THE LOOP ABSORPTION AND THE GENERALIZATION STRATEGIES FOR THE DEVELOPMENT OF LOGIC PROGRAMS AND PARTIAL DEDUCTION

MAURIZIO PROIETTI AND ALBERTO PETTOROSSI

We present a program transformation methodology which is based on the invention of the so-called eureka definitions necessary for improving program efficiency. We propose a strategy, called loop absorption, for the automatic generation of those definitions in the case of definite logic programs, and we show its use for partial deduction.

The problem of finding the eureka definitions is formalized as the search for suitable trees of clauses, called foldable U-trees, which are derived by unfolding the initial programs. It is possible, in general, to construct foldable U-trees if one uses the generalization rule. This rule should be applied with parsimony because it may reduce the efficiency of the derived programs. For overcoming this inconvenience, we propose a generalization strategy.

We also study the properties of that strategy, together with the loop absorption, and we show that some techniques for partial deduction correspond to particular ways of applying our program derivation methodology.

1. INTRODUCTION

Often, correctness and efficiency of programs are two conflicting requirements. In many cases, in fact, highly efficient programs can be proved correct at the expense of intricate proofs.

In order to overcome this difficulty, one may follow the program transformation methodology, by which the programmer is first asked to write a simple program, whose correctness can easily be shown, and then to improve its efficiency by applying transformation rules which preserve correctness. This methodology has...
been suggested in [3] for functional languages, but it can also be applied in the case of logic programs (see, for instance, [2, 11, 23, 25] and other papers listed in the References).

In this paper, we consider definite logic programs, and we develop some new techniques for their transformation. They are an improvement of those presented in [21]. We use the definition, unfolding, and folding rules described in the Appendix, and we also use some transformation strategies indicated in the following sections for guiding the application of the above rules and deriving more efficient programs.

Let us now recall, for the reader's convenience, the basic ideas of the program transformation methodology, as it can be described in the case of definite logic programs.

We are given the initial version of a program, which can also be viewed as a specification, and we want to derive a more efficient version. This may often be done by using the tupling strategy (see, for instance, [20]), which is applied as follows.

Suppose that in the body of a clause $C$ of our initial program, we have the atoms $A_1, \ldots, A_n$ whose evaluation determines some "unnecessary work," such as the repeated evaluation of common subgoals, the multiple visits of data structures, the construction of intermediate bindings, etc.

In that case, we introduce, using the definition rule, a new clause, say $D$, which defines a new predicate, usually called a eureka predicate.

The body of $D$ consists of the atoms $A_1, \ldots, A_n$, and the arguments of the head of $D$ are the distinct variables occurring in $A_1, \ldots, A_n$ and elsewhere in the clause $C$. We then replace, by performing a folding step, the atoms $A_1, \ldots, A_n$ in the body of $C$ by the head of $D$.

The above application of the definition and folding rules does not improve efficiency. However, efficiency improvements can indeed be obtained if we derive a recursive definition of the eureka predicate defined by the clause $D$. In fact, by doing so, we will avoid the "unnecessary work" during the evaluation of the atoms $A_1, \ldots, A_n$ at each level of recursion. These ideas will be made clear by the examples given below.

In order to derive the recursive definition of the eureka predicate, we first unfold the newly introduced clause $D$, thereby deriving a so-called unfolding tree of clauses, and we then look for an unfolding tree whose leaves are either clauses which cannot be further unfolded or can be folded using the eureka predicate. Those leaves will produce for us the recursive definition we wanted.

Sometimes the unfolding tree with the desired properties cannot be found if we allow ourselves to fold w.r.t. the initial clause $D$ only. In those cases, we introduce some auxiliary predicates (also called eureka predicates) so that we can perform extra folding steps. However, in order to achieve efficiency improvements, we are then required to find the recursive definitions for those auxiliary predicates as well.

More technical details of the program transformation process which we have now outlined will be given in the following sections, where we study the problem of finding the recursive definitions of the eureka predicates in an automatic way.

### 1.1. Partial Evaluation via Program Transformation: An Example

Before explaining the technicalities of our program transformation methodology, we want to make a preliminary study of its relationship with the partial evaluation
technique, also called partial deduction in the case of logic programs (see, for instance, [14, 19].

Partial evaluation works by importing some static information from the input data into the program. In more formal terms, the partial evaluation of a (definite) program \( \text{Prog} \) w.r.t. a goal \( G \) is an algorithm which produces a residual program \( \text{Prog}_I \), which is equivalent to \( \text{Prog} \) w.r.t. the goal \( G \). Here, and in the sequel, this notion of equivalence w.r.t. a goal means that \( \text{Prog} \cup \{G\} \) has a correct answer substitution \( \theta \) iff \( \text{Prog}_I \cup \{G\} \) does.

The residual program \( \text{Prog}_I \) is usually more efficient than the original one, simply because part of the input data of \( G \) already has been processed at partial evaluation time.

In the simple example we will now give, we see that partial evaluation can be performed by introducing a new predicate, and then finding its recursive definition, exactly as during the process of program transformation we have described above. Later on, we will provide a more formal comparison between our transformation strategies and some partial evaluation techniques.

Suppose, for instance, that we want to partially evaluate the following program:

\[
\begin{align*}
p([\ ], Y). \\
p([H|T], a) & \leftarrow p(T, a). \\
p([H|T], b) & \leftarrow p(T, b).
\end{align*}
\]

with respect to the goal \( \leftarrow p(X, a) \).

We follow the partial evaluation methodology as proposed in [15], which is equivalent to the standard one presented in [19].

Given a program \( \text{Prog} \) and an atomic goal \( \leftarrow G \), we consider the tautological clause \( G \leftarrow G \), which in our case is \( p(X, a) \leftarrow p(X, a) \). We then unfold once this clause w.r.t. the clauses of \( \text{Prog} \), and we get the following two clauses, called resultants:

\[
\begin{align*}
p([\ ], a). \\
p([H|T], a) & \leftarrow p(T, a).
\end{align*}
\]

which are the residual program \( \text{Prog}_I \) w.r.t. the goal \( \leftarrow p(X, a) \).

Notice that, in general, the set of resultants is a program which is not equivalent to the initial one w.r.t. the given goal (that is, the derived clauses are not a residual program, according to the terminology we have introduced above). However, in this example, the equivalence holds because every atom occurring in the body of the resultants (in our case, we have the atom \( p(T, a) \) only) is an instance of the goal w.r.t. which we have performed partial evaluation.

The reader may look at [19] for a more general condition which ensures that the unfolding process produces an equivalent program.

We now show that our simple example of partial evaluation can be rephrased in terms of the program transformation process we have described above.

We proceed as follows. We first introduce by the definition rule a clause whose body is the goal w.r.t. which partial evaluation should be performed, and whose head is a fresh predicate symbol.

The arguments of the head are the distinct variables of the body. In our case, we introduce the following clause:

D. \( \text{newp}(X) \leftarrow p(X, a) \).
We are now left with the problem of deriving the recursive definition of \( \text{newp}(X) \). By unfolding clause \( D \), we get

\[
\text{newp}([\ ]), \\
\text{newp}([H|T]) \leftarrow p(T, a).
\]

Now, the same syntactical condition which validates the partial evaluation process above, that is, the fact that \( p(T, a) \) is an instance of the body of \( D \), allows us to fold the atom \( p(T, a) \) w.r.t. the definition clause \( D \). We get the following program \( \text{Prog2} \):

\[
\text{newp}([\ ]), \\
\text{newp}([H|T]) \leftarrow \text{newp}(T).
\]

which has performances similar to (actually, higher than) those of \( \text{Prog1} \). The equivalence between \( \text{Prog2} \) and the initial version of the program derives from the fact that we have used the unfold/fold rules. Thus, we have that: \( \text{Prog} \cup \{ \leftarrow p(X, a) \} \) has a correct answer \( \theta \) iff \( \text{Prog2} \cup \{ \leftarrow \text{newp}(X) \} \) does.

Let us now compare the derivations by partial evaluation and program transformation shown above. We have two reasons for preferring the unfold/fold transformation method, and they are valid in general, and not simply in the above example.

The first reason is that the use of the transformation rules is sufficient to ensure the correctness of the derivation: extra conditions, like the ones in [19], are not needed.

The second reason is that the final program obtained by unfold/fold transformation is slightly more efficient because the newly introduced predicate \( \text{newp} \) does not have redundant arguments. In our case, \( \text{newp} \) has one argument only, while \( p \) has two arguments (although one may get \( \text{newp} \) from \( p \) by using ad hoc methods, such as the ones proposed in [8]).

As our example shows, the ability of the program transformation techniques to incorporate those related to partial evaluation is based on the possibility of performing some folding steps which cause the efficiency improvements.

One may argue that during program transformation, we have the extra problem of guiding the derivation for allowing the required folding steps. However, this is not an additional difficulty w.r.t. the partial evaluation techniques described in [19] because the conditions which validate the partial evaluation techniques are exactly those which allow us to perform the folding steps.

In Section 5, we will compare program transformation and partial evaluation in a more formal way, while in the following sections, we will present two powerful strategies for program transformation and the proof of some useful theoretical results.

1.2. The Loop Absorption Strategy in Action

We now present an introductory example of program derivation by transformation, where we will see in action a strategy, called \textit{loop absorption}, which we propose in this paper for driving the application of the transformation rules. By using loop absorption, we will automatically construct the recursive definitions of the eureka predicates necessary to perform the required folding steps.
This example also serves the purpose of motivating the introduction of some notions which are defined in the next section.

Example 1: List derivative. Let us consider the problem of: a) deleting from a list \( L \) of positive numbers the ones which are to the left of a larger number, and then b) deleting all elements of the residual list occurring in odd positions. It is assumed that the rightmost element of the list \( L \) is to the left of 0, and thus, it is never discarded after phase a).

The following logic program, called Even_Npderiv, solves that problem by constructing the list \( D \) which is obtained after the first deletion phase. (The name npderiv comes from the fact that \( D \) may be viewed as the list of values in \( L \) where the "derivative" is not positive.)

1.1 even_npderv\((L, E) \leftarrow npderiv(L, D), even(D, E).\)
1.2 npderiv([ ],[ ]).  
1.3 npderiv([X],[X]).  
1.4 npderiv([X,Y|T],[X|D]) \leftarrow \text{X} \geq \text{Y}, npderiv([Y|T], D).  
1.5 npderiv([X,Y|T], D) \leftarrow \text{X} < \text{Y}, npderiv([Y|T], D).  
1.6 even([ ],[ ]).  
1.7 even([X],[ ]).  
1.8 even(X,Y|T],[Y|E]) \leftarrow even(T,E).  

The above program is inefficient because the occurrence of the shared variable \( D \) in clause 1.1 determines the construction of an intermediate list whose value is not needed in the final result \( E \).

We may avoid the construction of the list \( D \) by deriving an equivalent definition for the predicate even_npderv without shared variables in the body of the defining clause. We apply the tupling strategy and we introduce the new clause:

\[ C. \ \text{newp}(L, E) \leftarrow npderiv(L, D), even(D, E). \]

whose body is made out of the atoms with shared variables in the body of clause 1.1, which determine the construction of the unnecessary intermediate binding for \( D \).

Since clause \( C \) is identical to clause 1.1, apart from the name of the head predicate, and since in the initial program only clause 1.1 has even_npderv as head predicate, we may avoid the introduction of clause \( C \) and we simply consider clause 1.1 as a definition clause. Thus, if required, we can perform folding steps using clause 1.1 (see the Appendix for the restrictions on the applicability of the folding rule).

Then we unfold clause 1.1, and continue the unfolding process, searching for a set \( S \) of clauses which satisfies the following condition:

\[ \text{each clause in } S \text{ either cannot be further unfolded or it can be folded.} \]

This condition characterizes our program transformation methodology as it will be formalized below.

If we obtain such a set of clauses, we can eliminate, by folding, the occurrences of the shared variable between npderiv and even, and thus we avoid the construction of the intermediate list \( D \).

If such a set cannot be obtained, we proceed with our program derivation process by introducing some suitable eureka predicates, as we will see below.
By unfolding \textit{npderiv} in clause 1.1, we get the following four clauses:

2. \texttt{even\_npderiv([ ]}, E) \leftarrow \texttt{even([ ], E)}.
3. \texttt{even\_npderiv([X], E) \leftarrow even([X], E)}.
4. \texttt{even\_npderiv([X,Y|T], E) \leftarrow X \geq Y, \texttt{npderiv([Y|T], D)}, \texttt{even([X|D], E)}).
5. \texttt{even\_npderiv([X,Y|T], E) \leftarrow X < Y, \texttt{npderiv([Y|T], D)}, \texttt{even(D, E)}}.

Clause 5 can be folded w.r.t. clause 1.1 because \texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{npderiv([Y|T], D)}, even(D, E)\}}}}} is an instance of the body of clause 1.1, and we get

5f. \texttt{even\_npderiv([X,Y|T], E) \leftarrow X < Y, even\_npderiv(Y|T), E)}.

Clauses 2 and 3 can be simplified by unfolding \texttt{even}, and we get

6. \texttt{even\_npderiv([ ], [ ])}.
7. \texttt{even\_npderiv([X], [ ])}.

Clause 4 cannot be folded, and therefore we continue the unfolding process by selecting the atom \texttt{even}. In what follows, we will discuss the general problem of selecting the atom to be unfolded, and we will propose a selection rule, called \texttt{SDR}. From clause 4, we get the following clauses:

8. \texttt{even\_npderiv([X,Y|T], [ ]) \leftarrow X \geq Y, npderiv([Y|T], [ ])}. 
9. \texttt{even\_npderiv([X,Y|T], [Z|E]) \leftarrow X \geq Y, npderiv([Y|T], [Z|D], even(D|], E)}.

Also, clauses 8 and 9 cannot be folded; therefore, we continue the unfolding process. By unfolding \texttt{npderiv} in clause 8, we get

10. \texttt{even\_npderiv([X,Y,Z|T], [ ]) \leftarrow X \geq Y, Y < Z, npderiv([Z|T], [ ])}.

