# Unfold/fold Transformations Preserving Termination Properties

Torben Amtoft *

Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Århus C, Denmark

**Abstract.** The unfold/fold framework constitutes the spine of many program transformation strategies. However, by unrestricted use of folding the target program may terminate less often than the source program. Several authors have investigated the problem of setting up conditions of syntactic nature, i.e. not based on some well-founded ordering of the arguments, which guarantee preservation of termination properties. These conditions are typically formulated in a way which makes it hard to grasp the basic intuition why they work, and in a way which makes it hard to give elegant proofs of correctness. The aim of this paper will be to give a more unified treatment by setting up a model which enables us to reason about termination preservation in a cleaner and more algebraic fashion. The model resembles a logic language and is parametrized with respect to evaluation order, but it should not be too difficult to transfer the ideas to other languages.

## 1 Introduction

The unfold/fold framework for program transformation dates back to (at least) [BD77] and has since been the subject of much interest, primarily aimed at making the process of finding "eureka"-definitions more systematic, e.g. [Wad90], [NN90], [PP91b]. Also supercompilation [Tur86] can be seen as a variant over the concept.

A major problem with the technique is that one, due to "too much folding", may risk that the program resulting from transformation (the *target program*) loops while the original (the *source program*) does not. A classical example is the following, expressed in a logic language:

*Example 1.* Suppose we have a source program $S_1$ containing the clauses

$p(X) \leftarrow q(X)$; $q(a) \leftarrow \Box$

Here the query $p(X)$ will succeed with answer substitution $\{X \rightarrow a\}$. However, if one *folds* the first clause of the program against itself, the target program $T_1$ will contain the clause

$p(X) \leftarrow p(X)$

and now the query $p(X)$ will loop (i.e. neither succeed or fail). □

* internet: tamtoft@daimi.aau.dk

By innocent abuse of terminology, we will say that a transformation is partially correct iff each time the target program terminates with some result also the source program terminates and with the same result; whereas we will define total correctness to mean partial correctness together with the condition that if the target program does not terminate then neither does the source program. Whether a transformation process itself terminates is beyond the scope of this paper, but e.g. [Wad90], [PP91b] address this problem for certain transformation strategies. For the related technique of *partial evaluation* (cf. [JSS89], [BD91]), the problem of ensuring termination of the partial evaluation process is addressed in [Hol91]. Upper bounds for the speed-up possible by applying unfold/fold transformations have been given in [Han91], [Hon91].

Several ways to guarantee total correctness have been proposed in the literature, e.g. [TS84], [KK90], [Sek91], [Kot85], [GS91], [PP91a]. They all work by putting forward some restrictions on the types of foldings allowed. For a more detailed description (and comparison with our approach), see Sect. 5.

The purpose of this paper is to present a model for unfold/fold-transformations which enables one to express conditions which are provably sufficient for total correctness. We want the model to include (most of) the results from the literature as special cases (after the frameworks in question have been encoded into our framework); and we want the model to have a clean algebraic structure.

Lack of space prevents us from giving detailed descriptions of how to translate other formalisms into our model - however, we hope our examples will provide the reader with sufficient intuition.

Our framework is primarily aimed at modeling logic programming - even though the machinery differs from the one usually used when treating logic languages, as done in e.g. [Llo84], [Søn89]. However, we believe that the main ideas can be carried over to other types of languages as well.

### 1.1 Two-level Transitions

The meaning of programs will be defined in terms of a transition semantics (cf. [Plo81]). The reason for this is that we feel this is more appropriate for capturing the essence of unfolding and folding: unfolding corresponds to a transition being made in the "right" direction; folding corresponds to a transition being made in the "wrong" direction. By using a denotational approach, this cannot be expressed directly. We believe that the reason why conditions for unfold/fold transformations to be termination preserving apparently is a more hot topic in the logic programming community than in the functional community is that in the former operational semantics (typically derivation trees) has a more respectable status than in the latter.

**Main Idea.** We will now, in a *very informal and simplified* way, present the basic intuition underlying this work. Execution of the source program on a goal configuration $C_1$ corresponds to a transition sequence

$$C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow \ldots$$

where each $C_i \rightarrow C_{i+1}$ corresponds to an atom in $C_i$ being unfolded. For the transition system in question, a "Church-Rosser property" says that if there is a transition

from $C$ to $C_1$ using $n_1$ inference steps (i.e. unfoldings), and also a transition from $C$ to $C_2$ using $n_2$ inference steps, then there exists a configuration $C'$ and numbers $n_1'$ and $n_2'$ such that there is a transition from $C_1$ to $C'$ using $n_1'$ inference steps, and a transition from $C_2$ to $C'$ using $n_2'$ inference steps. Moreover, it will hold that $n_1 + n_1' = n_2 + n_2'$ [2]

A clause in the target program takes the form

$$C_1 \xrightarrow{u} C' \xleftarrow{f} C_2$$

This means that first we have unfolded $C_1$ into $C'$ (using $u$ inference steps), and then folded back into $C_2$ - or equivalently unfolded $C_2$ into $C'$, using $f$ inference steps.

