ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

M P A Sellink (Ed)

2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997

Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997

An Algebraic Specification of a Transformation Tool for Prolog Programs

J.J. Brunekreef

Published in collaboration with the British Computer Society

BCS

Springer

ISBN: 3-540-76228-0

An Algebraic Specification of
a Transformation Tool for Prolog Programs

Jacob Brunekreef
Programming Research Group, University of Amsterdam
Amsterdam, The Netherlands

Abstract

The paper reports about a case study in algebraic specification. It describes TransLog, a transformation tool for pure Prolog programs. TransLog supports the interactive transformation of (a part of) a program. Six transformation steps are supported: unfolding, folding, goal definition, argument permutation (an instance of goal replacement), goal switching and definition elimination. As much as possible, pure Prolog semantics are preserved. The tool is equipped with navigation options, which allow the user to switch from the current program to another program in a transformation sequence.

TransLog has been implemented with the ASF+SDF Meta-environment. The paper focuses on the algebraic specification of the tool, not on the underlying theory of (logic) program transformation. The specification is presented at a global level, only the specification of the unfold and fold step is discussed more in depth.

1 Introduction

Program transformation is a technique used in software development. Execution by hand of a single transformation step already appears to be error-prone, even if the program is small. This makes tool support desirable. In this paper we present TransLog, a tool for the transformation of pure Prolog programs. Six basic transformation steps have been implemented: unfolding, folding, goal definition, argument permutation (an instance of goal replacement), goal switching and definition elimination. Furthermore, the tool is equipped with navigation options. These options allow the user to switch from the current program P_i to the previous program P_{i-1} or the next program P_{i+1} (if present) in a transformation sequence $P_0 \ldots P_{i-1},\ P_i,\ P_{i+1}, \ldots P_n$.

A transformation step is called correct with respect to a given semantics if the resulting program has the same semantics as the initial one. Most transformation steps have 'applicability conditions', which –in general– depend on the chosen semantics. We focus on the Prolog 'sequence of answer substitutions' semantics. So, order and multiplicity of answers do count.

The TransLog tool has been developed with the ASF+SDF Meta–environment ([8]). The algebraic specification technique is well-suited for specifying the operations that are required within the context of program transformation. There exist several other tools for the transformation of logic programs. We mention Spes ([2]), Transformer ([1]), PAL ([9]), Mixtus ([15]), Logimix ([12]), PADDY ([14]) and ECCE ([7]). Most of these tools have been implemented in Prolog. The advantages of Prolog-based systems are clear: the language is well-suited for the symbolic manipulations that are required, furthermore one gets syntax definition and unification for free. On the other hand, the ASF+SDF Meta-environment with its built-in parser, conditional equations, list matching properties and user interface primitives, constitutes a powerful programming environment for the specification of program transformations. We do not hold the opinion that one approach is definitely better than the other one. At several places in the paper we will pay attention to the (dis)advantages of the choice for the algebraic (ASF+SDF) approach.

The paper is organized as follows. First, in Section 2, we give an example of a transformation sequence in which a simple Prolog program is transformed into a more efficient program. In this section we also give an impression of what the TransLog tool looks like. The specification of the TransLog tool is presented in the Sections 3 to 7 at a global level, only two modules are discussed in more detail. The complete algebraic specification of the TransLog tool can

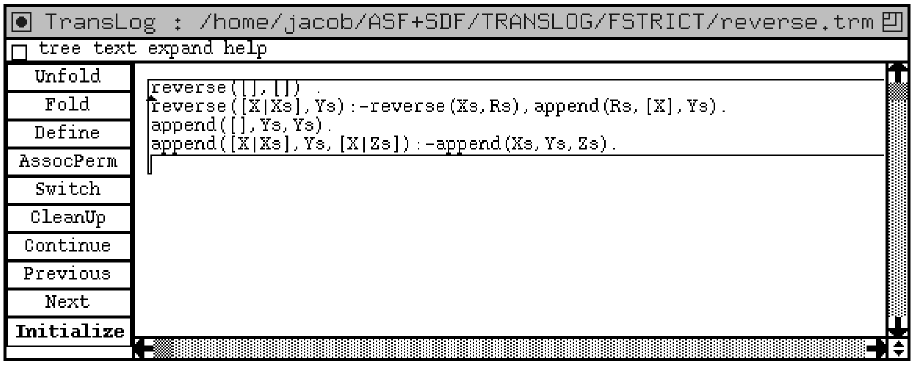

Figure 1: A TransLog window with the initial reverse program

be found in [5]. In Section 8 some conclusions are listed. The paper does not contain an introduction to the theory of (logic) program transformation, nor an answer to the question how the TransLog tool relates to the various theoretical considerations. For the first subject the reader is referred to [18, 16, 13]. For the second subject, see [4].

2 A transformation example

In the following example a program for list reversal is transformed into a more efficient version. The example serves as an informal introduction to most of the transformation steps that have been implemented in the TransLog tool. Figure 1 shows a TransLog window with the initial program. At the left side of the window ten buttons are present. Each button is related to a specific action on (a part of) the contents of the window. The upper six buttons are related to the six transformation steps. The Continue button is used in a dialogue. Dialogues are part of the Fold step and the Define step. The Previous and Next buttons represent the navigation options in a sequence of (transformed) programs. The Initialize button is used to start a transformation sequence.

