A transformation tool for pure Prolog programs – the algebraic specification –

Jacob Brunekreef

Programming Research Group, University of Amsterdam

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

e-mail: jacob@fwi.uva.nl

Abstract

This technical report contains the complete annotated algebraic specification of 'TransLog', a transformation tool for pure Prolog programs. TransLog supports the interactive transformation of (a part of) a program by means of buttons representing transformation steps. As much as possible, pure Prolog semantics are preserved. The choice of pure Prolog semantics induces a number of conditions that have to be satisfied before a particular transformation step is to be applied.

Keywords: logic programming, program transformation, algebraic specification.

1 Introduction

In this report we present the complete algebraic specification of *TransLog*, a tool for transformation of pure Prolog programs. The report does not contain an introduction to the theory of (logic) program transformation, nor a description of how the TransLog tool relates to the various theoretical considerations. For the first subject the reader is referred to [TS84], [Sek91] or [PP94]. For the second subject, see [Bru96].

In the TransLog tool six transformation steps have been implemented: unfolding, folding, goal definition, argument permutation (an instance of goal replacement), goal switching and definition elimination. A transformation step is correct (with respect to a given semantics) only if the resulting program has the same semantics as the original one. To achieve correctness most operations require appropriate applicability conditions which –in general– depend on the chosen semantics. We focus on the Prolog semantics given by the 'sequence of answer substitutions' semantics. The TransLog tool is equipped with two 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 $P_0 \ldots P_{i-1},\ P_i,\ P_{i+1}, \ldots P_n$.

The TransLog 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 focuses on the algebraic specification of the TransLog tool. Section 2 provides a very short introduction to the ASF+SDF formalism. The TransLog specification is presented in the Sections 3 to 8.

In [Bru95] a first version of the TransLog tool is described. This version did not include any test of applicability conditions, related to any semantical framework. So, the algebraic specification presented in that report is far more simple than the one presented here.

2 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 customized 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]).

3 TransLog specification outline

The complete TransLog specification consists of twenty-three 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 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 4.

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, list normalisation, etc. These functions are grouped in three modules:

- NormalisationFunction
- PrologFunctions
- LabeledProgram

Sets of (Prolog) terms are specified in a separate module TermSets. The syntax modules are presented in Section 5.

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

   - Substitution
   - Equations
   - Unification

   These modules are presented in Section 6.

4. Transformation modules.
The six TransLog transformation steps are specified in separate modules. So we have the modules

   - Unfold
   - Fold
   - Define
   - AssocPermutation
   - GoalSwitch
   - CleanUp

   The module VariantClause is imported by the modules Unfold and Fold. The seven transformation modules are presented in Section 7.

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

   - ErrorMessages
   - TransLog
   - Log
   - ButtonConditions
   - TransLog.seal

   The last module is written in the special purpose language SEAL. These modules are presented in Section 8.

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

Figure 1: TransLog modules import graph

4 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.

4.1 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

4.2 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
    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

5 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 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.

The Prolog syntax is defined in two modules, PrologTerms and PrologProgram. A number of functions on Prolog terms and constructs is defined in the modules NormalisationFunction and PrologFunctions. Sets of Prolog terms are specified in a separate module TermSets. In [Bru96] it is discussed that body goals in a program clause have to labeled *fold-allowing* or *not fold-allowing*. For each body goal in a program P_N the actual label value is determined by the transformation history leading from P_0 to P_N. This requires an 'internal program representation' in which labels are added to body terms. The syntax of this internal representation, as well as many functions on a 'labeled program', are specified in the module LabeledProgram.

5.1 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.

In the TransLog specification variables of a sort S are declared in the module M in which S is defined. These variables are exported, so they can be used in every module that imports the module M. In the module PrologTerms variables concerning the various term-sorts are declared and exported.

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**}

variables

V [′]* → VARIABLE
I [′]* → INTEGER
T [′]* → TERM
T [′]*“+” → {TERM “,”}+
T [′]*“*” → {TERM “,”}*
L [′]* → LIST
A [′]* → ATOM

5.2 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$*$ → PROGRAM
TERM "." → CLAUSE
TERM ":-" TERMLIST "." → CLAUSE
"(" CLAUSE ")" → CLAUSE {**bracket**}
"(" PROGRAM ")" → PROGRAM {**bracket**}
variables
C [$'$]$*$ → CLAUSE
C [$'$]$*$"$*$" → CLAUSE$*$
P [$'$]$*$ → PROGRAM

5.3 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 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.

The function *restore* restores the 'square bracket' representation of a Prolog list.

module NormalisationFunction
imports PrologTerms
exports
context-free syntax
norm "(" TERM ")" → TERM
restore "(" TERM ")" → TERM

equations

[n1] $\mathrm{norm}([\,]) = [\,]$
[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] $\mathrm{norm}(A(T)) = A(\mathrm{norm}(T))$

[n10] $$\frac{\mathrm{norm}(A(T^+)) = A(T''^+),\ \mathrm{norm}(A(T'^+)) = A(T'''^+)}{\mathrm{norm}(A(T^+, T'^+)) = A(T''^+, T'''^+)}$$

[n] $\mathrm{norm}(T) = T$ **otherwise**

[r1] $\text{restore}(.(T, [\,])) = [T]$
[r2] $\text{restore}(.(T, V)) = [T \mid V]$

[r3] $$\frac{\text{restore}(T') = L}{\text{restore}(.(T, T')) = [T \mid L]}$$

[r] $\text{restore}(T) = T$ **otherwise**

5.4 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, an integer? 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 a 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 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. The function that checks the syntax of a clause is exported, it will be used in the module that deals with the definition of a new clause (see Section 7.3).

  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 term, a clause or a program. The function *simplify* simplifies a predicate term $a(t_1, \ldots, t_n)$ to a term with the same predicate and n underscore-arguments. This function is needed in determining whether or not two terms hold the same predicate with the same arity. 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. The functions *varset* and *varset-T* produce the set ot variables that occur in a clause/a term. The sort TERM-SET is imported from a separate module TermSets. This module is presented in the following section.

module PrologFunctions
imports PrologProgram NormalisationFunction TermSets
exports
context-free syntax

isVar "(" TERM ")"	→ BOOL
isInteger "(" TERM ")"	→ BOOL
isEmpty "(" LIST ")"	→ BOOL
VARIABLE "$\sqsubset$" TERM	→ BOOL

isBodyTerm "(" TERM "," CLAUSE ")" → BOOL

syntaxCheck "(" PROGRAM ")" → BOOL
syntaxCheck-C "(" CLAUSE ")" → BOOL

simplify "(" TERM ")" → TERM
head "(" CLAUSE ")" → TERM
addcp "(" CLAUSE "," PROGRAM ")" → PROGRAM
removecp "(" CLAUSE "," PROGRAM ")" → PROGRAM
replacecp "(" PROGRAM "," CLAUSE "," PROGRAM ")" → PROGRAM
varset "(" CLAUSE ")" → TERM-SET
varset-T "(" {TERM ","}+ ")" → TERM-SET

hiddens

context-free syntax

VARIABLE "$\sqsubset_h$" TERM → BOOL
syntaxCheck-H "(" TERM ")" → BOOL
syntaxCheck-B "(" TERMLIST ")" → BOOL
syntaxCheck-BT "(" TERM ")" → BOOL

equations

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

[ii1] isInteger(I) = true
[ii] isInteger(T) = false **otherwise**

[ie1] isEmpty([ ]) = true
[ie] isEmpty(L) = false **otherwise**

[oc1] $V \sqsubset T = V \sqsubset_h \text{norm}(T)$
[oh1] $V \sqsubset_h A(T^*, V, T'^*) = \text{true}$

[oh2] $$\frac{V \sqsubset_h T = \text{true}}{V \sqsubset_h A(T^*, T, T'^*) = \text{true}}$$

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

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

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

[cc1] syntaxCheck-C(T .) = syntaxCheck-H(T)
[cc2] syntaxCheck-C(T :- T^+ .) = syntaxCheck-H(T) & syntaxCheck-B(T^+)

[bc1] syntaxCheck-B(T) = syntaxCheck-BT(T)

[bc2] $\text{syntaxCheck-B}(T,\ T^+) = \text{syntaxCheck-BT}(T)\ \&\ \text{syntaxCheck-B}(T^+)$

[hc1] $\text{syntaxCheck-H}(A) = \text{true}$
[hc2] $\text{syntaxCheck-H}(A(T^+)) = \text{true}$
[hc] $\text{syntaxCheck-H}(T) = \text{false}$ **otherwise**

[bt1] $\text{syntaxCheck-BT}(A) = \text{true}$
[bt2] $\text{syntaxCheck-BT}(A(T^+)) = \text{true}$
[bt3] $\text{syntaxCheck-BT}(V) = \text{true}$
[bt4] $\text{syntaxCheck-BT}(T\ A\ T') = \text{true}$
[bt] $\text{syntaxCheck-BT}(T) = \text{false}$ **otherwise**

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

[s1] $\text{simplify}(A(T)) = A(_)$

[s2] $$\frac{\text{simplify}(A(T^+)) = A(T''^+),\ \text{simplify}(A(T'^+)) = A(T'''^+)}{\text{simplify}(A(T^+,\ T'^+)) \ =\ A(T''^+,\ T'''^+)}$$

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

[h1] $\text{head}(T\ .) = T$
[h2] $\text{head}(T\ \text{:-}\ T^+\ .) = T$

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

[p1] $\text{replacecp}(C^*\ C\ C'^*,\ C,\ C''^*) = C^*\ C''^*\ C'^*$
[p] $\text{replacecp}(P,\ C,\ P') = P$ **otherwise**

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

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

[st7] $$\frac{\text{isEmpty}(L) = \text{true}}{\text{varset-T}(L) \ =\ \{\}}$$

[st8] $$\frac{\text{isEmpty}(L) = \text{false}}{\text{varset-T}(L) \ =\ \text{varset-T}(\text{norm}(L))}$$

[st9] $\text{varset-T}(T^+,\ T'^+) = \text{varset-T}(T^+) \cup \text{varset-T}(T'^+)$

5.5 TermSets

In various TransLog functions 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.

module TermSets
imports PrologTerms Booleans
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 ")" → TERM-SET {**bracket**}
  variables
    TS ['] * → TERM-SET

equations

[se1] $\{T^*, T, T'^*, T, T''^*\} = \{T^*, T, T'^*, T''^*\}$

[un1] $\{T^*\} \cup \{T'^*\} = \{T^*, T'^*\}$

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

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

[me1] $T \in \{T^*, T, T'^*\} = \text{true}$
[me] $T \in \{T^*\} = \text{false}$ **otherwise**

5.6 LabeledProgram

In this module the syntax of a Prolog program with labeled body goals is specified. The sort LATERM for a labeled body term is defined by LABEL * TERM. A label is specified as a sequence of two lower case characters. The syntax definition of a labeled program, a labeled clause, a labeled term-list is straightforward.