By unfolding \texttt{npderiv} in clause 9, we get the following three clauses:

11. \texttt{even\_npderiv([X,Y], [Y|E]) \leftarrow X \geq Y, even([ ], E)}.
12. \texttt{even\_npderiv([X,Y,Z|T], [Y|E]) \leftarrow X \geq Y, Z \geq G, ~ npderiv([Z|T], D), even(D, E)}.
13. \texttt{even\_npderiv([X,Y,Z|T], [V|E]) \leftarrow X \geq Y, Y \leq Z, npderiv([Z|T], [V|D]), even(D, E)}.

Clause 11 can be simplified, and we get

14. \texttt{even\_npderiv([X,Y], [Y]) \leftarrow X \geq Y}.

Clause 12 can be folded w.r.t. clause 1.1, while clauses 10 and 13 cannot be folded.

The patterns of atoms \texttt{"npderiv([Z|T], [ ]")} and \texttt{"npderiv([Z|T], [V|D]), even(D, E)"} in clauses 10 and 13 are instances (actually variants) of the patterns of the same atoms in clauses 8 and 9, respectively. We say that these patterns of atoms are recurrent, and that some loops for the patterns have been found.

We leave it to the reader to show that, for any choice of the atom to be unfolded, the unfolding process can never produce a set of clauses satisfying condition \texttt{\alpha}. We conclude that it is impossible to transform the Even\_Npderiv program and improve its efficiency by performing folding steps w.r.t. clause 1.1 only.

However, the derivation we have performed so far does not result in a complete failure. Indeed, we may apply the loop absorption strategy which uses the recurrent
patterns of atoms for introducing some extra eureka predicates. The clauses which define these predicates will allow us to perform some extra folding steps.

In order to clarify the application of the loop absorption strategy, let us analyze the unfolding process we have performed. It can be depicted in a tree-like fashion, as illustrated in Figure 1, where a solid arrow denotes an unfolding step, and the atom selected for unfolding has been underlined. A dashed arrow from clause \( x \) to the ancestor clause \( y \) denotes the fact that a pattern of atoms in clause \( x \) is an instance of a pattern in clause \( y \).

In the next section, we will formally introduce the representation of the unfolding process as a tree of clauses, which will be called Unfolding-tree (or U-tree for short). In Figure 1, a loop for a recurrent pattern of atoms is represented by a loop of arrows made out of one or more solid arrows “going down” in the U-tree and exactly one dashed arrow “going up”.

The reader may notice that each leaf of the tree of clauses depicted in Figure 1 either cannot be unfolded or it is on a loop for a recurrent pattern. A U-tree satisfying this property is an example of a foldable U-tree, which will be formally introduced in the next section.

![Figure 1](image-url)

**FIGURE 1.** An upper portion of the U-tree for \( \text{Even} \_\text{Npderiv} \) and clause 1.1 (\( e\_\text{npd}, \text{npd}, \) and \( e \) stand for \( \text{even} \_\text{npderiv}, \text{npderiv}, \) and \( \text{even}, \) respectively).
We will see in the sequel that the loop absorption strategy is always applicable when a foldable U-tree is found. The reader may refer to the Loop Absorption Procedure of the next section for a formal description of our strategy. Here, we simply show how it is used in our example. We have to take the recurrent patterns of atoms as the bodies of the clauses which define the eureka predicates to be introduced.

By applying the loop absorption strategy, we get the eureka predicates new1 and new2, defined as follows:

- **d1.** \( \text{new1}(Y, T) \leftarrow \text{npderiv}([Y|T],[\ ]). \)
- **d2.** \( \text{new2}(X, T, Y, E) \leftarrow \text{npderiv}([X|T],[Y|D]), \text{even}(D, E). \)

These definitions make it possible for us to perform folding steps on clauses 8 and 9, which are the clauses originating the loops for the recurrent patterns, and we get

- **8f.** \( \text{even\_npderiv}([X, Y|T],[\ ] \leftarrow X \geq Y, \text{new1}(Y, T). \)
- **9f.** \( \text{even\_npderiv}([X1, X|T],[Y|E]) \leftarrow X1 > X, \text{new2}(X, T, Y, E). \)

The program we have derived so far is made out of clauses 5f, 6, 7, 8f, 9f, together with the definitions of new1 and new2, and the clauses 1.2,. . ,1.8.

Now the program derivation process continues by taking d1 and d2 as initial clauses, and looking for the recursive definitions of the predicates new1 and new2, as we have done for the predicate even\_npderiv. This task can easily be accomplished because during the unfolding process, we have found recurrent patterns of atoms.

Indeed, for instance, the recurrent pattern of atoms "\( \text{npderiv}([Z|T],[\ ]) \)" which we have found during the unfolding process from clause 8 guarantees that the corresponding eureka definition d1 generates, by performing the same unfolding steps, a set of clauses satisfying condition (α), that is, each of them either cannot be unfolded or it can be folded w.r.t. a definition clause of the set \{1.1, d1, d2\}.

Thus, we may now replay the derivation process which we have performed from clause 8, and by unfolding clause d1, we get

- **d1.1** \( \text{new1}(Y, [Z|T]) \leftarrow Y < Z, \text{npderiv}([Z|T],[\ ]). \)

This clause can be folded w.r.t. clause d2, and we get

- **d1.1f** \( \text{new1}(Y, [Z|T]) \leftarrow Y < Z, \text{new1}(Z, T). \)

An analogous remark holds for the initial clause d2 w.r.t. the unfolding process starting from clause 9, and the recurrent pattern of atoms "\( \text{npderiv}([Z|T],[V|D]), \text{even}(D, E) \)".

Thus, we replay the derivation process which we have performed from clause 9. By unfolding clause d2, we get

- **d2.1** \( \text{new2}(X, [\ ], X, E) \leftarrow \text{even}([\ ], E). \)
- **d2.2** \( \text{new2}(X, [Y|T], X, E) \leftarrow X \geq Y, \text{npderiv}([Y|T], D), \text{even}(D, E). \)
- **d2.3** \( \text{new2}(X, [Y|T], Z, E) \leftarrow X < Y, \text{npderiv}([Y|T], [Z|D]), \text{even}(D, E). \)

Clause d2.1 can be simplified by unfolding even, and we get

- **d2.4** \( \text{new2}(X, [\ ], X, [\ ]). \)
Clause d2.2 can be folded w.r.t. clause 1.1, and we get
\[ d2.2f \quad \text{new}2(X, [Y|T], X, E) \leftarrow X \geq Y, \text{even_npderv}([Y|T], E). \]

Clause d2.3 can be folded w.r.t. clause d1, and we get
\[ d2.3f \quad \text{new}2(X, [Y|T], Z, E) \leftarrow X < Y, \text{new}2(Y, T, Z, E). \]

The program we have derived so far is made out of clauses 6, 7, 8f, 9f, 5f, d1.1f, d2.4, d2.2f, and d2.3f.

A further simplification is possible by observing that the predicate new1 has an empty model because the unique clause for it (that is, d1.1f) is recursive. Thus, clauses 8f and d1.1f can be discarded, and the final program is

6. \( \text{even_npderv}([ ], [ ]). \)
7. \( \text{even_npderv}([X], [ ]). \)
9f. \( \text{even_npderv}([XI, X[T], [Y|E]) \leftarrow XI \geq X, \text{new}2(X, T, Y, E). \)
5f. \( \text{even_npderv}([X,Y|T], E) \leftarrow X < Y, \text{even_npderv}([Y|T], E). \)
d2.4. \( \text{new}2(X, [ ], X, [ ]). \)
d2.2f. \( \text{new}2(X, [Y|T], X, E) \leftarrow X \geq Y, \text{even_npderv}([Y|T], E). \)
d2.3f. \( \text{new}2(X, [Y|T], Z, E) \leftarrow X < Y, \text{new}2(Y, T, Z, E). \)

The final version of the program avoids the construction of intermediate lists for computing the answer to a goal. Our computer experiments confirm that the derived program is more efficient than the initial version, both in time and space. For lists of about 600 elements for which the initial program version computes an intermediate list of about 300 elements, the improvements are about 10% in time and 20% in space.

In the rest of the paper, we will formally describe and develop the techniques which we have shown in action in the above example, and we will show how they can be used to perform the automatic transformation of large classes of programs.

In Section 2, we formally present the notion of a foldable U-tree. Then we present the loop absorption strategy which extracts from a foldable U-tree the required eureka predicates, together with their recursive definitions.

In Section 3, we study the problem, called the foldability problem, of performing suitable unfolding steps for finding foldable U-trees, and we show that it is unsolvable. We also show that for some classes of programs, we can find foldable U-trees by using the so-called SDR unfolding rule.

In Section 4, we introduce a new transformation rule, called the generalization rule, which can be used when we are not able to derive foldable U-trees by performing unfolding steps only. We give a procedure which uses the generalization rule for constructing foldable U-trees.

In Section 5, we show that the standard techniques for partial evaluation can be considered as a special case of our strategies for program transformation.

2. LOOP ABSORPTION FOR EXTRACTING PROGRAMS FROM FOLDABLE U-TREES

Let us start off by formalizing the process of unfolding a clause as a tree of clauses. Such a tree is called an Unfolding-tree. Since an unfolding step depends on the choice of the atom in the clause to be unfolded, the formalization of the unfolding
process also depends on the choice of a selection function from clauses to atoms, called *Unfolding-selection rule*, or *U-selection rule* for short. (Thus, the concepts of the U-tree and U-selection rule are analogous to those of the SLD-tree and computation rule [18], respectively.)

Let us introduce some preliminary definitions.

A *program* is a definite logic program [18] in which some *base predicates* have no explicit definition, that is, they do not occur in the heads of the program clauses. The nonbase predicates are called *defined predicates*. In the List Derivative Example above, we have implicitly assumed the base predicate to be $<$ and $>$, while the defined predicates are even\_npderiv, npderiv, and even.

The meaning of a defined predicate in a program can be given in the usual way in terms of the meanings of the base predicates. An atom with a defined predicate is called a *defined atom*.

**Definition 1.** Let $Prog$ be a program and $C$ a clause, possibly not occurring in $Prog$. We say that $C$ is a *success clause* (w.r.t. $Prog$) iff no defined predicate of $Prog$ occurs in the body of $C$.

We say that $C$ is a *failing clause* (w.r.t. $Prog$) iff in its body there is an atom $A$ whose predicate symbol is a defined predicate of $Prog$, and $A$ cannot be unified with any head in $Prog$.

In the trees of clauses we will consider below, we will use the standard irreflexive relations of *son*, *brother*, and *ancestor* of a node in a tree.

**Definition 2.** Let $Prog$ be a program, $C$ a clause, and $S$ a U-selection rule. An *Unfolding-tree* (U-tree, for short) for $\langle Prog, C \rangle$ via $S$ is a tree labeled by clauses and constructed as follows:

a) the root is labeled by the clause $C$, and

b) let $M$ be a node labeled by a nonfailing clause of the form: $H \leftarrow A_1, \ldots, A_h, \ldots, A_n$, and let $A_h$ be the defined atom selected by the U-selection rule $S$.

For each clause $A \leftarrow B_1, \ldots, B_s$ in $Prog$ such that there exists a most general unifier $\sigma$ of $A$ and $A_h$, $M$ has a son-node $N$ labeled by the clause $(H \leftarrow A_1, \ldots, A_{h-1}, B_1, \ldots, B_s, A_{h+1}, \ldots, A_n)\sigma$.

We will assume that the rule $S$ is a *partial* function which is uniquely determined by the set of defined atoms in the body of the clauses. Clauses for which $S$ is *not* defined are leaves of the U-tree. Success clauses and failing clauses can only be leaves of the U-tree.

Obviously, a U-tree may be infinite.

**Definition 3.** Given a tree $T$, we say that a *nonempty* tree $R$ is an *upper portion* of $T$ iff: a) the set of nodes of $R$ is contained in the set of nodes of $T$, b) if a node $N$ is in $R$, then also every ancestor of $N$ in $T$ is in $R$, and c) if a node $N$ is in $R$, then also every brother of $N$ in $T$ is in $R$.

The tree of clauses depicted in Figure 1 is an upper portion of the U-tree for the program Even\_npderiv and the clause 1.1 of Example 1. The atoms selected for unfolding are the underlined ones.
One can easily show that given a program \( \text{Prog} \), a clause \( C \), and the set \( L \) of leaves of an upper portion of the U-tree for \( \langle \text{Prog}, C \rangle \) via any U-selection rule, \( \text{Prog} \cup \{ C \} \) is equivalent to \( \text{Prog} \cup L \) w.r.t. any given goal.

**Definition 4.** Let \( \text{Prog} \) be a program, \( C \) a clause, and \( S \) a U-selection rule. A clause \( F \) in a U-tree for \( \langle \text{Prog}, C \rangle \) via \( S \) is said to be **foldable** iff there is an ancestor-clause \( A \) of \( F \) in the U-tree such that there exists a **subset of the defined atoms** in the body of \( F \) which is an **instance** of the set of all **defined atoms** in the body of \( A \).

The clause in the path from the root to \( F \), which is the **nearest** to the root satisfying the properties of \( A \), will be called the **oldest subsuming ancestor of** \( F \).

**Definition 5.** Let \( \text{Prog} \) be a program, \( C \) a clause, and \( S \) a U-selection rule. The U-tree for \( \langle \text{Prog}, C \rangle \) via \( S \) is said to be **foldable** iff it has a **finite** upper portion such that each leaf-clause is either: a) a success clause w.r.t. \( \text{Prog} \), or b) a failing clause w.r.t. \( \text{Prog} \), or c) a foldable clause. That portion will be called a **foldable upper portion** of the U-tree.

We will often consider the **minimal** foldable upper portion of a U-tree, that is, a foldable upper portion which contains no foldable upper portions different from itself.

With reference to the List Derivative example, the minimal foldable upper portion of a U-tree for \( \langle \text{Even\_Npderiv}, \text{clause 1.1} \rangle \) is depicted in Figure 1.

We will now present the Loop Absorption Procedure, which realizes the loop absorption strategy.