The definition of the reverse predicate displayed in Fig. 1 implies two list traversals: one for the recursive definition of the reverse predicate, one for the append predicate. In a sequence of transformation steps we will transform this program into a more efficient program with one list traversal. For spatial reasons we will not display the complete TransLog window after each transformation step. All displayed clauses are produced by the TransLog tool.

As a first step towards a more efficient program we apply the *define* step to add to the program a clause defining a new predicate:

```
revacc(Xs,Ys,Acc):-reverse(Xs,Rs),append(Rs,Acc,Ys).
```

The revacc predicate uses an accumulator: the third argument is supposed to contain the part of a list that has already been reversed. Next, we apply an *unfold* transformation step on the goal `reverse(Xs,Rs)` in the body of this clause. The goal `reverse(Xs,Rs)` is replaced by the body of a clause of which the head unifies with `reverse(Xs,Rs)`. Informally stated: a goal in the body of a clause is replaced by its 'definition'. As the goal `reverse(Xs,Rs)` unifies with the head of both defining clauses for the reverse predicate, we get two new clauses for the revacc predicate. In order to avoid identification of distinct variables, the variable `Rs` from the second reverse clause has been renamed to `Rs_n`.

```
revacc([],Ys,Acc):-append([],Acc,Ys).
revacc([X|Xs],Ys,Acc):-reverse(Xs,Rs_n),append(Rs_n,[X],Rs),
                       append(Rs,Acc,Ys).
```

When we apply the unfold transformation step to the append goal in the first clause listed above, this clause simplifies to:

```
revacc([],Ys,Ys).
```

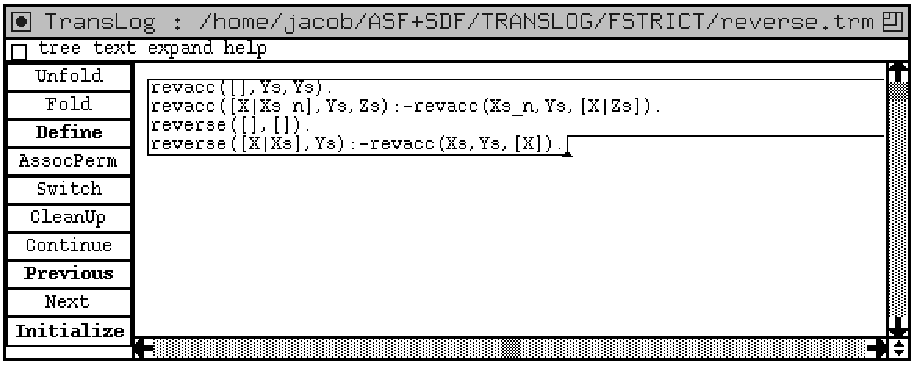

Figure 2: The final result of the transformation sequence

The body of the first append clause is empty, so the append goal in the body of the revacc clause vanishes.

Some predicates represent an associative operation. This holds for the append predicate: appending the lists L1 and L2 to a list R12, followed by appending R12 and L3, yields the same result as appending L2 and L3 to a list R23, followed by appending L1 and R23. So we have that the two sequences of goals

`append(L1,L2,R12),append(R12,L3,R123)` and

`append(L2,L3,R23),append(L1,R23,R123)`

are equivalent. This equivalence corresponds with the *AssocPerm* transformation step in which the predicate arguments are permuted. We apply this transformation to the append goals in the second clause of the revacc predicate. We get:

```
revacc([X|Xs],Ys,Acc):-reverse(Xs,Rs_n),append([X],Acc,Rs),
                       append(Rs_n,Rs,Ys).
```

We unfold the first append goal in the body of this clause:

```
revacc([X|Xs],Ys,Acc):-reverse(Xs,Rs_n),append([],Acc,Zs),
                       append(Rs_n,[X|Zs],Ys).
```

And unfold the resulting append goal (with the empty list):

```
revacc([X|Xs],Ys,Acc):-reverse(Xs,Rs_n),append(Rs_n,[X|Acc],Ys).
```

The body of this clause shows a close resemblance with the body of the original defining clause for the revacc predicate. We now apply a *fold* transformation step. Informally stated, this transformation step is the inverse operation of the unfold transformation step. A sequence of goals in the body of a clause (unifiable with the body of another clause) is replaced by a single goal (the head of that other clause, modulo some variable substitutions). We get:

```
revacc([X|Xs],Ys,Acc):-revacc(Xs,Ys,[X|Acc]).
```

We also apply a fold step on the second reverse clause:

```
reverse([X|Xs],Ys):-revacc(Xs,Ys,[X]).
```

With a *CleanUp* transformation step we remove all clauses that are not ‘reachable’ from the reverse clauses. We are left with the program that is displayed in Figure 2. The revacc predicate is recursively defined. The reverse predicate is defined in terms of the revacc predicate. In the definition of this predicate a list is traversed only once. The append predicate is not needed any more and has been removed from the program.

