following initial program.

```
c1:   z ←p, r.
c2:   p ←q, r.
c3:   q ←r, p.
```

Where $\mathtt{P_{new}} = \{\mathtt{c1}\}$ and $\mathtt{P_{old}} = \{\mathtt{c2}, \mathtt{c3}\}$. Notice that `r` is not defined anywhere, so everything fails and this program is left terminating. By unfolding `p` in `c1` we obtain the following clause:

```
c4:   z ←q, r, r.
```

By further unfolding `q` in `c4` we obtain:

```
c5:   z ←r, p, r, r.
```

Now we switch the first two atoms, obtaining:

```
c6:   z ←p, r, r, r.
```

Notice that this particular switch operation *does preserve left termination.* However, if we now fold the first two atoms, using clause `c1` for folding, we obtain the following:

```
c7:   z ←z, r, r.
```

which is not left terminating any longer. □

Here, we have a situation in which the switch operation does preserve left termination in a *local* way while left termination will subsequently be destroyed by the application of the fold operation. Notice also that such a fold operation satisfies any of the conditions **C1, ..., C4**. This shows that the switch operation requires applicability conditions which guarantee more than the termination properties of the actual program.

In [BCE96b] we propose a transformation system for definite programs based on unfold, fold and switch, which when applied to a left terminating moded program, yields a program which is left terminating as well. For this, we employ a new condition for the fold operation. Namely if $cl : H \leftarrow \mathbf{J}, \mathbf{B}, \mathbf{K}.$ is the clause to be folded in $\mathbf{B}$, instead of **C1**, we require that:

(C5) either *cl* defines an old atom, or one of the atoms in $\mathbf{J}$ or the leftmost in $\mathbf{B}$ comes from a previous unfolding.

(actually, this is not exactly the condition reported in [BCE96b], however, it is substantially equivalent to it, and this formulation allows us to compare it to the other ones reported here). This condition is clearly stronger than **C1** but weaker than **C3**. Then, the concept of transformation sequence is extended so that it includes the switch operation, for which specific applicability conditions are devised. The first applicability condition, introduced in [BCE96b] applies to moded programs and states that switching the atom A with B in the clause $cl : H \leftarrow \ldots, A, B, \ldots.$ is *allowed* if

(SW2) A is an *old* literal, $Var(Out(A)) \cap Var(In(B)) = \emptyset$, and A is *non-failing* in *cl*,

where *non-failing in cl* means that any instance of A, selected by the leftmost selection rule when *cl* is used in the resolution process, will eventually succeed[2]. The intuitive idea is that if A is non-failing, then it cannot "hide" any potential loops of the following atoms, hence we are allowed to move it to the right. A drawback is that the condition of being non-failing is generally non-computable, but for very particular classes of programs and queries [BC98], *noFD programs and queries*, which cannot have finitely failing LD-derivations, the non-failing property can be trivially guaranteed.

In [BCE00] we extend this transformation system for definite programs, by using the dual reasoning: If an atom B "never loops" then we should be able to move it *leftward.* This intuitive reasoning is not entirely true (the counterexample is still Example 3.10), however, it yields a new syntactic-based condition for guaranteeing the preservation of left termination, provided that the definition of B is never modified by the transformation. For this we need a new definition of initial program:

Definition 3.11 (Initial Program) We call a program P_0 an *initial program* if it can be partitioned into three programs P_{new}, P_{old} and P_{base}, such that the following conditions are satisfied:

(I1) $P_{new} \sqsupseteq (P_{old} \cup P_{base})$ and $P_{old} \sqsupseteq P_{base}$;

(I2) P_{new} is not recursive;

[2]Again, this is a conservative approximation of the more complex notion of non-failing used in [BCE96b].

(I3) all the literals in the bodies of the clauses of P_{old} are labelled "f", with the exception of literals defined in P_{base}; no other literal of the initial program is labelled.

We assume that the transformation does not affect the clauses in P_{base}. Then we obtain this new applicability condition for the switch operation: Switching the literal A with B in the clause $cl : H \leftarrow \ldots, A, B, \ldots$. is *allowed* if

(SW1) B is a *base* literal.

This condition allows for a complex theorem which has a number of different modular results on termination. To mention two of them which apply to definite programs: Let $P_0, \ldots, P_n$ be a transformation sequence in our system (where every fold satisfies **C5** and in which every switch operation is allowed) then we have that

- If P_0 is left terminating and P_{base} always left terminating, then P_n is also left terminating.
- If P_0 is well moded and left terminating and P_{base} well-terminating, then P_n is also left terminating.

