TransLog, an interactive tool for transformation of logic programs

Jacob Brunekreef

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

and

Programming Research Group, University of Amsterdam

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

e-mail: `jacob@fwi.uva.nl`

Abstract

This report describes 'TransLog', a prototype of a transformation tool for logic programs. The tool has been developed with the ASF+SDF Meta–environment, a programming environment generator tool based on algebraic specification. TransLog supports the interactive transformation of (a part of) a program by means of buttons representing transformation rules. This report contains the complete annotated algebraic specification of TransLog.

Keywords: logic programming, algebraic specification, program transformations.

1 Introduction

The subject of program transformation deals with transforming a simple-but-inefficient program into a more efficient program. An initial program P_0 is modified by applying a sequence of well-defined *transformation rules*, resulting in a program sequence $P_0 \ldots P_k$. As much as possible, the semantic properties of a program are maintained during a transformation sequence.

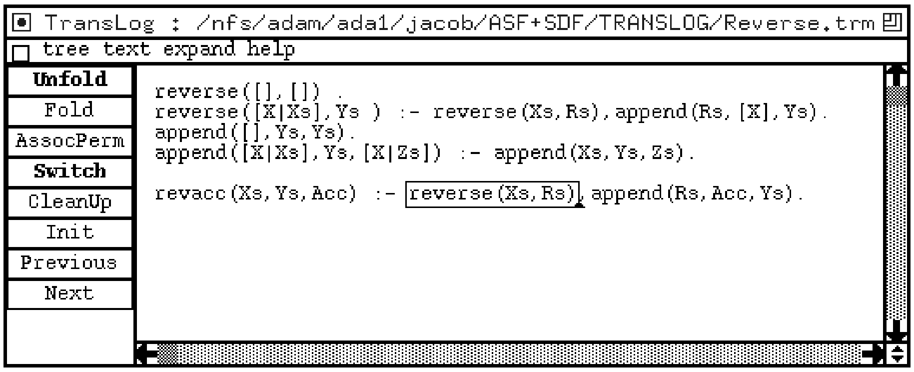
Program transformations have been studied since the seventies. We refer to the work of Burstall & Darlington [BD77], introducing transformations on recursion equations. In the field of logic programming, program transformations have been studied since the eighties. We refer to Tamaki & Sato [TS84] and Seki [Sek91]. A recent overview can be found in [PP94].

In this report we present *TransLog*, a tool for transformation of logic programs. A prototype of the tool has been developed by means of algebraic specification. This technique appears to be well-suited for specifying the kind of operations that are required within the context of program transformation. The specification has been created with the ASF+SDF Meta–environment of Klint et al. [Kli93]. This system offers an interactive development environment for the generation of programming environments. Moreover, a window-oriented user-interface can simply be defined within the system.

The report is structured as follows. In Section 2 the subject of transformation of logic programs is informally introduced. Section 3 provides a very short introduction to the ASF+SDF formalism. In this section also the choice for the algebraic approach is motivated. In Section 4 a number of design decisions is presented and discussed. The complete specification is presented in the Sections 5 to 10. A final section with some concluding remarks completes the report. The reader is supposed to be familiar with elementary logic (Prolog) programs and the basic concepts of algebraic specification.

Related work is presented by, among others, Komorowski [Kom90, KT94], Sahlin [Sah91], Alexandre et al. [ABFQ92] and Galagher [Gal86]. What is different in our work is the algebraic approach, opposite to the more or less Prolog-like approach in the cited work. In [EGP95] a program transformation system for a functional programming language is specified in ASF+SDF.

We conclude this introduction with an example of what a TransLog window looks like. It shows a Prolog program with one body goal selected and, at the left side, a number of buttons representing various transformations. Its contents will be explained in detail in Section 10.



2 Transformations of logic programs

We first present some transformations of logic programs in an informal way. In the following example, taken from unpublished work of A. Bossi, a program for list reversal is transformed into a more efficient version. See [PP94] pages 300–302 for a similar example. The example is given in the (Edinburgh) Prolog syntax.

Consider the following program for the reversal of a list:

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

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
```

This program traverses the list twice: once within the recursive reverse clause, once within the recursive append clause. As a first step towards a more efficient program we introduce 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 on the goal `reverse(Xs,Rs)` in the body of this clause. We replace the goal `reverse(Xs,Rs)` 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 clause for `reverse` 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).
```

By applying the same unfold transformation to the append goal in the first clause listed above, this clause simplifies to:

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

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, when applied in the (+,+,-) mode: 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 a transformation in which the predicate arguments are permuted. We apply this transformation to 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).
```

Now 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. Informally stated, this transformation is the inverse operation of the unfold transformation. 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]).
```

By removing all clauses that are not 'reachable' from the revacc clauses, we are left with the following program:

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

which represents a more efficient program for list reversion: in this program the list is traversed only once.

Summarising, in this small example we have applied four different transformations:

- Unfolding: a goal in the body of a clause is replaced by its 'defining body'. In case there is more than one unfolding clause, the unfolded clause is replaced by more than one clause. (In case there is *no* unfolding clause, the unfolded clause is simply removed from the program.)

- Folding: a sequence of goals in the body of a clause is replaced by a single goal. This goal corresponds with the head of a folding clause which has the replaced sequence of goals as a body.

- Associativity permutation: for 'associative predicates' the arguments may be permuted in a strict way.

- Program cleaning: all clauses that cannot be reached from a selected clause are removed from the program. This transformation may lead to a number of *definition eliminations*.

A formal definition of these transformations, as well as a discussion concerning the preservation of semantic properties, can be found in e.g. [Sek91, PP94].

3 The ASF+SDF Meta–environment

The ASF+SDF Meta–environment ([Kli93]) is an interactive development environment for the generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. The generation process is controlled by a definition of the language, which may include features such as syntax definition and checking, type checking, prettyprinting and execution of programs in the target language.

SDF is a shorthand for Syntax Definition Formalism. In SDF both the lexical syntax and the context–free syntax of a language can be defined in an algebraic style. ASF is a shorthand for Algebraic Specification Formalism. In ASF any function may be defined on terms, constructed according to the syntax defined in an SDF specification. In our case such a term will be a program, a clause, a goal. Functions will model transformations on these terms.

ASF+SDF specifications have a modular structure. Different parts of a specification can be written down in separate modules. In each module it is explicitly defined which part is visible to the 'outside world' and which part is hidden (local to the particular module). A module can be imported in another module. Only the visible part of an imported module (and of all modules imported in that module) is accessible to the importing module.

The ASF+SDF Meta–environment offers the possibility of syntax–directed editing of ASF+SDF specifications. As mentioned before, a specification consists of a series of modules. Each individual module can be created and edited by invoking a *module editor* for it. After each editing operation the implementation of a module is updated immediately. A lexical scanner, a parser, a pretty printer and a term rewriting system (the implementation of the set of equations) are derived from the module automatically.

For each module one or more *term editors* may be invoked in order to edit and evaluate terms defined by the particular module. A term editor uses the syntax defined in the module for parsing the textual representation of terms. The equations of the module can be applied to reduce a term to its normal form. A term editor can be customised by adding buttons or pull–down menus to the editor window. In this way the user-interface of a particular application can be designed. Such a user-interface is defined using the special purpose language SEAL ([Koo93]).

As noticed in the introduction, most work related to tooling of transformations on logic programs is based on Prolog. The advantages are clear: Prolog is suitable for the symbolic manipulations that are required, furthermore one gets some things (e.g. unification) for free. In this report an algebraic approach, based on the ASF+SDF Meta–environment, is investigated. This choice is based on the following arguments:

- Syntactic freedom. Although for the current prototype of TransLog a Prolog–like syntax has been chosen (see the coming section), future modifications of the syntax of the language can simply be defined.

- Transformations on terms (in our case: goals, clauses, programs) can very well be specified in a set of algebraic equations. ASF+SDF, with its conditional equations, list matching properties and default equations, is a powerful formalism for the specification of program transformations.

- Simple definition of the user-interface. With the user-interface specification language SEAL that comes with the ASF+SDF Meta-environment it is easy to customise a window with a number of buttons and/or menus.
- Exploration of a 'new' technique. Although algebraic specification itself is not new, its application to the domain of logic programming seems to be. Exploration of this technique is valuable in itself.

4 Design decisions for a transformation tool

A number of design decisions had to be made while specifying a prototype of a transformation tool for logic programs. In this section we shortly list and motivate the most important ones.

1. Select a *Prolog–like syntax* for the Logic Program language.
   We have chosen to adhere to the Prolog syntax as much as possible. In particular, we do not distinguish between function and relation symbols and allow the use of the same relation symbol in different arities. Moreover, clauses will be written with the ':-' operator and with a full stop at the end. Infix function symbols (relations) will be allowed in the body of a clause. The choice for a Prolog syntax means that any program from the TransLog tool can be exported to a Prolog environment without any difficulty.

2. Implement just a few transformations.
   A tool with a restricted functionality is already useful for getting an idea of what should be incorporated in the final version. In the current prototype the four transformations listed in Section 2 (unfolding, folding, associativity permutations and program cleaning) have been implemented. One more transformation has been added to this list: goal switching – the order of two adjacent goals in the body of a clause is switched.

3. No restrictions on transformations: "if it can be done somehow, just do it".
   Most transformations are only allowed if a set of conditions is satisfied, see [Sek91, PP94]. These conditions guarantee that a transformation is meaningful *and*, as much as possible, semantics preserving. Most notably, in the literature several sets of conditions are presented concerning the application of the folding transformation. In the prototype we give the responsibility for a correct use of a transformation to the user of the tool.

4. A user-interface based on buttons.
   The special purpose language SEAL that comes with the ASF+SDF Meta–environment offers two ways of customising a user window: buttons and/or pull–down menu's can be added to the window. As the number of different transformations is limited, we have chosen to give each transformation a separate button. It is clear that, when the number of transformations exceeds a certain value (say, 10), the larger selection capacity of pull–down menu's will become attractive.

5. Part of the user-interface: navigation options.
   The prototype is equipped with two simple navigation buttons: *previous* and *next*. These buttons 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 $P_0, \ldots, P_{i-1},\ P_i,\ P_{i+1}, \ldots, P_n$.

6. A transformation is performed by 'select–and–click'.
   A transformation of a program P_i in a term window includes the selection of a certain term (goal, clause) by placing the 'focus' (a rectangle) around the term, followed by the clicking on one of the buttons. After a few moments the resulting program P_{i+1} will appear in the term window. No editing (e.g. addition/removal of clauses) is needed. (However, we cannot prohibit it.)