**Loop Absorption Procedure.**

**Input.** A program \( \text{Prog} \), a definition clause \( D \), a U-selection rule \( S \), and the minimal foldable upper portion \( T \) of a U-tree for \( \langle \text{Prog}, D \rangle \) via \( S \).

**Output.** A set of clauses \( \text{TransfD} \) such that \( \text{Prog} \cup \{ D \} \) is equivalent to \( \text{Prog} \cup \text{TransfD} \) w.r.t. all goals involving predicates occurring in \( \text{Prog} \cup \{ D \} \).

1) Construct the set \( \text{DEF} \) of definition clauses which define the **eureka predicates** as follows. Set \( \text{DEF} \) to \( \{ D \} \). For every foldable clause \( P \) of \( T \), consider its oldest subsuming ancestor, say \( A \).

   Let \( A1 \) be a clause whose head predicate is a fresh symbol \( \text{newp} \) and whose body consists of the set \( B \) of defined atoms in the body of \( A \). The set of variables which are arguments of \( \text{newp} \) is the minimal set \( V \) such that both \( A \) and \( P \) can be folded w.r.t. \( A1 \).

   Add \( A1 \) to the set \( \text{DEF} \) unless there exists in \( \text{DEF} \) a clause, say \( A2 \), which differs from a variant of \( A1 \) only because of the name of the head predicate and/or the order of the arguments in the head.

2) For every clause \( E \) in \( \text{DEF} \), compute the recursive definition \( R_E \) of its head predicate as follows.

   a) Construct the minimal upper portion \( T_E \) of a U-tree for \( \langle \text{Prog}, E \rangle \) via \( S \) such that each leaf of \( T_E \) is either a success clause or a failing clause or it can be folded w.r.t. a clause in \( \text{DEF} \).
b) Collect all nonfailing clauses at the leaves of $T_E$, and perform on those clauses all possible folding steps w.r.t. clauses in $DEF$, thereby obtaining the set of clauses $R_E$.

3) Define $TransfD$ to be $\{R_E|E \in DEF\}$. □

Remarks. In the construction of $T_E$ at point 2a), we use the same selection rule $S$ used for $T$. Since we have assumed that the atom selected by $S$ in a clause is uniquely determined by the set of defined atoms in the body of that clause, we have that the tree $T_E$ can be constructed in a finite number of unfolding steps. Indeed, $T$ is a finite tree, and $T_E$ is a “replica” of an upper portion of the subtree of $T$ rooted in the oldest subsuming ancestor which generates the eureka definition $E$. The differences between $T_E$ and the corresponding portion of $T$ may only be in the heads and in the base atoms of the clauses. Thus, the Loop Absorption Procedure always terminates.

When performing a folding step at point 2b), we may prevent a different folding because the corresponding recurrent patterns may overlap. Thus, by “all possible folding steps” we actually mean “any maximal sequence of folding steps.”

After the application of the Loop Absorption Procedure, we may have the opportunity of improving the derived program $Prog \cup TransfD$ by deleting its sterile clauses. They are defined as follows.

**Definition 6.** Let $Prog$ be a program. For each clause $C$ in $Prog$ of the form $p_0(\ldots) \leftarrow p_1(\ldots), \ldots, p_n(\ldots)$, where $0 \leq n \leq r$ and $p_0, p_1, \ldots, p_n$ are the defined predicates in $C$, we generate the new clause $D$ of the form $p_0 \leftarrow p_1, \ldots, p_n$.

Let $ProgI$ be the set of all clauses generated in this way from $Prog$. We say that $p$ is a sterile predicate iff there is not an SLD-refutation of $ProgI \cup \{\leftarrow p\}$. A clause in $Prog$ is sterile iff a sterile predicate occurs in it.

**Fact 7.** Sterile clauses can be deleted without affecting the set of correct answer substitutions. □

Some more simplification steps can be performed on $ProgI \cup TransfD$. For instance, we can simplify equalities, arithmetic predicates (when the operands are fully instantiated), and we can eliminate any clause $C$ such that no predicate in $Prog \cup \{D\}$ depends on the head of $C$. (For the dependency relation between predicates, see, for instance, [15].)

**Example 2: List derivative revisited.** The program derivation we have shown in the introductory Example 1 is an application of the Loop Absorption Procedure. Let us see how the procedure works in that case. The minimal foldable upper portion of a U-tree for $Even_{nderv}$ and clause 1.1 is the one shown in Figure 1.

a) The set of eureka definitions $DEF$ is made out of the clauses:

1.1. $even_{nderv}(L, E) \leftarrow nderv(L, D), even(D, E)$.

d1. $new1(Y, T) \leftarrow nderv([Y|T], [\ ])$.

d2. $new2(X, T, Y, E) \leftarrow nderv([X|T], [Y|D]), even(D, E)$.

b) The tree $T_{11}$ is the upper portion of the U-tree depicted in Figure 1 whose leaves are the clauses 6, 7, 8, 9, and 5. The trees $T_{d1}$, $T_{d2}$ are depicted in Figures 2 and 3, respectively.
3. THE FOLDABILITY PROBLEM AND ITS SOLUTION FOR SOME CLASSES OF PROGRAMS

As we have shown in the previous sections, our transformation technique requires the construction of foldable U-trees. In this section, we will address the problem of determining some classes of programs for which those foldable U-trees can effectively be found.

We formalize the foldability problem as follows. Given a program $Prog$ and a definition clause $D$, is there a U-selection rule $S$ such that the U-tree for $(Prog, D)$ via $S$ is foldable?

**Theorem 8.** The foldability problem is not solvable, but partially solvable.

**Proof.** Let $Prog$ be a program and $D$ a definition clause. For each integer $k$, there is only a finite set, say $S_k$, of trees which have height $k$ and which are upper portions of U-trees for $(Prog, D)$. Thus, a semi-decision procedure for the foldability problem is given by the algorithm which generates the sets $S_1, S_2$, and so on, until it finds an $S_k$ containing a foldable upper portion.

We will show that the halting problem of Turing machines can be reduced to the foldability problem. (A simpler proof of this theorem can be based on the reduction of the termination problem of logic programs to the foldability problem. However, our proof has the advantage of being suitable for showing Theorem 11 as well.)
For each deterministic Turing machine $M$ and for each word $w$ over the input alphabet, we can construct a logic program $ProgM$ and a clause $D_w$ such that $M$ halts on the input $w$ (accepting or refusing it) iff there exists a U-selection rule $S$ such that the U-tree for $\langle ProgM, D_w \rangle$ via $S$ is foldable.

$ProgM$ is constructed as follows. We assume the definition of the Turing machine as in [12]. In particular, the tape is considered to be a semi-infinite sequence of cells which has a leftmost cell.

Consider the Turing machine $M = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$, where $Q$ is the set of states, $\Sigma$ is the alphabet, $\Gamma$ is the tape alphabet, $\delta$ is the transition function, $q_0 \in Q$ is the initial state, and $F$ is the set of final states. Suppose that $b$, $L$, and $R$ are the symbols denoting "blank", move-left", and "move-right", respectively, and they are not in $\Sigma$. Without loss of generality, we may assume that $\Gamma = \Sigma \cup \{ b \}$ and the machine $M$ does not print the blank symbol. Recall that, initially, the $k$ leftmost cells for a finite $k$ hold the input word in $\Sigma^*$, and all cells to the right of them hold blank symbols. We also assume that, initially, the tape-head is scanning the leftmost cell of the tape.

As a consequence of the fact that at each move $M$ prints a nonblank symbol, the nonblank symbols are always contiguous, and they fill a leftmost segment of the tape. During the computation, the tape-head scans either a nonblank cell or the cell immediately to the right of the rightmost nonblank cell.

For the construction of $ProgM$, we use: 1) the constants in $Q \cup \Sigma \cup \{ b, \text{nil}, 0 \}$, 2) the function symbols $\text{cons}$ and $\text{succ}$, and 3) the predicate symbol $\text{config}$.

The clauses of $ProgM$, describe the moves of $M$ from a configuration to the next one. Each clause has the form

$$\text{config}(q, \text{lefttape}, \text{symbol}, \text{righttape}, N) \leftarrow \text{config}(nextq, \text{nextlefttape}, \text{nextsymbol}, \text{nextrighttape}, \text{succ}(N)).$$

The head and the body of the clause represent the configurations of the machine before and after the move, respectively. The first argument of the predicate $\text{config}$ holds the state of $M$. The triple of terms $\langle \text{lefttape}, \text{symbol}, \text{righttape} \rangle$ represents the tape of $M$ before the move. In particular, suppose that the tape consists of the sequence of symbols: $\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \sigma_{k+1}, \ldots, \sigma_n, b, b, \ldots$, and suppose that the head-tape is on the $k$th cell. We have that

- $\text{lefttape}$ is the list $[\sigma_{k-1}, \ldots, \sigma_1]$ of the symbols on the tape in the reverse order, that is, the term $\text{cons}(\sigma_{k-1}, \ldots, \text{cons}(\sigma_1, \text{nil}) \ldots)$,
- $\text{symbol}$ is $\sigma_k$, and
- $\text{righttape}$ is the list $[\sigma_{k+1}, \ldots, \sigma_n]$.

Analogously, the triple of terms $\langle \text{nextlefttape}, \text{nextsymbol}, \text{nextrighttape} \rangle$ represents the tape after the move. The last argument of $\text{config}$ holds the number of moves done by the machine $M$.

In particular, for each transition $\delta(q_1, \sigma_1) = \langle q_2, \sigma_2, L \rangle$ (from state $q_1$ reading symbol $\sigma_1$ $M$ goes to state $q_2$, prints symbol $\sigma_2$, and the tape-head moves left), $ProgM$ has the clause

$$\text{config}(q_1, [\text{HeadLeftTape|TailLeftTape}], \sigma_1, \text{RightTape}, N) \leftarrow \text{config}(q_2, \text{TailLeftTape, HeadLeftTape}, [\sigma_2|\text{RightTape}], \text{succ}(N)).$$
For each transition $\delta(q, \sigma, R) = (q', \sigma', R')$, $ProgM$ has two clauses:

$$config(q, \text{LeftTape}, \sigma, [\text{HeadRightTape}\|\text{TailRightTape}], N) \leftarrow$$

$$config(q', [\sigma'], \text{LeftTape}], \text{HeadRightTape}, \text{TailRightTape}, \text{succ}(N)).$$

and

$$config(q, \text{LeftTape}, \sigma, [\phantom{1}], N) \leftarrow config(q', [\sigma'], \text{LeftTape}], b, [\phantom{1}], \text{succ}(N)).$$

For each final state $q_F \in F$, $ProgM$ has a unit clause:

$$config(q_F, \text{LeftTape}, \text{Symbol}, \text{RightTape}, N).$$

The clause $D_w$, which corresponds to the initial configuration of $M$, is

$$start \leftarrow config(q_0, [\phantom{1}], s, t, 0).$$

where $start$ is a new predicate symbol, and

1) if $w$ is empty, then $s = b$ and $t = [\phantom{1}]$, and
2) if $w = \sigma u$ with $\sigma \in \Sigma$ and $u \in \Sigma^*$, then $s = \sigma$ and $t$ is the list representing the word $u$.

Let us consider a U-tree $T_S$ for $\langle ProgM, D_w \rangle$ via a U-selection rule $S$. Since there is at most one atom in the body of any clause of $ProgM$, the body of each clause in the U-tree $T_S$ has at most one atom.

Let us consider the U-selection rule $L$ which returns the leftmost atom in the body of any given clause $C$, if $C$ is neither a success nor a failing clause. We have that the U-tree $T_S$ is an upper portion of the U-tree $T_L$ for $\langle ProgM, D_w \rangle$ via $L$. Indeed, $S$ may be undefined for some clause in $T_S$, and therefore, that clause will be a leaf of $T_S$.

Thus, there exists a foldable U-tree for $\langle ProgM, D_w \rangle$ via some U-selection rule iff the U tree $T_L$ is foldable. Now, notice that no clause can be foldable in this U-tree, because the last argument of $config$ is a ground term which grows at each unfolding step. Therefore, there exists a foldable U-tree for $\langle ProgM, D_w \rangle$ via some U-selection rule iff the U-tree $T_L$ is finite.

With every configuration $\gamma$ of the machine $M$ on input $w$, we can associate a clause $C_\gamma$ in $T_L$ of the form:

$$start \leftarrow config(q, \text{lefttape}, \sigma, \text{righttape}, n)$$

where $\langle \text{lefttape}, s, \text{righttape} \rangle$ is the triple of terms representing the tape of the configuration, as described above. We have that the machine $M$ moves from configuration $\gamma$ to configuration $\delta$ iff the clause $C_\delta$ is the son of the clause $C_\gamma$. Since $M$ is deterministic, each clause in $T_L$ has at most one son clause, and therefore the U-tree has one leaf only, if any.

$C_\gamma$ is a leaf of $T_L$ iff $\gamma$ is a configuration with no successor. Thus, the U-tree is finite iff $M$ halts on input $w$.

We will now show that the foldability problem is also unsolvable for a restricted classes of programs.

**Definition 9.** A term, an atom, or a conjunction of atoms is *linear* iff each variable occurs in it at most once.
Definition 10. A clause \( C \) of a program \( \text{Prog} \) is said to be an atom-linear clause (or AL-clause) iff each defined atom in \( C \) is linear. If an AL-clause is a definition clause, we will simply say that it is an AL-definition. \( \text{Prog} \) is said to be an atom-linear program (or AL-program) iff it is made out of AL-clauses only.

An AL-clause \( C \) of a program \( \text{Prog} \) is said to be a head-body-linear clause (or HBL-clause) iff the conjunction of the defined atoms in its body is linear.

\( \text{Prog} \) is said to be a head-body-linear program (or HBL-program) iff it is made of HBL-clauses only.