3 TransLog specification outline

The ASF+SDF specification of TransLog consists of twenty-three modules. In this paper we present the specification at a global level. Only a few interesting parts are completely displayed and discussed. This section describes the global structure of the specification. The set of TransLog modules can be divided in five subsets:

1. Basic modules. These modules contain some elementary specification of layout characters and booleans. They are present in almost every ASF+SDF specification. We pay no attention to these modules.

2. Syntax modules. The Prolog syntax is specified in two modules. The TransLog specification uses a number of functions on (a part of) a Prolog program. These functions are grouped in four more syntax modules. The syntax specification is discussed in Section 4.

3. Unification modules. The unification of Prolog terms is a basic operation in the TransLog specification. Three modules are concerned with unification. The specification of the unification part is discussed in Section 5.

4. Transformation modules. Each of the six TransLog transformation steps is specified in a separate module. In Section 6 we present the Unfold and Fold modules up to some detail. The modules related to the other transformation steps are briefly discussed.

5. User interface modules. The TransLog user interface is specified in five modules. They are discussed in Section 7.

4 Syntax

The TransLog tool operates on Prolog programs that are written in the Edinburgh syntax. The syntax definition is based on the SICStus Prolog manual ([17]). The SDF definition does not cover the complete Prolog syntax, but is merely a relevant subset:

- Only integer numbers are defined. Floating point numbers, binary numbers, octal numbers and hexadecimal numbers will not be recognized by the TransLog tool.

- Not all symbol-characters are allowed. A subset, directed to the definition of arithmetic operators, is specified.

- A semicolon between two goals (disjunction) is not defined.

- Infix notation of a predicate with two arguments (e.g. “X is 5”) is defined. However, no priorities are defined on various built-in operators. This means that terms like “X is 3+2” are ambiguous. This ambiguity has to be solved by the user.

In Prolog a list can be denoted in different ways, e.g. $[a, b, c]$ and $[a \mid [b, c]]$. In the TransLog specification a function is defined that rewrites all list terms to a cons-list: a full stop atom with two arguments, the first element of the list and a list with the remaining arguments. The same function is used to normalize a term with an infix operator to prefix notation. This normalization function is specified for efficiency reasons: by first normalizing a term to the normal form *atom(termlist)*, for many operations on this kind of terms the number of equations is significantly reduced.

Furthermore, the syntax part of the TransLog specification contains a number of basic functions on Prolog terms and programs. They can be grouped in four subsets:

- Boolean functions to test a property of a certain term: Is a term a variable, an integer? Is a list empty? etc.

- A boolean function to check the syntax of a Prolog program. It should be noticed that the correctness of the lexical syntax of Prolog terms and the basic syntax of a Prolog program is already guaranteed for any term of the sort PROGRAM. The function *syntaxCheck* only checks if a term is allowed as the head of a clause or as a goal in the body of a clause.

- Functions that remove a clause from a program, add a clause to a program, isolate the head of a clause, etc.
- For a correct Fold transformation step labeling of goals in the body of a clause is required. A set of functions is provided that add a label to a term, remove a label from a term or change the label of a term.

A transformation tool that is implemented in Prolog itself will have most of the syntax part 'for free'. However, the specification of the Prolog syntax in ASF+SDF is rather straightforward, as is the specification of the functions listed above. Therefore, we have not experienced the extra effort as a serious drawback.

5 Unification

The unification of two Prolog terms plays an important role in several transformation steps. The TransLog specification of unification is based on the Martelli–Montanari unification algorithm [11]. This algorithm has a set of equations $T_1 = T_2$ as input. In a number of steps this set is transformed in a new set of equations. When no more steps can be executed (and no failure is detected), the final set contains the *most general unifier* (mgu) of T_1 and T_2. The output of the TransLog unification function is not this set of equations, but a sequence of *substitutions* $V \mapsto T$, or a failure. This shows what this function is needed for: the resulting substitutions can be applied to clauses, clause bodies, terms, etc.

The unification part of the TransLog specification consists of three modules. Besides the unification algorithm, also the syntax of Prolog term equations and the effect of substitutions on Prolog terms is specified in these modules. A function *mgu* produces a sequence of substitutions. In the specification a substitution θ on a construct C is denoted by $C[i\ S]$, with i an identifier indicating on which construct a substitution is supposed to be performed (t for term, tl for term-list, es for equation sequence, etc.). S denotes a sequence of substitutions.

There exist similar unification specifications within the context of ASF+SDF, see e.g. Chapter 5 of [19].

The specification of the unification part of TransLog is rather straightforward. The specification of the unification function is close to the definition of the Martelli–Montanari unification algorithm.

6 Transformation steps

The TransLog tool offers six transformation steps for Prolog programs: unfold, fold, define, associativity permutation, goal switching and program clean up. Each transformation step is specified in a separate module. In this paper we present the core of the two most interesting modules, the Unfold module and the Fold module. The other modules are briefly discussed.