5 TransLog specification outline

The complete TransLog specification consists of twenty modules. In the subsequent sections these modules are presented and documented. This section describes the modular structure of the specification. The set of TransLog modules can be divided in five different subsets:

1. Basic modules.
   This subset contains two modules:

   - Layout
   - Booleans

   These modules contain some elementary specification of layout characters and booleans. They are presented in Section 6.

2. Syntax modules.
   The Prolog syntax, used in the TransLog specification, is specified in two modules:

   - PrologTerms
   - PrologProgram

   The TransLog specification uses a number of functions on a (part of a) Prolog program, like clause replacement, occur check, list normalisation, etc. These functions are grouped in two modules:

   - NormalisationFunction
   - PrologFunctions

   The syntax modules are presented in Section 7.

3. Unification modules.
   The unification of terms from the Prolog syntax is a basic operation in the TransLog specification. The following modules are concerned with unification:

   - Substitution
   - Equations
   - Unification

   These modules are presented in Section 8.

4. Transformation modules.
   The five TransLog transformations are specified in separate modules. So we have the modules

   - Unfold
   - Fold
   - AssocPermutation
   - GoalSwitch
   - CleanUp

   Two supporting modules are used by the Unfold module:

   - TermSets
   - VariantClause

The first module is also imported by the CleanUp module. The seven transformation modules are presented in Section 9.

5. User interface modules.
   The last subset contains four modules that are related with the TransLog user-interface:

   - TransLog
   - Log
   - ButtonConditions
   - TransLog.seal

   The module TransLog is very simple: only the various transformation modules are imported. The module Log defines a structure for storing and retrieving (a sequence of) programs. This structure is used for the implementation of the navigation properties of the TransLog tool. The module ButtonConditions specifies the boolean conditions that determine whether the various buttons have to be enabled/disabled. These buttons are defined in the module TransLog.seal. This module is written in the special purpose language SEAL. These modules are presented in Section 10.

The import-graph of the modules listed above (the module TransLog.seal excluded) is shown in Figure 1.

6 Basic modules

Some basic sorts and functions are used in (almost) every ASF+SDF specification. For reasons of completeness these simple modules are listed in this section.

6.1 Layout

This module defines some basic ASF+SDF layout syntax. Comments are defined as a character sequence starting with %% and ending with a newline character (with no newline character in the comment string). The space character, the tab character and the newline character are defined as layout characters.

module Layout
exports
  lexical syntax
    "%%"~[\n]*[\n] → LAYOUT
    [␣\t\n]   → LAYOUT

6.2 Booleans

The module Booleans provides the sort BOOL, the constants true and false, the operators | (logical or) and & (logical and) and the function not. Priorities are defined as one would expect: | < &.

module Booleans
imports Layout
exports
  sorts BOOL
  context-free syntax
    true   → BOOL
    false   → BOOL

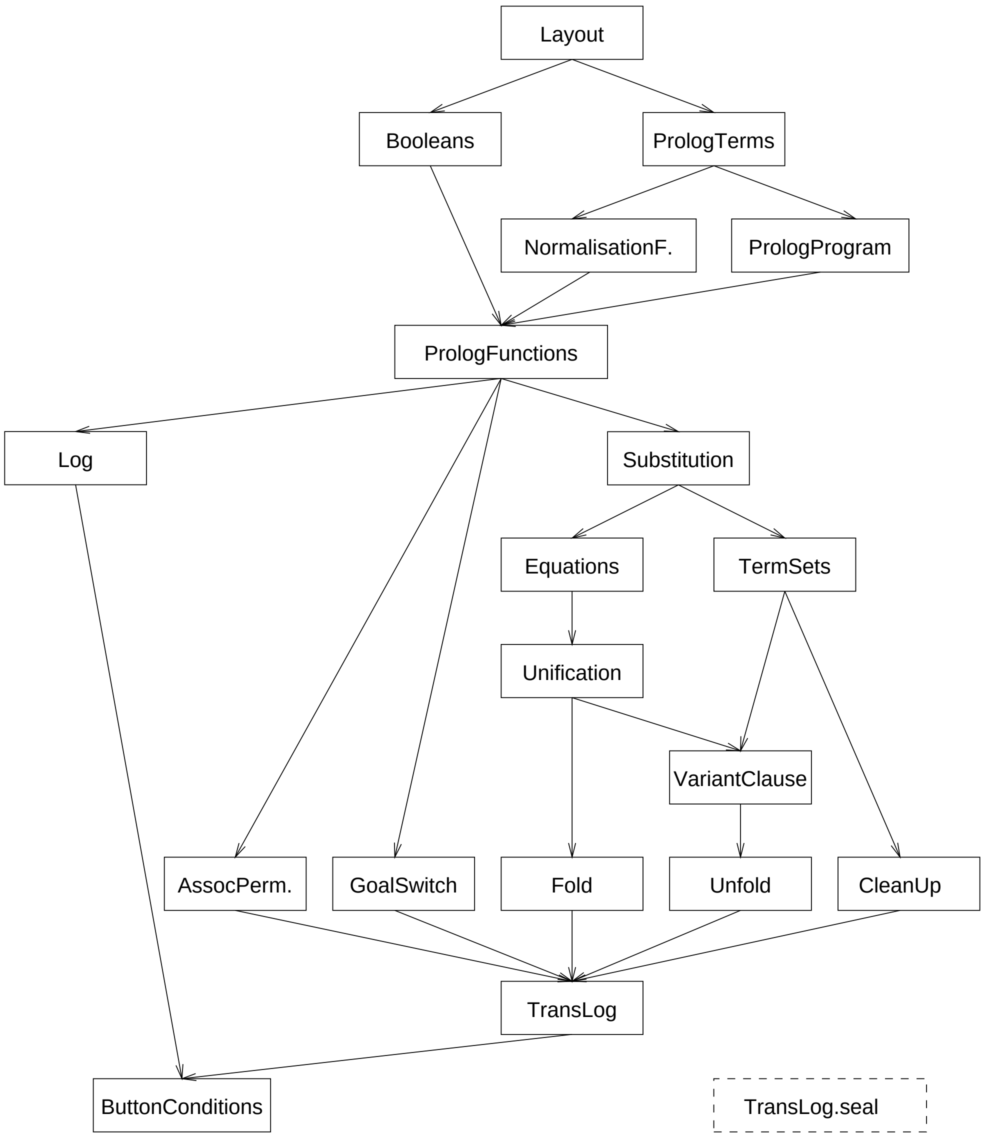


Figure 1: TransLog modules import graph

BOOL "|" BOOL → BOOL {**left**}
BOOL "&" BOOL → BOOL {**left**}
not "(" BOOL ")" → BOOL
"(" BOOL ")" → BOOL {**bracket**}
priorities
BOOL "|"BOOL → BOOL < BOOL "&"BOOL → BOOL

hiddens
variables
B [$'$]$*$ → BOOL

equations
[o1] true | B = true
[o2] false | B = B
[a1] true & B = B
[a2] false & B = false
[n1] not(false) = true
[n2] not(true) = false

7 Syntax modules

The TransLog tool operates on programs that are written in the (Edinburgh) Prolog syntax. The syntax definition in this section is based on the SICStus Prolog manual ([SIC92]). The definition does not cover the complete syntax, but is merely a relevant subset. Before we present the syntax modules, we first list some items that have *not* been implemented:

- Only integer numbers are defined. Floating point numbers, binary numbers, octal numbers and hexadecimal numbers will not be recognised 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.

The Prolog syntax is defined in two modules, showed and PrologProgram. A number of functions on Prolog terms and constructs is defined in the modules NormalisationFunction and PrologFunctions.

7.1 PrologTerms

The first module contains the definition of the lexical syntax of various Prolog 'tokens': atoms, numbers (integers), operator symbols and variables. From these basic elements the generic Prolog term (sort TERM) is defined. A term with an infix operator is also allowed. Furthermore, the various syntactical representations of the Prolog list-construct are defined. A term of the sort TERMLIST denotes a sequence of one or more Prolog terms.

module PrologTerms
imports Layout
exports
sorts TERM TERMLIST INTEGER VARIABLE ATOM OPSYM LIST
lexical syntax

[0-9]+	→ INTEGER	
[\−][0-9]+	→ INTEGER	
[A-Z_][a-zA-Z0-9_]∗	→ VARIABLE	
[a-z][a-zA-Z0-9_]∗	→ ATOM	
OPSYM+	→ ATOM	
“ ′ ”~[′\n\r]+“ ′ ”	→ ATOM	
“.”	→ ATOM	
“!”	→ ATOM	
“+”	→ OPSYM	
“\−”	→ OPSYM	
“∗”	→ OPSYM	
“/”	→ OPSYM	
“>”	→ OPSYM	
“<”	→ OPSYM	
“=”	→ OPSYM	

context-free syntax

INTEGER	→ TERM	
VARIABLE	→ TERM	
ATOM	→ TERM	
ATOM “(” TERMLIST “)”	→ TERM	
TERM ATOM TERM	→ TERM	
LIST	→ TERM	
“[” “]”	→ LIST	
“[” TERMLIST “]”	→ LIST	
“[” TERMLIST “|” LIST “]”	→ LIST	
“[” TERMLIST “|” VARIABLE “]”	→ LIST	
{TERM “,”}+	→ TERMLIST	
“(” TERM “)”	→ TERM	{**bracket**}
“(” TERMLIST “)”	→ TERMLIST	{**bracket**}

7.2 PrologProgram

The syntax of a Prolog program is specified in the module with the same name. The specification is straightforward. Two sorts are defined: PROGRAM and CLAUSE. A program is defined as a sequence of zero or more clauses. (Not: *one* or more clauses. Empty programs will sometimes be useful in other functions.) A clause is a unit clause (with only a head) or a clause with a head and a body. The symbol :- is defined as the connective between head and body. A body consists of a termlist (one or more terms, separated by commas).

It should be noticed that the syntax definition in this module is too ‘liberal’. For example, the clause

```
X :- [a,b,c],Y.
```

is correct with respect to the given syntax definition, all elementary terms are Prolog terms as defined in the previous module. In the module PrologFunctions the function syntaxCheck will be defined. This function actually determines whether or not a term of the sort PROGRAM is correct with respect to the Prolog syntax.

module PrologProgram
imports PrologTerms
exports
  sorts PROGRAM CLAUSE
  context-free syntax

CLAUSE∗	$\rightarrow$ PROGRAM	
TERM "."	$\rightarrow$ CLAUSE	
TERM ":-" TERMLIST "."	$\rightarrow$ CLAUSE	
"(" CLAUSE ")"	$\rightarrow$ CLAUSE	{**bracket**}
"(" PROGRAM ")"	$\rightarrow$ PROGRAM	{**bracket**}

7.3 NormalisationFunction

In Prolog different terms may represent the same list, e.g. $[a, b, c]$ and $[a \mid [b, c]]$. In this module a function *norm* is defined, that transforms all list terms to a standard normal form. In this normal form (also well-known in Prolog) a non-empty list is built from 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 normalise a term with an infix operator to prefix notation.