Example 3. The following clause of the \text{Even\_Npderiv} program (see Example 1) is an AL-clause:

\[
\text{npderiv}([X,Y], D) \leftarrow X < Y, \text{npderiv}([Y|T], D).
\]

The above clause is also an HBL-clause (assuming that \( < \) is a base predicate). The clause

\[
p(X,Y) \leftarrow p(X), p(Y).
\]

is an HBL-clause. The clause

\[
\text{npderiv}([X,Y|T],[X|D]) \leftarrow X \geq Y, \text{npderiv}([Y|T], D).
\]

is not an AL-clause because the variable \( X \) occurs twice in its head. However, it can be transformed into an equivalent AL-clause by using the equality predicate (to be considered as a base predicate) as follows:

\[
\text{npderiv}([X,Y|T],[X|D]) \leftarrow X \geq Y, X = X_1, \text{npderiv}([Y|T], D).
\]

Theorem 11. The foldability problem for HBL-programs is not solvable.

Proof. The halting problem of Turing machines can be reduced to the foldability problem for HBL-programs. Indeed, the program \( \text{ProgM} \) given in the proof of Theorem 8 is an HBL-program without base predicates.

In the remaining part of this section, we introduce the class of nonascending programs, and we show that foldable U-trees do exist for suitable subclasses of those programs.

Definition 12. Let \( X \) be a variable or a constant and let \( t \) be a term where \( X \) occurs. The depth of \( X \) in \( t \), denoted by \( \text{depth}(X,t) \), is defined by structural induction as follows:

- if \( t = X \), then \( \text{depth}(X,t) = 0 \), and
- if \( f \) is an \( n \)-ary function symbol and \( t = f(t_1, \ldots, t_n) \),
  then \( \text{depth}(X,t) = \max(\text{depth}(X,t_i)) \) \( X \) occurs in \( t_i \) and \( i = 1, \ldots, n \) + 1.

We denote by \( \text{height}(t) \) the value of \( \max(\text{depth}(X,t)) \) \( X \) is a variable or a constant occurring in \( t \). We denote by \( \text{vars}(t) \) the set of variables occurring in \( t \). We denote by \( \text{vars}(t) \cap \text{vars}(u) \) we have \( \text{depth}(X,t) \leq \text{depth}(X,u) \).

Notice that if \( t = f(t_1, \ldots, t_n) \) is a linear term and \( X \) is a variable occurring in \( t \), then the set \( \{\text{depth}(X,t_i) \} \) \( X \) occurs in \( t_i \) and \( i = 1, \ldots, n \) is a singleton.

The above definitions of \( \text{depth} \), \( \text{height} \), and \( \text{vars} \) and the definition of the \( \leq \) relation can be extended from terms to atoms by considering the predicate symbols as term constructors.
For proving a sufficient condition for the existence of foldable U-trees, we need a weak version of the Kruskal Tree Theorem [16], which is known as the Higman Lemma [10].

Given two finite words $u$ and $w$ over a finite alphabet, we say that $u$ is a subword of $w$ if $u$ can be obtained by deleting some occurrences of symbols in $w$.

**Lemma 13 (Higman).** If $\{w_i|i \geq 0\}$ is an infinite sequence of finite words over a finite alphabet, then there exist two indexes $i$ and $j$ such that $i < j$ and $w_i$ is a subword of $w_j$.

Suppose that $A_1$ and $A_2$ are two atoms such that $\text{vars}(A_1) \cap \text{vars}(A_2) \neq \emptyset$; then we will write $A_1 \sim A_2$. Given a clause $C$, we consider the reflexive and transitive closure $\approx$ of the relation $\sim$ on the defined atoms in the body of $C$. We denote by $\text{Part}(C)$ the partition of the set of defined atoms in the body of $C$ induced by the equivalence relation $\approx$.

**Theorem 14.** Let $\text{Prog}$ be a program, $D$ a definition, and $S$ a U-selection rule. Let us consider the U-tree $T$ for $\langle \text{Prog}, D \rangle$ via $S$. Suppose that $S$ is defined for all clauses in $T$, apart from success clauses or failing clauses, and that there exist two positive integers $H$ and $W$ such that for each clause $C$ in $T$,

a) $\max\{\text{height}(A) | A$ is a defined atom of the body of $C\} \leq H$, and

b) every sequence $\langle A_1, A_2, \ldots, A_m \rangle$ of different defined atoms of the body of $C$ such that $A_i \sim A_{i+1}$ for $i = 1, \ldots, m - 1$, is not longer than $W$, that is, $m \leq W$.

Then $T$ is a foldable U-tree.

**Proof.** By hypotheses a) and b), there exists a positive integer $\text{MaxN}$, depending on $H$ and $W$ only, such that, for each clause $C$ in the U-tree $T$, each block of equivalent atoms in $\text{Part}(C)$ has at most $\text{MaxN}$ atoms.

Recalling hypothesis a) and the fact that the set of predicate, function, and constant symbols occurring in $\text{Prog} \cup \{D\}$ from which we may construct the blocks in $\text{Part}(C)$ is finite, we have that each of those blocks is a variant of an element of a finite set $B$ of finite sets of atoms. The cardinality of $B$ depends on $H$, $W$, and on the number of predicate, function, and constant symbols occurring in $\text{Prog} \cup \{D\}$.

With each element of $B$ we associate a fresh new symbol $\sigma$. Let $\Sigma$ be the set of all such $\sigma$'s. Since $|\Sigma| = |B|$, $|\Sigma|$ is finite. Thus, with each clause and with each path of the U-tree, we can associate a word of $\Sigma^*$ and a sequence of words of $\Sigma^*$, respectively. By the Higman Lemma, if one of those sequences is infinite, then it has two elements, say $u$ and $w$, such that $u$ precedes $w$, and $u$ is a subword of $w$. Therefore, the word $w$ is associated with a foldable clause.

In the sequel, the sequences of different defined atoms of hypothesis b) of the above Theorem 14 will be called variable-chained sequences.

**Remark.** Condition b) of Theorem 14 cannot be dropped. Indeed, let us consider the case where $p(X,Y) \leftarrow p(X,Z), q(Z,Y)$ is in $\text{Prog}$, and $D$ is the clause $h(X,Y) \leftarrow p(X,Y), r(Y)$.

Suppose that $p$, $q$, and $r$ are defined predicates in $\text{Prog}$. By unfolding the leftmost atom in the body of any clause of the U-tree for $\langle \text{Prog}, D \rangle$, we get an infinite path consisting of the following clauses:

1. $h(X,Y) \leftarrow p(X,Y), r(Y)$. (which is clause $D$),
2. \( h(X,Y) \leftarrow p(X,Y_1), q(Y_1,Y), r(Y) \).
3. \( h(X,Y) \leftarrow p(X,Y_2), q(Y_2,Y_1), q(Y_1,Y), r(Y) \).

In this path, hypothesis a) of Theorem 14 is satisfied, but hypothesis b) is not. It is easy to see that no clause of the path is foldable.

Theorem 14 will be used below for showing the existence of foldable U-trees for some classes of programs which we now define.

**Definition 15.** Let \( \text{Prog} \) be an AL-program, and \( C \) a clause of \( \text{Prog} \) of the form

\[
H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n,
\]

where \( A_1, \ldots, A_m \) are the defined atoms.

The clause \( C \) is said to be **nonascending** iff \( A_i \leq H \) for \( i = 1, \ldots, m \). The program \( \text{Prog} \) is said to be nonascending iff all of its clauses are nonascending.

The clause \( C \) is said to be **strongly-nonascending** iff it is a nonascending HBL-clause. The program \( \text{Prog} \) is said to be strongly nonascending iff all of its clauses are strongly nonascending.

**Example 4.** The following clause of the Even_Npderiv program (see Example 1) is nonascending:

\[
\text{npderiv}( [X,Y,T], D ) \leftarrow X < Y, \text{npderiv}( [Y,T], D ).
\]

The above clause is also strongly-nonascending.

**Definition 16.** A **Synchronized Descent Rule** (simply called SDR) for an AL-program \( \text{Prog} \) is a partial function \( S \) from clauses to atoms which satisfies the following condition:

\[
(\mu) \text{ let } C \text{ be an AL-clause of the form } H \leftarrow \text{Body} \text{ and } \{A_1, \ldots, A_n\} \text{ be the set of defined atoms (w.r.t. } \text{Prog}) \text{ in } \text{Body}; \text{ if there exists } i \text{ in } \{1, \ldots, n\} \text{ such that } A_j \leq A_i \text{ for } j = 1, \ldots, n, \text{ then } S(C) = A_i, \text{ otherwise } S(C) \text{ is undefined.}
\]

In general, the choice of \( A_i \) for which \( S \) is an SDR is not unique.

For simplicity reasons, in the sequel we will refer to "the" SDR instead of "an" SDR when our statements are valid for any choice of SDR. However, different choices may determine different U-trees, and consequently, the programs obtained using the Loop Absorption Procedure may be different. We will not discuss this problem here.

**Lemma 17.** Let \( \text{Prog} \) be a nonascending program and \( C \) an AL-clause such that the SDR is defined for it. Let \( D \) be a clause which is obtained by unfolding \( C \) w.r.t. \( \text{Prog} \), according to the SDR. Then \( D \) is an AL-clause, and for each defined atom \( A \) in the body of \( D \), we have that

\[
\text{height}(A) \leq \max \{\text{height}(B) \mid B \text{ is a defined atom in } \text{Prog} \cup \{C\}\}.
\]

**Proof.** Let \( E \) be the clause of \( \text{Prog} \) which is used to produce \( D \). Suppose that \( C \) and \( E \) have the forms \( H \leftarrow A_1, \ldots, A_m, \ldots, A_n \) and \( K \leftarrow B_1, \ldots, B_n, \ldots, B_q \), respectively, where \( A_1, \ldots, A_m \) and \( B_1, \ldots, B_n \) are the defined atoms. We can assume that \( C \) and \( E \) do not share variables. Suppose also that \( A_i \) is the atom in \( \{A_1, \ldots, A_m\} \)
which is selected by $S$. Thus, $D$ has the form

$$(H \leftarrow A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m, \ldots, A_p)\sigma,$$

where $\sigma$ is the most general unifier of $A_i$ and $K$.

The fact that $D$ is an AL-clause is an immediate consequence of the hypothesis that both $C$ and $E$ are AL-clauses.

We will prove the remaining part of the thesis by showing that we have

a) for $r = 1, \ldots, m$:

$$\text{height}(A, \sigma) \leq \max\{\text{height}(A), \text{height}(K), \text{height}(A_r)\},$$

b) for $r = 1, \ldots, n$:

$$\text{height}(B, \sigma) \leq \max\{\text{height}(A), \text{height}(K), \text{height}(B_r)\}.$$

**Proof of a).** Let $\sigma$ be $(X_1/t_1, \ldots, X_n/t_n)$. We have that

$$\text{vars}(A) \cap \{X_1, \ldots, X_n\} \subseteq \text{vars}(A_r).$$

Now, if $\text{height}(A, \sigma) > \text{height}(A_r)$, then there exist $X_i \in \text{vars}(A_r)$ and a binding $X_j/t_i \in \sigma$ such that

$$\text{height}(A, \sigma) = \text{depth}(X_i, A) + \text{height}(t_i)$$

$$\leq \text{depth}(X_i, A_r) + \text{height}(t_i)$$

because (†) holds and $S$ is an SDR.

$$\leq \text{height}(A_r, \sigma).$$

Thus, we have that

$$\text{height}(A, \sigma) \leq \max\{\text{height}(A), \text{height}(A_r)\}.$$ 

By the hypothesis that $A_i$ and $K$ are linear atoms without common variables, we have that

$$\text{height}(A, \sigma) = \max\{\text{height}(A), \text{height}(K)\}.$$ 

The proof of b) is analogous to the above one (recall that $B_r \leq K$ for $r = 1, \ldots, n$). □

The following result ensures that, in the class of programs and clauses we will consider, the Loop Absorption Procedure is applicable, and we can automatically derive the auxiliary eureka predicates which are necessary for synthesizing efficient programs via transformation.

**Theorem 18.** Let $\text{Prog}$ be an HBL-program, $D$ an HBL-definition clause, and $S$ a $U$-selection rule which is defined for each clause, apart from success or failing clauses. Then the $U$-tree for $\langle \text{Prog}, D \rangle$ via $S$ is foldable.

**Proof.** By unfolding an HBL-clause using an HBL-clause, we get an HBL-clause. Thus, in the body of any clause $C$ in the $U$-tree, two defined atoms do not share any variables, that is, any variable-chained sequence has length 1. Therefore, $S$ is
an SDR and, by Lemma 17, for each defined atom $A$ in the body of $C$, we have that

$$\text{height}(A) \leq \max\{\text{height}(B) | B \text{ is a defined atom in } \text{Prog} \cup \{C\}\}.$$  

The thesis follows from Theorem 14. \qed

Now we want to consider the case where the initial program $\text{Prog}$ is strongly-nonascending, but the initial definition $D$ is not an HBL-clause, and thus we cannot make use of Theorem 18 above. This situation often occurs when we apply the tupling strategy as described in Section 1, and we introduce a new clause in whose body the atoms share some variables. In order to construct a foldable U-tree for $\langle \text{Prog}, D \rangle$ in that case as well, we may often use the following Theorem 20 which ensures the existence of a foldable U-tree for $\langle \text{Prog}, D \rangle$ under some suitable conditions.

**Definition 19.** Let $\text{Prog}$ be a program and $p_0$ an $n$-ary predicate in $\text{Prog}$. We say that $p_0$ is defined by induction on position $k_0$ ($\leq n$) iff for each clause in $\text{Prog}$ of the form

$$p_0(\ldots, t_0, \ldots) \leftarrow p_1(\ldots), \ldots, p_m(\ldots), \ldots, p_s(\ldots),$$

where $t_0$ is in position $k_0$ and $p_1, \ldots, p_m$ only are the defined predicates, we have that for each $i = 1, \ldots, m$, there exists a unique argument position $k_i$ such that

- the $k_i$th argument of $p_i$, say $t_i$, is a subterm of $t_0$ (the subterm relation is assumed to be reflexive), and all other arguments of $p_i(\ldots)$ do not have any variable in common with $t_0$,
- $p_i$ is defined by induction on position $k_i$.

The term $t_i$ occurring in the atom $p_i(\ldots, t_i, \ldots)$ at position $k_i$ is called the **inductive argument** of that atom. \qed

For instance, both predicates $\text{npderiv}$ and $\text{even}$ of Example 1 are defined by induction on positions 1 and 2.