6.1 Unfold

Within the context of Prolog semantics two particular unfold transformation steps are semantics preserving (cf. [13]):

1. *Leftmost unfolding.* The unfold transformation is applied to the leftmost goal in the body of the unfolded clause.
2. *Deterministic non-left-propagating unfolding.* There exists one unfolding clause. No undesirable variable substitutions propagate to the head of the unfolded clause or the goal(s) left of the unfolded goal in the body of the clause.

Unfolding the leftmost goal of a body is always allowed. In case of non-leftmost unfolding, determinism (the presence of at most one unfolding clause) is needed in order to maintain the order of the computed answers. Left-propagation of variable substitutions in body goals is forbidden, because the termination properties of a program may change. For example, let P_0 be the following program:

```
p(X):-q(X),r(X).    q(a):-q(a).    r(b).
```

Unfolding `r(X)` in the first clause results in the program P_1:

```
p(b):-q(b).    q(a):-q(a).    r(b).
```

The query "`?- p(X)`" is non-terminating for P_0, for P_1 the same query immediately fails.

The following algorithm implements the unfold transformation step. Let $C = T'$:- $T^*, T, T^{*\prime}$. be the unfolded clause. Let T be the body goal to-be-unfolded. Let P be the current program. The unfold transformation step is defined as follows:

for all clauses C' in the program P:
  if head(C') unifies with T (mgu(head$(C'), T) = \theta,\ \theta \neq$ fail)
    then if the applicability conditions are satisfied
      then construct a variant of C' with no variables in common with C
        construct a resolvent of C : $C'' = (T'$:- T^*, body variant of $C', T^{*\prime}.)\theta$
        add C'' to the resulting clause(s)
      else the result is an error message
if the result is an error message
  then return this message
  else in P replace C by the resulting clause(s).

We now turn to the algebraic specification of this algorithm in the module Unfold. We present the equations that represent the core of the algorithm given above. The 'work' is done by the function *unfoldResult*. This function has three arguments: the current (labeled) program, the body goal to-be-unfolded and the (labeled) clause that holds this body goal. The function outputs either the clauses that have to be added to the program as the result of the unfold operation, or an error message in case the applicability conditions are not satisfied. In the first case in the current program the unfolded clause is replaced by the new clause(s), in the second case only an error message is returned. See Section 7 for more details concerning error messages.

For performance reasons the equations below are written in a specification style in which the number of conditions is minimized. This style enlarges the number of (equations for) help-functions (the functions *unf1* to *unf4* in the equations below).

The function *unfoldResult* recursively investigates all clauses of the program. First, a variant of the investigated clause LC' is constructed that has no variables in common with the unfolded clause LC.

$$\text{unfoldResult}(,\ T,\ LC) =$$

$$\frac{\text{variant}(LC',\ LC) = LC''}{\text{unfoldResult}(LC'\ LC^*,\ T,\ LC)\ =\ \text{unf1}(LC''\ LC^*,\ T,\ LC)}$$

If the head of the investigated clause LC' is unifiable with the body goal to-be-unfolded T (the result of the *mgu-nv* function is not equal to *fail*), then the clause is selected as an unfolding clause and the applicability conditions are checked. Otherwise the next clause in the program is investigated.

$$\frac{\text{mgu-nv}(T,\ \text{lhead}(LC')) \neq \text{fail}}{\text{unf1}(LC'\ LC^*,\ T,\ LC)\ =\ \text{unf2}(LC'\ LC^*,\ T,\ LC)}$$

$$\frac{\text{mgu-nv}(T,\ \text{lhead}(LC')) = \text{fail}}{\text{unf1}(LC'\ LC^*,\ T,\ LC)\ =\ \text{unfoldResult}(LC^*,\ T,\ LC)}$$

If the body goal to-be-unfolded T is the leftmost goal of the body of the unfolded clause LC, then a resulting clause is constructed with the function *resolvent*. This clause is added to the other resulting clauses with the function *la-addcp*.

$$\frac{\text{leftmostUnfold}(T,\ LC) = \text{true}}{\text{unf2}(LC'\ LC^*,\ T,\ LC)\ =\ \text{la-addcp}(\text{resolvent}(LC',\ T,\ LC),\ \text{unfoldResult}(LC^*,\ T,\ LC))}$$

Otherwise, the conditions concerning deterministic unfolding and non-left propagating unfolding are checked. If one of these conditions is not satisfied, an error message is returned. If both conditions are satisfied, the resulting clause is constructed and returned as the result of the function *unfoldResult*.