The normalisation function is specified for efficiency reasons: by first normalising a term to the 'normal form' *atom(termlist)*, for many operations on this kind of terms the number of equations is significantly reduced.

module NormalisationFunction
imports PrologTerms
exports
  context-free syntax
    norm "(" TERM ")" $\rightarrow$ TERM
hiddens
  variables

V [′]∗	$\rightarrow$ VARIABLE
A [′]∗	$\rightarrow$ ATOM
T [′]∗	$\rightarrow$ TERM
T [′]∗"+"	$\rightarrow$ {TERM ","}+
L [′]∗	$\rightarrow$ LIST

equations

[n1]	norm([ ])	= [ ]
[n2]	$\mathrm{norm}([T])$	$= .(\mathrm{norm}(T), [\,])$
[n3]	$\mathrm{norm}([T,\ T^+])$	$= .(\mathrm{norm}(T), \mathrm{norm}([T^+]))$
[n4]	$\mathrm{norm}([T \mid L])$	$= .(\mathrm{norm}(T), \mathrm{norm}(L))$
[n5]	$\mathrm{norm}([T,\ T^+ \mid L])$	$= .(\mathrm{norm}(T), \mathrm{norm}([T^+ \mid L]))$
[n6]	$\mathrm{norm}([T \mid V])$	$= .(\mathrm{norm}(T),\ V)$
[n7]	$\mathrm{norm}([T,\ T^+ \mid V])$	$= .(\mathrm{norm}(T), \mathrm{norm}([T^+ \mid V]))$
[n8]	$\mathrm{norm}(T\ A\ T')$	$= A(\mathrm{norm}(T), \mathrm{norm}(T'))$
[n9]	$\mathrm{norm}(A(T))$	$= A(\mathrm{norm}(T))$

$$[\mathrm{n}10]\quad \frac{\mathrm{norm}(A(T^+)) \ = \ A(T^{+\prime\prime}),\ \ \mathrm{norm}(A(T^{+\prime})) \ = \ A(T^{+\prime\prime\prime})}{\mathrm{norm}(A(T^+,\ T^{+\prime})) \ = \ A(T^{+\prime\prime},\ T^{+\prime\prime\prime})}$$

[n] norm(T) = T **otherwise**

7.4 PrologFunctions

This module contains a number of functions on Prolog terms and programs. These functions will be needed in the coming modules. They can be grouped in three subsets:

- Boolean functions to test a property of a certain term (pair): Is a term a variable? Is a list empty? Does a variable occur in a term? The function $\sqsubset$ uses a help function $\sqsubset_h$. This function investigates whether or not a variable occurs in a normalised term (see the previous section).

  The function isBodyTerm investigates whether or not a term is part of the body of a clause.

- A boolean function to check the syntax of Prolog program. The function syntaxCheck checks whether or not a term of the sort PROGRAM is correct with respect to the Prolog syntax rules. This function uses a number of local help-functions that check the syntax of a clause, the head of a clause, the body of a clause and a term in the body of a clause.

  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 on a clause or a program. The function *head* produces the head term of a clause. The functions *addcp*, *removecp* and *replacecp* add a clause to a program, remove a clause from a program or replace a clause in a program.

module PrologFunctions
imports NormalisationFunction PrologProgram Booleans
exports
context-free syntax

isvar "(" TERM ")"	→ BOOL
is-empty "(" LIST ")"	→ BOOL
VARIABLE "$\sqsubset$" TERM	→ BOOL
isBodyTerm "(" TERM "," CLAUSE ")"	→ BOOL
syntaxCheck "(" PROGRAM ")"	→ BOOL
head "(" CLAUSE ")"	→ TERM
addcp "(" CLAUSE "," PROGRAM ")"	→ PROGRAM
removecp "(" CLAUSE "," PROGRAM ")"	→ PROGRAM
replacecp "(" PROGRAM "," CLAUSE "," PROGRAM ")"	→ PROGRAM

variables

V [']*	→ VARIABLE
T [']*	→ TERM
T [']*"+"	→ {TERM ","}+
T [']*"*"	→ {TERM ","}*
L [']*	→ LIST
A [']*	→ ATOM
C [']*	→ CLAUSE

C [$'$]$*$"$*$" $\rightarrow$ CLAUSE$*$
P [$'$]$*$ $\rightarrow$ PROGRAM

hiddens

context-free syntax

VARIABLE "$\sqsubset_h$" TERM $\rightarrow$ BOOL

scClause "(" CLAUSE ")" $\rightarrow$ BOOL
scHead "(" TERM ")" $\rightarrow$ BOOL
scBody "(" TERMLIST ")" $\rightarrow$ BOOL
scBodyTerm "(" TERM ")" $\rightarrow$ BOOL

equations

[iv1] isvar(V) = true
[iv] isvar(T) = false **otherwise**

[ie1] is-empty([ ]) = true
[ie] is-empty(L) = false **otherwise**

[oc1] $V \sqsubset T = V \sqsubset_h$ norm(T)

[oh1] $V \sqsubset_h A(T^*,\ V,\ T^{*\prime}) =$ true

[oh2] $$\frac{V \sqsubset_h T \ = \ \text{true}}{V \sqsubset_h A(T^*,\ T,\ T^{*\prime}) \ = \ \text{true}}$$

[oh] $V \sqsubset_h T =$ false **otherwise**

[ib1] isBodyTerm(T, T' :- T^*, T, $T^{*\prime}$.) = true
[ib] isBodyTerm(T, C) = false **otherwise**

[sc1] syntaxCheck() = true
[sc2] syntaxCheck(C C^*) = scClause(C) & syntaxCheck(C^*)

[cc1] scClause(T .) = scHead(T)
[cc2] scClause(T :- T^+ .) = scHead(T) & scBody(T^+)

[bc1] scBody(T) = scBodyTerm(T)
[bc2] scBody(T, T^+) = scBodyTerm(T) & scBody(T^+)

[hc1] scHead(A) = true
[hc2] scHead(A(T^+)) = true
[hc] scHead(T) = false **otherwise**

[bt1] scBodyTerm(A) = true
[bt2] scBodyTerm(A(T^+)) = true

[bt3]	scBodyTerm(V)	= true	
[bt4]	scBodyTerm($T\ A\ T'$)	= true	
[bt]	scBodyTerm(T)	= false	**otherwise**

[p1] $[T^+ \mid [\,]] = [T^+]$

[h1]	head(T .)	$= T$
[h2]	head(T :- T^+ .)	$= T$

[a1]	addcp(C, C^*)	$= C\ C^*$	
[r1]	removecp(C, $C^*\ C\ C^{*\prime}$)	$= C^*\ C^{*\prime}$	
[r]	removecp(C, P)	$= P$	**otherwise**
[p1]	replacecp($C^*\ C\ C^{*\prime}$, C, $C^{*\prime\prime}$)	$= C^*\ C^{*\prime\prime}\ C^{*\prime}$	
[p]	replacecp(P, C, P')	$= P$	**otherwise**

8 Unification modules

The unification of two terms plays an important role in most of the transformations specified in Section 9. In this section we present three modules that are related with unification. In the module Unification a most general unifier (mgu) of a list of term equations is constructed. The unification function is based on the Martelli–Montanari unification algorithm [MM82]. 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. The output of our unification function will not be a set of equations, but a set of substitutions (or a failure). This shows what this function is needed for: the resulting substitutions can be applied to clauses, bodies, terms, etc. In the module Substitution the syntax of a substitution is defined, as well as the result of a sequence of substitutions on a clause, a body, etc. In the module Equations the syntax of a set of equations is defined. Finally, in the module Unification the Martelli–Montanari unification algorithm is specified. Below we present these modules in more detail.

8.1 Substitution

In this module the substitution of a variable by a term, $V \mapsto T$, is defined. Such a substitution is considered to be a normal form of the sort SUB. Furthermore, a term of the sort SUBS denotes a sequence of zero or more substitutions. The constant *fail* of the sort SUBS denotes a failure of the unification algorithm.

Next, two binary operators on substitution sequences are specified. The $\otimes$ operator joins two substitution sequences in a 'smart' way: potential conflicts between two substitutions (e.g. two different substitutions for the same variable) are detected and handled. Furthermore, if one of the sequences is equal to the constant *fail*, the result of the $\otimes$ operator is made equal to *fail*. The $\oplus$ operator joins two substitution sequences in a simple way: the two sequences are simply glued together to one big sequence. The $\otimes$ operator uses two help-functions: *member?* investigates whether or not a variable is part of a substitution sequence, the function *get-term* produces the term that is to be substituted for a given variable.

In the final part of this module the effect of (a sequence of) substitutions on various Prolog constructs (clause, termlist (body) and term) is specified. A list of substitutions on a construct C

is denoted by $[c\ S]$.

module Substitution
imports PrologFunctions
exports
sorts SUBS SUB
context-free syntax

"(" VARIABLE "$\mapsto$" TERM ")"	$\rightarrow$ SUB	
"{" {SUB ","}* "}"	$\rightarrow$ SUBS	
fail	$\rightarrow$ SUBS	
SUBS "$\otimes$" SUBS	$\rightarrow$ SUBS	{**left**}
SUBS "$\oplus$" SUBS	$\rightarrow$ SUBS	{**left**}
"member?"(VARIABLE, SUBS)	$\rightarrow$ BOOL	
get-term(VARIABLE, SUBS)	$\rightarrow$ TERM	
CLAUSE "[c" SUBS "]"	$\rightarrow$ CLAUSE	
TERMLIST "[tl" SUBS "]"	$\rightarrow$ TERMLIST	
TERM "[t" SUBS "]"	$\rightarrow$ TERM	

variables
$S\ [']*$ $\rightarrow$ SUBS
$S\ [']*$"*" $\rightarrow$ {SUB ","}*

equations

[sj1] $\text{fail} \otimes S = \text{fail}$
[sj2] $S \otimes \text{fail} = \text{fail}$

[sj3] $$\frac{\text{member?}(V, S) = \text{true},\ \text{get-term}(V, S) = T}{\{(V \mapsto T), S^*\} \otimes S = \{S^*\} \otimes S}$$

[sj4] $$\frac{\text{member?}(V, S) = \text{true},\ \text{get-term}(V, S) \neq T}{\{(V \mapsto T), S^*\} \otimes S = \text{fail}}$$

[sj5] $$\frac{\text{member?}(V, S) = \text{false},\ \{S^*\} \otimes S = \{S^{*\prime}\}}{\{(V \mapsto T), S^*\} \otimes S = \{(V \mapsto T\,[\text{t}\ S]), S^{*\prime}\}}$$

[sj6] $\{\} \otimes S = S$

[ij1] $S \oplus \text{fail} = S$
[ij2] $\text{fail} \oplus S = S$
[ij3] $\{S^*\} \oplus \{S^{*\prime}\} = \{S^*, S^{*\prime}\}$

[m1] $\text{member?}(V, \{S^*, (V \mapsto T), S^{*\prime}\}) = \text{true}$
[m] $\text{member?}(V, S) = \text{false}$ **otherwise**

[g1] $\text{get-term}(V, \{S^*, (V \mapsto T), S^{*\prime}\}) = T$
[g] $\text{get-term}(V, S) = V$ **otherwise**

[cl1] $T\,.\,[\text{c}\ S] = T\,[\text{t}\ S]$.