**Theorem 20.** Let $\text{Prog}$ be a strongly-nonascending program and $D$ an AL-definition clause of the form $h(\ldots) \leftarrow a(\ldots, t, \ldots), b(\ldots, u, \ldots)$, where $t$ and $u$ occur at positions $h$ and $k$, respectively. Assume that

1. either $t$ is a subterm of $u$, or $u$ is a subterm of $t$,
2. $\text{vars}(a(\ldots, t, \ldots)) \cap \text{vars}(b(\ldots, u, \ldots)) = \text{vars}(t) \cap \text{vars}(u)$, and
3. $a$ and $b$ are defined by induction on positions $h$ and $k$, respectively.

If $S$ is an SDR then the U-tree for $\langle \text{Prog}, D \rangle$ via $S$ is foldable.

**Proof.** For expository purposes, let us assume that $h = k = 1$. Let us also assume that each predicate in $\text{Prog}$ is defined by induction on the first position. The general case is analogous. As in the proof of Theorem 14, let us consider the reflexive and transitive closure $\sim$ of the relation $\cdot$. For each clause $C$ in the U-tree, we also consider: 1) the partition $\text{Part}(C)$ of the set of defined atoms
in the body of C induced by \( \approx \), and 2) the following property \( \Pi(C) \):

For each block \( \{ p_1(t_1, \ldots), \ldots, p_m(t_m, \ldots) \} \) of \( \text{Part}(C) \), we have that:

a) each atom in the block is linear (that is, \( C \) is an AL-clause),

b) for each pair of indexes \( i \) and \( j \), \( \text{vars}(p_i(t_i, \ldots)) \cap \text{vars}(p_j(t_j, \ldots)) \) is equal to \( \text{vars}(t_i) \cap \text{vars}(t_j) \),

c) in the multiset \( \text{IndArgs} = \{ t_1, \ldots, t_m \} \) of the first arguments of \( p_1, \ldots, p_m \),

there exists a maximum, say \( \text{max}_t \), s.t. for \( i = 1, \ldots, m \), \( t_i \) is a subterm of \( \text{max}_t \),

d) if we assume that \( \text{max}_t \) is the term \( t_k \), then any two terms \( t_i \) and \( t_j \) in \( \text{IndArgs} \) s.t. \( i \not= j \), \( i \not= k \), and \( j \not= k \) have disjoint sets of variables.

We will show that property \( \Pi(C) \) holds for every clause \( C \) in the U-tree for \( \langle \text{Prog}, D \rangle \) via any U-selection rule. The proof is by induction on the construction of the U-tree.

**Basis.** \( \Pi(D) \) holds by hypothesis.

In particular, condition d) trivially holds because \( |\text{IndArgs}| = 2 \).

**Induction.** Suppose that property \( \Pi(C) \) holds for a clause \( C \) in the U-tree for \( \langle \text{Prog}, D \rangle \). We have to prove that \( \Pi \) holds for every son-clause of \( C \). If \( S \) is not defined for \( C \), then \( C \) is a leaf, and the thesis trivially holds. Otherwise let \( A \) be the atom selected by \( S \) for unfolding. Let \( \{ p_1(t_1, \ldots), \ldots, p_m(t_m, \ldots) \} \) be the block of \( \text{Part}(C) \) to which the atom \( A \) belongs. Without loss of generality, we may assume that \( A \) is \( p_i(t_1, \ldots) \).

Let \( p_j(s_1, \ldots, r_n(s_1, \ldots), \ldots, r_n(s_n, \ldots)) \) be the program clause used for unfolding, where \( r_1, \ldots, r_n \) are the defined predicates. We may assume that this clause does not have any variable in common with \( C \). By hypothesis iii), we have that \( s_1, \ldots, s_n \) are subterms of \( s \). Let \( \sigma \) be the most general unifier of \( p_i(t_1, \ldots) \) and \( p_j(s_1, \ldots) \). Let \( C I \) be the clause obtained by unfolding. We have to show that \( \Pi(CI) \) holds.

We have that

\[
\text{Part}(CI) = (\text{Part}(C) - \{ p_i(t_1, \ldots), \ldots, p_m(t_m, \ldots) \}) \cup \\
\text{Part}\{ r_1(s_1, \ldots)\sigma, \ldots, r_n(s_n, \ldots)\sigma, p_2(t_2, \ldots)\sigma, \ldots, p_m(t_m, \ldots)\sigma \}
\]

because the sets of variables in

\[
(\text{Part}(C) - \{ p_i(t_1, \ldots), \ldots, p_m(t_m, \ldots) \}) \text{ and in } \\
\{ r_1(s_1, \ldots)\sigma, \ldots, r_n(s_n, \ldots)\sigma, p_2(t_2, \ldots)\sigma, \ldots, p_m(t_m, \ldots)\sigma \}
\]
are disjoint.

For the blocks in \( \text{Part}(C) - \{ p_i(t_1, \ldots), \ldots, p_m(t_m, \ldots) \} \) properties a), b), and d) hold by inductive hypothesis. Let us complete the proof for the blocks in

\[
\text{Part}\{ r_1(s_1, \ldots)\sigma, \ldots, r_n(s_n, \ldots)\sigma, p_2(t_2, \ldots)\sigma, \ldots, p_m(t_m, \ldots)\sigma \}.
\]

**Point a.** It follows from the inductive hypothesis a) and the fact that by unfolding an AL-clause using an HBL-clause we get an AL-clause.

**Point b.** Let us consider the atoms \( r_i(s_i, \ldots)\sigma, p_j(t_j, \ldots)\sigma, p_k(t_k, \ldots)\sigma \), where \( 1 \leq i \leq n, 2 \leq j, k \leq m, j \not= k \). We have to show that

b1) \( \text{vars}(r_i(s_i, \ldots)\sigma) \cap \text{vars}(p_j(t_j, \ldots)\sigma) = \text{vars}(s_i\sigma) \cap \text{vars}(t_j\sigma) \), and

b2) \( \text{vars}(p_j(t_j, \ldots)\sigma) \cap \text{vars}(p_k(t_k, \ldots)\sigma) = \text{vars}(t_j\sigma) \cap \text{vars}(t_k\sigma) \).
By the hypothesis iii), and by the inductive hypotheses a) and b), we have that
the following six sets of variables are pairwise disjoint:

\[ V_1 = \text{vars}(t_1) \cup \text{vars}(t_j) \cup \text{vars}(t_k). \]

\[ V_2 = \{ V \mid V \text{ occurs in an argument of } p_i(t_1, \ldots) \text{ different from } t_1 \} \]
\[ \quad - \text{vars}(p_i(t_1, \ldots)) - \text{vars}(t_1). \]

\[ V_3 = \{ V \mid V \text{ occurs in an argument of } p_j(t_j, \ldots) \text{ different from } t_j \} \]
\[ \quad = \text{vars}(p_j(t_j, \ldots)) - \text{vars}(t_j). \]

\[ V_4 = \{ V \mid V \text{ occurs in an argument of } p_k(t_k, \ldots) \text{ different from } t_k \} \]
\[ \quad = \text{vars}(p_k(t_k, \ldots)) - \text{vars}(t_k). \]

\[ W_1 = \text{vars}(s) \cup \text{vars}(s_i) = \text{vars}(s). \]

\[ W_2 = \{ V \mid V \text{ occurs in an argument of } p_i(s, \ldots) \text{ different from } s \}
\quad \text{or } V \text{ occurs in an argument of } r_i(s_i, \ldots) \text{ different from } s_i \}
\[ \quad = (\text{vars}(p_i(s, \ldots)) - \text{vars}(s)) \cup (\text{vars}(r_i(s_i, \ldots)) - \text{vars}(s_i)). \]

By the inductive hypothesis a), the atom \( p_i(t_1, \ldots) \) is linear. Since \( p_i(s, \ldots) \) is
the head of an HBL-clause, it is a linear atom. Thus, \( \sigma \) is the union of four disjoint
sets of bindings: \( \sigma_1 \) and \( \sigma_2 \) (which unify \( t_1 \) and \( s \)), and \( \sigma_3 \) and \( \sigma_4 \) [which unify the
remaining arguments of \( p_i(t_1, \ldots) \) and \( p_i(s, \ldots) \)], such that

\( \sigma_1 \) consists of bindings of the form \( X_1/f_1 \), where \( X_1 \in V_1 \) and \( \text{vars}(f_1) \subseteq W_1. \)

\( \sigma_2 \) consists of bindings of the form \( X_2/f_2 \), where \( X_2 \in W_1 \) and \( \text{vars}(f_2) \subseteq V_1. \)

\( \sigma_3 \) consists of bindings of the form \( X_3/f_3 \), where \( X_3 \in V_2 \) and \( \text{vars}(f_3) \subseteq W_2. \)

\( \sigma_4 \) consists of bindings of the form \( X_4/f_4 \), where \( X_4 \in W_2 \) and \( \text{vars}(f_4) \subseteq V_2. \)

Now, we have that

\[ \text{vars}(s_i) \subseteq W_1 \text{ and } \text{vars}(s_i, \sigma) \subseteq W_1 \cup V_1; \]

\[ \text{vars}(t_j) \subseteq V_1 \text{ and } \text{vars}(t_j, \sigma) \subseteq V_1 \cup W_1; \]

\[ \text{vars}(r_i(s_i, \ldots)) \subseteq \text{vars}(s_i) \cup W_2 \text{ and } \text{vars}(r_i(s_i, \ldots) \sigma) \]
\[ \subseteq \text{vars}(s_i, \sigma) \cup W_2 \cup V_2; \text{ and } \]

\[ \text{vars}(p_i(t_j, \ldots)) \subseteq \text{vars}(t_j) \cup V_3 \text{ and } \text{vars}(p_i(t_j, \ldots) \sigma) \subseteq \text{vars}(t_j, \sigma) \cup V_3. \]

Thus, we have that

\[ \text{vars}(r_i(s_i, \ldots) \sigma) \cap \text{vars}(p_i(t_j, \ldots) \sigma) \subseteq \text{vars}(s_i, \sigma) \cap \text{vars}(t_j, \sigma) \]
because

\[ (\text{vars}(r_i(s_i, \ldots) \sigma) \cap \text{vars}(p_i(t_j, \ldots) \sigma)) \]
\[ \subseteq ((\text{vars}(s_i, \sigma) \cup (W_2 \cup V_2)) \cap (\text{vars}(t_j, \sigma) \cup V_3)) \]
\[ = \text{vars}(s_i, \sigma) \cap \text{vars}(t_j, \sigma) \]
(use distributivity of $\cap$ over $\cup$ and the fact that

$$(W_2 \cup V_2) \cap V_3 = \text{vars}(s, \sigma) \cap V_3 = \text{vars}(t_j, \sigma) \cap (W_2 \cup V_2) = \emptyset.$$ 

Obviously, we also have that

$$\text{vars}(r_j(s_1, \ldots) \sigma) \cap \text{vars}(p_j(t_1, \ldots) \sigma) \supseteq \text{vars}(s, \sigma) \cap \text{vars}(t_j, \sigma).$$

Thus, point b1) is proved. Point b2) follows from the inductive hypothesis b) and from the assumption that the two clauses involved in the unfolding step do not share any variable. Therefore, point b) holds for clause CI.

**Point c.** Let $\text{max}_t$ be the maximum term in $\{t_1, \ldots, t_m\}$ w.r.t. the subterm ordering relation. We first show that after an unfolding step, $(\text{max}_t) \sigma$ is linear. If we unfold an atom which does not contain $\text{max}_t$, then $(\text{max}_t) \sigma$ is linear by point a). If we unfold an atom where $\text{max}_t$ occurs, then $(\text{max}_t) \sigma$ is a linear term because it is the most general common instance of $\text{max}_t$ and $s$, which are linear terms without common variables.

Moreover, by hypothesis iii), each element of the set $\{s_1 \sigma, \ldots, s_n \sigma\}$ is a subterm of $(\text{max}_t) \sigma$, and by inductive hypothesis c), each element of $\{t_2 \sigma, \ldots, t_m \sigma\}$ is a subterm of $(\text{max}_t) \sigma$. Therefore, two elements of $\text{IndArgs} = \{s_1 \sigma, \ldots, s_n \sigma, t_2 \sigma, \ldots, t_m \sigma\}$ are either comparable in the subterm ordering relation or they do not share any variable.

Thus, by point b), each block of

$$\text{Part}((r_j(s_1, \ldots) \sigma, \ldots, r_n(s_n, \ldots) \sigma, p_2(t_2, \ldots) \sigma, \ldots, p_m(t_m, \ldots) \sigma))$$

contains an atom whose inductive argument is the maximum among the inductive arguments of all atoms in that block. (Recall that two subterms of a linear term are either comparable in the subterm relation or they do not have common variables.)

**Point d.)** Let us say that two terms are disjoint iff their sets of variables are disjoint. We consider the following two cases d1 and d2.

**Case d1.** Suppose that the inductive argument $t_j$ of the atom $p_j(t_1, \ldots)$ selected by $S$ is a maximum in the multiset of inductive arguments $\{t_1, \ldots, t_m\}$. By hypothesis iii), we have that $s_i \sigma, \ldots, s_n \sigma$ are pairwise disjoint subterms of the linear term $s \sigma = t_i \sigma$. By inductive hypothesis, we also have that $t_2 \sigma, \ldots, t_m \sigma$ are pairwise disjoint subterms of $t_i \sigma$. Let $u$ be the inductive argument of an atom $A$ such that $u$ is the maximum among the inductive arguments of the atoms in the block, say B, of $\text{Part}((r_j(s_1, \ldots) \sigma, \ldots, r_n(s_n, \ldots) \sigma, p_2(t_2, \ldots) \sigma, \ldots, p_m(t_m, \ldots) \sigma))$ where $A$ occurs.

Thus, $u$ is either (case d1.1) the term $s_i \sigma$ for some $i = 1, \ldots, n$ or (case d1.2) the term $t_j \sigma$ for some $j = 2, \ldots, m$.