Now suppose the target program loops. This means that there exists an infinite sequence $C_1 \ldots C_i \ldots$ where each step from $C_i$ to $C_{i+1}$ represents an application of a clause in the target program, i.e. for all $i$ there exists $C_i'$, $u_i$, $f_i$ such that

$$C_i \xrightarrow{u_i} C_i' \xleftarrow{f_i} C_{i+1}$$

In order for the transformation in question to be termination preserving, we must demand the source program to loop on $C_1$ as well. Our task is to find sufficient conditions for this. First we can draw the Church-Rosser completion of the situation, as sketched in Fig. 1. Now, if for all $i$ we have $u_i > f_i$ then the path

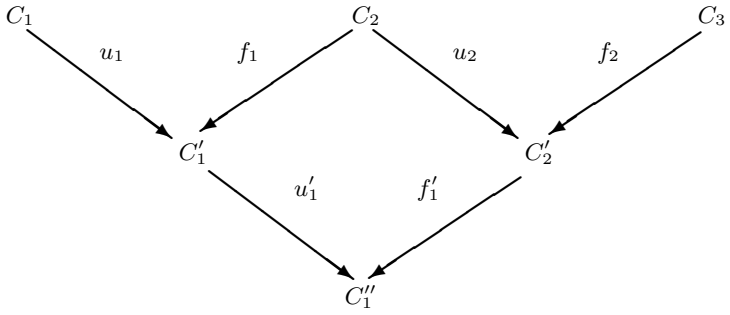


**Fig. 1.** A Church-Rosser completion.

$$C_1 \to C_1' \to C_1'' \to \ldots$$

will represent infinitely many inference steps - too see this, note that e.g.

$$\begin{aligned} u_1 + u_1' &> f_1 + u_1' = u_2 + f_1' \\ &> f_2 + f_1' \end{aligned}$$

[2] In order for such a simple relation to hold one must, of course, assume that no copying or destruction of "redices" takes place.

Observe that the $u_i$'s and $f_i$'s do not necessarily have to denote the total number of inference steps. By e.g. letting them denote only the number of steps where certain predicates are unfolded, the same reasoning applies. More generally, one can - as will be done in this paper - assign a *weight* to each predicate symbol.

Of course, several features are not accounted for in the treatment above - we will repair on this in the following.

**Configurations.** A *basic configuration* is a sequence of goals together with some information about which values the variables in the goals can assume. One usually represents this information as a substitution, cf. [Llo84]. As substitutions are hard to reason about from an algebraic point of view (even though e.g. [Søn89] and [Pal89] show that certain sets of substitutions carry some structure), we will represent the information as a family of sets of ground values. As an example take the goal sequence *(p(X),q(Y),r(Z))* together with the substitution $\{X \rightarrow f(Y), Z \rightarrow a\}$ Assuming that $\mathcal{D}$ is a universal data domain, this in our framework could be represented as the goal sequence *(p,q,r)* together with the $\mathcal{D}$-indexed family where the $d$'th element is the singleton set $\{(f(d), d, a)\}$ - on the other hand, one might also use the family consisting of one element only, namely the set $\{(f(d), d, a) | d \in \mathcal{D}\}$. The latter representation will be needed for dealing with "variables occurring on the right hand side but not on the left hand side"; we will elaborate on this issue later in this section.

A *configuration* is a multiset of basic configurations (whose "information families" are indexed over the same set). A configuration can be thought of as representing the current frontier of the search tree (we need multiset instead of just set in order to model multiple identical solutions).

A basic configuration is *failure* if its information family consists of empty sets only. A non-failure basic configuration is *solved* if the goal sequence is empty. A configuration where all basic configurations are non-failure is said to be *pruned*. A pruned configuration where all basic configurations are solved is said to be in *normal form*. Given a configuration $C$, we define $\mathsf{P}(C)$ as the configuration obtained by removing all failure basic configurations from $C$. If $\mathsf{P}(C) = \emptyset$ we say $C$ is failure.

We define some operations on configurations:

**+:** $C_1$+$C_2$ is just the union (taken as multisets) of $C_1$ and $C_2$ (only defined if the information families are indexed over the same set). Intuitively, this models "or".

**&:** Intuitively, this models "and". On basic configurations, this is defined as follows: let $B_i$, $i = 1, 2$, consist of goal sequence $H_i$ and an information family $Q_i$ indexed over $K_i$. Now $B_1 \& B_2$ consists of the goal sequence $H_1 H_2$ together with an information family $Q$ indexed over $K_1 \times K_2$, where $Q$ is given by

$$Q(k_1, k_2) = \{d_1 \times d_2 | d_1 \in Q(k_1), d_2 \in Q(k_2)\}$$

The extension to (non-basic) configurations is determined by the desire to make & distributive over +.

**$\mathcal{U}$_(_):** Intuitively, this models instantiation. Let $B$ be a basic configuration with goal sequence $H$ and an information family $Q$ indexed over $K$. Let $s$ be a mapping

from some other index set $K'$ into $\mathcal{P}(K)$. Then $\mathcal{U}_s(B)$ is the goal sequence $H$ together with the $K'$-indexed information family $Q'$, where

$$Q'(k') = \bigcup_{k \in s(k')} Q(k)$$

As an example of how these operations work together, consider how the goal sequence *p(X),q(X)* will be represented in our framework. Define $B_1$ as the basic configuration consisting of goal sequence $p$ and $\mathcal{D}$-indexed information family $Q_1$ given by $Q_1(d) = \{d\}$; and let $B_2$ consist of goal sequence $q$ and (again) $Q_1$. Now $B = B_1 \& B_2$ will consist of goal sequence *(p,q)* and $\mathcal{D} \times \mathcal{D}$-indexed family $Q$, where $Q(d_1, d_2) = \{(d_1, d_2)\}$. Define $s : \mathcal{D} \to \mathcal{D} \times \mathcal{D}$ by $s(d) = (d, d)$. Then $B' = \mathcal{U}_s(B)$ will contain goal sequence *(p,q)* and $\mathcal{D}$-information family $Q'$, where

$$Q'(d) = \bigcup_{(d_1,d_2) \in s(d)} Q(d_1, d_2) = Q(d, d) = \{(d, d)\}$$

enabling us to conclude that *p(X),q(X)* can be represented as $\mathcal{U}_s(B_1 \& B_2)$.