In this module also a number of functions is specified which attach labels to a program, a clause, etc. (the functions *set-label*), to remove labels from a program, a clause, etc. (the functions *strip-label*) and to change a label in a program, a clause, etc. (the functions *change-label*).

The functions *lhead*, *lvarset*, *la-replacecp*, *la-addcp* and *la-removecp* are the counterparts of the same functions on non-labeled programs in the module PrologFunctions.

The function *la-version* produces the labeled version of a given unlabeled clause, taken from a given labeled program. Finally, the boolean function *isDefined* determines if a predicate is defined in a labeled program.

Remark: the module ErrorMessages is imported. Error messages are not needed at this moment, the one and only reason for importing this module here is avoiding multiple inputs in the various transformation modules later on.

module LabeledProgram
imports ErrorMessages
exports
sorts LABEL LATERM LATERMLIST LACLAUSE LAPROGRAM
lexical syntax
[a-z][a-z] → LABEL
context-free syntax

LABEL "*" TERM	→ LATERM	
{LATERM ","}+	→ LATERMLIST	
TERM "."	→ LACLAUSE	
TERM ":-" LATERMLIST "."	→ LACLAUSE	
LACLAUSE*	→ LAPROGRAM	
"(" LATERM ")"	→ LATERM	{**bracket**}
"(" LATERMLIST ")"	→ LATERMLIST	{**bracket**}
"(" LACLAUSE ")"	→ LACLAUSE	{**bracket**}
"(" LAPROGRAM ")"	→ LAPROGRAM	{**bracket**}
set-label "(" LABEL ";" PROGRAM ")"	→ LAPROGRAM	
set-label-C "(" LABEL ";" CLAUSE ")"	→ LACLAUSE	
set-label-TL "(" LABEL ";" TERMLIST ")"	→ LATERMLIST	
set-label-T "(" LABEL ";" TERM ")"	→ LATERM	
strip-label "(" LAPROGRAM ")"	→ PROGRAM	
strip-label-C "(" LACLAUSE ")"	→ CLAUSE	
strip-label-TL "(" LATERMLIST ")"	→ TERMLIST	
strip-label-T "(" LATERM ")"	→ TERM	
change-label "(" LABEL ";" LAPROGRAM ")"	→ LAPROGRAM	
change-label-C "(" LABEL ";" LACLAUSE ")"	→ LACLAUSE	
change-label-TL "(" LABEL ";" LATERMLIST ")"	→ LATERMLIST	
change-label-T "(" LABEL ";" LATERM ")"	→ LATERM	
lhead "(" LACLAUSE ")"	→ TERM	
lvarset "(" LACLAUSE ")"	→ TERM-SET	
la-replacecp "(" LAPROGRAM "," LACLAUSE "," LAPROGRAM ")"	→ LAPROGRAM	
la-addcp "(" LACLAUSE "," LAPROGRAM ")"	→ LAPROGRAM	
la-removecp "(" LACLAUSE "," LAPROGRAM ")"	→ LAPROGRAM	
la-version-C "(" CLAUSE ";" LAPROGRAM ")"	→ LACLAUSE	
isDefined "(" TERM ";" LAPROGRAM ")"	→ BOOL	

variables

LA ['] *	→ LABEL
LP ['] *	→ LAPROGRAM
LC ['] *	→ LACLAUSE
LC ['] * "*"	→ LACLAUSE*
LT ['] *	→ LATERM
LT ['] * "+"	→ {LATERM ","}+

LT [′]∗“∗” → {LATERM “,”}∗

equations

[slp1] set-label(LA;) =

$$[\text{slp2}]\ \frac{\text{set-label-C}(LA;\ C) = LC,\ \ \text{set-label}(LA;\ C^*) = LC^*}{\text{set-label}(LA;\ C\ C^*)\ =\ LC\ LC^*}$$

[slc1] set-label-C(LA; T .) = T .
[slc2] set-label-C(LA; T :- T^+ .) = T :- set-label-TL(LA; T^+) .

$$[\text{slt1}]\ \frac{\text{set-label-T}(LA;\ T) = LT}{\text{set-label-TL}(LA;\ T)\ =\ LT}$$

$$[\text{slt2}]\ \frac{\text{set-label-T}(LA;\ T) = LT,\ \ \text{set-label-TL}(LA;\ T^+) = LT^+}{\text{set-label-TL}(LA;\ T,\ T^+)\ =\ LT,\ LT^+}$$

[slt3] set-label-T(LA; T) = $LA * T$

[rlp1] strip-label() =

$$[\text{rlp2}]\ \frac{\text{strip-label-C}(LC) = C,\ \ \text{strip-label}(LC^*) = C^*}{\text{strip-label}(LC\ LC^*)\ =\ C\ C^*}$$

[rlc1] strip-label-C(T .) = T .
[rlc2] strip-label-C(T :- LT^+ .) = T :- strip-label-TL(LT^+) .

$$[\text{rlt1}]\ \frac{\text{strip-label-T}(LT) = T}{\text{strip-label-TL}(LT)\ =\ T}$$

$$[\text{rlt2}]\ \frac{\text{strip-label-T}(LT) = T,\ \ \text{strip-label-TL}(LT^+) = T^+}{\text{strip-label-TL}(LT,\ LT^+)\ =\ T,\ T^+}$$

[rlt3] strip-label-T($LA * T$) = T

[clp1] change-label(LA;) =

$$[\text{clp2}]\ \frac{\text{change-label-C}(LA;\ LC) = LC',\ \ \text{change-label}(LA;\ LC^*) = LC'^*}{\text{change-label}(LA;\ LC\ LC^*)\ =\ LC'\ LC'^*}$$

[clc1] change-label-C(LA; T .) = T .
[clc2] change-label-C(LA; T :- LT^+ .) = T :- change-label-TL(LA; LT^+) .

$$[\text{ctl1}]\quad \frac{\text{change-label-T}(LA;\ LT) = LT'}{\text{change-label-TL}(LA;\ LT)\ =\ LT'}$$

$$[\text{ctl2}]\quad \frac{\text{change-label-T}(LA;\ LT) = LT',\ \ \text{change-label-TL}(LA;\ LT^+) = LT'^+}{\text{change-label-TL}(LA;\ LT,\ LT^+)\ =\ LT',\ LT'^+}$$

$$[\text{clt1}]\quad \text{change-label-T}(LA;\ LA'\ *\ T) = LA\ *\ T$$

$$[\text{lh1}]\quad \text{lhead}(T\ .) = T$$
$$[\text{lh2}]\quad \text{lhead}(T\ \text{:-}\ LT^+\ .) = T$$

$$[\text{lv1}]\quad \frac{\text{strip-label}(LC) = C}{\text{lvarset}(LC)\ =\ \text{varset}(C)}$$

$$[\text{lrc1}]\quad \text{la-replacecp}(LC^*\ LC\ LC'^*,\ LC,\ LC''^*) = LC^*\ LC''^*\ LC'^*$$
$$[\text{lrc}]\quad \text{la-replacecp}(LP,\ LC,\ LP') = LP \quad \textbf{otherwise}$$

$$[\text{lac1}]\quad \text{la-addcp}(LC,\ LC^*) = LC\ LC^*$$

$$[\text{lrc1}]\quad \text{la-removecp}(LC,\ LC^*\ LC\ LC'^*) = LC^*\ LC'^*$$
$$[\text{lrc}]\quad \text{la-removecp}(LC,\ LP) = LP \quad \textbf{otherwise}$$

$$[\text{lv1}]\quad \frac{\text{strip-label-C}(LC) = C}{\text{la-version-C}(C;\ LC^*\ LC\ LC'^*)\ =\ LC}$$

$$[\text{idf1}]\quad \frac{\text{simplify}(T) = T',\ \ \text{lhead}(LC) = T'',\ \ \text{simplify}(T'') = T'}{\text{isDefined}(T;\ LC^*\ LC\ LC'^*)\ =\ \text{true}}$$

$$[\text{idf}]\quad \text{isDefined}(T;\ LP) = \text{false} \quad \textbf{otherwise}$$

6 Unification modules

The unification of two terms plays an important role in several transformations specified in Section 7. 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.

6.1 Substitution

In this module the syntax of a 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: 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: *isMember* investigates whether or not a variable is part of a substitution sequence, the function *getTerm* produces the term that is to be substituted for a given variable.

The function *isVarRenaming*, together with the local function *varTOvar*, checks if a sequence of substitutions is a proper variable renaming: every variable is renamed to a variable, no two different variables are renamed to the same variable.

In the final part of this module the effect of (a sequence of) substitutions on various constructs (clause, termlist (body), termset and term) is specified. A list of substitutions on a construct C is denoted by $[c\ S]$. The last three exported functions define the effect of a substitution on a labeled program construct.

module Substitution
imports LabeledProgram
exports
  sorts SUBS SUB
  context-free syntax
    "(" VARIABLE "↦" TERM ")" → SUB
    "{" {SUB ","}* "}" → SUBS
    fail → SUBS

    SUBS "⊗" SUBS → SUBS {**left**}
    SUBS "⊕" SUBS → SUBS {**left**}
    isMember "(" VARIABLE "," SUBS ")" → BOOL
    getTerm "(" VARIABLE "," SUBS ")" → TERM

    isVarRenaming "(" SUBS ")" → BOOL

    CLAUSE "[c" SUBS "]" → CLAUSE
    TERMLIST "[tl" SUBS "]" → TERMLIST
    TERM "[t" SUBS "]" → TERM
    TERM-SET "[ts" SUBS "]" → TERM-SET

    LACLAUSE "[lc" SUBS "]" → LACLAUSE
    LATERMLIST "[ltl" SUBS "]" → LATERMLIST
    LATERM "[lt" SUBS "]" → LATERM
  variables
    S ['] → SUBS
    S ['] "*" → {SUB ","}*
hiddens
  context-free syntax

varTOvar "(" SUBS ")" → BOOL

equations

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

[sj3] $$\frac{\text{isMember}(V, S) = \text{true},\ \ \text{getTerm}(V, S) = T}{\{(V \mapsto T),\ S^*\} \otimes S\ =\ \{S^*\} \otimes S}$$