In case d1.1 the inductive arguments of the atoms in block $B$ are among the pairwise disjoint terms $t_2 \sigma, \ldots, t_m \sigma$ because the terms $s_1 \sigma, \ldots, s_{i-1} \sigma, s_{i+1} \sigma, \ldots, s_n \sigma$ are disjoint from $s_i \sigma$, and (by point b) the atoms to which they belong occur in different blocks.

Case d1.2 is analogous to case d1.1. Therefore, in case d1, point d) holds.

**Case d2.** Suppose that the inductive argument $t_j$ is *not* the maximum of the multiset $\{t_1, \ldots, t_m\}$. Let $t_k$, $k \neq 1$ be the maximum of $\{t_1, \ldots, t_m\}$. 


By inductive hypothesis, each term among $t_2, \ldots, t_{k-1}, t_{k+1}, \ldots, t_m$ is disjoint from $t_1$, and by point b) we have that $t_2\sigma = t_2, \ldots, t_{k-1}\sigma = t_{k-1}, t_{k+1}\sigma = t_{k+1}, \ldots, t_m\sigma = t_m$. Therefore, each element in $\{t_2\sigma, \ldots, t_{k-1}\sigma, t_{k+1}\sigma, \ldots, t_m\sigma\}$ is disjoint from $t_1\sigma$. Since the terms $s_1\sigma, \ldots, s_n\sigma$ are pairwise disjoint subterms of $s\sigma = t_1\sigma$, we have that also in case d2, point d) holds. (Notice that the existence of two terms $u$ and $v$ in IndArgs such that $u = v$ implies that the multiset IndArgs is $\{u, v\}$.)

Property $\Pi(C)$ implies that the Synchronized Descent Rule $S$ is defined for $C$, and it selects an atom with maximal inductive argument. Therefore, by Lemma 17, for every clause $C$ in the U-tree for $\langle \text{Prog}, D \rangle$ via $S$, and for every defined atom $G$ in the body of $C$, we have

$$\text{height}(G) \leq \max\{\text{height}(B) | B \text{ is a defined atom in Prog } \cup \{D\}\}.$$

We will now show that the number of atoms belonging to a block, say $B$, in Part($C$) is bounded by a value $M$, depending on $\text{height}(\text{max})$, where $\text{max}$ is the maximum inductive argument of an atom, call it $\text{MaxA}$, in the block.

Indeed, by point b) of Property $\Pi(C)$, each atom in the block $B$ has a subterm of $\text{max}$ as inductive argument. By point d) of Property $\Pi(C)$, any two atoms different from $\text{MaxA}$ have disjoint inductive arguments, and the number of pairwise disjoint subterms of $\text{max}$ is bounded by a number depending on $\text{height}(\text{max})$.

Thus, recalling (‡) above, we have that the number of atoms in a block is bounded by a positive integer $W$. This implies that the length of the longest variable-chained sequence in the U-tree is smaller than $W$, and therefore, by Theorem 14, we get the thesis.

The reader may notice that the above result can easily be extended to the case where the definition clause $D$ has more than two atoms in the body.

Example 5: List derivative revisited. Let us consider again the Even-Npderiv program of Example 1. Let Prog be the set of clauses: (clause1.2, ..., clause1.8), and let $D$ be clause 1.1. The pair $\langle \text{Prog}, D \rangle$ does not satisfy the hypotheses of Theorem 20 because clauses 1.3, 1.4, and 1.8 are not strongly-nonascending. However, they can be transformed into the following strongly-nonascending clauses by using the equality predicate:

- $1.3^* \text{npderiv}([X], [X]) \leftarrow X = X1.$
- $1.4^* \text{npderiv}(X, Y[T], [X1|D]) \leftarrow X = X1, X \geq Y, \text{npderiv}(Y[T], D).$
- $1.8^* \text{even}([X, Y[T]], [Y|E]) \leftarrow Y = Y1, \text{even}(T, E).$

Now $\text{Progl} = \{1.2, 1.3^*, 1.4^*, 1.5, 1.6, 1.7, 1.8^*\}$ satisfies the hypotheses of Theorem 20.

The reader may verify that by using an SDR, we get a foldable upper portion of the U-tree for $\langle \text{Progl}, \text{clause 1.1} \rangle$ which is equal to that for $\langle \text{Prog}, \text{clause 1.1} \rangle$ shown in Figure 1, except for some equality predicates in the bodies of the clauses.

By applying the Loop Absorption Procedure, simplifying the equality predicates, and eliminating the sterile predicates, we get exactly the final program version of Example 1. □
4. THE GENERALIZATION STRATEGY AND THE TRANSFORMATION TECHNIQUE FOR UNRESTRICTED PROGRAMS

We now introduce one more rule, called the \textit{generalization rule}, which can be applied for constructing a foldable U-tree when it is \textit{not} possible to obtain it by performing unfolding steps only. This is necessary only if the program at hand is outside the classes considered in Theorems 18 and 20.

We will apply that rule according to the so-called \textit{generalization strategy} as indicated in the Tree Construction Procedure below. By doing so, we can \textit{always} get a foldable U-tree, but we may lose the ability of performing some useful transformations, as we will explain later.

\textit{Definition 21.} The application of the \textit{generalization + equality introduction rule} (or \textit{generalization rule}, for short) to a clause \( C \) of the form \( H \leftarrow A_1, \ldots, A_n \) consists in deriving the new clause \( \text{Gen}C \) of the form \( H \leftarrow \text{Gen}A_1, \ldots, \text{Gen}A_n, X_1 \leftarrow t_1, \ldots, X_r \leftarrow t_r \), where \( \langle \text{Gen}A_1, \ldots, \text{Gen}A_n \rangle \theta = (A_1, \ldots, A_n) \), \( \theta = \{X_1/t_1, \ldots, X_r/t_r\} \), and \( \theta \) is not a variable renaming. We will say that the clause \( \text{Gen}C \) is a \textit{generalization} of the clause \( C \).

Generalization steps (i.e., applications of the generalization rule), together with the definition, unfolding, and folding steps, preserve the set of correct answer substitutions.

The process of repeatedly applying the unfolding rule and/or the generalization rule, starting from a given clause, can be represented as a tree of clauses. By an extension of the old notion, we will call \textit{U-tree} that tree of clauses. In this way, we do not need to modify the Loop Absorption Procedure, even though we have modified the notion of U-tree.

When we apply the generalization rule to a given clause, we generate a son-clause, where the equality predicates should be considered as base predicates. Thus, they will \textit{never} be unfolded when constructing the remaining portion of the U-tree.

A U-tree, in this extended notion, is said to be foldable according to Definition 5.

Let us now recall that Theorem 14 ensures the foldability of a U-tree if, in the bodies of its clauses, some conditions on the height of the defined atoms and on the length of their variable-chained sequences are satisfied.

By performing suitable generalization steps, it is possible to transform any clause into an equivalent one for which those conditions hold. Consider, for instance, the following clause:

\[ C. \ h(X, Y) \leftarrow p(t(X, s(Y))), q(t(Y, Z)), r(Z). \]

where we assume that \( h, p, q, \) and \( r \) are defined predicates.

Clause \( C \) has the atom \( p(t(X, s(Y))) \) whose height is 3. In that clause, the variable-chained sequence \( \langle p(t(X, s(Y))), q(t(Y, Z)), r(Z) \rangle \) has length 3. Clause \( C \) can be transformed into an equivalent one, namely,

\[ h(X, Y) \leftarrow p(t(X, Y1)), Y1 = s(Y), q(t(Y, Z)), r(Z). \]

where \( 2 \) is the maximal height for each defined atom in the body, and \( 2 \) is also the maximal length of the variable-chained sequence of the defined atoms in the body.
The following procedure constructs a foldable U-tree for any given \( \langle \text{program, clause} \rangle \) pair by performing unfolding and generalization steps.

**Procedure for the Construction of Foldable U-Trees by Generalization (or Tree Construction Procedure, for short).**

**Input.** A program \( \text{Prog} \), a definition clause \( D \), and a U-selection rule \( S \) which is defined for all clauses, apart from success or failing clauses. Two positive integers \( H \) and \( W \), called *vertical* and *horizontal bounds*, respectively.

**Output.** The minimal foldable upper portion \( T \) of a U-tree for \( \langle \text{Prog}, D \rangle \).

Set \( T \) to the tree whose unique node is labeled by the clause \( D \).

While \( T \) is not foldable do

- **take a leaf-clause, say \( C \), which is neither a success clause, nor a failing clause, nor a foldable clause, and**

- if in the body of \( C \) there exists *either* a defined atom \( A \) higher than \( H \) or a variable-chained sequence of defined atoms longer than \( W \)

  then apply the generalization rule to \( C \) in such a way that the derived clause \( \text{Gen}C \) does not exceed the bounds \( H \) and \( W \), and for any other generalization of \( C \), say \( Z \), which does not exceed \( H \) and \( W \), \( \text{Gen}C \) is *not* a generalization of \( Z \) (in this sense \( \text{Gen}C \) is a most concrete generalization of \( C \) w.r.t. \( H \) and \( W \))

else unfold \( C \) according to \( S \).

The Tree Construction Procedure is always terminating. Indeed, for each clause \( C \) in \( T \), from which we have to perform an unfolding step (*not* a generalization step), we have that

1) \( \max \{ \text{height}(A) | A \text{ is a defined atom in the body of } C \} \leq H \), and

2) each variable-chained sequence of defined atoms in \( C \) is not longer than \( W \).

Since a single unfolding step determines a bounded growth of the height of the atoms and the length of the variable-chained sequences, we have that, for every clause \( C \) in \( T \), the above properties 1) and 2) hold if we replace \( H \) and \( W \) by some suitable \( H' \) and \( W' \).

As a consequence of Theorem 14, the Tree Construction Procedure always terminates, and thus we have the following result.

**Theorem 22.** Given any program \( \text{Prog} \) and any clause \( D \), there exists a foldable U-tree for \( \langle \text{Prog}, D \rangle \), and it can be constructed using unfolding and generalization steps.

Notice that generalization steps should be performed with parsimony because they may reduce the efficiency of the derived programs. It is indeed the case that, although the generalization rule guarantees the termination of the Tree Construction Procedure, it introduces new variables instead of instantiated terms, and thus it may prevent some simplification steps because some information about the structure of the terms is lost.

We will not address here the problem of the optimal choice of the vertical and horizontal bounds for the application of the generalization rule. Various heuristics can be proposed. Our experience is that a good choice of those bounds is

- a) \( H \) is not lower than \( \max \{ \text{height}(A) | A \text{ is a defined atom in } \text{Prog} \cup \{D\} \} \), and
b) \( W \) is not lower than the length of the longest variable-chained sequence of defined atoms in the body of a clause in \( \text{Prog} \cup \{ D \} \).

A motivation for this choice is provided by the following fact.
Suppose that during the application of the Tree Construction Procedure, we unfold an atom \( A \) in a clause \( C \) w.r.t. a program clause \( E \). If the vertical bound \( H \) (or the horizontal bound \( W \)) is smaller than the height of some defined atom (or the length of some variable-chained sequence) in the body of \( E \), then the next step in the construction of \( T \) is a generalization step.
An analogous phenomenon may happen if, before unfolding the atom \( A \) in the clause \( C \) w.r.t. the clause \( E \), we have that: i) the vertical bound is smaller than the height of the head of \( E \), and ii) the atom \( A \) shares some variables with other defined atoms in the body of \( C \).
Thus, our choice for the bounds \( H \) and \( W \) may avoid too many generalization steps.
Notice that by unfolding only predicates which are not recursively defined, we cannot generate an infinite U-tree, and thus we cannot produce an unbounded growth of the height of the atoms and/or the length of the variable-chained sequences. Therefore, before fixing the values of the vertical and the horizontal bounds, we may assume that we have unfolded the predicates which are nonrecursive.
The examples of the next section show that our simple heuristics for the choice of \( H \) and \( W \) is very satisfactory in practice.

5. PARTIAL EVALUATION VIA PROGRAM TRANSFORMATION: A FORMAL COMPARISON AND AN EXTENDED EXAMPLE

In this section, we apply our program transformation methodology to the case of partial evaluation. We show in a formal way that our transformation techniques are an extension of the partial evaluation methods as described in [19]. We also present some examples which illustrate the use of our techniques for performing partial evaluation in an automatic way.
We will first recall (and rephrase, in our terminology) some definitions and results given in [19].

Definition 23. Given a set \( A \) of atoms, a clause \( C \) is said to be \( A \)-closed iff each defined atom in the body of \( C \) is an instance of an atom in \( A \). Analogously, a set of clauses is said to be \( A \)-closed iff each clause is \( A \)-closed.

Theorem 24. Let \( \text{Prog} \) be a definite program, \( G \) a definite goal, and \( A \) a finite set of atoms. Suppose that \( G \) is \( A \)-closed and, for each \( B \in A \), \( T_B \) is an upper portion of a \( U \)-tree for \( \langle \text{Prog}, B \leftarrow B \rangle \) such that: a) \( T_B \) has been obtained by unfolding steps only, and b) the set \( L_B \) of all nonfailing leaves of \( T_B \) is \( A \)-closed. Then the union of all sets \( L_B \), for \( B \in A \), is equivalent to \( \text{Prog} \) w.r.t. the goal \( G \).

The above result provides the justification for the following Algorithm 1 for performing partial evaluation.

Algorithm 1. Input. A program \( \text{Prog} \) and a goal \( G \). Output. The residual program \( \text{Prog}^1 \).
1a) Find a finite set \( A \) of atoms and, for every \( B \in A \), find an upper portion \( T_B \) of a U-tree for \( \langle \text{Prog}, B \leftarrow B \rangle \) such that
   i) \( G \) is \( A \)-closed,
   ii) for each \( B \in A \), the set \( L_B \) of all nonfailing leaves of \( T_B \) is \( A \)-closed.

1b) \( \text{Progl} \) is the union of all sets \( L_B \) for \( B \in A \).

A concrete instance of Algorithm 1 is the procedure for partial evaluation presented in [5].