Summarizing, in [BCE96a, BCE00] we propose a transformation system for definite programs with fold satisfying **C5** and a switch operation with special applicability conditions. Such system – intuitively speaking – maintains left termination together with the usual declarative properties.

Our system is appropriate for the paradigm *logic programming + leftmost selection rule.* At the same time it is not suitable for Prolog with built-ins: For instance it employs an unfolding operation which allows for left-propagation. Moreover *the switch operation can cause permutations in the sequence of answers substitutions*, which, as we have seen before, in the case of full Prolog can have serious consequences on the operational behaviour of the program.

3.2.3 Prolog and Partial Evaluation

In general, the term partial evaluation indicates a transformation system which does not include the fold operation. To this rule there are important exceptions. In the first place, the PADDY system [Pre92] employs a fold operation which is based on the system of Proietti-Pettorossi [PP91] (yet with a different unfolding). Sahlin's Mixtus [Sah93] is an automatic partial evaluator for *full* Prolog which incorporates a fold operation. The latter operation employs further restrictions, among which that the folded conjunction (**B** in Definition 3.6) must consist of only one atom. Sahlin states: "Our experience indicates that the most important class of programs to be partially evaluated, the interpreters, folding for composite goals does not seem to be required for getting satisfactory results".

Finally, we should mention the work on *conjunctive partial deduction.* [SGJ+99]. Partial deduction in its usual form (i.e. without a fold operation) cannot achieve certain optimizations which are possible by unfold/fold transformations. Conjunctive partial deduction is an extension of partial deduction which allows for optimizations which are typical of an unfold/fold system, for instance tupling and deforestation. Intuitively speaking, this is achieved by extending the paradigm to one in which a head of a clause might consist of a *conjunction* of atoms. In its pure form, conjunctive partial deduction is correct wrt the declarative semantics of a program (basically wrt the least Herbrand model and the Computed Answer Substitutions semantics, however Finite Failure and other declarative semantics can also be accommodated). Clearly, these semantics are independent of the selection rule.

The authors in [SGJ+99] address also the problem of conjunctive partial deduction in presence of a fixed left-to-right selection rule. Interestingly, the authors come to the conclusion that one should "limit the splitting to contiguous atoms only", otherwise one might degrade program's performances or even introduce nontermination. Without getting into the details of the splitting operation, we notice that this is very similar to *forbidding any switching* of two atoms.

4 An Extended System: Replacement

We can obtain a much more powerful transformation system by adding to unfold and fold a further transformation operation: The replacement. Replacement allows one to substitute a sequence of literals in a clause body by an "equivalent" sequence of literals. What "equivalent" means depends on the chosen observables. We give here a very general definition.

Definition 4.1 (Replacement) Let $\mathbf{B}'$ be a sequence of literals defined in P, $c : H \leftarrow \mathbf{A}, \mathbf{B}, \mathbf{C}.$ be a clause in P and let $c' : H \leftarrow \mathbf{A}, \mathbf{B}', \mathbf{C}.$
Let X be the set of *common variables* and Y be the set of *private variables in* $\mathbf{B}$ *and* $\mathbf{B}'$, namely $X = Var(\mathbf{B}) \cap Var(\mathbf{B}')$ and $Y = Var(\mathbf{B}, \mathbf{B}') \setminus X$.
Replacing $\mathbf{B}$ *by* $\mathbf{B}'$ *in* c consists in replacing c by c' if

(R1) *the variables in* Y *are local wrt* c *and* c', that is $Var(H, \mathbf{A}, \mathbf{C}) \cap Y = \emptyset$;

(R2) $\mathbf{B}$ *and* $\mathbf{B}'$ *are equivalent wrt the chosen semantics*, that is (with an extended notation on existential quantifiers) $\exists Y \mathbf{B} \equiv_S \exists Y \mathbf{B}'$, where S is a specified semantics.

Replacement may be used for applying algebraic laws as shown in the next example.