**Transitions.** If there is a transition between configurations $C$ and $C'$, the index sets of the information families of $C$ and $C'$ must be isomorphic. There exist some operations on transitions:

**+:** If $t_1$ is a transition from $C_1$ to $C_1'$, and if $t_2$ is a transition from $C_2$ to $C_2'$, then $t_1 + t_2$ will be a transition from $C_1 + C_2$ to $C_1' + C_2'$. This models "or-parallelism", cf. [Gre87].

**&:** If $t_1$ is a transition from $C_1$ to $C_1'$, and if $t_2$ is a transition from $C_2$ to $C_2'$, then $t_1 \& t_2$ will be a transition from $C_1 \& C_2$ to $C_1' \& C_2'$. This models "and-parallelism".

**$\mathcal{U}_{\_}(\_)$:** If $t$ is a transition from $C_1$ to $C_2$, and $s$ is a function (having an appropriate functionality), $\mathcal{U}_s(t)$ will be a transition from $\mathcal{U}_s(C_1)$ to $\mathcal{U}_s(C_2)$.

**$\star$:** If $t_1$ is a transition from $C_1$ to $C_2$, and $t_2$ is a transition from $C_2$ to $C_3$, then $t_1 \star t_2$ is a transition from $C_1$ to $C_3$. This modeles sequential execution.

**$Id\_$:** For all configurations $C$, there exists an identity transition $Id_C$ from $C$ to $C$.

**$\mathsf{P}(\_)$:** If $t$ is a transition from $C_1$ to $C_2$, then $\mathsf{P}(t)$ (the pruning of $t$) is a transition from $\mathsf{P}(C_1)$ to $\mathsf{P}(C_2)$.

Not all transitions are valid. We have a hierarchy as follows:

1. The source program is represented as *rules at level 0*. For instance, if $p$ is defined by the clauses

   *p(a) ←□ and p(X) ←q(f(X,Y))*

   then this will be represented as a transition from the configuration *[p],*$\{\{d\} | d \in \mathcal{D}\}$ to the configuration consisting of the basic configurations $B_1$ and $B_2$, where $B_1 = ([], Q_1)$, $B_2 = ([q], Q_2)$ and where the information families $Q_1$ and $Q_2$, both indexed by $\mathcal{D}$, are defined by $Q_1(a) = \{()\}$, $Q_1(d) = \emptyset$ for $d \neq a$, $Q_2(d) = \{f(d, d') | d' \in \mathcal{D}\}$.

2. As soon as the rules at level 0 have been given, the *valid transitions at level 1* (also called the 1-valid transitions) and the *t-valid transitions at level 1* (also called the 1t-valid transitions) will be fixed. The intuition behind the 1-valid transitions is that they should model standard execution of the source program; therefore the set of 1-valid transitions is defined to be the closure of the set of level 0 rules wrt. the operations $+$, $\&$, $\mathcal{U}\_(\_)$, $\star$, $Id\_$ and $\mathsf{P}(\_)$.
   The intuition behind the 1t-valid transitions is that they should model transformation ("symbolic evaluation") of the source program, thus the set of 1t-valid transitions will be the closure (wrt. the operations $+$, $\&$, $\mathcal{U}\_(\_)$, $\star$, $Id\_$, $\mathsf{P}(\_)$) of the set consisting of the level 0 rules *together with* "legal reversals" of those. What constitutes a legal reversal will be defined in the following.
   If there is a 1-valid transition from $C$ to $D$ we write $1 \vdash_u C \to D$; if there is a 1t-valid transition from $C$ to $D$ we write $1 \vdash C \to D$. One can also define the closure of the set of legal reversals of level 0 rules (without the level 0 rules themselves); if there is a transition in this set from $C$ to $D$ we write $1 \vdash_f C \to D$.
3. Among all the 1t-valid transitions, some are chosen to be *rules at level 1*. These rules can be interpreted as the target program.
4. As soon as the rules at level 1 have been given, the *valid transitions at level 2* (also called the 2-valid transitions) will be fixed as the closure of the level 1 rules (wrt. the operations $+$, $\&$, $\mathcal{U}\_(\_)$, $\star$, $Id\_$, $\mathsf{P}(\_)$). Intuitively, 2-valid transitions model (standard) execution of the target program. If there is a 2-valid transition from $C$ to $D$, we write $2 \vdash C \to D$.

A key point of our approach is that "standard evaluation" (the 1-valid transitions) is a special case of "symbolic evaluation" (the 1t-valid transitions) - this greatly facilitates reasoning about the properties of the target program. This lack of distinction between standard evaluation and symbolic evaluation comes almost for free in a logic language, but also in the functional world one gains from viewing the latter as a generalization of the former [DP88]. However, an important difference between standard and symbolic evaluation is that during symbolic evaluation any atom in the goal sequence may be unfolded, whereas during standard evaluation one for efficiency reasons often chooses a fixed strategy, typically the strategy always to unfold the leftmost goal - this strategy will be denoted $\mathcal{LR}$.