[cl2] T :- T^+ . [c S] = T [t S] :- T^+ [tl S] .

[tl1] $$\frac{T\,[\mathrm{t}\ S] \;=\; T'}{T\,[\mathrm{tl}\ S] \;=\; T'}$$

[tl2] $$\frac{\begin{array}{r}T^+\,[\mathrm{tl}\ S] \;=\; T^{+\prime\prime},\\ T^{+\prime}\,[\mathrm{tl}\ S] \;=\; T^{+\prime\prime\prime}\end{array}}{T^+,\ T^{+\prime}\,[\mathrm{tl}\ S] \;=\; T^{+\prime\prime},\ T^{+\prime\prime\prime}}$$

[at1] A [t S] = A
[at2] $A(T^+)$ [t S] = $A(T^+$ [tl S])
[at3] $T\ A\ T'$ [t S] = T [t S] $A\ T'$ [t S]

[va1] V [t S] = get-term(V, S)

[in1] `I` [t S] = `I`

[li1] [ ] [t S] = [ ]
[li2] [T^+] [t S] = [T^+ [tl S]]

[li3] $$\frac{L\,[\mathrm{t}\ S] \;=\; L'}{[T^+ \mid L]\,[\mathrm{t}\ S] \;=\; [T^+\,[\mathrm{tl}\ S] \mid L']}$$

[li4] $$\frac{V\,[\mathrm{t}\ S] \;=\; V'}{[T^+ \mid V]\,[\mathrm{t}\ S] \;=\; [T^+\,[\mathrm{tl}\ S] \mid V']}$$

[li5] $$\frac{V\,[\mathrm{t}\ S] \;=\; L}{[T^+ \mid V]\,[\mathrm{t}\ S] \;=\; [T^+\,[\mathrm{tl}\ S] \mid L]}$$

8.2 Equations

This module defines the syntax of a set of equations of Prolog terms. This set of equations is the basic input of the Martelli-Montanari unification algorithm, which will be specified in the following section. In order to avoid a conflict with the built-in ASF+SDF =-operator for algebraic equations, in the syntax of term equations the operator $\equiv$ is used between two terms.

The sort EQ is related to a single equation, the sort EQS is related to a sequence of equations. On both sorts a substitution function is defined. In the defining equations these substitutions are reduced to substitutions on separate terms (which have been defined in the previous section).

module Equations
imports Substitution
exports
  sorts EQ EQS
  context-free syntax
    TERM "≡" TERM → EQ
    "{" {EQ ","}* "}" → EQS

EQ "[e" SUBS "]" $\rightarrow$ EQ
EQS "[es" SUBS "]" $\rightarrow$ EQS
variables
$E\,[']*$ $\rightarrow$ EQ
$E\,[']*$"+" $\rightarrow$ {EQ ","}+
$E\,[']*$"*" $\rightarrow$ {EQ ","}*
$EQS\,[']*$ $\rightarrow$ EQS

equations

[eq1] $T \equiv T'$ [e S] = T [t S] $\equiv$ T' [t S]

[es1] {} [es S] = {}
[es2] $\{E\}$ [es S] = $\{E$ [e S]$\}$

$$\text{[es3]}\quad \frac{\{E^{+}\}\ [\text{es } S] = \{E^{+\prime\prime}\},\quad \{E^{+\prime}\}\ [\text{es } S] = \{E^{+\prime\prime\prime}\}}{\{E^{+}, E^{+\prime}\}\ [\text{es } S] = \{E^{+\prime\prime}, E^{+\prime\prime\prime}\}}$$

8.3 Unification

The unification of Prolog terms is specified according to the Martelli-Montanari algorithm ([MM82]). The original algorithm takes a set of term equations as input and produces a modified set of equations as output. As mentioned before, in our specification a sequence of *substitutions* is produced instead of a set of equations.

The algorithm contains six conditional steps. In an iterative process these steps are applied, producing an output result. This result may be a failure (no mgu found). Before giving the specification, we first give a concise description of the unification algorithm.

Take an equation of a sequence and perform the associated action:

1. $a(t_1, \ldots, t_n) \equiv a(v_1, \ldots, v_n)$: replace the equation by the equations $t_1 \equiv v_1, \ldots, t_n \equiv v_n$.
2. $a(t_1, \ldots, t_n) \equiv b(v_1, \ldots, v_m)$ with $a \neq b$ or $n \neq m$: halt with failure.
3. $v \equiv v$: delete the equation.
4. $t \equiv v$ where t is not a variable: replace the equation by the equation $v \equiv t$.
5. $v \equiv t$ where $v \neq t$, v does not occur in t and v occurs elsewhere: perform the substitution $v \mapsto t$ everywhere except in the current equation.
6. $v \equiv t$ where $v \neq t$ and v occurs in t: halt with failure.

The algorithm terminates when no more action can be performed or when a failure arises.
Note: In the original version of the algorithm an equation is nondeterministically chosen from the set of equations. In our version we always will take the first equation of the remaining set. This difference does not influence the final result.

In the module Unification three functions are specified. The function mgu determines the most general unifier (sequence of substitutions) for the list of equations that is given in its input argument. The function mgu-nv determines the most general unifier of two non-variable terms. The local function mgu with two arguments is used while producing the result of the exported function mgu.

The equations for the function mgu closely resemble the algorithm described above. The equations for the function mgu-nv need no further explanation.

module Unification
imports Equations
exports
context-free syntax
mgu "(" EQS ")" → SUBS
mgu-nv "(" TERM "," TERM ")" → SUBS
hiddens
context-free syntax
mgu "(" EQS "," SUBS ")" → SUBS

equations
[mg1] mgu(EQS) = mgu(EQS, {})

[m1] $$\frac{\text{isvar}(T) = \text{false},\ \text{isvar}(T') = \text{false},\ \text{mgu-nv}(T, T') = S',\ S' \neq \text{fail}}{\text{mgu}(\{T \equiv T', E^*\}, S) = \text{mgu}(\{E^*\}\ [\text{es } S'], S \otimes S')}$$

[m2] $$\frac{\text{isvar}(T) = \text{false},\ \text{isvar}(T') = \text{false},\ \text{mgu-nv}(T, T') = S',\ S' = \text{fail}}{\text{mgu}(\{T \equiv T', E^*\}, S) = \text{fail}}$$

[m3] $\text{mgu}(\{V \equiv V, E^*\}, S) = \text{mgu}(\{E^*\}, S)$

[m4] $$\frac{\text{isvar}(T) = \text{false}}{\text{mgu}(\{T \equiv V, E^*\}, S) = \text{mgu}(\{V \equiv T, E^*\}, S)}$$

[m5] $$\frac{V \neq T,\ V \sqsubset T = \text{false}}{\text{mgu}(\{V \equiv T, E^*\}, S) = \text{mgu}(\{E^*\}\ [\text{es } \{(V \mapsto T)\}], S \otimes \{(V \mapsto T)\})}$$

[m6] $$\frac{V \neq T,\ V \sqsubset T = \text{true}}{\text{mgu}(\{V \equiv T, E^*\}, S) = \text{fail}}$$

[mg] mgu(EQS, S) = S **otherwise**

[nv1] mgu-nv(`I`, `I`) = {}
[nv2] mgu-nv(A, A) = {}
[nv3] $\text{mgu-nv}(A(T), A(T')) = \text{mgu}(\{T \equiv T'\})$

[nv4] $$\frac{\text{mgu}(\{T \equiv T'\}) = S}{\text{mgu-nv}(A(T, T^+), A(T', T^{+\prime})) = S \otimes \text{mgu-nv}(A(T^+\ [\text{tl } S]), A(T^{+\prime}\ [\text{tl } S]))}$$

[nv5] $\text{mgu-nv}(T\ A\ T', T''\ A\ T''') = \text{mgu}(\{T \equiv T'', T' \equiv T'''\})$
[nv6] mgu-nv([ ], [ ]) = {}

[nv7] $$\frac{\text{is-empty}(L) = \text{false},\ \text{is-empty}(L') = \text{false}}{\text{mgu-nv}(L, L') = \text{mgu-nv}(\text{norm}(L), \text{norm}(L'))}$$

[nv] $\text{mgu-nv}(T, T') = \text{fail}$ **otherwise**

9 Transformation modules

The TransLog tool offers five transformations of Prolog programs: unfold, fold, associativity permutation, goal switch and program clean up. Each transformation is specified in a separate module. At the end of this section two supporting modules will be presented.

9.1 Unfold

Before presenting the algebraic specification of the unfold transformation, we first give an informal description of the algorithm that is applied.

Let $C = T''$:- $T^*, T, T^{*\prime}$. be the unfolded clause. Let T be the body term to-be-unfolded. Let P be the current program. Then the unfold transformation 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 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 program P
remove C from the program P.