Now we will show that by using the loop absorption and the generalization strategies, it is possible to construct an algorithm, called Algorithm 2, which transforms programs as the above Algorithm 1 does. The only relevant difference between the two algorithms is that the nonfailing leaves of the U-trees from which we extract the residual program \( \text{Progl} \) are closed w.r.t. a set of conjunctions of atoms rather than a set of atoms.

Obviously, that difference may often be an advantage of our methodology because it allows us to better exploit the interactions among various atoms of the goal being partially evaluated. In particular, we will see that the Partial Evaluation Procedure we describe below can be applied starting from a clause whose body is the set of the defined atoms in a given goal \( G \), rather than starting from an individual clause for each defined atom in \( G \), as required by Algorithm 1.

**Definition 25.** Let \( Q \) be a set of clauses and \( \text{DEF} \) a set of definition clauses. We say that \( Q \) is \( \text{DEF-closed} \) iff the set of all defined atoms in the body of each clause in \( Q \) can be partitioned into conjunctions of atoms which can be folded w.r.t. clauses in \( \text{DEF} \).

Now let us consider the following algorithm, Algorithm 2, which corresponds to Algorithm 1.

**Algorithm 2. Input.** A program \( \text{Prog} \) and a goal \( G \). **Output.** The residual program \( \text{Progl} \).

2a.1) Find a finite set \( \text{DEF} \) of definitions, and for each \( D \in \text{DEF} \), find the upper portion \( T_D \) of a U-tree for \( \langle \text{Prog}, D \rangle \) such that
   i) \( G \) is \( \text{DEF-closed} \),
   ii) for each \( D \in \text{DEF} \), the set \( L_D \) of all nonfailing leaves of \( T_D \) is \( \text{DEF-closed} \).

2a.2) Fold all subsets of defined atoms occurring in the bodies of the clauses of \( L_D \) w.r.t. clauses in \( \text{DEF} \).

2b) \( \text{Progl} \) is the union of all sets \( L_D \) for \( D \in \text{DEF} \).

A particular implementation of Algorithm 2 is provided by the following Partial Evaluation Procedure which uses our Tree Construction and Loop Absorption Procedures for obtaining the required foldable U-trees and extracting the residual program.

**Partial Evaluation Procedure.**

**Input.** A program \( \text{Prog} \) and a goal \( G \). **Output.** The residual program \( \text{Progl} \).

Let \( DG \) be a clause whose body is the set of defined atoms in \( G \), and the head
consists of a fresh predicate symbol whose arguments are the variables occurring in $G$.

Initialize $\text{DEFin}$ to $\{DG\}$, $\text{DEFout}$ to $\emptyset$, and $\text{Prog}l$ to $\emptyset$.

While there exists a clause, say $C$, in $\text{DEFin}$ do:

1) Construct the minimal foldable upper portion $T$ of a U-tree for $\langle \text{Prog}, C \rangle$ by applying the Tree Construction Procedure (with fixed U-selection rule, vertical bound, and horizontal bound).

2) Apply the Loop Absorption Procedure, taking $T$ as input, add the derived program to $\text{Prog}l$, and add both $C$ and the derived eureka definitions to $\text{DEFout}$.

3) Delete $C$ from $\text{DEFin}$.

4) Take a clause $D$ in $\text{Prog}l$, and consider the set $\text{NotFolded}_D$ of defined atoms in the body of $D$ whose predicate does not occur in the head of a clause in $\text{DEFin} \cup \text{DEFout}$.

5) Consider a block $B$ in the partition $\text{Part}(\text{NotFolded}_D)$ induced by the equivalence relation $\approx$ on $\text{NotFolded}_D$. Add to $\text{DEFin}$ a clause $N$ whose head predicate is a fresh symbol $\text{newp}$ and whose body is $B$. The set of variables which are arguments of $\text{newp}$ is the minimal set $V$ such that $D$ can be folded w.r.t. $N$.

6) Perform all possible folding steps on clauses of $\text{Prog}l$ w.r.t. the clauses in $\text{DEFin} \cup \text{DEFout}$.

The reader may easily verify that at the end of the Partial Evaluation Procedure, 1) the definitions in the set $\text{DEFout}$, 2) the upper portions of the U-trees constructed by the various applications of the Tree Construction Procedure, and 3) the program $\text{Prog}l$ do correspond to: 1) the set $\text{DEF}$, 2) the trees $T_D$ with $D \in \text{DEF}$, and 3) the program $\text{Prog}l$ of Algorithm 2 above.

We conclude this section by presenting three examples where we will see our Partial Evaluation Procedure in action.

In Example 6, we give an introductory demonstration of our methodology. In Example 7, we show the use of the generalization rule for the construction of the foldable upper portion of a U-tree. Finally, in Example 8, we consider a more complex case, where we compare the efficiency of the residual program derived by using our techniques w.r.t. that of the program derived by using the methods indicated in [5, 19].

**Example 6.** Let us partially evaluate the following program $\text{Times}$:

\[
times(0, N, 0).
\]

\[
times(s(M), N, T) \leftarrow \times(M, N, T), plus(N, T, T).
\]

\[
plus(0, N, N).
\]

\[
plus(s(M), N, s(P)) \leftarrow plus(M, N, P).
\]

w.r.t. the goal $\leftarrow \times(N, s(s(0)), T)$. We apply the Partial Evaluation Procedure by introducing the new clause (which plays the role of the clause $DG$):

1. \text{newl}(N, T) \leftarrow \times(N, s(s(0)), T).

Thus, initially $\text{DEFin}$ is \{clause 1\}, while $\text{DEFout}$ and $\text{Prog}l$ are empty. We construct the minimal foldable upper portion of a U-tree for $\langle \text{Times}, \text{clause 1} \rangle$ by using the Tree Construction Procedure.
In that procedure, we use the U-selection rule which selects the leftmost atom in the body of a clause. The vertical and horizontal bounds $H$ and $W$ are chosen according to the heuristics illustrated in the previous section.

We indeed take $H = 3$ because it is the maximal value in \{height(A)|A is a defined atom in $\text{Times} \cup \{\text{clause1}\}\}$, and $W = 2$ because it is the length of the longest variable-chained sequence of defined atoms in the body of a clause in $\text{Times} \cup \{\text{clause1}\}$.

That choice of the bounds does not determine any generalization step, and the resulting minimal foldable upper portion of the U-tree for $\langle \text{Times}, \text{clause 1} \rangle$ is shown in Figure 4.

The output of the Loop Absorption Procedure is the following program:

1. $\text{newl}(N, T) \leftarrow \text{times}(N, s(s(0)), T)$.

2. $\text{newl}(0, 0)$.

3. $\text{newl}(s(M), T) \leftarrow \text{newl}(M, TI), \text{plus}(s(s(0)), TI, T)$.

which is added to $\text{Prog}1$.

Since $\text{newl}$ is in $\text{DEFin} \cup \text{DEFout}$, the only atom in $\text{NotFolded}_{\text{clause3}}$ is $\text{plus}(s(s(0)), TI, T)$. Therefore, we add to $\text{DEFin}$ the following new clause:

4. $\text{new2}(TI, T) \leftarrow \text{plus}(s(s(0)), TI, T)$.

and we fold plus in clause 3 w.r.t. clause 4. We get the clause

5. $\text{newl}(s(M), T) \leftarrow \text{newl}(M, TI), \text{new2}(TI, T)$.

Thus, at the end of the first loop of the Partial Evaluation Procedure, we have that i) $\text{DEFin} = \{\text{clause 4}\}$, ii) $\text{DEFout} = \{\text{clause 1}\}$, and iii) $\text{Prog}1 = \{\text{clause 2}, \text{clause 5}\}$.

We now execute the while-loop of the Partial Evaluation Procedure for clause 4. By applying the Tree Construction Procedure to the pair $\langle \text{Times}, \text{clause 4} \rangle$ we construct a foldable U-tree (see Figure 5), and we get the clause

6. $\text{new2}(T, s(s(T)))$.

Now, all predicates in the current program $\text{Prog}1$ occur in $\text{DEFin} \cup \text{DEFout}$ as

\[
\begin{align*}
\text{new2}(T1, T) & \leftarrow \text{plus}(s(s(0)), T1, T). \\
\text{new2}(T1, T) & \leftarrow \text{plus}(s(0), T1, T). \\
\text{new2}(T1, s(T)) & \leftarrow \text{plus}(0, T1, T). \\
\text{new2}(T, s(s(T))) & \leftarrow \text{plus}(0, T1, T).
\end{align*}
\]
well, and therefore, no new predicate is added to DEFin. After deleting clause 4, DEFin becomes empty. Therefore, the Partial Evaluation Procedure halts, and it computes the final program Progl = {clause 2, clause 5, clause 6}.

We now present an example where the use of the generalization rule is indeed necessary for the construction of the minimal foldable upper portion of a U-tree.

Example 7. Let us consider the following program, called Timesl, which is a tail recursive version of the program shown in the previous example, Example 6.

\[
times(M, N, T) \leftarrow \times l(M, N, A, T).
\]
\[
\times l(0, N, T, T).
\]
\[
\times l(s(M), N, A, T) \leftarrow plus(N, A, P), \times l(M, N, P, T).
\]
\[
plus(0, N, N).
\]
\[
\times l(s(M), N, s(P)) \leftarrow plus(M, N, P).
\]

Suppose that we want to partially evaluate Timesl w.r.t. the goal \(\times (M, s(s(0)), T)\). We introduce the clause

1. newl(M, T) \leftarrow times(M, s(s(0)), T).

Then we construct the minimal foldable upper portion of a U-tree for \(\langle\text{Timesl}, \text{clause 1}\rangle\) by using the Tree Construction Procedure.

In that procedure, we choose the bounds \(H\) and \(W\) according to the heuristics indicated in the previous section. Thus, we have

\[
H = \text{height}(\times (M, s(s(0)), T)) = 3
\]
\[
W = \text{length}((\text{plus}(N, A, P), \times l(M, N, P, T))) = 2.
\]

The minimal foldable upper portion of a U-tree for \(\langle\text{Timesl}, \text{clause 1}\rangle\) constructed using the Tree Construction Procedure is depicted in Figure 6. In that figure, solid arrows denote one or more unfolding steps, and the edges labeled by "generalization" correspond to applications of the generalization rule, which is applied when the height of a defined atom is larger than 3, as indicated in the Tree Construction Procedure.

The reader may easily verify that, without generalization steps, no foldable upper portion of a U-tree can be found for \(\langle\text{Timesl}, \text{clause 1}\rangle\).

By applying the Loop Absorption Procedure to the minimal foldable upper portion, we have constructed, we get the following residual program, after the simplification of some equalities.

\[
\text{newl}(0, 0).
\]
\[
\text{newl}(s(0), s(s(0))).
\]
\[
\text{newl}(s(s(M)), T) \leftarrow \text{new2}(M, s(s(0)), T).
\]
\[
\text{new2}(0, X, s(s(X))).
\]
\[
\text{new2}(s(M), X, T) \leftarrow \text{new2}(M, s(s(X)), T).
\]

The predicate new2 has been introduced because of the loop for the recurrent pattern consisting of the atom \(\times l(M, s(s(0)), s(s(Y)), T)\) (see Figure 6).

In our final example, we would like to show that our techniques, based on the loop absorption and the generalization strategies, are able to perform automatic partial evaluation of nontrivial programs.

Example 8: Searching a string in a text. Let us consider the strings of characters \(S\)
and \( T \). We say that \( S \) occurs in the “text string” \( T \) at position \( P \) iff there exist two (possibly empty) substrings \( \text{Before}S \) and \( \text{After}S \) of \( T \) such that: a) \( T \) is the concatenation of \( \text{Before}S \), \( S \), and \( \text{After}S \) and b) the length of \( \text{Before}S \) is \( P \).

We would like to compute the leftmost occurrence of \( S \) in \( T \), that is, the minimum \( P \) such that \( S \) occurs in \( T \) at position \( P \). Here is a logic program, called \text{Firstin}, which solves that problem.

\[
\text{firstin}(T, S, \text{Position}) \leftarrow \text{fi}(T, [\ ], S, 0, \text{Position}).
\]
\[
\text{fi}(X, Y, [\ ], \text{Acc}, s(\text{Position})), \text{length}(Y, L), \text{plus}(L, \text{Position}, \text{Acc}).
\]
\[
\text{fi}([C|\text{RestofText}], \text{OldPref}, \text{OldSuff}, \text{Acc}, \text{Position}) \leftarrow
\text{newstring}(C, \text{OldPref}, \text{OldSuff}, \text{NewPref}, \text{NewSuff}),
\text{fi}(\text{RestofText}, \text{NewPref}, \text{NewSuff}, s(\text{Acc}), \text{Position}).
\]
\[
\text{newstring}(C, \text{OldPref}, [C|\text{RestOldSuff}], \text{NewPref}, \text{RestOldSuff}) \leftarrow
\text{append}(\text{OldPref}, [C], \text{NewPref}).
\]
\[
\text{newstring}(C, \text{OldPref}, [D|\text{RestOldSuff}], \text{NewPref}, \text{NewSuff}) \leftarrow
C \neq D, \text{append}(\text{OldPref}, [C], H), \text{append}(V, \text{NewPref}, H), \text{append}(\text{NewPref}, R, \text{OldPref}),
\text{append}(R, [D|\text{RestOldSuff}], \text{NewSuff}).
\]
\[
\text{length}([\ ], 0).
\]
\[
\text{length}([H|T], s(L)) \leftarrow \text{length}(T, L).
\]
\[
\text{plus}(0, N, N).
\]
\[
\text{plus}(s(M), N, s(P)) \leftarrow \text{plus}(M, N, P).
\]
\[
\text{append}([\ ], L, L).
\]
\[
\text{append}([H|T], L[H|TI]) \leftarrow \text{append}(T, L, TI).
\]

The only base predicate in this program is \(*\).

This program is a slight modification of the one in [17] which tests whether or
not a string occurs in a text. In our program, the top predicate \textit{firstin} has the extra argument \textit{Position} which tells us where the string \textit{S} occurs in \textit{T}.

The understanding of the behavior of the above program is not necessary for the application of the transformation techniques presented in the previous section. They are syntactically based, and their success does not rely on any human intuition.