**Legal Reversals.** We are left with the question of what constitutes a legal reversal of a level 0 rule. Let the level 0 rule (defining $p$) be a transition from basic configuration $B = ([p], \{\{d\} | d \in \mathcal{D}\})$ to configuration $B_1 + \ldots + B_n$. Now suppose $s(p, i)$ (a function of $p$ and $i$) is such that

- the information family of $\mathcal{U}_{s(p,i)}(B_i)$ consists of non-empty sets only
- $\mathcal{U}_{s(p,i)}(B'_i)$ is failure for all $i' \neq i$.

Then there will be a transition (a legal reversal) from $\mathcal{U}_{s(p,i)}(B_i)$ to $\mathcal{U}_{s(p,i)}(B)$.
Some noteworthy points:

- To motivate why we demand the information family of $\mathcal{U}_{s(p,i)}(B_i)$ to consist of non-empty sets only, observe that otherwise there might be a fold transition from a failure configuration into a non-failure configuration, something which

would render our theorems about termination preservation void: referring back to Fig. 1, this corresponds to e.g. $C_1''$ being failure while $C_3$ not. In the standard framework, the condition can be explained as follows: if there is a program clause $p(a) \leftarrow q(a)$ and one encounters the goal $q(a)$, one should fold it back into $p(a)$ and not into $p(X)$ - even though unifying the goal $p(X)$ against the above clause gives us the goal $q(a)$ as result.

- To motivate why we demand that $\mathcal{U}_{s(p,i)}(B_i')$ is failure for all $i' \neq i$, observe that otherwise we would have a 1t-valid transition from $\mathcal{U}_{s(p,i)}(B_i)$ to $\mathcal{U}_{s(p,i)}(B_i) + \mathcal{U}_{s(p,i)}(B_i')$ the latter basic configuration *not* being failure. In the standard framework, this is modeled by the requirement that only one clause defining the predicate folded against should match.
- Recall that e.g. the clause $p(X) \leftarrow q(f(X,Y))$ is translated into a transition from $([p], \{\{d\} | d \in \mathcal{D}\}$ to $([q], Q)$, where $Q(d) = \{f(d,d') | d' \in \mathcal{D}\}$. Thus $Q$ does *not* consist of singletons only, and hence - switching back and forth between our framework and the standard framework - it will be impossible to instantiate $Y$ by means of the $\mathcal{U}_{\_}(\_)$-operation. This gives a nice solution to the problems arising when folding against a clause containing variables not occurring in the head - a problem to which some incorrect solutions have been proposed in the literature (and yet proved correct!), for a survey see [GS91].

### 1.2 An Overview of This Paper

In Sect. 2 we give a semantics for the language sketched above and state some of its properties. In Sect. 3 we state some conditions ensuring preservation of semantics. These conditions will be "global", i.e. not relate to a single rule at level 1. In Sect. 4 we for each such global condition states a local condition, i.e. a condition concerning the form of a level 1-rule, which implies the global condition (and hence termination preservation). In Sect. 5 we discuss other approaches to the problem of ensuring total correctness and relate them to our model. Section 6 concludes. Some important theorems will, due to lack of space, be given without proof (or only a loose sketch of such); these will appear in [Amt92].

## 2 Two-level Semantics

First the Church-Rosser property can be formulated:

**Theorem 1.** *If* $1 \vdash_u C \to C_1$ *and* $1 \vdash_u C \to C_2$, *with* $C_1$ *and* $C_2$ *pruned, then there exists* $D$ *such that* $1 \vdash_u C_1 \to D$ *and* $1 \vdash_u C_2 \to D$.

This proposition only holds because we imposed no ordering on the elements of a configuration, which thus is a *multiset* and not a *sequence* of basic configurations. In [PP91a] one wants to model the fact the standard PROLOG explores the branches in sequential order, and therefore unfolding of the *leftmost* atom only is allowed (unless extra conditions are satisfied).

Suppose we have a function $\mathcal{E}$ which from a *basic* configuration $B$ and a configuration $C$ in normal form (where there is a transition from $B$ to $C$) extracts the "final result" $\mathcal{E}(B,C)$. $\mathcal{E}(B,C)$ will correspond to the multiset of answer substitutions. For

instance, if $C$ is a singleton and contains a $\mathcal{D}$-indexed information family $Q$, then we would naturally define

$$\mathcal{E}(B, C) = \{d | Q(d) \neq \emptyset\}$$

Let $\mathcal{V}$ be the codomain of $\mathcal{E}$, extended with a bottom element $\perp$ such that for all $v \in \mathcal{V}$, $\perp \leq v$ - thus making $\mathcal{V}$ a flat domain.

**Definition 2.** If there exists a pruned $C$ in normal form such that there is a 1-valid transition from $B$ to $C$, then $[\![\mathrm{B}]\!]_1 = \mathcal{E}(B, C)$. If there exists no such $C$, $[\![\mathrm{B}]\!]_1 = \perp$.

Exploiting Theorem 1, one can easily check that this is well-defined. Notice that a program which e.g. returns an answer and then loops is identified with a program which loops without producing any answer. It should not be too hard to modify the semantics so it distinguishes between such programs.

The following theorem comes almost immediately from the definitions:

**Theorem 3.** *If* $2 \vdash C \to D$*, then also* $1 \vdash C \to D$*.*

Next comes a most crucial theorem, stating that an arbitrarily mixed sequence of unfoldings and foldings is equivalent to a sequence of unfoldings followed by a sequence of foldings - this bears some similarities to proof normalization [GLT89].