[sj4] $$\frac{\text{isMember}(V, S) = \text{true},\ \ \text{getTerm}(V, S) \neq T}{\{(V \mapsto T),\ S^*\} \otimes S\ =\ \text{fail}}$$

[sj5] $$\frac{\text{isMember}(V, S) = \text{false},\ \ \{S^*\} \otimes S = \{S'^*\}}{\{(V \mapsto T),\ S^*\} \otimes S\ =\ \{(V \mapsto T\ [\text{t}\ S]),\ S'^*\}}$$

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

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

[m1] $\text{isMember}(V, \{S^*,\ (V \mapsto T),\ S'^*\}) = \text{true}$
[m] $\text{isMember}(V, S) = \text{false}$ **otherwise**

[g1] $\text{getTerm}(V, \{S^*,\ (V \mapsto T),\ S'^*\}) = T$
[g] $\text{getTerm}(V, S) = V$ **otherwise**

[ir1] $\text{isVarRenaming}(\{S^*,\ (V \mapsto T),\ S'^*,\ (V' \mapsto T),\ S''^*\}) = \text{false}$
[ir] $\text{isVarRenaming}(S) = \text{varTOvar}(S)$ **otherwise**

[vtv1] $\text{varTOvar}(\{\}) = \text{true}$
[vtv] $\text{varTOvar}(\{(V \mapsto T),\ S^*\}) = \text{isVar}(T)\ \&\ \text{varTOvar}(\{S^*\})$ **otherwise**

[cl1] $T\ .\ [\text{c}\ S] = T\ [\text{t}\ S]\ .$
[cl2] $T\ \text{:-}\ T^+\ .\ [\text{c}\ S] = T\ [\text{t}\ S]\ \text{:-}\ T^+\ [\text{tl}\ S]\ .$

[tl1] $$\frac{T\ [\text{t}\ S] = T'}{T\ [\text{tl}\ S]\ =\ T'}$$

[tl2] $$\frac{T^+\ [\text{tl}\ S] = T''^+,\ \ T'^+\ [\text{tl}\ S] = T'''^+}{T^+,\ T'^+\ [\text{tl}\ S]\ =\ T''^+,\ T'''^+}$$

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

[va1] $V\,[\mathrm{t}\ S] = \mathrm{getTerm}(V,\ S)$

[in1] $I\,[\mathrm{t}\ S] = I$

[li1] $[\,]\ [\mathrm{t}\ S] \quad = [\,]$
[li2] $[T^+]\ [\mathrm{t}\ S] = [T^+\ [\mathrm{tl}\ S]]$

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

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

[ts3] $$\frac{\{T^+\}\ [\mathrm{ts}\ S] = \{T''^+\},\ \{T'^+\}\ [\mathrm{ts}\ S] = \{T'''^+\}}{\{T^+,\ T'^+\}\ [\mathrm{ts}\ S]\ =\ \{T''^+,\ T'''^+\}}$$

[lcl1] $T\,.\ [\mathrm{lc}\ S] = T\,[\mathrm{t}\ S]\ .$

[lcl2] T :- $LT^+\ .\ [\mathrm{lc}\ S] = T\,[\mathrm{t}\ S]$:- $LT^+\ [\mathrm{ltl}\ S]\ .$

[ltl1] $$\frac{LT\,[\mathrm{lt}\ S] = LT'}{LT\,[\mathrm{ltl}\ S]\ =\ LT'}$$

[ltl2] $$\frac{LT^+\ [\mathrm{ltl}\ S] = LT''^+,\ LT'^+\ [\mathrm{ltl}\ S] = LT'''^+}{LT^+,\ LT'^+\ [\mathrm{ltl}\ S]\ =\ LT''^+,\ LT'''^+}$$

[lat1] $LA * T\,[\mathrm{lt}\ S] = LA * T\,[\mathrm{t}\ S]$

6.2 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 ≡ 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 "]" → EQ
    EQS "[es" SUBS "]" → EQS
  variables
    E [′]* → EQ
    E [′]*"+" → {EQ ","}+
    E [′]*"*" → {EQ ","}*
    EQS [′]* → EQS

equations

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

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

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

6.3 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 actions 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 four 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 function *mgu-tl* determines the most general unifier of two term-lists. 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 restore function is needed because the most general unifier for lists is determined by first normalising the list representation (see Section 5.3). The equations for the functions *mgu-nv* and *mgu-tl* need no further explanation.

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

equations

[mg1] $\text{mgu}(EQS) = \text{mgu}(EQS, \{\})$

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

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

[mg] $\text{mgu}(EQS,\ S) = S$ **otherwise**

[nv1] $\text{mgu-nv}(I,\ I) = \{\}$

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

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

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

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

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

[tl1] $\text{mgu-tl}(T^+;\ T'^+) = \text{mgu-nv}(\texttt{a}(T^+),\ \texttt{a}(T'^+))$

7 Transformation modules

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

7.1 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 goal 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 ($\text{mgu}(\text{head}(C'), T) = \theta,\ \theta \neq \text{fail}$)
    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 program P
remove C from the program P.

The function *unfold* has three input-arguments: the current (labeled) program, the body goal to-be-unfolded and the clause that holds this body goal (the unfolded clause). The result of this function is the transformed program. The 'work' is done by the function *unfoldResult*. This function produces 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 the error message is returned. An error message consists of two clauses, one clause containing a standard error message, the other clause containing a specific indication of the kind of error. Both clauses are labeled *el* ('error label'). This label has no meaning.

The function *unfoldResult* recursively investigates all clauses of the program. If the head of a certain clause is unifiable with the body goal to-be-unfolded (the result of the *mgu-nv* function is not equal to *fail*), then the clause is selected as an unfolding clause and the unfold applicability conditions are checked. First it is investigated if the body goal to-be-unfolded is the leftmost goal of the body of the unfolded clause. If this is the case, then a resulting clause is constructed. Otherwise, the conditions on deterministic unfolding and non-left propagating unfolding are checked. In the case that one of these conditions is not satisfied, an error message is returned. If both conditions are satisfied, a resulting clause is constructed and added to the result of the function *unfoldResult*.

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 goal, a variant of this clause is constructed that has no variables in common with the clause C in which the body goal 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 goal are unifiable and the applicability conditions are satisfied, 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 goal to-be-unfolded by the body of the unfolding clause. To be more precise: the body goal is replaced by the body of the *variant* of the unfolding clause; furthermore the substitutions that result from the unification of the unfolded goal 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 goal vanishes. If this goal is the only body goal 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 body goals.

The functions *leftmostUnfold*, *noMoreUclauses* and *isVariant* are applied while checking the applicability conditions. The definition of these functions is straightforward.

module Unfold
imports VariantClause
exports
context-free syntax
unfold "(" LAPROGRAM "," TERM "," CLAUSE ")" → LAPROGRAM
unfoldResult "(" LAPROGRAM "," TERM "," LACLAUSE ")" → LAPROGRAM
resolvent "(" LACLAUSE "," TERM "," LACLAUSE ")" → LACLAUSE
hiddens
context-free syntax
unf "(" LAPROGRAM "," LACLAUSE "," LAPROGRAM ")" → LAPROGRAM
ucl1 "(" LAPROGRAM "," TERM "," LACLAUSE ")" → LAPROGRAM
ucl2 "(" LAPROGRAM "," TERM "," LACLAUSE ")" → LAPROGRAM
ucl3 "(" LAPROGRAM "," TERM "," LACLAUSE ")" → LAPROGRAM
ucl4 "(" LACLAUSE "," TERM "," LACLAUSE ")" → LAPROGRAM

leftmostUnfold "(" TERM "," LACLAUSE ")" → BOOL
noMoreUclauses "(" TERM "," LAPROGRAM ")" → BOOL
isVariant "(" TERM "," LACLAUSE "," SUBS ")" → BOOL
nonex "(" {LATERM ","}* ")" → BOOL

equations

[ufd1] $$\frac{\text{la-version-C}(C;\, LP) = LC,\ \ \text{unfoldResult}(LP,\, T,\, LC) = LP'}{\text{unfold}(LP,\, T,\, C)\ =\ \text{unf}(LP,\, LC,\, LP')}$$

[uf1] $$\frac{\text{strip-label}(LP') = P,\ \ \text{isErrorMessage}(P) = \text{false}}{\text{unf}(LP,\, LC,\, LP')\ =\ \text{la-replacecp}(LP,\, LC,\, LP')}$$

[uf1] $\mathrm{unf}(LP,\ LC,\ LP') = LP'$ **otherwise**

[ur1] $\mathrm{unfoldResult}(,\ T,\ LC) =$

$$[\text{ur2}]\ \frac{\mathrm{variant}(LC',\ LC) = LC''}{\mathrm{unfoldResult}(LC'\ LC^*,\ T,\ LC)\ =\ \mathrm{ucl1}(LC''\ LC^*,\ T,\ LC)}$$

$$[\text{uch1}]\ \frac{\text{mgu-nv}(T,\ \mathrm{lhead}(LC')) = \mathrm{fail}}{\mathrm{ucl1}(LC'\ LC^*,\ T,\ LC)\ =\ \mathrm{unfoldResult}(LC^*,\ T,\ LC)}$$

$$[\text{uch2}]\ \frac{\text{mgu-nv}(T,\ \mathrm{lhead}(LC')) \neq \mathrm{fail}}{\mathrm{ucl1}(LC'\ LC^*,\ T,\ LC)\ =\ \mathrm{ucl2}(LC'\ LC^*,\ T,\ LC)}$$

$$[\text{uch3}]\ \frac{\mathrm{leftmostUnfold}(T,\ LC) = \mathrm{true}}{\mathrm{ucl2}(LC'\ LC^*,\ T,\ LC)\ =\ \text{la-addcp}(\mathrm{resolvent}(LC',\ T,\ LC),\ \mathrm{unfoldResult}(LC^*,\ T,\ LC))}$$

$$[\text{uch4}]\ \frac{\mathrm{leftmostUnfold}(T,\ LC) = \mathrm{false}}{\mathrm{ucl2}(LC'\ LC^*,\ T,\ LC)\ =\ \mathrm{ucl3}(LC'\ LC^*,\ T,\ LC)}$$

$$[\text{uch5}]\ \frac{\mathrm{noMoreUclauses}(T,\ LC^*) = \mathrm{false}}{\mathrm{ucl3}(LC'\ LC^*,\ T,\ LC)\ =\ \text{set-label}(\texttt{el};\ \text{standard derrorm})}$$

$$[\text{uch6}]\ \frac{\mathrm{noMoreUclauses}(T,\ LC^*) = \mathrm{true}}{\mathrm{ucl3}(LC'\ LC^*,\ T,\ LC)\ =\ \mathrm{ucl4}(LC',\ T,\ LC)}$$

$$[\text{uch7}]\ \frac{\begin{array}{c}\text{mgu-nv}(T,\ \mathrm{lhead}(LC')) = S,\ \ \mathrm{isVariant}(T,\ LC,\ S) = \mathrm{true},\\ \mathrm{resolvent}(LC',\ T,\ LC) = LC''\end{array}}{\mathrm{ucl4}(LC',\ T,\ LC)\ =\ LC''}$$

$$[\text{uch8}]\ \frac{\text{mgu-nv}(T,\ \mathrm{lhead}(LC')) = S,\ \ \mathrm{isVariant}(T,\ LC,\ S) = \mathrm{false}}{\mathrm{ucl4}(LC',\ T,\ LC)\ =\ \text{set-label}(\texttt{el};\ \text{standard verrorm})}$$