The function unfold has three input-arguments: the current program, the body term to-be-unfolded and the clause that holds this body term (the unfolded clause). The result of this function is the transformed program. The 'work' is done by the function u-clauses. This function produces the clauses that have to be added to the program as the result of the unfold operation. This function recursively investigates all clauses of the program. If the head of a certain clause is unifiable with the selected body term (the result of the mgu-nv function is not equal to fail), then a new clause is constructed and added to the result of the function u-clauses. In this process some auxiliary operations (functions) are needed. First, before investigating whether or not the head of a clause C' is unifiable with the selected body term, a variant of this clause is constructed that has no variables in common with the clause C in which the body term resides. This is done with the function variant, which is specified in a separate module VariantClause, see below. If the head of (the variant of) a clause and the selected body term are unifiable, then a new clause for the program (a partial result of the unfold operation) is constructed with the function resolvent. This function replaces the body term to-be-unfolded by the body of the unfolding clause. To be more precise: the body term is replaced by the body of the *variant* of the unfolding clause; furthermore the substitutions that result from the unification of the unfolded term and the head of the variant clause are applied to the new clause. A special case arises when the unfolding clause is a unit clause (the body of the unfolding clause is empty). In that case the unfolded body term vanishes. If this term is the only body term of the unfolded clause, then the unfolded clause also becomes a unit clause. Of course, in both cases the required substitutions have to be performed on the new clause. The local boolean function nonex tests on the presence of (more) body terms.

module Unfold
imports VariantClause
exports
  context-free syntax
    unfold "(" PROGRAM "," TERM "," CLAUSE ")" → PROGRAM
    u-clauses "(" PROGRAM "," TERM "," CLAUSE ")" → PROGRAM
    resolvent "(" CLAUSE "," TERM "," CLAUSE ")" → CLAUSE
hiddens
  context-free syntax

nonex "(" {TERM ","}∗ ")" → BOOL

equations

[uf1] $$\frac{\text{u-clauses}(P,\ T,\ C)\ =\ P'}{\text{unfold}(P,\ T,\ C)\ =\ \text{replacecp}(P,\ C,\ P')}$$

[uc1] $\text{u-clauses}(,\ T,\ C) =$

[uc2] $$\frac{\text{mgu-nv}(T, \text{head}(\text{variant}(C',\ C)))\ =\ \text{fail}}{\text{u-clauses}(C'\ C^*,\ T,\ C)\ =\ \text{u-clauses}(C^*,\ T,\ C)}$$

[uc3] $$\frac{\text{variant}(C',\ C)\ =\ C'',\ \text{mgu-nv}(T, \text{head}(C''))\ \neq\ \text{fail}}{\begin{array}{l}\text{u-clauses}(C'\ C^*,\ T,\ C)\ = \\ \text{addcp}(\text{resolvent}(C'',\ T,\ C), \text{u-clauses}(C^*,\ T,\ C))\end{array}}$$

[re1] $$\frac{\text{mgu-nv}(T,\ T')\ =\ S}{\begin{array}{l}\text{resolvent}(T \text{ :- } T^+\ .\ ,\ T',\ T'' \text{ :- } T^*,\ T',\ T^{*\prime}\ .)\ = \\ T'' \text{ :- } T^*,\ T^+,\ T^{*\prime}\ .\ [\text{c } S]\end{array}}$$

[re2] $$\frac{\text{nonex}(T^*,\ T^{*\prime})\ =\ \text{false},\ \text{mgu-nv}(T,\ T')\ =\ S}{\text{resolvent}(T\ .\ ,\ T',\ T'' \text{ :- } T^*,\ T',\ T^{*\prime}\ .)\ =\ T'' \text{ :- } T^*,\ T^{*\prime}\ .\ [\text{c } S]}$$

[re3] $$\frac{\text{nonex}(T^*,\ T^{*\prime})\ =\ \text{true},\ \text{mgu-nv}(T,\ T')\ =\ S}{\text{resolvent}(T\ .\ ,\ T',\ T'' \text{ :- } T^*,\ T',\ T^{*\prime}\ .)\ =\ T''\ .\ [\text{c } S]}$$

[ne1] $\text{nonex}() = \text{true}$
[ne2] $\text{nonex}(T^+) = \text{false}$

9.2 Fold

Before presenting the Fold module we first discuss some general aspects of the fold operation. In the literature this operation is bound to a set of restrictive conditions. Only if all conditions are met, a fold operation is allowed. In this way certain semantic properties of the program are maintained. We refer to [Sek91, PP94] for more details. As stated in Section 4, in this first prototype we have not included any condition on a transformation operation ("an operation is allowed if it can be performed somehow"). However, for the fold operation a choice had to be made what program (clauses) may be used as 'folding clauses'. In most cases the intended folding clause is not part of the current program. In the example, given in Section 2, the folding clause was the original revacc clause, which was not part of the transformed program in which the folding operation was executed. In order to keep in line with the 'select and click' user-interface (see Section 4), we have chosen to require that a folding clause has to be part of the initial program P_0 of a transformation sequence. No extra clauses have to be added during a transformation sequence.

Another aspect of the folding operation is the number of body terms that is involved. Consider the following (part of an) initial program P_0:

```
p :- a,b.
q :- a,b,c.
```

Now the clause `r :- a,b,c,d.` can be folded to `r :- p,c,d.`, but also to `r :- q,d.`. For practical reasons (which will be explained in Section 10), we have chosen to leave the solution of this ambiguity to the TransLog system. So, in the case above the system will make an (as far as the user may notice) undeterministic choice what clause is to be used as the folding clause. This aspect will be improved in a future version of the system, see also Section 11.

We give the following definition of the fold transformation: Let $C = T$:-$T^*, T^{+\prime}, T^{*\prime}$. be the clause to-be-folded, with $T^{+\prime}$ the body term(s) to-be-folded. Let P_i be the initial program from which a folding clause may be selected. Let P be the current program. Then the fold transformation is defined as follows:

Let $C' = T''$:-T^+. be a clause in P_i.
If T^+ unifies with $T^{+\prime}$ (mgu(T^+,$T^{+\prime}$) $= \theta,\ \theta \neq$ fail)
then in P replace C by $C'' = T$:-$T^*, (T'')\theta, T^{*\prime}$.

We now turn to the actual specification of the fold operation in the module with the same name. The function fold has three arguments: the folded clause with the body terms to-be-folded, the program containing the clauses that may be used for folding and the current program. The help function body-mgu determines the most general unifier of two bodies by a pairwise search for the mgu of two body terms. Which clause (body) from the third argument of the fold function is used for folding is left to the ASF+SDF system: the subterm $C^*\ T''$:- T^+. $C^{*\prime}$ denotes a program with *a* clause that has a body T^+. While replacing the clause to-be-folded by the resulting clause, the head of the folding clause is subjected to the substitutions that result from the mgu of the two sequences of body terms. The definition of the function body-mgu is straightforward.

module Fold
imports Unification
exports
context-free syntax
fold "(" CLAUSE "," PROGRAM "," PROGRAM ")" → PROGRAM
body-mgu "(" TERMLIST ";" TERMLIST ")" → SUBS

equations

[fld1]
$$\frac{\text{body-mgu}(T^+;\ T^{+\prime}) \ =\ S,\ S \neq \text{fail}}{\text{fold}(T :\text{-}\ T^*,\ T^{+\prime},\ T^{*\prime}\ .\ ,\ C^*\ T''\ :\text{-}\ T^+\ .\ C^{*\prime}\ ,\ P) \ = \ \text{replacecp}(P,\ T :\text{-}\ T^*,\ T^{+\prime},\ T^{*\prime}\ .\ ,\ T :\text{-}\ T^*,\ T''\ [\text{t}\ S],\ T^{*\prime}\ .)}$$

[bm1] body-mgu(T; T') = mgu-nv(T, T')

[bm2]
$$\frac{\begin{array}{c}\text{body-mgu}(T^+;\ T^{+\prime\prime}) \ =\ S,\ S \neq \text{fail},\\ \text{body-mgu}(T^{+\prime};\ T^{+\prime\prime\prime}) \ =\ S',\ S' \neq \text{fail}\end{array}}{\text{body-mgu}(T^+,\ T^{+\prime};\ T^{+\prime\prime},\ T^{+\prime\prime\prime}) \ =\ S \otimes S'}$$

[bm] body-mgu(T^+; $T^{+\prime}$) = fail **otherwise**

9.3 AssocPermutation

Some predicates represent an associative operation. For these predicates argument permutation is allowed for two adjacent goals. In Section 2 and Section 4 the example of appending two lists has been mentioned. For this transformation we look at predicates with arity 3, in 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 '+,+,-' mode and the '-,-,+' mode. As this transformation is very simple we will not give a separate definition, but we will turn to the specification immediately.

The function assperm has three input arguments: the first body term that is involved in this transformation, the clause that holds this body term and the current program. The help functions aph1 and aph2 do the job for the two modes, they produce a new clause. Note: as will be explained in Section 10, the function *assperm* is only called when this transformation *is* possible, so one of the two help functions will always produce a result.

module AssocPermutation
imports PrologFunctions
exports
  context-free syntax
    assperm "(" TERM "," CLAUSE "," PROGRAM ")" → PROGRAM
hiddens
  context-free syntax
    aph1 "(" TERM "," CLAUSE ")" → CLAUSE
    aph2 "(" TERM "," CLAUSE ")" → CLAUSE

equations

[ap1] $$\frac{\text{aph1}(T,\ C)\ =\ C'}{\text{assperm}(T,\ C,\ C^{*}\ C\ C^{*\prime})\ =\ C^{*}\ C'\ C^{*\prime}}$$

[ap2] $$\frac{\text{aph2}(T,\ C)\ =\ C'}{\text{assperm}(T,\ C,\ C^{*}\ C\ C^{*\prime})\ =\ C^{*}\ C'\ C^{*\prime}}$$

[ah1] $\text{aph1}(A(T,\ T',\ V),\ T''' \text{ :- } T^{*},\ A(T,\ T',\ V),\ A(V,\ T'',\ V'),\ T^{*\prime}\ .)\ =$
$T''' \text{ :- } T^{*},\ A(T',\ T'',\ V),\ A(T,\ V,\ V'),\ T^{*\prime}\ .$

[ah2] $\text{aph2}(A(V,\ V',\ T),\ T' \text{ :- } T^{*},\ A(V,\ V',\ T),\ A(V'',\ V''',\ V),\ T^{*\prime}\ .)\ =$
$T' \text{ :- } T^{*},\ A(V'',\ V,\ T),\ A(V''',\ V',\ V),\ T^{*\prime}\ .$

9.4 GoalSwitch

While preparing the body of a clause for a certain transformation (folding, associativity permutation), it may be required to switch the order of two adjacent body terms. This is a potentially dangerous operation, because of shared variables or changing termination properties. However, according to our motto "if it can be done somehow, just do it", we will not restrict this transformation by binding it to any condition. This makes the specification of the switch function very simple. The input arguments of this function represent the body term that has to be switched with its right-hand neighbour, the clause that holds the body term and the current program. The local help function gsw is only used for reasons of clarity, the operation can also be specified in one single (but long) equation.

module GoalSwitch
imports PrologFunctions
exports
  context-free syntax

switch "(" TERM "," CLAUSE "," PROGRAM ")" $\rightarrow$ PROGRAM
hiddens
context-free syntax
gsw "(" TERM "," CLAUSE ")" $\rightarrow$ CLAUSE

equations

[gs1] $$\frac{\text{gsw}(T,\, C) \;=\; C'}{\text{switch}(T,\, C,\, C^*\; C\; C^{*\prime}) \;=\; C^*\; C'\; C^{*\prime}}$$

[gh1] $\text{gsw}(T,\, T'' \text{ :- } T^*,\, T,\, T',\, T^{*\prime}\ .) = T'' \text{ :- } T^*,\, T',\, T,\, T^{*\prime}\ .$

9.5 CleanUp

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 new clauses that contain the transformed version of the program. Therefore they safely can be removed. We give the following definition of this transformation. Let P be the current program. Let T be the selected head term. Then the cleanUp transformation is defined as follows:

Construct a set of goals TS that can be reached from the clause with the head T.
(The set TS only contains atoms with their arity.)
Select from the program P all clauses of which the head (atom + arity) is an element of TS.
These clauses constitute the resulting program.