**Theorem 4.** *If* $1 \vdash C \to D$ *with* $C$*,*$D$ *pruned, there exists* $E$ *such that* $1 \vdash_u C \to E$*,* $1 \vdash_f E \to D$ *(and then also* $1 \vdash_u D \to E$*).*

*Proof.* (A sketch only) Consider the transition sequence leading from $C$ to $D$ (which we can assume leads through pruned configurations only). If no folding precedes an unfolding, we are through. Otherwise, the situation is that some $C_i$ has been folded into $C_{i+1}$ and then unfolded into $C_{i+2}$. As a folding represents an inverse unfolding, we can apply theorem 1 to find $C'_{i+1}$ such that $C_i$ and $C_{i+2}$ both can be unfolded into $C'_{i+1}$. Moreover, it is not too hard to see that the latter unfolding can be reversed into a folding from $C'_{i+1}$ into $C_{i+2}$.

By repeating the process above, we eventually gets a transition sequence in the desired form. □

If $D$ is in normal form, we from $1 \vdash_u D \to E$ conclude $D = E$. We then, by combining with Theorem 3, get

**Corollary 5.** *If* $2 \vdash C \to D$*, with* $C$ *pruned and* $D$ *in normal form,* $1 \vdash_u C \to D$*.*

**Definition 6.** If there exists a $C$ in normal form such that there is a 2-valid transition from $B$ to $C$, then $[\![\mathrm{B}]\!]_2 = \mathcal{E}(B, C)$. If there exists no such $C$, $[\![\mathrm{B}]\!]_2 = \perp$.

Using Corollary 5 we see that this is well-defined. Moreover, we get a theorem expressing that transformation is *partial correct*:

**Theorem 7.** *If* $[\![B]\!]_2 = v \neq \perp$*, then also* $[\![B]\!]_1 = v$ *– i.e. for all* $B$*,* $[\![B]\!]_2 \leq [\![B]\!]_1$*.*

In order to give conditions for *total correctness* we need some definitions. We say that $B$ *loops at level 1* iff there exists an infinite sequence of pruned, non-empty configurations $C_1 \ldots C_n \ldots$ with $B = C_1$ such that for all $i$ there exists a 1-valid transition from $C_i$ to $C_{i+1}$ where this transition represents (at least) one unfolding step. Similarly, we can define that $B$ loops at level 2.

Notice that it does not necessarily hold that $[\![\mathrm{B}]\!]_2 = \bot$ (or $[\![\mathrm{B}]\!]_1 = \bot$) implies that $B$ loops at level 2 (level 1), since a given configuration may be *stuck*, i.e. it is not in normal form but it is not possible to unfold any goals. However, we shall assume that the rules are "exhaustive", excluding such behavior.

As is well known, it may happen that $B$ loops at level 1 (due to some ineffective strategy) while $[\![\mathrm{B}]\!]_1 \neq \bot$. On the other hand, if we (cf. [Llo84]) define a *fair* strategy as a strategy which sooner or later unfolds any goal, we have

**Fact 8.** *If $B$ loops at level 1 (2) by a fair strategy, then $[\![B]\!]_1 = \bot$ ($[\![B]\!]_2 = \bot$).*

## 3 Global Conditions for Total Correctness

We start by proposing some conditions which may be helpful for proving total correctness. Here $B$ will be a basic configuration.

**Definition 9.** We say that $\mathcal{F}$ (for *F*air) is satisfied iff for any $B$ it holds that if $B$ loops at level 2 by a fair strategy then it also loops at level 1 by a fair strategy.

We say that $\mathcal{L}$ (for *L*eftmost) is satisfied iff for any $B$ it holds that if $B$ loops at level 2 by the $\mathcal{LR}$ strategy then it also loops at level 1 by the $\mathcal{LR}$ strategy.

We say that $\mathcal{A}$ (for *A*ny) is satisfied iff for any $B$ it holds that if $B$ loops at level 2 by some strategy then it also loops at level 1 by some strategy.

**Theorem 10.** *Suppose $\mathcal{F}$ holds. Then for any $B$, $[\![B]\!]_2 = [\![B]\!]_1$.*

*Proof.* Suppose $[\![\mathrm{B}]\!]_2 = \bot$. Then $B$ loops at level 2 by a fair strategy, and by $\mathcal{F}$ also at level 1 by a fair strategy. By Fact 8, $[\![\mathrm{B}]\!]_1 = \bot$. Now combine with Theorem 7. □

In order to model e.g. PROLOG, a semantics capturing that evaluation employs the $\mathcal{LR}$ strategy seems better suited:

**Definition 11.** If there exists a $C$ in normal form such that there is a 1-valid (2-valid) transition, using the $\mathcal{LR}$ strategy, from $B$ to $C$, then $[\![\mathrm{B}]\!]^L_1 = \mathcal{E}(B, C)$ ($[\![\mathrm{B}]\!]^L_2 = \mathcal{E}(B, C)$). If there exists no such $C$, $[\![\mathrm{B}]\!]^L_1 = \bot$ ($[\![\mathrm{B}]\!]^L_2 = \bot$).