[lu1] $\mathrm{leftmostUnfold}(T,\ T'$:- $LA * T,\ LT^*$.$) = \mathrm{true}$
[lu] $\mathrm{leftmostUnfold}(T,\ LC) = \mathrm{false}$ **otherwise**

$$[\text{nmu1}]\ \frac{\text{mgu-nv}(T,\ \mathrm{lhead}(LC)) = S,\ \ S \neq \mathrm{fail}}{\mathrm{noMoreUclauses}(T,\ LC^*\ LC\ LC'^*)\ =\ \mathrm{false}}$$

[nmu] $\mathrm{noMoreUclauses}(T,\ LP) = \mathrm{true}$ **otherwise**

$$[\text{iv1}]\ \frac{\text{strip-label-TL}(LT^+) = T^+}{\begin{array}{l}\mathrm{isVariant}(T,\ T'\ \text{:-}\ LT^+,\ LA * T,\ LT^*\ .\ ,\ S)\ =\\ \mathrm{isVarRenaming}(\text{mgu-tl}(T',\ T^+;\ T',\ T^+\ [\mathrm{tl}\ S]))\end{array}}$$

[iv] $\mathrm{isVariant}(T,\ LC,\ S) = \mathrm{false}$ **otherwise**

$$[\text{re1}]\quad \frac{\text{mgu-nv}(T,\ T') = S,\ \ \text{change-label-TL}(\texttt{fa};\ LT^{+}) = LT'^{+}}{\begin{array}{l}\text{resolvent}(T\ \texttt{:-}\ LT^{+}\ .\ ,\ T',\ T''\ \texttt{:-}\ LT^{*},\ LA * T',\ LT'^{*}\ .) \ = \\ T''\ \texttt{:-}\ LT^{*},\ LT'^{+},\ LT'^{*}\ .\ [\text{lc}\ S]\end{array}}$$

$$[\text{re2}]\quad \frac{\text{nonex}(LT^{*},\ LT'^{*}) = \text{false},\ \ \text{mgu-nv}(T,\ T') = S}{\text{resolvent}(T\ .\ ,\ T',\ T''\ \texttt{:-}\ LT^{*},\ LA * T',\ LT'^{*}\ .) \ =\ T''\ \texttt{:-}\ LT^{*},\ LT'^{*}\ .\ [\text{lc}\ S]}$$

$$[\text{re3}]\quad \frac{\text{nonex}(LT^{*},\ LT'^{*}) = \text{true},\ \ \text{mgu-nv}(T,\ T') = S}{\text{resolvent}(T\ .\ ,\ T',\ T''\ \texttt{:-}\ LT^{*},\ LA * T',\ LT'^{*}\ .) \ =\ T''\ .\ [\text{lc}\ S]}$$

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

7.2 Fold

In [Bru96] it is explained that the applicability conditions for a fold transformation step are very complex. Following Pettorossi and Proietti, the TransLog tool applies Tamaki & Sato folding. 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, the leftmost body goal of the folded clause is labeled *fold-allowing*.

Before looking at the Fold module in detail we first present the 'algorithm' that is applied.

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. The fold transformation step is defined as follows. For reference purposes the lines of the algorithm are numbered.

1. if there exists a clause D in P whose body is unifiable with T^{+}
2. then if the Tamaki & Sato conditions are satisfied
3. then apply a (reversible) folding step
4. else if there exists a clause D in P_{new} whose body is unifiable with T^{+}
5. then if the predicate of head(D) is defined in P
6. then if the Tamaki & Sato conditions are satisfied
7. then if the head predicate of C is an old predicate or the leftmost body goal of C is labeled *fold-allowing*
8. then apply the folding step
9. else report error: no old predicate or label *fold-allowing*
10. else report error: violation of T&S conditions
11. else report error: folding clause with undefined head predicate
12. 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 $C' = T$:- $T^*, \mathrm{head}(D)\theta, T^{*\prime}.$, with θ the mgu of T^+ and the body of the folding clause D.

We now turn to the actual specification of the fold operation in the module with the same name. The function *fold* has five arguments: the number of body goals to-be-folded, the first body goal of the sequence of body goals to-be-folded, the clause with the body terms to-be-folded, the (labeled) program P_{new}, containing the clauses resulting from previous define transformation steps, and the current (labeled) program P. The result of the *fold* function is a labeled program, containing the transformed program or an error message.

The various "prompt-functions" are applied in a dialogue in which the user is prompted to specify the number of body goals that have to be involved in the folding step. This dialogue is explained in Section 8.5, where the user interface is specified.

In the specification of the fold algorithm listed above, a number of local (help) functions are applied. The following table shows the correspondence between the line numbers of the fold algorithm and local functions. Help functions are explained afterwards.

Number	function(s)
1.	test on conditions in first equation of *revfold*
2.	test on conditions in first equation of *rfld1*
3.	*laFoldedClause*
4.	*revFold* or *rfld1* default equation, test on conditions in first equation of *newFold*
5.	test on conditions in first equation of *nfld1*
6.	test on conditions in first equation of *nfld2*
7.	test on conditions in first equation of *nfld3*
8.	*laFoldedClause*
9.	error message: *nfld3* default equation
10.	error message: *nfld2* default equation
11.	error message: *nfld1* default equation
12.	error message: *newFold* default equation

Several help-functions are applied. In the list below these functions are summarized.

- *selectBodyPart* extracts a given number (1–5) of body goals from the body of a given clause. The second argument of the function indicates the first body goal of the required sub-sequence.
- *foldingClause* tries to find a folding clause in a program P, given a sequence of body goals from the folded clause. In case no folding clause is found, the pseudo-clause `'fail'.` is returned.
- *intVarSet* returns a term set with the *internal* variables of a clause: variables that occur in the body, but not in the head.
- *restrict* filters out of a sequence of substitutions those substitutions that a concerned with a variable that is a member of a given term set.
- *image* groups the 'images' of a substitution sequence together in a term set.
- *nonRecursive* is a boolean function that checks if the head predicate of a given clause occurs only once in a given program (that is, in the given clause, which is part of the program).
- *onceAndNonr* is a help function for the previous function. This function receives two input arguments: a simplified term and a (stripped) program, equal to the original program without the clause in which the original predicate is defined.
- *laFoldedClause* produces the result of a successfull folding step: a clause in which a sequence of body goals is replaced by a single head of a folding clause. The label of the new body goal is set to 'fa', fold-allowing.

- *finalConditions* implements the 'final test': is the head predicate of the folded clause an old predicate or is the leftmost body goal labeled 'fold-allowing'. This function has two arguments: the folded clause and P_{new}. It is supposed that a predicate that is not defined in P_{new} is an old predicate.

module Fold
imports VariantClause
exports
context-free syntax

fold "(" TERM "," TERM "," CLAUSE "," LAPROGRAM "," LAPROGRAM ")"	→ LAPROGRAM
foldPrompt "(" ")"	→ PROGRAM
fprompt1	→ CLAUSE
fprompt2	→ CLAUSE
fprompt3	→ CLAUSE

hiddens
context-free syntax

revFold "(" TERMLIST ";" CLAUSE ";" LAPROGRAM ";" LAPROGRAM ")"	→ LAPROGRAM
rfld1 "(" TERMLIST ";" CLAUSE ";" CLAUSE ";" LAPROGRAM ";" LAPROGRAM ")"	→ LAPROGRAM
newFold "(" TERMLIST ";" CLAUSE ";" LAPROGRAM ";" LAPROGRAM ")"	→ LAPROGRAM
nfld1 "(" TERMLIST ";" CLAUSE ";" CLAUSE ";" LAPROGRAM ";" LAPROGRAM ")"	→ LAPROGRAM
nfld2 "(" TERMLIST ";" CLAUSE ";" CLAUSE ";" LAPROGRAM ";" LAPROGRAM ")"	→ LAPROGRAM
nfld3 "(" TERMLIST ";" CLAUSE ";" CLAUSE ";" LAPROGRAM ";" LAPROGRAM ";" SUBS ")"	→ LAPROGRAM
selectBodyPart "(" TERM ";" TERM ";" CLAUSE ")"	→ TERMLIST
foldingClause "(" TERMLIST ";" PROGRAM ")"	→ CLAUSE
intVarSet "(" CLAUSE ")"	→ TERM-SET
restrict "(" SUBS ";" TERM-SET ")"	→ SUBS
image "(" SUBS ")"	→ TERM-SET
nonRecursive "(" CLAUSE ";" LAPROGRAM ")"	→ BOOL
onceAndNonr "(" TERM ";" PROGRAM ")"	→ BOOL
laFoldedClause "(" TERMLIST ";" TERM ";" LACLAUSE ")"	→ LACLAUSE
finalConditions "(" LACLAUSE ";" LAPROGRAM ")"	→ BOOL

equations

[f1] $$\frac{\text{selectBodyPart}(T;\ T';\ C) = T^{+}}{\text{fold}(T,\ T',\ C,\ LP,\ LP')\ =\ \text{revFold}(T^{+};\ C;\ LP;\ LP')}$$

[rf1] $$\frac{\begin{array}{c}\text{strip-label}(LP') = P,\ \text{removecp}(C,\ P) = P',\\ \text{foldingClause}(T^{+};\ P') = C',\ \ C' \neq \texttt{'fail'}\ .,\\ \text{variantC}(C',\ C) = C''\end{array}}{\text{revFold}(T^{+};\ C;\ LP;\ LP')\ =\ \text{rfld1}(T^{+};\ C;\ C'';\ LP;\ LP')}$$