The transformation is specified in the module CleanUp. This module imports the module TermSets, which will be presented in section 9.6. The only visible function is the function cleanUp. This function has two input arguments: a (head) term and the current program. The head term indicates from which head (predicate) the 'reachability analysis' has to be started. This analysis is performed by the local function goalset. This function produces a set of goals (predicates) that can be reached from the given head term. Only clauses that contain a definition for these predicates are selected for the result of the cleanUp function. This selection is performed by the function select-clauses.

With respect to the goal set, two terms have to be considered different if the predicate is different or the arity of the predicates is different. Therefore, before adding a goal to the goal set, the goal is first 'simplified': all its arguments are replaced by underscores. This is done by the function simplify.

The required goal set is constructed in a straightforward way: starting from a given head term (the first argument of the function goalSet), each program clause that has the same (simplified) head is investigated. Each of the body terms of these clauses may lead to new elements in the goal set. Once a clause has been investigated, it is removed from the investigated program. In this way infinite loops, caused by recursive clauses, are avoided.

The function select-clauses selects the clauses for the resulting program by testing, for each clause in the current program, whether or not its (simplified) head is present in the constructed goal set. If this is the case, the clause is added to the resulting program. If not, the clause is skipped.

module CleanUp
imports TermSets
exports
context-free syntax
cleanUp "(" TERM "," PROGRAM ")" $\rightarrow$ PROGRAM
hiddens
context-free syntax

goalset "(" TERMLIST "," PROGRAM "," PROGRAM ")" $\rightarrow$ TERM-SET
simplify "(" TERM ")" $\rightarrow$ TERM
select-clauses "(" TERM-SET "," PROGRAM ")" $\rightarrow$ PROGRAM

equations

[cu1] $$\frac{\text{goalset}(T,\, P,\, P) \;=\; \text{TS}}{\text{cleanUp}(T,\, P) \;=\; \text{select-clauses}(\text{TS},\, P)}$$

[gs1] $\text{goalset}(T, \;, P) = \{\}$

[gs2] $$\frac{\text{simplify}(T) \;=\; T'',\;\; \text{simplify}(T') \;=\; T''}{\text{goalset}(T,\, T' \,.\, C^* \,,\, P) \;=\; \{T''\} \cup \text{goalset}(T,\, C^*,\, P)}$$

[gs3] $$\frac{\text{simplify}(T) \;=\; T'',\;\; \text{simplify}(T') \;=\; T'',\;\; \text{removecp}(T' \text{ :- } T^+ \;. \;,\, P) \;=\; P'}{\text{goalset}(T,\, T' \text{ :- } T^+ \,.\, C^* \,,\, P) \;=\; \{T''\} \cup (\text{goalset}(T,\, C^*,\, P') \cup \text{goalset}(T^+,\, P',\, P'))}$$

[gs4] $\text{goalset}(T^+,\, T^{+\prime},\, P,\, P) = \text{goalset}(T^+,\, P,\, P) \cup \text{goalset}(T^{+\prime},\, P,\, P)$
[gs] $\text{goalset}(T,\, C\; C^*,\, P) = \text{goalset}(T,\, C^*,\, P)$ **otherwise**

[si1] $\text{simplify}(A(T)) = A(\_)$

[si2] $$\frac{\text{simplify}(A(T^+)) \;=\; A(T^{+\prime\prime}),\;\; \text{simplify}(A(T^{+\prime})) \;=\; A(T^{+\prime\prime\prime})}{\text{simplify}(A(T^+,\, T^{+\prime})) \;=\; A(T^{+\prime\prime},\, T^{+\prime\prime\prime})}$$

[si] $\text{simplify}(T) = T$ **otherwise**

[sc1] select-clauses(TS,) =

[sc2] $$\frac{\text{simplify}(\text{head}(C)) \in \text{TS} \;=\; \text{true},\;\; \text{select-clauses}(\text{TS},\, C^*) \;=\; P}{\text{select-clauses}(\text{TS},\, C\; C^*) \;=\; \text{addcp}(C,\, P)}$$

[sc3] $$\frac{\text{simplify}(\text{head}(C)) \in \text{TS} \;=\; \text{false}}{\text{select-clauses}(\text{TS},\, C\; C^*) \;=\; \text{select-clauses}(\text{TS},\, C^*)}$$

9.6 TermSets

For various reasons *sets* of Prolog terms are needed. In the module TermSets the associated sort TERM-SET and some basic set operations – union, intersection, set difference, membership – are specified. Also a substitution function on term sets is specified. This function routes a substitution '[ts S]' on a term set to a substitution '[t S]' on each of the elements of the set. A substitution on terms has already been defined in the module Substitution (Section 8.1).

module TermSets
imports Substitution
exports
sorts TERM-SET
context-free syntax

"{" {TERM ","}* "}" → TERM-SET
TERM-SET "∪" TERM-SET → TERM-SET {**assoc**}
TERM-SET "∩" TERM-SET → TERM-SET
TERM-SET "\" TERM-SET → TERM-SET
TERM "∈" TERM-SET → BOOL
TERM-SET "[ts" SUBS "]" → TERM-SET

"(" TERM-SET ")" → TERM-SET {**bracket**}

variables
TS [′]* → TERM-SET

equations

[se1] $\{T^*, T, T^{*\prime}, T, T^{*\prime\prime}\} = \{T^*, T, T^{*\prime}, T^{*\prime\prime}\}$

[un1] $\{T^*\} \cup \{T^{*\prime}\} = \{T^*, T^{*\prime}\}$

[is1] $\{T^*, T, T^{*\prime}\} \cap \{T^{*\prime\prime}, T, T^{*\prime\prime\prime}\} = \{T\} \cup \{T^*, T^{*\prime}\} \cap \{T^{*\prime\prime}, T^{*\prime\prime\prime}\}$
[is] $\mathrm{TS} \cap \mathrm{TS}' = \{\}$ **otherwise**

[di1] $\{T^*, T, T^{*\prime}\} \setminus \{T^{*\prime\prime}, T, T^{*\prime\prime\prime}\} = \{T^*, T^{*\prime}\} \setminus \{T^{*\prime\prime}, T, T^{*\prime\prime\prime}\}$
[di] $\mathrm{TS} \setminus \mathrm{TS}' = \mathrm{TS}$ **otherwise**

[me1] $T \in \{T^*, T, T^{*\prime}\} = \mathrm{true}$
[me] $T \in \{T^*\} = \mathrm{false}$ **otherwise**

[ts1] $\{\}\ [\mathrm{ts}\ S] = \{\}$
[ts2] $\{T\}\ [\mathrm{ts}\ S] = \{T\ [\mathrm{t}\ S]\}$

[ts3] $$\frac{\{T^+\}\ [\mathrm{ts}\ S] = \{T^{+\prime\prime}\},\ \{T^{+\prime}\}\ [\mathrm{ts}\ S] = \{T^{+\prime\prime\prime}\}}{\{T^+, T^{+\prime}\}\ [\mathrm{ts}\ S] = \{T^{+\prime\prime}, T^{+\prime\prime\prime}\}}$$

9.7 VariantClause

In the unfold transformation a body term of the unfolded clause is replaced by the body of the unfolding clause. To avoid unwanted identification of different variables with accidentally the same name, in the unfold transformation a *variant* of the unfolding clause is used in which all variables that also occur in the unfolded clause are renamed. This variant clause is created in the module with the same name by the function variant that takes both the unfolding clause and the unfolded clause as input arguments.

We will shortly explain the single equation for this function. With the function varset a set is created with the variables from a clause. With the function varsub a sequence of substitutions is deduced from two variable sets. For each variable from the unfolding clause C that is also present in the unfolded clause C' a substitution is added. In this substitution the variable is replaced by a new variable that is not present in the variable set of C' or in the variable set of C. A new variable name is created by repeated addition of _n to the old variable name until a unique variable name is derived. The substitutions are applied to the clause C. The resulting variant clause has no variables in common with the clause C'.

module VariantClause
imports Unification TermSets
exports
  context-free syntax
    variant "(" CLAUSE "," CLAUSE ")" → CLAUSE
hiddens
  context-free syntax
    newvar "(" VARIABLE "," TERM-SET ")" → TERM
    prime "(" VARIABLE ")" → VARIABLE

    varset "(" CLAUSE ")" → TERM-SET
    tvarset "(" {TERM ","}+ ")" → TERM-SET

    varsub "(" TERM-SET "," TERM-SET ")" → SUBS
  variables
    "c" → CHAR+

equations

[va1] $$\frac{\text{varset}(C) = \text{TS},\ \text{varset}(C') = \text{TS}'}{\text{variant}(C, C') = C\,[\text{c varsub}(\text{TS} \cap \text{TS}', \text{TS} \cup \text{TS}')]}$$

[sv1] $\text{varset}(T\ .) = \text{tvarset}(T)$
[sv2] $\text{varset}(T \text{ :- } T^{+}\ .) = \text{tvarset}(T) \cup \text{tvarset}(T^{+})$

[st1] $\text{tvarset}(\mathtt{I}) = \{\}$
[st2] $\text{tvarset}(V) = \{V\}$
[st3] $\text{tvarset}(A) = \{\}$
[st4] $\text{tvarset}(A(T)) = \text{tvarset}(T)$
[st5] $\text{tvarset}(A(T^{+})) = \text{tvarset}(T^{+})$
[st6] $\text{tvarset}(T\ A\ T') = \text{tvarset}(T) \cup \text{tvarset}(T')$

[st7] $$\frac{\text{is-empty}(L) = \text{true}}{\text{tvarset}(L) = \{\}}$$

[st8] $$\frac{\text{is-empty}(L) = \text{false}}{\text{tvarset}(L) = \text{tvarset}(\text{norm}(L))}$$

[st9] $\text{tvarset}(T^{+},\ T^{+\prime}) = \text{tvarset}(T^{+}) \cup \text{tvarset}(T^{+\prime})$

[vs1] $\text{varsub}(\{\}, \text{TS}) = \{\}$

[vs2] $$\frac{\text{newvar}(V, \text{TS}) = V'}{\text{varsub}(\{V,\ T^{*}\}, \text{TS}) = \{(V \mapsto V')\} \otimes \text{varsub}(\{T^{*}\}, \text{TS})}$$

$$[\text{nv1}]\quad \frac{\text{prime}(V) \in \text{TS} \ = \ \text{false}}{\text{newvar}(V, \text{TS}) \ = \ \text{prime}(V)}$$

$$[\text{nv2}]\quad \frac{\text{prime}(V) \in \text{TS} \ = \ \text{true}}{\text{newvar}(V, \text{TS}) \ = \ \text{newvar}(\text{prime}(V), \text{TS})}$$

[pr1] prime(variable(c)) = variable(c "_" "n")

10 User interface modules

Four modules are related with the user-interface of the TransLog system. One of these modules, TransLog.seal, is no ASF+SDF specification, but a SEAL script. SEAL ([Koo93]) is the user-interface specification language for the definition of a user-interface of ASF+SDF term windows.