$$\frac{\text{leftmostUnfold}(T,\, LC) = \text{false}}{\text{unf2}(LC'\; LC^*,\, T,\, LC) \;=\; \text{unf3}(LC'\; LC^*,\, T,\, LC)}$$

$$\frac{\text{noMoreUclauses}(T,\, LC^*) = \text{true}}{\text{unf3}(LC'\; LC^*,\, T,\, LC) \;=\; \text{unf4}(LC',\, T,\, LC)}$$

$$\frac{\text{noMoreUclauses}(T,\, LC^*) = \text{false}}{\text{unf3}(LC'\; LC^*,\, T,\, LC) \;=\; \text{set-label}(\texttt{el};\, \text{standard derrorm})}$$

$$\frac{\text{mgu-nv}(T,\, \text{lhead}(LC')) = S,\;\; \text{isVariant}(T,\, LC,\, S) = \text{true},\;\; \text{resolvent}(LC',\, T,\, LC) = LC''}{\text{unf4}(LC',\, T,\, LC) \;=\; LC''}$$

$$\frac{\text{mgu-nv}(T,\, \text{lhead}(LC')) = S,\;\; \text{isVariant}(T,\, LC,\, S) = \text{false}}{\text{unf4}(LC',\, T,\, LC) \;=\; \text{set-label}(\texttt{el};\, \text{standard verrorm})}$$

The boolean functions *leftmostUnfold*, *noMoreUclauses* and *isVariant* check the applicability conditions. The definition of these functions is straightforward.

6.2 Fold

The applicability conditions for a fold transformation step are very complex. Following Pettorossi and Proietti, the TransLog tool applies Tamaki & Sato folding ([18]). This folding puts some restrictions on multiple occurrences of variables and the predicate symbol in the head of the folding clause. For preservation of Prolog semantics it is furthermore required that a folding step either is *reversible*, or, in case the head of the folded clause is a new predicate[1], the leftmost body goal of the folded clause is labeled *fold-allowing*. A folding step is reversible if the folding clause is in the same program P_i as the folded clause and these clauses do not coincide. (So, an unfold step in the resulting program P_{i+1} may result in the program P_i again.) According to the second condition, body goals are labeled *fold-allowing* or *not-fold-allowing*. Initially, the body goals of P_0 are labeled *fold-allowing*. A transformation step may change the label of a body goal. See [13] for details.

The following algorithm implements the fold transformation step.

Let $C = T$:- $T^*, T^+, T^{*\prime}$. be the folded clause. Let T^+ be the body goals to-be-folded. Let P be the current program, let P_{new} be the program consisting of new defined clauses.

If there exists a clause D in P whose body is unifiable with T^+
  then if the Tamaki & Sato conditions are satisfied
    then apply a (reversible) folding step
  else if there exists a clause D in P_{new} whose body is unifiable with T^+
    then if the predicate of head(D) is defined in P
      then if the Tamaki & Sato conditions are satisfied
        then if the head predicate of C is an old predicate or the leftmost body goal of C
            is labeled *fold-allowing*
          then apply the folding step
          else report error: no old predicate or label *fold-allowing*
        else report error: violation of T&S conditions
      else report error: folding clause with undefined head predicate

[1] A *new* predicate is not defined in P_0, but added to the program with a *define* transformation step. The predicates defined in P_0 are called *old* predicates.

else report error: no folding clause present in P or P_{new}

A folding step implies the replacement of the clause C by the clause $C' = T$:- T^*, head$(D)\theta, T^{*\prime}$., with θ the mgu of T^+ and the body of the folding clause D.

The fold algorithm has been implemented in a set of conditional equations. As with the unfold transformation step, we only present the equations that specify the core of the algorithm.

The function *fold* has five arguments: the number of body goals to-be-folded (stored in a term T), the first body goal of the sequence of body-goals to-be-folded T', the folded clause C, the program P_{new} with the clauses resulting from define transformation steps (LP), and the current (labeled) program LP'. After selection of the appropriate part of the body of the folded clause, reversible folding is tried first.

$$\frac{\mathrm{selectBodyPart}(T;\ T';\ C) = T^+}{\mathrm{fold}(T,\ T',\ C,\ LP,\ LP') \ = \ \mathrm{revFold}(T^+;\ C;\ LP;\ LP')}$$

This means that the current program (stripped from its labels and without the folded clause) is searched for a folding clause. If this clause is found, a variant of this clause is subjected to further investigations. If not, clauses from P_{new} are investigated with the function *newFold*.

$$\frac{\begin{array}{c}\mathrm{strip\text{-}label}(LP') = P,\ \ \mathrm{removecp}(C,\ P) = P',\ \ \mathrm{foldingClause}(T^+;\ P') = C',\\ C' \neq \texttt{'fail'}\ .,\ \ \mathrm{variantC}(C',\ C) = C''\end{array}}{\mathrm{revFold}(T^+;\ C;\ LP;\ LP') \ = \ \mathrm{rfld1}(T^+;\ C;\ C'';\ LP;\ LP')}$$

$$\mathrm{revFold}(T^+;\ C;\ LP;\ LP') = \mathrm{newFold}(T^+;\ C;\ LP;\ LP') \quad \textbf{otherwise}$$

Further investigations: are the Tamaki & Sato conditions satisfied? If so, a new clause is constructed. This clause replaces the folded clause in the current program.