[rf] revFold(T^+; C; LP; LP') = newFold(T^+; C; LP; LP') **otherwise**

[r1f1] $$\frac{\text{mgu-tl}(T'^+;\ T^+) = S,\ S \neq \text{fail},\ \text{intVarSet}(T' \text{ :- } T'^+ .) = TS,\ \text{restrict}(S;\ TS) = S',\ \text{isVarRenaming}(S') = \text{true},\ \text{image}(S') = TS',\ TS' \cap \text{varset-T}(T,\ T^*,\ T'^*,\ T'\ [\text{t}\ S]) = \{\},\ \text{nonRecursive}(T' \text{ :- } T'^+ .\ ;\ LP') = \text{true},\ \text{la-version-C}(T \text{ :- } T^*,\ T^+,\ T'^* .\ ;\ LP') = LC,\ \text{laFoldedClause}(T^+;\ T'\ [\text{t}\ S];\ LC) = LC'}{\text{rfld1}(T^+;\ T \text{ :- } T^*,\ T^+,\ T'^* .\ ;\ T' \text{ :- } T'^+ .\ ;\ LP;\ LP') \ =\ \text{la-replacecp}(LP',\ LC,\ LC')}$$

[r1f] rfld1(T^+; C; C'; LP; LP') = newFold(T^+; C; LP; LP') **otherwise**

[nf1] $$\frac{\text{strip-label}(LP) = P,\ \text{foldingClause}(T^+;\ P) = C',\ C' \neq \texttt{'fail'} .,\ \text{variantC}(C',\ C) = C''}{\text{newFold}(T^+;\ C;\ LP;\ LP') \ =\ \text{nfld1}(T^+;\ C;\ C'';\ LP;\ LP')}$$

[nf] newFold(T^+; C; LP; LP') = set-label(`el`; standard ferrorm) **otherwise**

[n1f1] $$\frac{\text{isDefined}(T;\ LP') = \text{true}}{\text{nfld1}(T^+;\ C;\ T \text{ :- } T'^+ .\ ;\ LP;\ LP') \ =\ \text{nfld2}(T^+;\ C;\ T \text{ :- } T'^+ .\ ;\ LP;\ LP')}$$

[n1f] nfld1(T^+; C; C'; LP; LP') = set-label(`el`; standard f2errorm) **otherwise**

[n2f1] $$\frac{\text{mgu-tl}(T'^+;\ T^+) = S,\ S \neq \text{fail},\ \text{intVarSet}(T' \text{ :- } T'^+ .) = TS,\ \text{restrict}(S;\ TS) = S',\ \text{isVarRenaming}(S') = \text{true},\ \text{image}(S') = TS',\ TS' \cap \text{varset-T}(T,\ T^*,\ T'^*,\ T'\ [\text{t}\ S]) = \{\},\ \text{nonRecursive}(T' \text{ :- } T'^+ .\ ;\ LP) = \text{true}}{\begin{array}{c}\text{nfld2}(T^+;\ T \text{ :- } T^*,\ T^+,\ T'^* .\ ;\ T' \text{ :- } T'^+ .\ ;\ LP;\ LP') \ = \\ \text{nfld3}(T^+;\ T \text{ :- } T^*,\ T^+,\ T'^* .\ ;\ T' \text{ :- } T'^+ .\ ;\ LP;\ LP';\ S)\end{array}}$$

[n2f] nfld2(T^+; C; C'; LP; LP') = set-label(`el`; standard f3errorm) **otherwise**

[n3f1] $$\frac{\text{la-version-C}(T \text{ :- } T^*,\ T^+,\ T'^* .\ ;\ LP') = LC,\ \text{finalConditions}(LC;\ LP) = \text{true},\ \text{laFoldedClause}(T^+;\ T'\ [\text{t}\ S];\ LC) = LC'}{\text{nfld3}(T^+;\ T \text{ :- } T^*,\ T^+,\ T'^* .\ ;\ T' \text{ :- } T'^+ .\ ;\ LP;\ LP';\ S) \ =\ \text{la-replacecp}(LP',\ LC,\ LC')}$$

[n3f] nfld3(T^+; C; C'; LP; LP'; S) = set-label(`el`; standard f4errorm) **otherwise**

[sbp1] selectBodyPart(1; T; T' :- T^*, T, T'^* .) = T
[sbp2] selectBodyPart(2; T; T' :- T^*, T, T'', T'^* .) = T, T''
[sbp3] selectBodyPart(3; T; T' :- T^*, T, T'', T''', T'^* .) = T, T'', T'''
[sbp4] selectBodyPart(4; T; T' :- T^*, T, T'', T''', T'''', T'^* .) = T, T'', T''', T''''

[sbp5] selectBodyPart(5; T; T' :- T^*, T, T'', T''', T'''', T''''', T'^* .)
= T, T'', T''', T'''', T'''''

[fc1] $$\frac{\text{mgu-tl}(T^+;\ T'^+) = S,\ \ S \neq \text{fail}}{\text{foldingClause}(T^+;\ C^*\ T \text{ :- } T'^+\ .\ C'^*)\ =\ T \text{ :- } T'^+\ .}$$

[fc] $\text{foldingClause}(T^+;\ P) = \texttt{'fail'}\ .$ **otherwise**

[iv1] $\text{intVarSet}(T \text{ :- } T^+\ .) = \text{varset-T}(T^+)\ \setminus\ \text{varset-T}(T)$

[re1] $\text{restrict}(\{\};\ TS) = \{\}$

[re2] $$\frac{V \in\ TS = \text{true},\ \ \text{restrict}(\{S^*\};\ TS) = S'}{\text{restrict}(\{(V \mapsto T),\ S^*\};\ TS)\ =\ \{(V \mapsto T)\} \oplus S'}$$

[re] $\text{restrict}(\{(V \mapsto T),\ S^*\};\ TS) = \text{restrict}(\{S^*\};\ TS)$ **otherwise**

[im1] $\text{image}(\{\}) = \{\}$
[im2] $\text{image}(\{(V \mapsto T),\ S^*\}) = \{T\} \cup \text{image}(\{S^*\})$

[nr1] $$\frac{\begin{array}{c}\text{simplify}(T) = T'',\ \ \text{simplify}(T') = T'',\\ \text{strip-label}(LC^*\ LC'^*) = P,\ \ \text{onceAndNonr}(T'';\ P) = \text{true}\end{array}}{\text{nonRecursive}(T \text{ :- } T^+\ .\ ;\ LC^*\ T' \text{ :- } LT^+\ .\ LC'^*)\ =\ \text{true}}$$

[nr] $\text{nonRecursive}(C;\ LP) = \text{false}$ **otherwise**

[onr1] $$\frac{\text{simplify}(T') = T}{\text{onceAndNonr}(T;\ C^*\ T'\ .\ C'^*)\ =\ \text{false}}$$

[onr2] $$\frac{\text{simplify}(T') = T}{\text{onceAndNonr}(T;\ C^*\ T' \text{ :- } T^+\ .\ C'^*)\ =\ \text{false}}$$

[onr3] $$\frac{\text{simplify}(T') = T}{\text{onceAndNonr}(T;\ C^*\ T'' \text{ :- } T^*,\ T',\ T'^*\ .\ C'^*)\ =\ \text{false}}$$

[onr] $\text{onceAndNonr}(T;\ P) = \text{true}$ **otherwise**

[laf1] $\text{laFoldedClause}(T;\ T';\ T'' \text{ :- } LT^*,\ LA * T,\ LT'^*\ .)$
$= T'' \text{ :- } LT^*, \texttt{fa} * T',\ LT'^*\ .$

[laf2] $\text{laFoldedClause}(T,\ T^+;\ T';\ T'' \text{ :- } LT^*,\ LA * T,\ LT'^*\ .)$
$= \text{laFoldedClause}(T^+;\ T';\ T'' \text{ :- } LT^*,\ LT'^*\ .)$

[fcd1] $$\frac{\text{isDefined}(T;\ LP) = \text{false}}{\text{finalConditions}(T \text{ :- } LT^+\ .\ ;\ LP)\ =\ \text{true}}$$

[fcd2] $$\frac{\text{isDefined}(T;\ LP) = \text{true}}{\text{finalConditions}(T \texttt{ :- fa } * T',\ LT^*\ .\ ;\ LP)\ =\ \text{true}}$$

[fcd] finalConditions(LC; LP) = false **otherwise**

[fp1] fprompt1 = `'Give the number (1-5) of atoms to be folded'.`
[fp2] fprompt2 = `'Type the number on a new line, followed by a dot'.`
[fp3] fprompt3 = `'Focus on the number and select the button Continue'.`
[fp4] foldPrompt ( ) = fprompt1 fprompt2 fprompt3

7.3 Define

A new clause (predicate definition) can be added to a program by the Define transformation step. The function *Define* has three arguments: the new clause, the current program and the program P_{new}, consisting of previously defined new clauses. 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 be defined in the current program or in P_{new}. Finally, the new clause has to be non-recursive. Before the new clause is added to the program its body goals are labeled *fold-forbidden* (ff).

The function *newDefine* adds a new clause to P_{new}. No conditions are checked. The structure of the user interface guarantees that this addition will be harmless. See for details Section 8.5.