10.1 TransLog

The TransLog module is very simple: it serves as a collecting module for the various transformations. Adding a new transformation in a later phase only means adding the related module to the import list of the TransLog module.

module TransLog
imports Unfold Fold AssocPermutation GoalSwitch CleanUp

10.2 Log

In Section 4 it was decided that the TransLog system should be equipped with navigation properties with respect to a sequence of transformed programs. Therefore, during the process of transforming an initial program P_0 the intermediate results (programs) have to be stored somewhere. As we shall see in Section 10.4, the only way to do this is to store programs in a separate ASF+SDF window. The module Log provides the necessary sort and functions for storing a sequence of programs. The sort PLIST represents a list of one or more programs, separated by the symbol $\diamond$. A list of programs can be initialised with an empty program, denoted by the function l-empty. The function l-append appends a program at the end of a list. The function l-delete removes the last program from a list. The function l-init has two arguments: a clause and a program. The result of this function is a program that consists of the clause and all clauses that come after it in the program. This function is used to store a set of clauses that can be used for a fold transformation at a later stage. The resulting list of clauses is stored as the first program in a list. The function l-first returns this first program.

module Log
imports PrologFunctions
exports
  sorts PLIST
  context-free syntax
    {PROGRAM "$\diamond$"}+ → PLIST

    l-empty "(" ")" → PROGRAM
    l-append "(" PLIST "," PROGRAM ")" → PLIST

l-delete "(" PLIST ")" → PLIST
l-init "(" CLAUSE "," PROGRAM ")" → PLIST
l-first "(" PLIST ")" → PROGRAM

"(" PLIST ")" → PLIST {**bracket**}

variables

P [′]*"+" → {PROGRAM "⋄"}+

equations

[ap1] l-append(P^+, P) = $P^+ \diamond P$

[de1] l-delete($P^+ \diamond P$) = P^+

[in1] l-init(C, C^* C $C^{*\prime}$) = C $C^{*\prime}$

[fi1] l-first($P \diamond P^+$) = P

10.3 ButtonConditions

A part of the TransLog user-interface is built on buttons, one for each transformation. The language SEAL offers the opportunity to enable/disable a button according to a certain condition. In the module ButtonConditions these conditions are specified in boolean functions e-....

The function e-unfold yields *true* when the term-to-be-unfolded is one of the goals in the body of a clause and the term is not a variable.

The function e-fold yields *true* when the first program of a stored program list (produced by the function l-first) contains a clause with a body that is unifiable with (a part of) the body of the folded clause. The success or failure of the unification is determined by the local function fold-mgu. This function uses the previously defined function body-mgu that determines the mgu of two bodies.

The function e-asperm yields *true* when its two arguments have the same predicate (atom) and the 'configuration' of the arguments corresponds with the intended permutation.This is checked in the local functions asperm1 and asperm2.

The function e-switch only checks if the selected term resides in the body of a clause and if there exists a next term in this body.

The function e-clean yields *true* when the term in its first argument is the head of a clause, given in its second argument.

The equations for the functions e-previous and e-next are obvious: they yield *true* when a previous program or a next program exists, otherwise they yield *false*.

module ButtonConditions
imports TransLog Log
exports
context-free syntax

e-unfold "(" TERM "," CLAUSE ")" → BOOL
e-fold "(" CLAUSE "," PLIST ")" → BOOL
e-asperm "(" TERM "," CLAUSE ")" → BOOL
e-switch "(" TERM "," CLAUSE ")" → BOOL
e-clean "(" TERM "," CLAUSE ")" → BOOL
e-previous "(" PLIST ")" → BOOL

e-next "(" PLIST ")" $\rightarrow$ BOOL

hiddens

context-free syntax

fold-mgu "(" CLAUSE "," PROGRAM ")" $\rightarrow$ SUBS
asperm1 "(" TERM "," TERM ")" $\rightarrow$ BOOL
asperm2 "(" TERM "," TERM ")" $\rightarrow$ BOOL

equations

$$[\text{eu1}]\quad \frac{\text{isBodyTerm}(T,\ C)\ =\ \text{true}}{\text{e-unfold}(T,\ C)\ =\ \text{not}(\text{isvar}(T))}$$

[eu] $\text{e-unfold}(T,\ C) = \text{false}$ **otherwise**

$$[\text{ef1}]\quad \frac{\text{l-first}(P^{+})\ =\ P,\ \text{fold-mgu}(C,\ P)\ \neq\ \text{fail}}{\text{e-fold}(C,\ P^{+})\ =\ \text{true}}$$

[ef] $\text{e-fold}(C,\ P^{+}) = \text{false}$ **otherwise**

[ea1] $\text{e-asperm}(T,\ T'\ \text{:-}\ T^{*},\ T,\ T'',\ T^{*\prime}\ .) = \text{asperm1}(T,\ T'')\ |\ \text{asperm2}(T,\ T'')$
[ea] $\text{e-asperm}(T,\ C) = \text{false}$ **otherwise**
[ap1] $\text{asperm1}(A(T,\ T',\ V),\ A(V,\ T'',\ V')) = \text{true}$
[ap1] $\text{asperm1}(T,\ T') = \text{false}$ **otherwise**
[ap2] $\text{asperm2}(A(V,\ V',\ T),\ A(V'',\ V''',\ V)) = \text{true}$
[ap2] $\text{asperm2}(T,\ T') = \text{false}$ **otherwise**

[es1] $\text{e-switch}(T,\ T''\ \text{:-}\ T^{*},\ T,\ T',\ T^{*\prime}\ .) = \text{true}$
[es] $\text{e-switch}(T,\ C) = \text{false}$ **otherwise**

[ec1] $\text{e-clean}(T,\ T\ .) = \text{true}$
[ec2] $\text{e-clean}(T,\ T\ \text{:-}\ T^{+}\ .) = \text{true}$
[ec] $\text{e-clean}(T,\ C) = \text{false}$ **otherwise**

[ep1] $\text{e-previous}(P \diamond P^{+}) = \text{true}$
[ep] $\text{e-previous}(P^{+}) = \text{false}$ **otherwise**

[en1] $\text{e-next}(\text{l-empty}\ (\ ) \diamond P^{+}) = \text{true}$
[en] $\text{e-next}(P^{+}) = \text{false}$ **otherwise**

$$[\text{fm1}]\quad \frac{\text{body-mgu}(T^{+};\ T^{+\prime})\ =\ S,\ S\ \neq\ \text{fail},\ \ T^{+}\ \neq\ T^{*},\ T^{+\prime},\ T^{*\prime}}{\text{fold-mgu}(T\ \text{:-}\ T^{*},\ T^{+\prime},\ T^{*\prime}\ .\ ,\ C^{*}\ \ T'\ \text{:-}\ T^{+}\ .\ C^{*\prime})\ =\ S}$$

[fm] $\text{fold-mgu}(C,\ P) = \text{fail}$ **otherwise**

10.4 TransLog.seal

The user-interface of the TransLog tool is defined in the SEAL module TransLog.seal. In this module eight buttons are defined for a TransLog term window. We will not go into the details of the language SEAL. The script below can be understood in a global sense without this knowledge. The generic structure of a button definition can be read as

```
button <button-name>
when <condition>
enable
  <operations on focus, windows, etc.>
doc : <some comments>
```

The condition determines whether or not the button has to be enabled. After each change in the TransLog term window this condition is evaluated. If the outcome of this evaluation is *false*, the button is disabled ('dimmed') and clicking on it has no effect. If the outcome is *true*, the button is enabled (highlighted) and after clicking on it the specified operations will be performed. These operations comprise focus operations (e.g. "move the focus one position upwards in the abstract syntax tree"), assignments (e.g. "assign the contents of a focus to a focus variable" or "assign the result of a function call from some other module to the focus"), window creation (for the Log windows).