Example 4.2 Let a program P contain the clause

c: `p(`l_1, l_2, l_3`,Z)` $\leftarrow$ `app(`l_1`,` l_2`, Y1), app(Y1,` l_3`, Z).`

where *app* is the usual append predicate and l_1, l_2, l_3 are lists of fixed length. Let $\mathbf{B} = app(l_1, l_2, Y1), app(Y1, l_3, Z)$ and $\mathbf{B}' = app(l_2, l_3, Y2), app(l_1, Y2, Z)$. Replacing $\mathbf{B}$ by $\mathbf{B}'$ in c satisfies both the syntactic condition and the equivalence one. In fact this replacement corresponds to applying the *associative property* of append.

Replacement can also be used to eliminate or add literals to a clause body, what is called respectively *thin* and *fatten* operation in [BC93].

Clearly a transformation system including replacement has a much greater power and it allows for a deep restructuring of the initial program. A typical use of replacement is transforming non-linear recursive predicates into linear ones by introducing accumulators or difference-lists. It is also clear that such transformations are less automatizable and require more guidance from the programmer.

Notice also that fold can be considered as a special case of replacement. A transformation system containing replacement could then drop the fold operation. This was actually done by Cook and Gallagher in [CG94]. In fact they propose an elegant transformation system for definite programs based on two basic operations only: A particular form of unfold that we already described in Section 3.1 and replacement.

The basic difference between folding and replacement is that in the first one there has to be a *folding clause*, which makes the operation of “syntactic nature”, on the other hand, the replacement allows one to exchange any two sequences of literals, provided he can prove their equivalence. It is then a very general operation, whose applicability is typically undecidable.

4.0.4 Declarative Properties

The first requirement in Definition 4.1 is syntactic correctness, namely private variables Y must not produce different bindings of $\mathbf{B}$ and $\mathbf{B}'$ with their contexts c and c'. The second requirement imposes the equivalence of $\mathbf{B}$ and $\mathbf{B}'$ wrt common variables in the chosen semantics. Many different instances of this second condition can be found in the literature.

- [GS91] requires that for all grounding θ, $P \models \mathbf{B}\theta$ implies $P \setminus \{c\} \models \mathbf{B}'\theta'$ and vice-versa that for all grounding θ, $P \models \mathbf{B}'\theta$ implies $P \setminus \{c\} \models \mathbf{B}\theta'$, where θ and θ' coincide on the variables occurring in $H, \mathbf{A}, \mathbf{C}$. This guarantees the preservation of the least Herbrand Model, M_P. They study replacement also in the context of general programs.

- In [Mah87], given that no predicate in $\mathbf{B}$ and $\mathbf{B}'$ depends on $Pred(H)$, the equivalence must be provable in *comp*(P); then it guarantees the preservation of the Success Set and of the Ground Finite Failure set in definite programs.

- In [BCE96a] a simultaneous replacement of many sequences of literals in many clauses is defined. The (rather complex) condition to be satisfied allows for dependencies between replaced sequences of literals and modified clauses and still it guarantees to preserve the Fitting’s and Kunen’s semantics of general programs.

- In [PP94b] the equivalence condition is parametric wrt the semantics and it depends on $P \setminus \{c\}$. In order to prove it, Pettorossi and Proietti propose unfold/fold proofs for definite and gneral programs which are naturally parametric wrt the semantics.

Termination Properties

Let us consider now more procedural properties such as termination ones. First notice that when we consider control issues, the order of literals in the bodies becomes relevant also for replacement. Note also that replacement itself can be used for reordering literals.

As previously mentioned, Cook and Gallagher in [CG94] define a transformation system for definite programs based only on a particular unfold and replacement. Such replacement depends on (semantic equivalence +) termination analysis, namely it requires a property of termination (or left termination if we consider the Prolog leftmost selection rule) which must hold on the resulting program. This condition ensures that the system preserves the Success Set. If a similar termination property is required also on the program to be transformed, then the system preserves also the Finite Failure Set. The authors suggest to check such termination properties a posteriori, by means of any known technique for verifying termination properties, but they also claim that one could devise sufficient conditions for the applicability of such replacement.

In [BC97] we extend our simple unfold/fold system for definite programs presented in [BC94], which preserves universal termination of a query with *LD-resolution*. For reordering atoms in the bodies, we introduce also a replacement operation. In order to guarantee the non-increasing property also for replacement, besides conditions **R1** and **R2** (referred to the semantics given by Computed Answer Substitutions plus universal termination for a query), we impose a further restriction on replacement, namely that

(R3) $\mathbf{B}'$ *is non-increasing in* c *wrt* $\mathbf{B}$ *in* P.