The various prompt-functions are used while prompting the user to enter the definition of a new clause. The local function *nonRec* checks if a clause is non-recursive: the head predicate may not occur in the body of the clause.

module Define
imports LabeledProgram
exports
context-free syntax
define "(" CLAUSE "," LAPROGRAM "," LAPROGRAM ")" → LAPROGRAM
newDefine "(" CLAUSE "," LAPROGRAM ")" → LAPROGRAM
definePrompt "(" ")" → PROGRAM
dprompt1 → CLAUSE
dprompt2 → CLAUSE
context-free syntax
nonRec "(" CLAUSE ")" → BOOL

equations

[de1] $$\frac{\text{syntaxCheck-C}(C) = \text{false}}{\text{define}(C,\ LP,\ LP')\ =\ \text{set-label}(\texttt{el};\ \text{standard nerrorm})}$$

[de2] $$\frac{\begin{array}{c}\text{syntaxCheck-C}(C) = \text{true},\ \text{head}(C) = T,\ \text{isDefined}(T;\ LP') = \text{false},\\ \text{isDefined}(T;\ LP) = \text{false},\ \text{nonRec}(C) = \text{true},\ \text{set-label-C}(\texttt{ff};\ C) = LC\end{array}}{\text{define}(C,\ LP,\ LP')\ =\ \text{la-addcp}(LC,\ LP)}$$

[de] define(C, LP, LP') = set-label(`el`; standard rerrorm) **otherwise**

$$[\text{nr1}]\quad \frac{\text{simplify}(T) = T'',\ \text{simplify}(T') = T''}{\text{nonRec}(T \text{ :- } T^*,\ T',\ T''^*\ .)\ =\ \text{false}}$$

[nr] nonRec(C) = true **otherwise**

$$[\text{nd1}]\quad \frac{\text{set-label-C}(\texttt{ff};\ C) = LC}{\text{newDefine}(C,\ LP)\ =\ \text{la-addcp}(LC,\ LP)}$$

[dp1] dprompt1 = `'Define a new clause on a new line'` .
[dp2] dprompt2 = `'Focus on it and select the button Continue'` .
[dp3] definePrompt ( ) = dprompt1 dprompt2

7.4 AssocPermutation

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 incorporated in the TransLog tool. It is supposed that `p` is a predicate that represents an associative operation, defined with recursion on its first argument (like the well-known `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 '+,+,-' 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 goal that is involved in this transformation, the clause that holds this body goal and the current program. The boolean functions *apCheck-i* and *apCheck-o* check if the arguments are in the right position and if the crucial variables are local to the two goals involved. The help-functions *aph-i* and *aph-o* do the job for the two modes, they produce a new clause. An error message is returned if the required permutation is not allowed.

module AssocPermutation
imports LabeledProgram
exports
  context-free syntax
    assperm "(" TERM "," CLAUSE "," LAPROGRAM ")" → LAPROGRAM
hiddens
  context-free syntax
    apCheck-i "(" TERM "," CLAUSE ")" → BOOL
    apCheck-o "(" TERM "," CLAUSE ")" → BOOL
    aph-i "(" TERM "," LACLAUSE ")" → LACLAUSE
    aph-o "(" TERM "," LACLAUSE ")" → LACLAUSE

equations

$$[\text{ap1}]\quad \frac{\text{apCheck-i}(T,\ C) = \text{true},\ \text{la-version-C}(C;\ LP) = LC,\ \text{aph-i}(T,\ LC) = LC'}{\text{assperm}(T,\ C,\ LP)\ =\ \text{la-replacecp}(LP,\ LC,\ LC')}$$

$$[\text{ap2}]\quad \frac{\text{apCheck-o}(T,\ C) = \text{true},\ \text{la-version-C}(C;\ LP) = LC,\ \text{aph-o}(T,\ LC) = LC'}{\text{assperm}(T,\ C,\ LP)\ =\ \text{la-replacecp}(LP,\ LC,\ LC')}$$

[ap] assperm(T, C, LP) = set-label(`nf`; standard aerrorm) **otherwise**

[aci1] $$\frac{V \in \text{varset-T}(T,\ T^*,\ T'^*) = \text{false}}{\text{apCheck-i}(A(T',\ T'',\ V),\ T \text{ :- } T^*,\ A(T',\ T'',\ V),\ A(V,\ T''',\ V'),\ T'^*\ .)\ =\ \text{true}}$$

[aci] $\text{apCheck-i}(T,\ C) = \text{false}$ **otherwise**

[ahi1] $\text{aph-i}(A(T',\ T'',\ V),\ T \text{ :- } LT^*,\ LA * A(T',\ T'',\ V),\ LA' * A(V,\ T''',\ V'),\ LT'^*\ .)$
$= T \text{ :- } LT^*,\ LA * A(T'',\ T''',\ V),\ LA' * A(T',\ V,\ V'),\ LT'^*\ .$

[aco1] $$\frac{V \in \text{varset-T}(T',\ T^*,\ T'^*) = \text{false}}{\text{apCheck-o}(A(V,\ V''',\ T),\ T' \text{ :- } T^*,\ A(V,\ V''',\ T),\ A(V',\ V'',\ V),\ T'^*\ .)\ =\ \text{true}}$$

[aco] $\text{apCheck-o}(T,\ C) = \text{false}$ **otherwise**

[aho1] $\text{aph-o}(A(V,\ V''',\ T),\ T' \text{ :- } LT^*,\ LA * A(V,\ V''',\ T),\ LA' * A(V',\ V'',\ V),\ LT'^*\ .)$
$= T' \text{ :- } LT^*,\ LA * A(V',\ V,\ T),\ LA' * A(V'',\ V''',\ V),\ LT'^*\ .$

7.5 GoalSwitch

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 cases. Therefore, we have incorporated this transformation step in the TransLog tool. Furthermore, we do not restrict this transformation by any applicability condition. This makes the specification of the *switch* function very simple. The input arguments of this function represent the body goal that has to be switched with its right-hand neighbor, the clause that holds the body goal and the current program. The local help function *swTerms* is only used for reasons of clarity, the operation can also be specified in one single (but long) equation.

module GoalSwitch
imports LabeledProgram
exports
context-free syntax
switch "(" TERM "," CLAUSE "," LAPROGRAM ")" → LAPROGRAM
hiddens
context-free syntax
swTerms "(" TERM "," LACLAUSE ")" → LACLAUSE

equations

[gs1] $$\frac{\text{la-version-C}(C;\ LP) = LC,\ \ \text{swTerms}(T,\ LC) = LC'}{\text{switch}(T,\ C,\ LP)\ =\ \text{la-replacecp}(LP,\ LC,\ LC')}$$

[st1] $\text{swTerms}(T,\ T'' \text{ :- } LT^*,\ LA * T,\ LA' * T',\ LT'^*\ .)$
$= T'' \text{ :- } LT^*,\ LA' * T',\ LA * T,\ LT'^*\ .$

7.6 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 clauses that contain the definition of relevant predicates. Therefore, they safely can be removed. We give the following definition of this transformation step. Let P be the current program. Let T be the selected head term. Let TS be the set of goals TS that can be reached from the clause with head T. (The set TS only contains predicates with their arity.) Then, select from P all the clauses of which the head (predicate + arity) is an element of TS. The resulting sequence of clauses constitutes the program that is the result of the CleanUp transformation step.

The transformation is specified in the module CleanUp. 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 *selectClauses*.

With respect to the goal set, two goals 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*, specified in the module PrologFunctions.

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 goals 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 *selectClauses* 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 Substitution
exports
  context-free syntax
    cleanUp "(" TERM "," LAPROGRAM ")" → LAPROGRAM
hiddens
  context-free syntax
    goalset "(" TERMLIST "," PROGRAM "," PROGRAM ")" → TERM-SET
    selectClauses "(" TERM-SET "," LAPROGRAM ")" → LAPROGRAM

equations

[cu1] $$\frac{\text{strip-label}(LP) = P,\ \text{goalset}(T, P, P) = TS}{\text{cleanUp}(T, LP) = \text{selectClauses}(TS, LP)}$$

[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] $\text{goalset}(T^+,\ T'^+,\ P,\ P) = \text{goalset}(T^+,\ P,\ P) \cup \text{goalset}(T'^+,\ P,\ P)$
[gs] $\text{goalset}(T,\ C\ C^*,\ P) \quad = \text{goalset}(T,\ C^*,\ P) \quad$ **otherwise**

[sc1] $\text{selectClauses}(TS,\ ) =$

[sc2] $$\frac{\text{simplify}(\text{lhead}(LC)) \in TS = \text{true},\ \text{selectClauses}(TS,\ LC^*) = LP}{\text{selectClauses}(TS,\ LC\ LC^*)\ =\ \text{la-addcp}(LC,\ LP)}$$

[sc3] $$\frac{\text{simplify}(\text{lhead}(LC)) \in TS = \text{false}}{\text{selectClauses}(TS,\ LC\ LC^*)\ =\ \text{selectClauses}(TS,\ LC^*)}$$

7.7 VariantClause

In the unfold and fold transformation steps a variant of the (un)folding clause is needed in order to avoid unwanted identification of variables with accidentally the same name. This variant clause is created in the module with the same name by the function *variant* that takes both the (un)folding clause and the (un)folded clause as input arguments.

We will shortly explain the single equation for this function. With the function *lvarset* a set is created with the variables from a (labeled) clause. With the function *varsub* a sequence of substitutions is deduced from two variable sets. For each variable V from the clause LC that is also present in the clause LC', a substitution is added. In this substitution the variable V is replaced by a new variable that is not present in the variable set of LC', nor in the variable set of LC. A new variable name is created by repeated addition of the character sequence `_n` to the old variable name until a unique variable name is derived. The derived substitutions are applied to the clause LC. The resulting variant clause has no variables in common with the clause LC'.

module VariantClause
imports Unification
exports
  context-free syntax
    variant "(" LACLAUSE "," LACLAUSE ")" → LACLAUSE
    variantC "(" CLAUSE "," CLAUSE ")" → CLAUSE
hiddens
  context-free syntax
    newVar "(" VARIABLE "," TERM-SET ")" → TERM
    prime "(" VARIABLE ")" → VARIABLE
    varSub "(" TERM-SET "," TERM-SET ")" → SUBS
  variables
    "c" → CHAR+

equations

[va1] $$\frac{\text{lvarset}(LC) = TS,\ \text{lvarset}(LC') = TS'}{\text{variant}(LC,\ LC')\ =\ LC\,[\text{lc varSub}(TS \cap TS',\ TS \cup TS')]}$$

[vc1] $$\frac{\text{varset}(C) = TS,\ \text{varset}(C') = TS'}{\text{variantC}(C,\ C')\ =\ C\,[\text{c varSub}(TS \cap TS',\ TS \cup TS')]}$$

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

$$[\text{vs2}]\quad \frac{\text{newVar}(V,\ TS) = V'}{\text{varSub}(\{V,\ T^*\},\ TS)\ =\ \{(V \mapsto V')\} \otimes \text{varSub}(\{T^*\},\ TS \cup \{V'\})}$$

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

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

$$[\text{pr1}]\quad \text{prime}(\text{variable}(c)) = \text{variable}(c\ \texttt{"_"}\ \texttt{"n"})$$

8 User interface modules

Five modules are related with the user interface of the TransLog tool. The module ErrorMessages contains the messages that are displayed when a transformation step is not allowed. The module TransLog groups together all the transformation steps. The module Log defines a number of functions which are related to the storage of a transformation sequence. The module ButtonConditions contains the conditions which determine if a button in the TransLog window is enabled or disabled. The last module, TransLog.seal, is not an 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.

8.1 ErrorMessages

When the applicability conditions that are related to a specific transformation step fail, an error message is displayed instead of a transformed program. The choice has been made to give error messages the syntax of a Prolog program (clause). This makes it easy to present an error message in a program window.