**Fact 12.** *For all $B$, $[\![B]\!]^L_1 \leq [\![B]\!]_1$, $[\![B]\!]^L_2 \leq [\![B]\!]_2$.*

As is well known, we cannot hope for $[\![\mathrm{B}]\!]^L_2 \leq [\![\mathrm{B}]\!]^L_1$ to hold in general (unless some tight restrictions are made). This corresponds, in the functional world, to the fact that the call-by-name nature of symbolic evaluation may increase the domain of termination in a call-by-value language. Now let us consider what must be fulfilled in order for the opposite inclusion to hold:

**Theorem 13.** *Suppose $\mathcal{L}$ is fulfilled. Then for all $B$, $[\![B]\!]^L_1 \leq [\![B]\!]^L_2$.*

*Proof.* If $[\![\mathrm{B}]\!]_2^L = v \neq \bot$, then by Fact 12 $[\![\mathrm{B}]\!]_2 = v$. By Theorem 7, $[\![\mathrm{B}]\!]_1 = v$; so by Fact 12 $[\![\mathrm{B}]\!]_1^L \leq v$. If $[\![\mathrm{B}]\!]_2^L = \bot$, the result follows from the assumption. □

Finally, one may ask what use the condition $\mathcal{A}$ is. Now, suppose $B$ is such that if $B$ loops at level 1 by some strategy then $B$ loops at level 1 by any strategy (we say that $B$ is strategy-independent). Then we can reason as follows: Suppose $[\![\mathrm{B}]\!]_2 = \bot$. Therefore $B$ loops at level 2 by any strategy. If $\mathcal{A}$ holds, $B$ loops at level 1 by some strategy. By assumption, $B$ loops at level 1 by any strategy; thus $[\![\mathrm{B}]\!]_1 = \bot$. We have shown

**Theorem 14.** *If $\mathcal{A}$ holds, and $B$ is strategy-independent, then $[\![B]\!]_2 = [\![B]\!]_1$.*

## 4 Local Conditions for Total Correctness

This section will be devoted to formulating conditions F, L, A for the 1t-valid transition sequences leading to the level-1-rules, such that F implies $\mathcal{F}$, L implies $\mathcal{L}$ and A implies $\mathcal{A}$.

In order to reason about 1t-valid transition sequences we represent them as *U-mirrors*, which models the fact (cf. Theorem 4) that such a sequence is equivalent to a sequence of unfoldings followed by a sequence of foldings. Before stating a formal definition we will give an example.

*Example 2.* Let the source program $S_2$ be given by (as it is the control aspect which has our primary interest, we use substitutions to describe data)

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

We can transform the definition of $p$ by *first* unfolding $q$ (into $q$) and *then* fold back into $p$. By leaving $q$ and $r$ unchanged, we arrive at the target program $T_2$:

$p(a) \leftarrow p(a)$; $q(a) \leftarrow q(a)$; $r(b) \leftarrow \Box$

In Fig. 2 is depicted a U-mirror $u$ corresponding to the transformation on $p$.

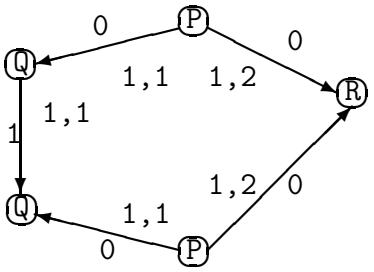


**Fig. 2.** The U-mirror $u$, corresponding to $T_2$.

We see that $u$ has an arrow from the topmost node labeled $q$ to the node below (representing the unfolding of $q$), and the folding step is represented by the arrows going *up* from the bottom node.

As can be seen, also the arcs are labeled. For instance the arc from $p$ to $r$ is labeled with $(1,2)$ (inside the diagram) and with 0 (outside the diagram). The label $(1,2)$ means that the 2nd conjunct in the 1st rule (in this case, there are no other rules) for $p$ is $r$. The label 0 is the *weight* of the arc, more about that later. □

Now we will embark on giving a formal definition of U-mirrors. First, the control aspect (ctr. the data aspect) of a source program must be represented:

**Definition 15.** A control representation of a source program consists of

- A universe $U$ of predicate symbols. Let $U' = U \cup \{\Box\}$
- A function $OI$ (for Or Indices) which for each $G \in U$ returns a non-empty index set $OI(G)$.
- A function $AI$ (for And Indices) which for each $G \in U$ and each $i \in OI(G)$ returns a non-empty index set $AI(G,i)$. This set must be equipped with a total order $\prec$, $j_1 \prec j_2$ modelling "$j_1$ is to the left of $j_2$".
- A function $P$ which for each $G \in U$, each $i \in OI(G)$ and each $j \in AI(G,i)$ returns $P(G,i,j) \in U'$. If $P(G,i,j) = \Box$, then $AI(G,i)$ is a singleton.
- A function $W$ which for each $G \in U$, each $i \in OI(G)$ and each $j \in AI(G,i)$ returns $W(G,i,j)$, a non-negative integer.

Except for the weight part it is rather obvious how one, given a source program, arrives at a control representation:

*Example 3.* For the source program $S_2$ from Example 2 we have among others $U = \{p,q,r\}$, $OI(p) = \{1\}$ (or any one-element set), $AI(p,1) = \{1,2\}$ (or any two-element set) with $1 \prec 2$, $P(p,1,1) = q$, $P(p,1,2) = r$, $P(r,1,1) = \Box$. □

The weights, on the other hand, can be assigned in a completely arbitrary fashion - but if one wants to prove that a given transformation is correct using the techniques given in this paper, one may have to be a bit clever about the assignment. In Fig. 2, one has e.g. made the choices $W(p,1,2) = 0$, $W(q,1,1) = 1$.