This basically means that for any substitution θ, instantiating only common variables, any partial LD-derivation of $\mathbf{B}'\theta$ is not longer than a corresponding one of $\mathbf{B}\theta$.

We also study how typing information and the well-typing property can simplify the verification of such applicability conditions for replacement. In fact the major problem is how to verify in practice such applicability conditions for preserving universal termination since they are semantic conditions and operational in style, not decidable in general.

5 Conclusions

Virtuous programming methodology which consists in focusing on correctness of programs at first and on their efficiency only afterwards, fits particularly well with logic programming [Kow79, Dev90]. This encourages the application of transformation systems to logic programs both for synthesizing a correct program from a logic specification and for optimizing it.

The main requirements for a practical transformation systems are on one hand to guarantee the preservation of interesting program properties and on the other hand to be supported by an automatic or semi-automatic tool. Among interesting properties the most basic are captured by declarative semantics, but nondeclarative properties are also extremely relevant, such as the termination of the program.

In this paper we give a short description of the systems proposed for logic program transformation. In particular, we focus on systems able to preserve termination and other nondeclarative properties in Prolog programs.

We consider at first simple unfold/fold systems and then more powerful ones including the replacement or at least the switch operation. Transformation systems can include other basic transformation operations, either obtainable through a combination of the previous ones or completely independent from them. Since any transformation system proposes his own set of basic operations, we decided to restrict our comparison only to the main ones: New definition, unfold, fold and replacement. Such set of operations gives rise to systems which are powerful enough for dealing with most applications of program transformation.

Since we focus on nondeclarative properties of Prolog programs, we have not given a detailed account of the various results on declarative semantic, neither we consider transformation systems dealing with extended logic programming paradigms, such as CLP, or with modified interpreters. For a rather complete panorama of recent proposals in the field of logic programs transformations the LOPSTR proceedings are a good reference [LOP].

References

[AB91] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3&4):335–363, 1991.

[AD93] C. Aravidan and P. M. Dung. On the correctness of Unfold/Fold transformation of normal and extended logic programs. Technical report, Division of Computer Science, Asian Institute of Technology, Bangkok, Thailand, April 1993.

[AD94] C. Aravidan and P. M. Dung. Partial Deduction of Logic Programs w.r.t. Well-Founded Semantics. *New Generation Computing*, 13:45–74, 1994.

[Amt92] T. Amtoft. Unfold/fold transformations preserving termination properties. In *Proceedings PLILP'92*, number 631 in Lecture Notes in Computer Science, pages 187–201. Springer-Verlag, 1992.

[AP90] K. R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In J.W. Lloyd, editor, *Symposium on Computional Logic*, pages 150–176, Berlin, 1990. Springer-Verlag.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[Apt97] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[Bau89] M. Baudinet. *Logic Programming Semantics: Techniques and Applications.* PhD thesis, Stanford University, Stanford, California, 1989.

[BC93] 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.

[BC94] A. Bossi and N. Cocco. Preserving universal termination trough unfold/fold. In G. Levi and M. Rodríguez-Artalejo, editors, *Proc. Fourth Int'l Conf. on Algebraic and Logic Programming*, volume 850 of *Lecture Notes in Computer Science*, pages 269–286. Springer-Verlag, Berlin, 1994.

[BC97] A. Bossi and N. Cocco. Replacement can preserve termination. In J. Gallagher, editor, *Proc. Sixth Workshop on Logic Program Synthesis and Transformation*, volume 1207 of *Lecture Notes in Computer Science*, pages 104–129. Springer-Verlag, Berlin, 1997.

[BC98] A. Bossi and N. Cocco. Programs without failures. In R. Fuchs, editor, *Proc. Seventh Workshop on Logic Program Synthesis and Transformation*, volume 1463 of *Lecture Notes in Computer Science*, pages 28–48. Springer-Verlag, Berlin, 1998.

[BCE92] A. Bossi, N. Cocco, and S. Etalle. Transforming Normal Programs by Replacement. In A. Pettorossi, editor, *Meta Programming in Logic - Proceedings META'92*, volume 649 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, Berlin, 1992.

[BCE96a] A. Bossi, N. Cocco, and S. Etalle. Simultaneous replacement in normal programs. *Journal of Logic and Computation*, 6(1):79–120, February 1996.