$$\frac{\begin{array}{c}\mathrm{mgu\text{-}tl}(T'^+;\ T^+) = S,\ \ S \neq \mathrm{fail},\ \ \mathrm{tsFolding}(T';\ T'^+;\ T^*;\ T'^*;\ S) = \mathrm{true},\\ \mathrm{la\text{-}version\text{-}C}(T \text{ :- } T^*,\ T^+,\ T'^*\ .\ ;\ LP') = LC,\ \ \mathrm{laFoldedClause}(T^+;\ T'\ [\mathrm{t}\ S];\ LC) = LC'\end{array}}{\mathrm{rfld1}(T^+;\ T \text{ :- } T^*,\ T^+,\ T'^*\ .\ ;\ T' \text{ :- } T'^+\ .\ ;\ LP;\ LP') \ = \ \mathrm{la\text{-}replacecp}(LP',\ LC,\ LC')}$$

If the T&S conditions are not satisfied, the clauses from P_{new} are investigated as well.

$$\mathrm{rfld1}(T^+;\ C;\ C';\ LP;\ LP') = \mathrm{newFold}(T^+;\ C;\ LP;\ LP') \quad \textbf{otherwise}$$

Search P_{new} for a folding clause. If such a clause is found, then continue.

$$\frac{\mathrm{strip\text{-}label}(LP) = P,\ \ \mathrm{foldingClause}(T^+;\ P) = C',\ \ C' \neq \texttt{'fail'}\ .,\ \ \mathrm{variantC}(C',\ C) = C''}{\mathrm{newFold}(T^+;\ C;\ LP;\ LP') \ = \ \mathrm{nfld1}(T^+;\ C;\ C'';\ LP;\ LP')}$$

Otherwise, return an error message.

$$\mathrm{newFold}(T^+;\ C;\ LP;\ LP') = \mathrm{set\text{-}label}(\texttt{el};\ \mathrm{standard\ ferrorm}) \quad \textbf{otherwise}$$

Continuation: test if the head predicate of the folding clause is defined in the current program. If so, then continue.

$$\frac{\mathrm{isDefined}(T;\ LP') = \mathrm{true}}{\mathrm{nfld1}(T^+;\ C;\ T \text{ :- } T'^+\ .\ ;\ LP;\ LP') \ = \ \mathrm{nfld2}(T^+;\ C;\ T \text{ :- } T'^+\ .\ ;\ LP;\ LP')}$$

Otherwise, return an error message.

$$\mathrm{nfld1}(T^+;\ C;\ C';\ LP;\ LP') = \mathrm{set\text{-}label}(\texttt{el};\ \mathrm{standard\ f2errorm}) \quad \textbf{otherwise}$$

Continuation: check the T&S conditions. If OK, then continue.

$$\frac{\text{mgu-tl}(T'^{+};\ T^{+}) = S,\ \ S \neq \text{fail},\ \ \text{tsFolding}(T';\ T'^{+};\ T^{*};\ T'^{*};\ S) = \text{true}}{\begin{array}{c}\text{nfld2}(T^{+};\ T \text{ :- } T^{*},\ T^{+},\ T'^{*}\ .\ ;\ T' \text{ :- } T'^{+}\ .\ ;\ LP;\ LP')\ = \\ \text{nfld3}(T^{+};\ T \text{ :- } T^{*},\ T^{+},\ T'^{*}\ .\ ;\ T' \text{ :- } T'^{+}\ .\ ;\ LP;\ LP';\ S)\end{array}}$$

Otherwise, return an error message.

$$\text{nfld2}(T^{+};\ C;\ C';\ LP;\ LP') = \text{set-label}(\texttt{el};\ \text{standard f3errorm}) \quad \textbf{otherwise}$$

Continuation: the 'final' conditions are checked: is the head predicate of the folded clause an old predicate or is the left-most body goal labeled 'fold-allowing'? If the final test succeeds, in the current program the folded clause is replaced by the resulting clause.

$$\frac{\begin{array}{c}\text{la-version-C}(T \text{ :- } T^{*},\ T^{+},\ T'^{*}\ .\ ;\ LP') = LC,\ \ \text{finalConditions}(LC;\ LP) = \text{true}, \\ \text{laFoldedClause}(T^{+};\ T'\ [\text{t}\ S];\ LC) = LC'\end{array}}{\text{nfld3}(T^{+};\ T \text{ :- } T^{*},\ T^{+},\ T'^{*}\ .\ ;\ T' \text{ :- } T'^{+}\ .\ ;\ LP;\ LP';\ S)\ =\ \text{la-replacecp}(LP',\ LC,\ LC')}$$

If the final test fails, an error message is displayed.

$$\text{nfld3}(T^{+};\ C;\ C';\ LP;\ LP';\ S) = \text{set-label}(\texttt{el};\ \text{standard f4errorm}) \quad \textbf{otherwise}$$

6.3 Other transformation steps

In the remainder of this section we give a brief description of the other transformation modules.

A new clause (predicate definition) can be added to a program by the define transformation step. This step is specified in the module Define. Several checks are performed before a new clause is added to the current program. First, the syntax is checked. If this check fails, an error message is returned instead of a new program. Next, it is checked if the predicate defined in the new clause indeed is a new predicate: it should not already have been defined in the current program or in previously defined new clauses. Finally, the new clause has to be non-recursive. Before the new clause is added to the program its body goals are labeled *not-fold-allowing*.