**Definition 16.** A *goal sequence* $(J,H)$ is a totally ordered index set $J$ and a mapping $H$ from $J$ to $U$.

**Definition 17.** A U-forest over a goal sequence $(J,H)$ is a $J$-indexed family of trees where

- All nodes are labeled by a member of $U'$.
- All arcs are labeled by a direction label $(i,j)$ and a weight label $w$.
- For all $j \in J$, the root of the $j'$th tree must be labeled by $H(j)$
- Let $N$ be a node which is not a leaf. Then its label $G$ must belong to $U$ (i.e. not be $\Box$), and there must exist an $i' \in OI(G)$ such that $(i,j)$ is the label of an arc from N if and only if $i = i'$, $j \in AI(G,i)$. The arcs from $N$ inherit the ordering of $AI(G,i)$.
- If $a$ is an arc with direction label $(i,j)$ from a node labeled $G$ to a node $N$, then the label of $N$ must be $P(G,i,j)$, and the weight label of $a$ must be $W(G,i,j)$
- There is a total ordering among the leaves (and thus also among the paths, where a path starts at a root and ends at a leaf), determined in the "natural way" by the ordering on $J$ and the ordering on the arcs leaving each node.

We say that a U-forest is *foldy* iff no nodes are labeled $\Box$. The intuition is that if one has a clause say $r \leftarrow \Box$ then one is not allowed to make a fold transition from $p$ to $p,r$.

We define the *weight* of a path as the sum of the weight labels encountered when walking along the path. We define the *weight* of a U-forest as the sum of all the weight labels occurring in the tree.

**Definition 18.** A U-mirror $(u, v, \sim)$ over goal sequences $H$ and $H'$ consists of a U-forest $u$ over $H$, a foldy U-forest $v$ over $H'$, and a bijection $\sim$ between the paths of $u$ *not* containing $\Box$ and the paths of $v$ which preserves the ordering on these, and which satisfies that if $p \sim q$ then $p$ and $q$ end with the same symbol.

A path in $(u, v, \sim)$ is *either* of the form $(p, q)$ with $p \in u$, $q \in v$ and $p \sim q$ *or* is of the form $p$, where $p \in u$ and $p$ ends with $\Box$. There is a natural total order on paths.

Referring back to Fig. 2, it can be checked that $u$ is a U-mirror over $(p)$ and $(p)$.

We can assign weights to paths in a U-mirror as follows: if the path is of form $(p, q)$, the weight is the difference between the weight of $p$ and the weight of $q$; otherwise the path is of form $p$ and its weight is simply the weight of $p$. Likewise, we can define the weight of the U-mirror as the difference between the weight of $u$ and the weight of $v$.

Finally, we are in position to formulate F, L and A:

**Definition 19.** Given a U-mirror $m$. Then

- $m$ satisfies F iff all paths have weight $\geq 1$
- $m$ satisfies L iff the leftmost path has weight $\geq 1$
- $m$ satisfies A iff $m$ has weight $\geq 1$.

*Example 4.* Referring back to Fig. 2, $u$ satisfies L and A but not F.

We say that a transition from $B$ to $B_1 + \ldots + B_n$ satisfies F if for all $i \in \{1 \ldots n\}$, the U-mirror representing how to get from $B$ to $B_i$ satisfies F. Similarly for L and A.

**Theorem 20.** *Suppose all rules at level 1 satisfy* F. *Then $\mathcal{F}$ holds, and thus $[\![B]\!]_2 = [\![B]\!]_1$ for all $B$, cf. Theorem 10.*

This theorem is essentially the main content of (but is more general than) [Sek91]

**Theorem 21.** *Suppose all rules at level 1 satisfy* L. *Then $\mathcal{L}$ holds, and thus $[\![B]\!]_2^L \geq [\![B]\!]_1^L$ for all $B$, cf. Theorem 13.*

More specific versions of this theorem are stated in [PP91a] and [Han91].

**Theorem 22.** *Suppose all rules at level 1 satisfy* A. *Then $\mathcal{A}$ holds, and thus $[\![B]\!]_2 = [\![B]\!]_1$ for all* strategy-independent *$B$, cf. Theorem 14.*

As the correctness condition for folding, [TS84] and [KK90] essentially use (a special case of) A.

For proofs of Theorem 20, 21 and 22 (which all elaborate on the intuition presented in section 1), see [Amt92].

*Example 5.* Continuing from Example 4, we by Theorem 21 can conclude that $\mathcal{L}$ holds - which is as expected since $S_2$ loops on *p(t)* by the $\mathcal{LR}$ strategy iff $t$ unifies with $a$ iff $T_2$ loops on *p(t)* by the $\mathcal{LR}$ strategy.

On the other hand, our theorems do not allow us to conclude that $\mathcal{F}$ holds - and neither it does, since $S_2$ terminates by a fair strategy on *p(t)* for any $t$ while $T_2$ loops (by any strategy) on e.g. *p(a)*.

Notice that if we e.g. had chosen $W(q, 1, 1) = 0$, we had not been able to infer that $\mathcal{L}$ holds.