In the module a number of error messages is defined as unit clauses: a quoted atom, followed by a full stop. Each specific error message is preceded by the 'standard' message *The selected transformation is not allowed.* This part of an error message is used in the boolean function *isErrorMessage*. This function determines whether or not a given program is an error message (consists of a sequence of unit clauses with quoted atoms, with the first clause the 'standard' error message). This function is used in the module Log.

module ErrorMessages
imports PrologFunctions
exports
  context-free syntax
    standard → CLAUSE
    derrorm → CLAUSE
    verrorm → CLAUSE
    ferrorm → CLAUSE
    f2errorm → CLAUSE
    f3errorm → CLAUSE
    f4errorm → CLAUSE
    nerrorm → CLAUSE
    rerrorm → CLAUSE

aerrorm → CLAUSE

isErrorMessage "(" PROGRAM ")" → BOOL

equations

[st1] standard = `'The selected transformation is not allowed'`.

[de1] derrorm = `'No deterministic unfolding of non-left body term possible'`.

[ve1] verrorm = `'No non-left-propagating unfolding of non-left body term possible'`.

[fe1] ferrorm = `'No folding clause present in current program or Pnew'`.

[fe2] f2errorm = `'No definition of head of folding clause in current program'`.

[fe3] f3errorm = `'T&S conditions fail on folding clause from Pnew'`.

[fe4] f4errorm = `'No old head predicate or fold-allowing leftmost body goal'`.

[ne1] nerrorm = `'Syntax error in new clause'`.

[re1] rerrorm = `'New clause is not correct'`.

[ae1] aerrorm = `'No proper permutation of arguments possible'`.

[ie1] $$\frac{C = \text{standard}}{\text{isErrorMessage}(C\ C^*)\ =\ \text{true}}$$

[ie] isErrorMessage(P) = false **otherwise**

8.2 TransLog

The TransLog module is very simple: it serves as a collecting module for the various transformation steps. 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 Define AssocPermutation GoalSwitch CleanUp

8.3 Log

The TransLog system offers navigation functions 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. Furthermore, a sequence of clauses defining new predicates (clauses resulting from the Define transformation step) has to be stored. We will call this sequence P_{new}. Because we can go back in a transformation sequence, P_{new} is related to the sequence. E.g. when a Previous command is issued after a Define transformation step, the newly defined clause has to be removed from P_{new}. Therefore we have chosen to store program*duo*'s instead of single programs.

From the sections on the Fold transformation step it may be clear that *labeled* programs have to be stored: each body goal in a clause is labeled *fold-allowing* or *fold-forbidden*. Before displaying a program in a TransLog window, the labels are stripped from the body goals.

Like other algebraic specification formalisms, the ASF+SDF system does not offer 'state variables'. So, in an ongoing transformation sequence a (duo)program cannot be stored in a variable. We have chosen to store a program sequence in two separate ASF+SDF windows, named LogA and LogB. 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. The window LogA contains the 'past and presence' of the sequence: the programs $P_0, P_1, \ldots, P_i$. The window LogB contains the 'future' of the sequence: the programs $P_{i+1}, \ldots, P_n$. In case $i = n$, the LogB window is empty.

A program in a transformation sequence is stored together with the actual contents of P_{new} in a construct of the sort DUOPROGRAM. The sort DUOLIST represents a sequence of the sort DUOPROGRAM. The constant *l-dummy* represents a dummy duoprogram, which can be used for initialization of a sequence of duoprograms. The various append functions append a duoprogram to a list. The function *l-appendP* appends a (transformed) program together with the last P_{new} in a list to the list. So, P_{new} is not changed. The function *l-appendN* appends a (transformed) program together with a new version of P_{new} to a list. This function is used after a Define transformation step. The function *l-appendA* simply appends a duoprogram to a list. The function *l-appendB* only appends a duoprogram to a list if the second program of the duo is not an error message. The need for this function is explained in Section 8.5. The function *l-delete* removes the last duoprogram from a list. The function *l-initialize* initializes a list of duoprograms with a dummy value, followed by a duoprogram that consists of an empty program P_{new} and the initial program P_0 with all labels set to *fold-allowing*. The functions *l-lastP* and *l-lastN* return the last program or the last P_{new} from a list.

The three *l-fold* functions are used during the execution of a Fold transformation step. The function *l-fold-store* stores the first body goal to-be-folded and the folded clause temporarily in an artificial program. As we will see later on, this program is stored in the LogB window. The label *ff* for the folded clause has no meaning, it is added because only labeled programs can be stored in the window. The functions *l-fold-term* and *l-fold-clause* retrieve the term and clause from the artificial program.

module Log
imports LabeledProgram
exports
  sorts DUOPROGRAM DUOLIST
  context-free syntax
    LAPROGRAM "∘" LAPROGRAM → DUOPROGRAM
    {DUOPROGRAM "⋄"}+ → DUOLIST

    l-dummy "(" ")" → DUOPROGRAM
    l-appendP "(" DUOLIST "," LAPROGRAM ")" → DUOLIST
    l-appendN "(" DUOLIST "," LAPROGRAM ","

LAPROGRAM ")" → DUOLIST
l-appendA "(" DUOLIST "," DUOPROGRAM ")" → DUOLIST
l-appendB "(" DUOLIST "," DUOPROGRAM ")" → DUOLIST
l-delete "(" DUOLIST ")" → DUOLIST
l-initialize "(" PROGRAM ")" → DUOLIST
l-lastP "(" DUOLIST ")" → LAPROGRAM
l-lastN "(" DUOLIST ")" → LAPROGRAM
l-fold-store "(" TERM "," CLAUSE ")" → DUOPROGRAM
l-fold-term "(" DUOLIST ")" → TERM
l-fold-clause "(" DUOLIST ")" → CLAUSE

"(" DUOPROGRAM ")" → DUOPROGRAM {**bracket**}
"(" DUOLIST ")" → DUOLIST {**bracket**}

variables

$LP\ [']*$ → LAPROGRAM
$DP\ [']*$ → DUOPROGRAM
$DP\ [']*$"*" → {DUOPROGRAM "$\diamond$"}$*$
$DP\ [']*$"+" → {DUOPROGRAM "$\diamond$"}+

equations

[apP1] $$\frac{DP^* = DP'^* \diamond LP \circ LP'}{\text{l-appendP}(DP^*,\ LP'') \ =\ DP^* \diamond LP \circ LP''}$$

[apN1] $$\text{l-appendN}(DP^*,\ LP,\ LP') = DP^* \diamond LP \circ LP'$$

[apA1] $$\text{l-appendA}(DP^*,\ DP) = DP^* \diamond DP$$

[apB1] $$\frac{\text{strip-label}(LP') = P,\ \text{isErrorMessage}(P) = \text{false}}{\text{l-appendB}(DP^*,\ LP \circ LP') \ =\ DP^* \diamond LP \circ LP'}$$

[apB2] $$\frac{\text{strip-label}(LP') = P,\ \text{isErrorMessage}(P) = \text{true}}{\text{l-appendB}(DP^*,\ LP \circ LP') \ =\ DP^*}$$

[de1] $$\text{l-delete}(DP^* \diamond DP) = DP^*$$

[in1] $$\frac{\text{set-label}(\texttt{fa};\ P) = LP}{\text{l-initialize}(P) \ =\ \text{l-dummy}\ (\ ) \diamond\ \circ\ LP}$$

[la1] $$\text{l-lastP}(DP^* \diamond LP \circ LP') = LP'$$

[lan1] $$\text{l-lastN}(DP^* \diamond LP \circ LP') = LP$$

[lfs1] $$\text{l-fold-store}(T,\ C) = T\ .\ \circ\ \text{set-label-C}(\texttt{ff};\ C)$$

[lft1] $$\frac{LP = T\ .}{\text{l-fold-term}(LP \circ LP') \ =\ T}$$

[lft] $$\text{l-fold-term}(DP^*) = \texttt{'error'} \quad \textbf{otherwise}$$

[lfc1] $$\text{l-fold-clause}(LP \circ LC) = \text{strip-label-C}(LC)$$
[lfc] $$\text{l-fold-clause}(DP^*) \quad = \texttt{'error'}\ .\quad \textbf{otherwise}$$

8.4 ButtonConditions

A part of the TransLog user interface is built on buttons, one for each transformation step. Buttons are also used for navigation functions on a program sequence and initialization of the storage of program sequences in windows. 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-...*.

It is possible to test the complete set of applicability functions for a transformation step A in the function *e-A*. In that case a button is ony 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 made 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 make 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. In the case 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 window –when the system re-evaluates the button conditions– the user will have to wait. We have chosen to avoid this delay.

We now turn to the functions of the module ButtonConditions. The function *e-unfold* yields *true* when the goal to-be-unfolded is one of the goals in the body of a clause and the goal is not a variable. The same holds for the function *e-fold*.

The function *e-define* yields *true* when its argument is a program, not a message. This is checked in the boolean function *noMessage*, explained below.

The function *e-asperm* yields *true* when its second argument (a clause) holds two consecutive goals with the same predicate and the same arity (3). The first argument of the function indicates the first body goal.

The function *e-switch* only checks if the selected goal resides in the body of a clause and if there exists a right neighbor goal 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 condition *isInteger* in the first equation is needed because of the dialogue that is used in the fold transformation step. This dialogue is explained in Section 8.5.

The functions *e-foldcon* and *e-defcon* determine whether or not the Continue button is enabled. The Fold step and the Define step require a simple dialogue with the user. Both dialogs have the format –question (from TransLog) → answer (from the user) → transformed program– . When the user has entered an answer (and selected an appropriate term), only the Continue button will be enabled. In case of the Fold transformation step the function *e-foldcon* checks if the required number of body goals to-be-folded is present in the folded clause. This number is restricted to the interval [1–5]. The local function *bodyTermsPresent* does the work. In case of the Define transformation step the function *e-defcon* checks if the TransLog window contains the prompt related with this step, followed by a newly defined clause.

The equations for the functions *e-previous* and *e-next* are obvious: they yield *true* when there exists a previous program or a next program in a sequence of programs, otherwise they yield *false*. The Previous button is also not enabled when the first part of a dialogue (the Fold prompt or the Define prompt) is in the TransLog window.

The function *e-init* yields *true* when its argument (a program) is not an error message or a dialogue message.