The replacement of the goal sequence `p(A,B,E),p(E,C,D)`' by the sequence '`p(B,C,F),p(A,F,D)`' is specified in the module AssocPermutation. It is supposed that the predicate `p` represents an associative operation, defined with recursion on its first argument (like the append predicate). For this transformation we look at predicates with arity 3, of which an operation on the first two arguments produces the third argument. Sometimes, as with list concatenation, the inverse operation mode is also possible: given the third argument, extract the first two arguments. This leads to two modes of operation: the 'i,i,o' mode and the 'o,o,i' mode (with 'i' for input argument and 'o' for output argument). Both modes have been implemented in the TransLog tool.

The AssocPerm transformation step is an instance of a *goal replacement* step. In general the validity of a goal replacement step is undecidable. In the TransLog tool no applicability conditions are checked. The user has to be aware of the fact that this transformation step is not semantics preserving in all cases.

While preparing the body of a clause for a certain transformation step (folding, associativity permutation), it may be required to switch the order of two adjacent body goals. This is a potentially dangerous operation, because of shared variables or changing termination properties. However, the switching step is needed in many examples. Therefore, we have incorporated this transformation step in the TransLog tool.

At the end of a transformation sequence the current program often contains several clauses that are of no use any more. They are not reached from the clauses that contain the definition of relevant predicates. Therefore, they safely can be removed.

In the module CleanUp a transformation step with the same name has been implemented as semantics preserving *with respect to a particular selected predicate*. The clause(s) defining this predicate, as well as the clauses defining the predicates that can be reached from these clauses, are not removed from the program by a CleanUp transformation step. All other clauses are removed.

The specification of the transformation steps represents the core of the TransLog tool. The list-matching properties and conditional equations make the ASF+SDF formalism very suitable for this job. However, a critical remark is in order. Like every 'declarative' formalism, ASF+SDF lacks the notion of state variables. So, it is not possible to (temporary) store the result of a computation. This induces a specification style in which results have to be computed twice or are carried along as function parameters, leading to functions with many parameters. The equations presented in this section illustrate this point.

7 User interface

In the design of the TransLog tool a choice has been made for a 'select-and-click' user interface. The execution of a transformation step includes the selection of an expression (program, clause, goal), followed by clicking on a button that represents a transformation step. The folding and goal definition transformation steps require some additional editing. The specification of the user interface comprises

1. The storage of a transformation sequence $P_0 \ldots P_n$.

2. The displaying of an error message when necessary.

3. The 'definition' of the user buttons.

In this section we will briefly discuss these items.

The TransLog tool offers navigation options with respect to a sequence of transformed programs. Therefore, during the process of transforming an initial program P_0 the intermediate results (labeled programs) have to be stored somewhere. Before displaying a program in a TransLog term-window, the labels are stripped from the program.

As ASF+SDF does not offer 'state variables', a program cannot be stored in a global variable during a sequence of transformation steps. We have chosen to store a program sequence in two separate ASF+SDF term-windows. Suppose, we have a transformation sequence $P_0, P_1, \ldots, P_i, \ldots, P_n$, with P_i the current program that is displayed in the TransLog window. One storage window contains the 'past and presence' of the sequence: the programs $P_0, P_1, \ldots, P_i$. The other storage window contains the 'future' of the sequence: the programs $P_{i+1}, \ldots, P_n$. In case $i = n$, the second window is empty. The two-window solution has been chosen because of its 'efficiency': after a transformation step the new program is appended at the end of a sequence of programs, instead of inserted somewhere in a sequence. The two storage windows are created by clicking on the Initialize button at the beginning of a transformation sequence. Their contents are of no interest for a user.

If a transformation step is not allowed an error message is displayed in the TransLog window. The choice has been made to give an error message the syntax of a Prolog program (clause). This makes it easy to present an error message in a program window. Error messages are defined as unit clauses: a quoted atom, followed by a full stop. All error messages have the same first line, the unit clause `'The selected transformation is not allowed'.`. A second line explains why, e.g. `'No deterministic unfolding of non-left body term possible'.`, or `'T&S conditions fail on folding clause from Pnew'.`.

The ten buttons at the left side of a TransLog program window have been defined in a script written in the special-purpose language SEAL ([10]). SEAL offers the opportunity to enable/disable a button according to a condition that is based on the actual contents of (a part of) the TransLog window. These button conditions are specified in boolean ASF+SDF functions. It would have been possible to include in these functions the complete set of applicability conditions for all transformation steps. In that case a button would only be enabled if the transformation step is allowed. We have chosen not to follow this approach, for three reasons:

1. While determining the outcome of the applicability conditions, a lot of 'computations' have to be performed which are also useful for the required transformation step (e.g. the computation of an mgu). If we separate the test

on the conditions from the execution of a transformation step, we will have to perform this kind of computations twice. In case of integration of the test and the transformation step, in most cases the computation is required only once.

2. By integrating the test on conditions and the execution of a transformation step it is possible to return a to-the-point error message instead of a transformed program. If the conditions are checked before enabling/disabling a button, this is not possible.