## 5 Related Work

- In the literature on unfold/fold transformations in logic languages transformation typically proceeds in a "step by step fashion"; after a goal in the body of a clause has been unfolded the clause is *deleted* from the program and *replaced* by the clause resulting from the unfolding - this is the approach taken in e.g. [GS91], [KK90], [PP91a], [Sek91], [TS84]. As pointed out in [GS91], one by applying this method loses some power - to see this, consider the clause $C = p(f(X)) \leftarrow p(X)$. By our or similar techniques one is able to derive the clause $C' : p(f(f(f(X)))) \leftarrow p(X)$ but this is impossible by the step-by-step method, since one - after having unfolded $C$ against itself obtaining $p(f(f(X))) \leftarrow p(X)$ - has lost $C$. Aside from being less powerful, we also think that the step-by-step strategy conceptually is much less clean than our approach - a similar view being held in [Tur86].
- In the literature, one is typically (contrary to our framework) not allowed to fold against a (direct or indirect) recursive predicate [KK90], [PP91a], [Sek91], [TS84]. This mirrors the view that folding corresponds to abbreviation, a view also held in [Han91]. On the other hand, as pointed out in [PP91b] this is essentially no restriction for applications.
- [TS84] and [KK90] divide the predicates into two classes: the *new* (corresponding to "eureka-definitions") and *old*, where folding is allowed against new predicates only. In order for folding to be valid, they require that in the body of the clause defining a new predicate (where only old predicates can occur) at least one predicate must have been unfolded. We can translate this into our framework: new predicates are assigned weight 0, old predicates are assigned weight 1, and A must hold. As we have seen in Sect. 3, this condition is (too) weak, since failing branches may convert to loops.
- [Sek91] improves on the above, essentially by demanding that F must hold (still when new predicates have been assigned weight 0 and old predicates weight 1).
- [Kot85] treats a functional language, where there apparently is no branching. The situation is that first a number of unfoldings are made, then some laws are applied (not catered for by our framework), then some foldings are made. It is claimed that folding is safe if the number of unfoldings is greater than the number of foldings. In some sense, this can be modeled in our framework by assigning all predicates weight 1.
- [GS91] allows folding against *existing* clauses (recall clauses are deleted after having been unfolded) only (not allowing a clause to be folded against itself).

This greatly limits the applications, since it seems impossible to arrive at recursive definitions of eureka-predicates. On the other hand, it becomes possible to give a relatively simple proof of termination preservation.
- In contrast to the authors mentioned so far, [PP91a] impose an order on a sequence of goals, i.e. consider PROLOG's $\mathcal{LR}$ strategy. The crucial condition on folding is that the leftmost atom has been unfolded. Again by assigning the predicates folded against weight 0 and the others 1, the essence of this translates into our condition L.

## 6 Final Remarks

We hope this paper has contributed to giving a better understanding of the notion of folding, making the duality between folding and unfolding more explicit. Still, not every unfolding can be reversed into a folding, and even though we cannot hope for this, the set of legal reversals might be extended:

- If there is a transition from $B$ to $B_1+B_2$, we in order to make a fold step from $B_1$ to $B$ had to ensure that $B_2$ was failure. Under some conditions it will be enough if there is a transition from $B_2$ to a failure configuration - such an extension is essential if one is to model folding against clauses derived during transformation but not present in the original source program.
- If there is a transition from $B$ to $B_1+B_2$, we may even consider folding the *non-singleton* configuration $B_1+B_2$ back into $B$ - this bears some similarity to the process of converting a nondeterministic automaton into a deterministic equivalent.

Work in progress includes setting up a model for a *functional* language in which results similar to the ones presented here can be obtained.

Thanks are due to Jens Palsberg, Jesper Träff and Brian Mayoh for reading drafts of this paper; also thanks to the anonymous referees for useful suggestions.

## References

[Amt92] Torben Amtoft. Unfold/fold transformations preserving termination properties. To appear as a technical report from DAIMI, University of Aarhus, Denmark, 1992.

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

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.

[DP88] John Darlington and Helen Pull. A program development methodology based on a unified approach to execution and transformation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 117–131. North-Holland, 1988.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Gre87] Steve Gregory. *Parallel Logic Programming in PARLOG - the language and its implementation*. Addison-Wesley, 1987.

[GS91] P. A. Gardner and J. C. Shepherdson. Unfold/fold transformations of logic programs. In *Computational Proofs: Essays in honour of Alan Robinson*. 1991.

[Han91] Torben Amtoft Hansen. Properties of unfolding-based meta-level systems. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9)*, 1991.

[Hol91] Carsten Kehler Holst. Finiteness analysis. In John Hughes, editor, *International Conference on Functional Programming Languages and Computer Architecture*. Springer Verlag, LNCS no 523, August 1991.

[Hon91] Zhu Hong. How powerful are folding/unfolding transformations. Technical Report CSTR-91-2, Department of Computer Science, Brunel University, January 1991.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[KK90] Tadashi Kawamura and Tadashi Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformation. *Theoretical Computer Science*, 75:139–156, 1990.

[Kot85] Laurent Kott. Unfold/fold program transformations. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12. Cambridge University Press, 1985.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[NN90] Hanne Riis Nielson and Flemming Nielson. Eureka definitions for free! or disagreement points for fold/unfold transformations. In Neil D. Jones, editor, *ESOP 90, Copenhagen, Denmark. LNCS 432*, pages 291–305, May 1990.

[Pal89] Catuscia Palamidessi. Algebraic properties of idempotent substitutions. Technical Report TR-33/89, University of Pisa, 1989.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, University of Aarhus, Denmark, September 1981.

[PP91a] Maurizio Proietti and Alberto Pettorossi. Semantics preserving transformation rules for Prolog. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9)*, 1991.

[PP91b] Maurizio Proietti and Alberto Pettorossi. Unfolding - Definition - Folding, in this order, for avoiding unnecessary variables in logic programs. In *Proceedings of PLILP 91, Passau, Germany (LNCS 528)*, August 1991.

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

[Søn89] Harald Søndergaard. Semantics-based analysis and transformation of logic programs. Technical Report 89/22, DIKU, University of Copenhagen, Denmark, 1989.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/fold transformation of logic programs. In *Proceedings of 2nd International Logic Programming Conference, Uppsala*, pages 127–138, 1984.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

This article was processed using the LaTeX macro package with LLNCS style