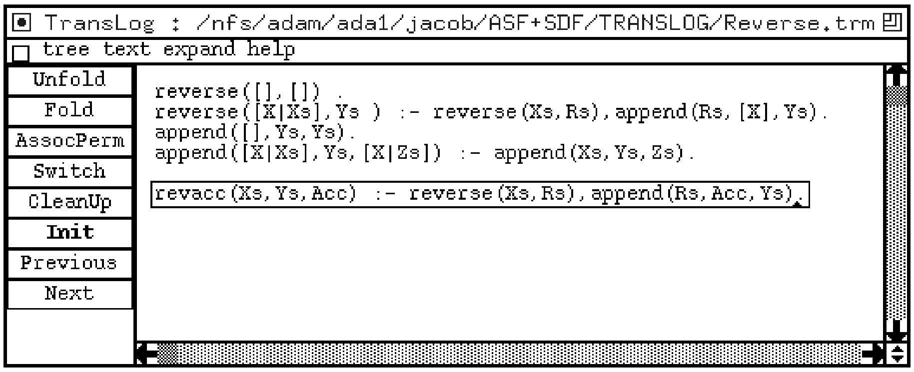
We give the text of the module without further comments.

```
Configuration for language TransLog is

button Unfold
when ButtonConditions : e-unfold(focus, focus up,up) and "LogA" . focus is PLIST
enable
  BtVar := focus;
  ClauseVar := focus up, up;
  RootVar := focus root;
  "LogA" . focus root;
  while "LogA" . focus next is PROGRAM do "LogA" . focus next od;
  LogVar := "LogA" . focus;
  "LogA" . focus :=  Log : l-append(LogVar,RootVar);
  "LogB" . focus root;
  "LogB" . focus := Log : l-empty();
  focus root;
  focus := Unfold : unfold(RootVar, BtVar, ClauseVar)
doc : "Unfold transformation on Prolog Program"

button Fold
when ButtonConditions : e-fold(focus up,up , "LogA" . focus) and PrologFunctions :
isBodyTerm(focus, focus up,up)
enable
  ClauseVar := focus up , up;
  RootVar := focus root;
  LogVar := "LogA" . focus;
  FoldVar := Log : l-first(LogVar);
  "LogA" . focus root;
  while "LogA" . focus next is PROGRAM do "LogA" . focus next od;
  "LogA" . focus :=  Log : l-append(LogVar,RootVar);
  "LogB" . focus root;
  "LogB" . focus := Log : l-empty();
  focus root;
```

```
  focus := Fold : fold(ClauseVar,FoldVar,RootVar)
doc : "Fold transformation on Prolog Program"

button AssocPerm
when ButtonConditions : e-asperm(focus , focus up,up) and "LogA" . focus is PLIST
enable
  BtVar := focus;
  ClauseVar := focus up , up;
  RootVar := focus root;
  "LogA" . focus root;
  while "LogA" . focus next is PROGRAM do "LogA" . focus next od;
  LogVar := "LogA" . focus;
  "LogA" . focus :=  Log : l-append(LogVar,RootVar);
  "LogB" . focus root;
  "LogB" . focus := Log : l-empty();
  focus root;
  focus := AssocPermutation : assperm(BtVar, ClauseVar, RootVar)
doc : "Permutation of atom arguments with mode +,+,-. or -,-,+"

button Switch
when ButtonConditions : e-switch(focus, focus up,up) and "LogA" . focus is PLIST
enable
  BtVar := focus;
  ClauseVar := focus up , up;
  RootVar := focus root;
  "LogA" . focus root;
  while "LogA" . focus next is PROGRAM do "LogA" . focus next od;
  LogVar := "LogA" . focus;
  "LogA" . focus :=  Log : l-append(LogVar,RootVar);
  "LogB" . focus root;
  "LogB" . focus := Log : l-empty();
  focus root;
  focus := GoalSwitch : switch(BtVar, ClauseVar, RootVar)
doc : "Switch on two adjacent bodyterms (shared vars allowed)"

button CleanUp
when ButtonConditions : e-clean(focus, focus up) and "LogA" . focus is PLIST
enable
  HeadVar := focus;
  RootVar := focus root;
  "LogA" . focus root;
  while "LogA" . focus next is PROGRAM do "LogA" . focus next od;
  LogVar := "LogA" . focus;
  "LogA" . focus :=  Log : l-append(LogVar,RootVar);
  "LogB" . focus root;
  "LogB" . focus := Log : l-empty();
  focus root;
  focus := CleanUp : cleanUp(HeadVar,RootVar)
doc : "Remove obsolete clauses from a program"

button Init
when focus is CLAUSE and PrologFunctions : syntaxCheck(focus root)
enable
  ClauseVar := focus;
  RootVar := focus root;
```

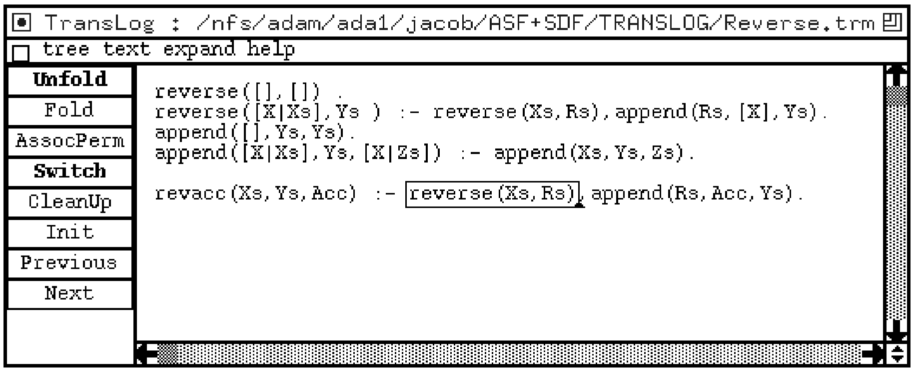
```
  create("LogA", Log : l-init(ClauseVar,RootVar));
  create("LogB", Log : l-empty())
doc : "Initialise log windows with part of initial program P0 and empty list."

button Previous
when ButtonConditions : e-previous("LogA" . focus)
enable
  RootVar := focus root;
  "LogB" . focus down;
  while "LogB" . focus next is PROGRAM do "LogB" . focus next od;
  LogVar := "LogB" . focus;
  "LogB" . focus := Log : l-append(LogVar,RootVar);
  "LogB" . focus root;
  "LogA" . focus down;
  while "LogA" . focus next is PROGRAM do "LogA" . focus next od;
  LogVar := "LogA" . focus;
  focus := LogVar;
  "LogA" . focus root;
  LogVar := "LogA" . focus;
  "LogA" . focus :=  Log : l-delete(LogVar)
doc: "One step back in transformation sequence."

button Next
when ButtonConditions : e-next("LogB" . focus)
enable
  RootVar := focus root;
  "LogA" . focus down;
  while "LogA" . focus next is PROGRAM do "LogA" . focus next od;
  LogVar := "LogA" . focus;
  "LogA" . focus :=  Log : l-append(LogVar,RootVar);
  "LogA" . focus root;
  "LogB" . focus down;
  while "LogB" . focus next is PROGRAM do "LogB" . focus next od;
  LogVar := "LogB" . focus;
  focus := LogVar;
  "LogB" . focus root;
  LogVar := "LogB" . focus;
  "LogB" . focus :=  Log : l-delete(LogVar)
doc: "One step forward in transformation sequence."
```

We conclude this section with some examples of what a TransLog term window looks like. The window below contains the initial reverse program from the example presented in Section 2.

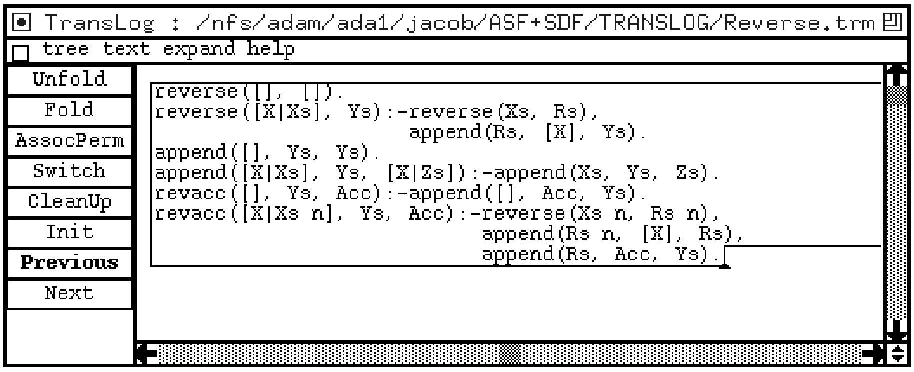


The focus is placed around the clause for the `revacc` predicate. At the left side of the window the eight buttons, defined in the SEAL script above, are present. Initially, the Init button is the only one that is enabled (highlighted).

The second window shows the same program just before the first unfold transformation. The focus is placed around the reverse goal in the body of the revacc clause, the Unfold button and the Switch button are enabled. (This window has also been showed in the Introduction.)



The third window shows the result of the unfold transformation on the selected goal. The unfolded clause is replaced by two new clauses.



11 Experiences and future plans

The prototype of the transformation tool TransLog has been constructed within a short time (about two man–months). The ASF+SDF Meta–environment showed to be a useful tool for both the formal specification of the syntax of the target language (Prolog) and the specification of transformation operations on terms of this language (programs). Several examples from the literature have been used as test–input for the TransLog tool. No errors were observed.

The ASF+SDF Meta–environment is built on a Lisp interpreter (INRIA's Le-Lisp). The execution time of the evaluation of functions (say, the response time after clicking on a transformation button) is slower than one would wish (5 to 10 seconds on a fast machine). This performance aspect can be improved by compilation of (parts of) the specification.

The ASF+SDF term windows have a generic built–in prettyprint facility. Unfortunately, this facility ruins the layout of a Prolog program. Therefore, a customized pretty printer for Prolog programs had to be created with the prettyprint tools that come with the ASF+SDF Meta-environment. These tools are described in [vdBV95].

The user-interface language SEAL has some shortcomings with respect to our purposes. Most noteworthy, SEAL does not offer 'global state variables'. So, for example, it is not possible to save the initial program P_0 of a transformation sequence. This feature would be helpful with respect to the implementation of the Fold transformation.

The work on the TransLog tool continues. Here are some plans for the near future:

- Develop a more strict (semantics preserving) variant of TransLog, which takes into account the various conditions that are known from the literature.
- Improve the selection of the goals (body terms) to-be-folded. In the current version of TransLog any clause (in the saved part of the initial program) with a body that unifies with one or more body terms from the folded clause is accepted for folding. In a future version the number of body terms involved in a folding transformation will be selected by the user of the system (e.g. in a simple dialogue).
- Extend the syntax definition: more attention for Prolog built–in predicates (e.g. no unfolding allowed on these predicates).
- Extend the system to constraint logic programs.
- More attention for efficient specifications. Until now, attention merely has been paid to correct specifications, not to efficient specifications. Probably, a redesign with respect to this topic will reduce the response times. On the other hand, a prototype like TransLog is not designed for high performance purposes.
- Investigate the incorporation of 'transformation strategies': well-defined sequences of transformations that can automatically be applied to a certain class of programs.

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

References

[ABFQ92] 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.

[BD77] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[EGP95] S. Eijkelkamp, D. Geluk, and M. Polling. Program Transformation using ASF+SDF. Master's thesis, University of Amsterdam, 1995.

[Gal86] J. Gallagher. Transforming logic programs by specialising interpreters. In B. du Boulay, D. Hogg, and L. Steels, editors, *Advances in Artificial Intelligence - II*, pages 313–326. North Holland, 1986.

[Kli93] P. Klint. A meta–environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Kom90] J. Komorowski. Towards a Programming Methodology Founded on Partial Deduction. In L.G. Aiello, editor, *ECAI90, Proceedings of the 9th European Conference on AI*, pages 404–409. Pitman Publ., 1990.

[Koo93] 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.

[KT94] J. Komorowski and S. Trcek. Towards Refinement of Definite Logic Programs. In Z.W. Raś and M. Zemankova, editors, *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, LNCS 869, pages 315–325. Springer–Verlag, 1994.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[PP94] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19, 20:261–320, 1994.

[Sah91] D. Sahlin. *An automatic partial evaluator for full Prolog.* PhD thesis, The Royal Institute of Technology (KTH) / SICS, 1991.

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

[SIC92] SICS – Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog User's Manual*, 1992.

[TS84] H. Tamaki and T. Sato. Unfold/Folf Transformation of Logic Programs. In S.-Å. Tärnlund, editor, *Proceedings of the 2nd International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

[vdBV95] M.G.J. van den Brand and E. Visser. Generation of Formatters for Context-free Languages. Technical Report P9506, Programming Research Group, University of Amsterdam, 1995. Submitted for publication elsewhere.