3. Evaluating the applicability conditions for all transformation steps requires a considerable amount of time. This means that after every update of the TransLog term-window –when the system re-evaluates the button conditions– the user will have to wait. We have chosen to avoid this delay.

The contents of the TransLog term-window (displaying the actual program in a transformation sequence) are 'pretty-printed', so they are presented to the user in a readable format. To this extent a separate pretty-printer has been defined, using the tools described in [3].

The design and implementation of the user interface part of the TransLog tool has been a rather complicated job. Both the storage of a program sequence and the definition of the button conditions were non-trivial problems. The two-window solution of the storage problem is rather artificial. Although SEAL is a more or less imperative language, the scope of an assignment is restricted to the definition of a single button. A global assignment would have solved the storage problem in a more elegant way.

8 Concluding remarks

We summarize the conclusions drawn at the end of the previous sections. Furthermore, we mention some topics for future research.

Most of the existing transformation tools for logic programs have been implemented in Prolog. The advantages are clear: no syntax definition is needed, basic operations like unification are 'for free'. In our experience, on this point the algebraic approach of ASF+SDF is not really at a disadvantage. The SDF syntax definition is small and simple (also due to the simple syntax of Prolog). With respect to the extension of the syntax with 'labeled programs' (as needed for a folding step), an explicit syntax specification is an advantage.

With its conditional equations and list matching properties, ASF+SDF is very suitable for the specification of (conditional) transformation steps on programs. A few critical remarks: due to the absence of state variables in an algebraic specification, intermediate results cannot be stored for later use. This implies that some computations have to be performed more than once, e.g. in the evaluation of a condition and in the computation of a result. We notice that similar conclusions can be found in [6].

Compared to other transformation tools TransLog is small, both in its lines of code (23 modules with about 325 equations) and its functionality. What is lacking most is a more or less automatized 'transformation strategy'. In the example given in Section 2, the user is responsible for the sequence of selected transformation steps. This selection process can more or less be automatized in an incorporated transformation strategy. For a general unfold/fold transformation system (like TransLog), this is not simple, further research will be needed.

The performance of the TransLog tool can be characterized as satisfying. The execution of a single transformation step takes a few seconds on a fast machine. We emphasize that performance has not been a major topic in the design of the TransLog tool.

Acknowledgements Krzysztof Apt (CWI and University of Amsterdam) is acknowledged for his initiating and stimulating role with respect to the work described in this paper. The ASF+SDF specification has benefited from unpublished work of Arie van Deursen (CWI) on unification. Thanks to Eelco Visser (University of Amsterdam) for commenting the TransLog specification.

References

[1] O. Aioni. A System for the Automatic Transformation of Logic programs. Technical Report Report R 0195, Electronics Department, University of Roma Tor Vergata, 1995.

[2] F. Alexandre, K. Bsaïes, J.P. Finance, and A. Quéré. Spes: A System for Logic Program Transformation. In *Proceedings of the International Conference on Logic Programming and Automated reasoning (LPAR '92)*, LNCS 624, pages 445–447. Springer–Verlag, 1992.

[3] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[4] J.J. Brunekreef. A transformation tool for pure Prolog programs. In J.P. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96*, number 1207 in LNCS, pages 130–145. Springer-Verlag, 1996.

[5] J.J. Brunekreef. A Transformation Tool for Pure Prolog Programs –The Algebraic Specification–. Technical Report P9607, Programming Research Group, University of Amsterdam, 1996.

[6] S. Eijkelkamp, D. Geluk, and M. Polling. Program Transformation using ASF+SDF. Master's thesis, University of Amsterdam, 1995.

[7] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive Partial Deduction in Practice. In J.P. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96*, number 1207 in LNCS. Springer-Verlag, 1996.

[8] P. Klint. A meta–environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[9] J. Komorowski. A Prolegomenon to Partial Deduction. *Fundamenta Informaticae*, 18(1):41–64, 1993.

[10] J.W.C. Koorn. Connecting semantic tools to a syntax–directed user–interface. In H.A. Wijshof, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'93*, pages 217–228. SION, 1993.

[11] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[12] T.Æ. Mogensen and A. Bondorf. Logimix: A Self-Applicable Partial Evaluator for Prolog. In *LOPSTR '92 – Proceedings of the second international workshop on Logic Program Synthesis and Transformation*, Workshops in Computing, pages 214–227. Springer–Verlag, 1992.

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

[14] S. Prestwitch. Online Partial Deduction of Large Programs. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*, pages 111–118. ACM press, 1993.

[15] D. Sahlin. The Mixtus Approach to Automatic Partial Evaluation of Full Prolog. In S. Debray and M. Hermenegildo, editors, *Proc. of the North American Conf. on Logic Programming*, pages 377–398, 1990.

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

[17] SICS – Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog User's Manual*, 1992.

[18] H. Tamaki and T. Sato. Unfold/Fold Transformation of Logic Programs. In S.-Å. Tärnlund, editor, *Proceedings of the 2nd International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

[19] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping – an Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co, Singapore, 1996.