[BCE96b] A. Bossi, N. Cocco, and S. Etalle. Transformation of Left Terminating Programs: the Reordering Problem. In M. Proietti, editor, *LOPSTR95 – Fifth International Workshop on Logic Program Synthesis and Transformation*, number 1048 in LNCS, pages 33–45. Springer-Verlag, 1996.

[BCE00] A. Bossi, N. Cocco, and S. Etalle. Transformation of Left Terminating Programs. In A. Bossi, editor, *Ninth International Workshop on Logic Program Synthesis and Transformation*, number 1817 in LNCS, pages 156–175. Springer-Verlag, 2000.

[BCER01] A. Bossi, N. Cocco, S. Etalle, and S. Rossi. Termination in a hierarchy of general logic programs. Technical Report CS-2001-05, Dipartimento di Informatica, Università Ca' Foscari Di Venezia, Italy, March 2001.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BE94] A. Bossi and S. Etalle. Transforming Acyclic Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1081–1096, July 1994.

[BR89] M. Bugliesi and F. Rossi. Partial evaluation in Prolog: Some improvements about cut. In E.L. Lusk and R.A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference 1989, Cleveland, Ohio, October 1989*, pages 645–660. MIT Press, 1989.

[Cav89] L. Cavedon. Continuity, consistency and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *6 International Conference on Logic Programming*, pages 571–584. MIT press, 1989.

[CG94] J. Cook and J.P. Gallagher. A transformation system for definite programs based on termination analysis. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*, pages 51–68. Springer-Verlag, 1994.

[Cla78] K. L. Clark. Negation as failure rule. In H. Gallaire and G. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[Dev90] Y. Deville. *Logic Programming. Systematic Program Development.* Addison-Wesley, 1990.

[DM88] W. Drabent and J. Maluszynski. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

[EBC99] S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.

[FLMP93] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[Gal91] J. P. Gallagher. A system for specializing logic programs. Technical Report 91-32, University of Bristol, 1991.

[GS91] P.A. Gardner and J.C. Shepherdson. Unfold/fold transformations of logic programs. In J-L Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.

[JM84] N. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 281–288, 1984.

[KK90] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. *Theoretical Computer Science*, 75(1&2):139–156, 1990.

[Kow79] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.

[LMS98] M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalisation and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):208–258, 1998.

[LOP] International Workshops on Logic Program Synthesis and Transformation. http://www.cs.man.ac.uk/ kung-kiu/lopstr/.

[LS88] G. Levi and G. Sardu. Partial evaluation of metaprograms in a multiple worlds logic language. *New Generation Computing*, 6(2,3):227–247, 1988.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[Mah87] M.J. Maher. Correctness of a logic program transformation system. IBM Research Report RC13496, T.J. Watson Research Center, 1987.

[PP91] M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91), New Haven, CT (U.S.A.) (SIGPLAN NOTICES, Vol.26 (9))*, pages 274–284. ACM press, 1991.

[PP94a] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19(20):261–320, 1994.

[PP94b] M. Proietti and A. Pettorossi. Total correctness of a goal replacement rule based of the unfold/fold proof method. In M. Alpuente, editor, *Proc. 1994 Joint Conference on Declarative Programming GULP-PRODE'94*, pages 347–358. Springer-Verlag, 1994.

[Pre92] S. Prestwich. The PADDY partial deduction system. Technical Report 92-6, ECRC GmbH, Munich, Germany, 1992.

[Pre93] S. Prestwich. An Unfold Rule for full Prolog. In K.K. Lau and T. Clement, editors, *Proceedings LOPSTR'92*, Workshops in Computing, pages 199–213. Springer-Verlag, Berlin, 1993.

[Sah93] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, (12):7–51, 1993.

[Sek91] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86(1):107–139, 1991.

[Sek93] H. Seki. Unfold/fold transformation of general logic programs for the Well-Founded semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.

[SGJ$^+$99] D. De Schreye, R. Glück, Jesper 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.

[ST84] T. Sato and H. Tamaki. Transformational logic program systhesis. In *International Conference on Fifth Generation Computer Systems, Tokyo, Japan, November 1984*, pages 195–201. ICOT, 1984.

[TS84] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.

[VP86] T. Vasak and J. Potter. Characterization of Terminating Logic Programs. In *Proc. Third IEEE Int'l Symp. on Logic Programming*, pages 140–147. IEEE Comp. Soc. Press, 1986.