Suppose now that we want to partially evaluate our program with respect to the string \textit{S} = \{a, b\}. We will derive the residual program, which gives us the leftmost position where \{a, b\} occurs in the text \textit{T}.

This derivation will be performed by applying our Partial Evaluation Procedure, which in this case will determine some generalization steps.

In fact, the reader may notice that we cannot use Theorems 18 and 20, which ensure that no generalization step is necessary, because the given program is not nonasc ending. Indeed, in the third clause of the program, the fourth argument of the predicate \textit{fi} is \textit{Acc} in the head and \textit{s(Acc)} in the body.

We initialize \textit{DEFin} to the singleton made out of the following clause:

\[ 1. \text{firstin}(\text{Text}, \text{Position}) \leftarrow \text{firstin}([a,b], \text{Position}). \]

We apply our Tree Construction Procedure which constructs the minimal foldable upper portion of a U-tree for \(\langle\text{Firstin}, \text{clause 1}\rangle\). In that procedure, we choose the selection rule, called \textit{LeftS}, which selects the leftmost atom in the body of the current clause. We also establish suitable values for the vertical and the horizontal bounds \(H\) and \(W\). In this respect, we follow the heuristics presented in the previous section. Thus, we unfold the nonrecursive predicate \textit{newstring} in the body of the second clause for the predicate \textit{fi}, and we get

\[
\begin{align*}
\text{fi}([C|\text{RestofText}], & \text{OldPref}, [C|\text{RestOldSuff}], \text{Acc}, \text{Position}) \leftarrow \\
& \text{append}(\text{OldPref}, [C], \text{NewPref}), \\
& \text{fi}(\text{RestofText}, \text{NewPref}, \text{RestOldSuff}, \text{s(Acc)}, \text{Position}).
\end{align*}
\]

\[
\begin{align*}
\text{fi}([C|\text{RestofText}], & \text{OldPref}, [D|\text{RestOldSuff}], \text{Acc}, \text{Position}) \leftarrow \\
& C \neq D, \text{append}(\text{OldPref}, [C], H), \text{append}(V, \text{NewPref}, H), \\
& \text{append}(\text{NewPref}, R, \text{OldPref}), \\
& \text{append}(R, [D|\text{RestOldSuff}], \text{NewSuff}), \\
& \text{fi}(\text{RestofText}, \text{NewPref}, \text{NewSuff}, \text{s(Acc)}, \text{Position}).
\end{align*}
\]

If we consider the above two clauses, the initial program version, and clause 1, then the highest defined atom has height 3, while the longest variable-chained sequence has length 5. Thus, we assume \(H = 3\) and \(W = 5\).

The relevant parts of the minimal foldable upper portion \(T_I\) of a U-tree for \(\langle\text{Firstin}, \text{clause 1}\rangle\) are depicted in Figure 7, where a solid arrow between two clauses denotes one or more unfolding and/or generalization steps, and the numbers stand for the following clauses:

\begin{align*}
C_1. \text{firstin}(\text{Text}, \text{Position}) & \leftarrow \text{firstin}(\text{Text}, [a, b], \text{Position}). \\
C_2. \text{firstin}([a, b], \text{R}, \text{s(0)}). \\
C_3. \text{firstin}([a, a, C], \text{R}, P) & \leftarrow a \neq b, \text{newstring}(C, [a], [b], \text{NP, NS}), \\
& \text{fi}(R, \text{NP, NS, s(s(s(0)))), P). \\
C_4. \text{firstin}([a, C, D], \text{R}, P) & \leftarrow C \neq b, \text{newstring}(D, [ ]), [a, b], \text{NP, NS}), \\
& \text{fi}(R, \text{NP, NS, s(s(X))), P), X = s(0)).
\end{align*}
Let us now examine the path of that U-tree which connects the root and the leaf-clause C11. That path is generated by unfolding clause 1, thereby obtaining clause C3, where the height of the atom $\text{fi}(R, NP, NS, s(s(s(O))))$, $X = s(0)$. Thus, according to the Tree Construction Procedure, we generalize the occurrence of $s(0)$ in that atom, and we get clause C7.

By continuing the unfolding process from C7, we get clause C9, and then clause C11 which determines a loop with clause C7 as the oldest subsuming ancestor, the recurrent pattern of atoms being "newstring($C, [a], [b], NP, NS$), $\text{fi}(R, NP, NS, s(s(s(X))))$, $Y = s(0)$."
We now apply the Loop Absorption Procedure to extract a program from the
minimal foldable upper portion $T_l$ depicted in Figure 7. That procedure deter-
mines the introduction of four eureka predicates: new1, new2, new3, and new4. The
clauses defining those predicates are

N1. $\text{new1}(C, T, A, P) \leftarrow \text{newstring}(C, [a], [b], NP, NS), \text{fi}(T, NP, NS, s(s(A)), P)$.
N2. $\text{new2}(C, T, A, P) \leftarrow \text{newstring}(C, [a, b], NP, NS), \text{fi}(T, NP, NS, s(s(A)), P)$.
N3. $\text{new3}(T, A, P) \leftarrow \text{fi}(T, [a, b], s(s(A)), P)$.
N4. $\text{new4}(T, A, P) \leftarrow \text{fi}(T, [a, b], s(s(A)), P)$.

The bodies of the clauses N1, N2, N3, and N4 above are made out of the set of
defined atoms in the bodies of the clauses C7, C4, C10, and C14, respectively,
which are the oldest subsuming ancestors of foldable clauses in $T_l$ (see Figure 7).
By simplifying the equalities produced by the generalization steps, we get the
following program, called Firstabin:

\begin{align*}
\text{firstabin}(\{a, b\}, R, s(0)). \\
\text{firstabin}(\{a, a, D\}, R, s(0)). \\
\text{firstabin}(\{a, C, D\}, R, s(0)). \\
\text{firstabin}(\{a, C, D\}, R, s(0)). \\
\text{new1}(b, T, P, s(P)). \\
\text{new1}(\{\{a, b\}, \{a, b\}, \{a, b\}\}, s(0), P). \\
\text{new1}(\{a, b, a, b\}, s(0), P). \\
\text{new1}(\{a, b, a, b\}, s(0), P).
\end{align*}

All defined atoms in the bodies of the above clauses have been folded, and
therefore, no new clause is added to $\text{DEFin}$, which becomes empty. The Partial
Evaluation Procedure terminates, and its final output is the above program.

Now we would like to compare the program derived by using our transfor-
mational approach with the residual program one may obtain by using the partial
evaluation technique of Lloyd and Shepherdson [19], as specified by Algorithm 1.
Let us consider the following set $A$ of three atoms:

\[ A = \{ \text{firstin}(T, [a, b], P), \text{fi}(T, [a, b], A, P), \text{fi}(T, [a, b], A, P) \}. \]

Let us also consider the corresponding three clauses:

a1. $\text{firstin}(T, [a, b], P) \leftarrow \text{firstin}(T, [a, b], P).
\text{a2. } \text{fi}(T, [a, b], A, P) \leftarrow \text{fi}(T, [a, b], A, P).
\text{a3. } \text{fi}(T, [a, b], A, P) \leftarrow \text{fi}(T, [a, b], A, P).$
We then construct the finite upper portions of the U-trees for \langle Firstin, clause a1 \rangle, \langle Firstin, clause a2 \rangle, and \langle Firstin, clause a3 \rangle, whose leaves are the following clauses:

\begin{align*}
  \text{firstin}(T, [a, b], P) & \leftarrow \text{fi}(T, [ ], [a, b], 0, P). \\
  \text{fi}([a[T], [ ], [a, b], A, P) & \leftarrow \text{fi}(T, [a], [b], s(A), P). \\
  \text{fi}([C[T], [ ], [a, b], A, P) & \leftarrow C \neq a, \text{fi}(T, [ ], [a, b], s(A), P). \\
  \text{fi}([b[T], [a], [b], s(P), s(P)). \\
  \text{fi}([a[T], [a], [b], A, P) & \leftarrow \text{fi}(T, [a], [b], s(A), P). \\
  \text{fi}([C[T], [a], [b], A, P) & \leftarrow C \neq b, \text{fi}(T, [ ], [a, b], s(A), P). 
\end{align*}

The above clauses are A-closed, and therefore, they are a residual program, called \text{LS\_Firstubin}, of \text{Firstin} w.r.t. the goal \leftarrow \text{firstin}(T, [a, b], P).

This residual program, computed according to the standard partial evaluation techniques, is simpler than the \text{Firstubin} program. This fact basically depends on the choice which has been made for providing the above set \( A \) of atoms. But that choice has been guided by human intuition, while through our Partial Evaluation Procedure, we have found the required eureka definitions in an automatic way.

Some computer experiments show that the program \text{Firstubin} is slightly more efficient than the program \text{LS\_Firstubin}. This is due to the fact that in \text{Firstubin}, we use different predicates for different instances of the same atom, and thus we reduce the time spent during unification.

6. CONCLUSIONS

We have presented a program transformation methodology which makes use of the loop absorption and generalization strategies for the automatic derivation of logic programs.

This methodology basically consists of finding the so-called foldable U-tree associated with a given logic program. From that tree, we then extract an improved program version. We have also established some theorems which ensure the existence of foldable U-trees for suitable initial programs.

The notion of foldability presented here is different from the ones in [21] and [22]. In the first paper, the authors assume that a set of definitions is given, and they study the problem of constructing the upper portion of a U-tree such that all leaves can be folded w.r.t. the given definitions. Here, on the contrary, we do not assume that a set of definitions is given in advance, but we generate them by solving the foldability problem (see Section 3).

The notion of foldability in [22] is a bit stronger than the one considered here. In that paper, the authors say that a clause in a U-tree is foldable if the set (not a subset) of all defined atoms in its body is an instance of the set of the defined atoms in the body of an ancestor clause. As we have shown in Sections 3 and 4, this weakening of the foldability notion allows us to find larger classes of programs in which our transformation methodology is successful.

In Section 5, we have shown that the standard partial evaluation techniques can be considered as an instance of our program transformation methodology. To this regard, we have introduced the Partial Evaluation Procedure which, together with the Tree Construction and the Loop Absorption Procedures, automatically computes the residual program w.r.t. a given goal.
The main advantage of performing partial evaluation by using our transformational approach (based on semantics preserving rules) is that we do not need to check extra conditions to prove the equivalence between the initial program and the residual program. Other minor advantages which concern efficiency have been mentioned when commenting on the examples.

It is worthwhile noticing that not all equivalent programs can be derived by the unfold/fold transformation rules used in this paper (in this sense, they are not complete). However, those rules can be strengthened by exploiting the properties of the predicates at hand, such as for instance, associativity, commutativity, etc. Indeed, the results achieved by various research groups in this direction show that the unfold/fold rules provide a strong basis for building powerful systems for the automatic derivation of efficient programs (see [7] in the case of functional languages).

As mentioned in the Appendix, during program transformation, we preserve the semantics defined as the set of correct answer substitutions of definite logic programs. We leave for further research the extension of our techniques to the case of Prolog programs (where the depth-first search of the SLD-tree affects the semantics) and general logic programs (where the language includes negation [18]).

APPENDIX

We list the transformation rules which we use for program derivation. For their application, we assume that two different clauses do not share any common variable.

- **Definition Rule.** It consists of adding to the current program version a new clause, say \( \text{newp}(X_1, \ldots, X_k) \leftarrow A_1, \ldots, A_n \), where \( \text{newp} \) is a fresh predicate defined in terms of already existing predicates.

  The arguments \( X_1, \ldots, X_k \) are distinct variable symbols occurring in \( A_1, \ldots, A_n \). Notice that recursive definitions are not allowed. A clause introduced by applying the definition rule is considered to be a definition clause (see the folding rule below).

- **Unfolding Rule.** Let \( C \) be a clause of the form \( H \leftarrow A_1, \ldots, A_m, A_\alpha \), and \( S \) a set of clauses of the form \( (K_j \leftarrow B_{j1}, \ldots, B_{jm}) | j = 1, \ldots, r \) such that, for \( j = 1, \ldots, r \), \( K_j \) and \( A_\alpha \) are unifiable, with most general unifier \( \theta_j \).

  The result of unfolding \( A_j \) (\( 1 \leq i \leq m \)) in \( C \) w.r.t. \( S \) is the following set \( U \) of clauses \( \{(H \leftarrow A_1, \ldots, A_{i-1}, B_{j}, B_{j1}, \ldots, B_{jm}, A_{i+1}, \ldots, A_m) | j = 1, \ldots, r \} \).

  If \( C \) is a clause in the current program \( P \) and \( S \) consists of all clauses in \( P \) whose heads are unifiable with \( A_\alpha \), then we can replace \( C \) in \( P \) by the set \( U \).

  A clause obtained by unfolding is not considered to be a definition clause.

- **Folding Rule.** Let \( C \) be a clause of the form \( H \leftarrow A_1, \ldots, A_n, A_{n+1}, \ldots, A_r \) and \( D \) a clause of the form \( K \leftarrow B_1, \ldots, B_n \). Suppose that there exists a substitution \( \sigma \) such that \( A_i = B_i \sigma \), for all \( i = 1, \ldots, n \). We fold the atoms \( A_1, \ldots, A_n \) in \( C \) w.r.t. \( D \) by substituting for \( C \) the clause \( F \) of the form \( H \leftarrow K \sigma, A_{n+1}, \ldots, A_r \).

  This rule is applied only if: 1) by unfolding the atom \( K \sigma \) in \( F \) w.r.t. \( D \) we obtain a variant of the clause \( C \), 2) the clause \( D \) is a definition clause.
(possibly not occurring in the current program), and 3) $C$ is not a definition clause.

Our rules are slightly different from the ones presented in [24, 13, 2]. However, it can be shown that by using the rules according to our formulation, we preserve the least Herbrand model semantics of the predicates occurring in the initial version of the program. It can also be shown that the set of correct answer substitutions is preserved. Moreover, in the case of definite programs, our rules are more powerful than the ones proposed in [9] because they allow us to fold w.r.t. clauses which do not occur in the current program version, and this is the case when we want to obtain recursive definitions.

---

REFERENCES