The local function *noMessage* checks if its argument (a program) is an error message or a dialogue message. Only the first clause of a program is inspected.

module ButtonConditions
imports TransLog Log
exports
context-free syntax
e-unfold "(" TERM "," CLAUSE ")" → BOOL
e-fold "(" TERM "," CLAUSE ")" → BOOL
e-define "(" PROGRAM "," PROGRAM ")" → BOOL
e-asperm "(" TERM "," CLAUSE ")" → BOOL
e-switch "(" TERM "," CLAUSE ")" → BOOL
e-clean "(" TERM "," CLAUSE ")" → BOOL
e-foldcon "(" TERM "," DUOLIST ")" → BOOL
e-defcon "(" PROGRAM ")" → BOOL
e-previous "(" PROGRAM "," DUOLIST ")" → BOOL
e-next "(" DUOLIST ")" → BOOL
e-init "(" PROGRAM ")" → BOOL
hiddens
context-free syntax
bodyTermsPresent "(" TERM "," TERM "," CLAUSE ")" → BOOL
noMessage "(" PROGRAM ")" → BOOL

equations

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

[eu] e-unfold(T, C) = false **otherwise**

[ef1] $$\frac{\text{isBodyTerm}(T,\ C) = \text{true}}{\text{e-fold}(T,\ C)\ =\ \text{not}(\text{isVar}(T))}$$

[ef] e-fold(T, C) = false **otherwise**

[ed1] e-define(P, P) = noMessage(P)

[ea1] e-asperm($A(T,\ T',\ T'')$, T''' :- T^*, $A(T,\ T',\ T'')$, $A(T'''',\ T''''',\ T''''''),\ T'^*$.) = true

[ea] e-asperm(T, C) = false **otherwise**

[es1] e-switch(T, T'' :- T^*, T, T', T'^* .) = true
[es] e-switch(T, C) = false **otherwise**

[ec1] $$\frac{\text{isInteger}(T) = \text{false}}{\text{e-clean}(T,\ T\ .)\ =\ \text{true}}$$

[ec2] e-clean(T, T :- T^+ .) = true
[ec] e-clean(T, C) = false **otherwise**

$$[\text{efc1}]\ \frac{\text{l-fold-term}(DP^*) = T',\ \text{l-fold-clause}(DP^*) = C}{\text{e-foldcon}(T,\ DP^*)\ =\ \text{bodyTermsPresent}(T,\ T',\ C)}$$

[efc] e-foldcon(T, DP^*) = false **otherwise**

$$[\text{edc1}]\ \frac{C = \text{dprompt1},\ \ C' = \text{dprompt2}}{\text{e-defcon}(C\ C'\ C'')\ =\ \text{true}}$$

[edc] e-defcon(C^*) = false **otherwise**

$$[\text{ep1}]\ \frac{P \neq \text{foldPrompt ( )},\ \ P \neq \text{definePrompt ( )},\ \ DP^+ = DP' \diamond DP'^+}{\text{e-previous}(P,\ DP \diamond DP^+)\ =\ \text{true}}$$

[ep] e-previous(P, DP^+) = false **otherwise**

[en1] e-next(l-dummy ( ) $\diamond$ DP^+) = true
[en] e-next(DP^+) = false **otherwise**

[ei1] e-init(P) = noMessage(P)

[bp1] bodyTermsPresent(1, T, T' :- T^*, T, T'^* .) = true
[bp2] bodyTermsPresent(2, T, T' :- T^*, T, T'', T'^* .) = true
[bp3] bodyTermsPresent(3, T, T' :- T^*, T, T'', T''', T'^* .) = true
[bp4] bodyTermsPresent(4, T, T' :- T^*, T, T'', T''', T'''', T'^* .) = true
[bp5] bodyTermsPresent(5, T, T' :- T^*, T, T'', T''', T'''', T''''', T'^* .) = true
[bp] bodyTermsPresent(T, T', C) = false **otherwise**

$$[\text{nm1}]\ \frac{C = \text{standard}}{\text{noMessage}(C\ C^*)\ =\ \text{false}}$$

$$[\text{nm2}]\ \frac{C = \text{fprompt1}}{\text{noMessage}(C\ C^*)\ =\ \text{false}}$$

$$[\text{nm3}]\ \frac{C = \text{dprompt1}}{\text{noMessage}(C\ C^*)\ =\ \text{false}}$$

[nm] noMessage(P) = true **otherwise**

8.5 TransLog.seal

The user interface of the TransLog tool is defined in the SEAL module TransLog.seal. In this module ten 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 will not go into the details of every single button, but we will discuss some general aspects. First, many buttons have an enabling condition that consists of a function call to one of the *e-...* functions of the module ButtonConditions and a test on the sort of the term in the focus of the LogA window. This second test yields *true* only when a LogA window has been created. In this way it is guaranteed that at startup time only the Initialize button will be enabled.

For most buttons the operations section starts with a sequence of assignments in which various constructs (a term, a clause, a –possibly labeled– program) are assigned to SEAL variables. Then the transformation function is called, the result of the call is assigned to another SEAL variable (`NewVar`). This result is always a labeled program. A stripped version of this program is assigned to the focus of the TransLog window, the labeled version is appended (together with a possibly new version of P_{new}) to the sequence of duoprograms in the LogA window. At the moment of this append operation the focus of the LogA window is placed around the last duoprogram stored so far, and not around the whole sequence of duoprograms. The new focus contents will be this duoprogram with the new duoprogram appended to it. This is done for performance reasons: after an update of the contents of a focus only these contents are (again) pretty-printed by the SEAL system. This takes a lot of time. The focus operation `"LogA" . focus previous , next , next` places the focus again at the last duoprogram in the sequence. After a transformation step the contents of the LogB window ("the future") are flushed.

The Fold button and the Define button have a simple script. Only a dialogue prompt is placed in the TransLog window. In the operations section of the Fold button this is preceded by the storage of some information in the LogB window. The remaining part is found in the script of the Continue button. After some general assignments the distinction between a 'Fold continue' and a 'Define continue' is made in two while loops. Each loop contains the necessary SEAL code for the required operation.

The scripts for the Previous button and the Next button are straightforward: a program is taken from a LogA/LogB window, displayed in the TransLog window (without labels), and stored in a LogB/LogA window. The Initialize button takes care of the creation and initialization of the LogA window and LogB window. In the enabling condition of this function also the syntax of the program is checked. If the syntax is not OK, the Initialize button will not be enabled.

```
Configuration for language TransLog is

button Unfold
when ButtonConditions : e-unfold(focus, focus up,up) and "LogA" . focus is DUOPROGRAM
enable
  BtVar := focus;
  ClauseVar := focus up, up;
  DuoVar := "LogA" . focus;
  ProgVar := "LogA" . focus down , next;
  NewVar := Unfold : unfold(ProgVar, BtVar, ClauseVar);
  focus root;
  focus := LabeledProgram : strip-label(NewVar);
  "LogA" . focus :=  Log : l-appendP(DuoVar,NewVar);
  "LogA" . focus previous , next , next;
  "LogB" . focus root;
   "LogB" . focus := Log : l-dummy()
doc : "Unfold transformation on Prolog Program"

button Fold
when ButtonConditions : e-fold(focus,focus up,up) and "LogA" . focus is DUOPROGRAM
enable
  BtVar := focus;
  ClauseVar := focus up , up;
  "LogB" . focus root;
  "LogB" . focus := Log : l-fold-store(BtVar,ClauseVar);
  focus root;
  focus := Fold : foldPrompt()
doc : "First part of Fold transformation on Prolog Program"

button Define
when ButtonConditions : e-define(focus, focus root) and "LogA".focus is DUOPROGRAM
enable
  focus root;
  focus := Define : definePrompt()
doc : "First part of Define transformation on Prolog Program"

button AssocPerm
when ButtonConditions : e-asperm(focus , focus up,up) and "LogA" . focus is DUOPROGRAM
enable
  BtVar := focus;
  ClauseVar := focus up , up;
  DuoVar := "LogA" . focus;
  ProgVar := "LogA" . focus down , next;
  NewVar := AssocPermutation : assperm(BtVar, ClauseVar, ProgVar);
  focus root;
  focus := LabeledProgram : strip-label(NewVar);
  "LogA" . focus := Log : l-appendP(DuoVar,NewVar);
  "LogA" . focus previous , next , next;
  "LogB" . focus root;
  "LogB" . focus := Log : l-dummy()
doc : "Permutation of atom arguments with mode +,+,-. or -,-,+"

button Switch
when ButtonConditions : e-switch(focus, focus up,up) and "LogA" . focus is DUOPROGRAM
enable
```

```
  BtVar := focus;
  ClauseVar := focus up , up;
  DuoVar := "LogA" . focus;
  ProgVar := "LogA" . focus down , next;
  NewVar := GoalSwitch : switch(BtVar, ClauseVar, ProgVar);
  focus root;
  focus := LabeledProgram : strip-label(NewVar);
  "LogA" . focus :=  Log : l-appendP(DuoVar,NewVar);
  "LogA" . focus previous , next , next;
  "LogB" . focus root;
  "LogB" . focus := Log : l-dummy()
doc : "Switch on two adjacent bodyterms (shared vars NOT allowed)"

button CleanUp
when ButtonConditions : e-clean(focus, focus up) and "LogA" . focus is DUOPROGRAM
enable
  DuoVar := "LogA" . focus;
  ProgVar := "LogA" . focus down , next;
  HeadVar := focus;
  NewVar := CleanUp : cleanUp(HeadVar,ProgVar);
  focus root;
  focus := LabeledProgram : strip-label(NewVar);
  "LogA" . focus :=  Log : l-appendP(DuoVar,NewVar);
  "LogA" . focus previous , next , next;
  "LogB" . focus root;
  "LogB" . focus := Log : l-dummy()
doc : "Remove obsolete clauses (with respect to selected head predicate"

button Continue
when ButtonConditions : e-foldcon(focus , "LogB" . focus) or ButtonConditions : e-defcon(focus root)
enable
  FocusVar := focus;
  FRootVar := focus root;
  LBVar := "LogB" . focus;
  DuoVar := "LogA" . focus;
  ProgVar := "LogA" . focus down , next;
  PnewVar := "LogA" . focus down;
  while ButtonConditions : e-foldcon(FocusVar, LBVar) do
   NumberVar := focus;
   TempVar := "LogB" . focus;
   BtVar := Log : l-fold-term(TempVar);
   ClauseVar := Log : l-fold-clause(TempVar);
   NewVar := Fold : fold(NumberVar,BtVar,ClauseVar,PnewVar,ProgVar);
   focus root;
   focus := LabeledProgram : strip-label(NewVar);
   "LogA" . focus := Log : l-appendP(DuoVar,NewVar);
   "LogA" . focus previous , next , next;
   FocusVar := focus
  od;
  while ButtonConditions : e-defcon(FRootVar) do
   ClauseVar := focus;
   NewVar := Define : define(ClauseVar,ProgVar,PnewVar);
   NewPnewVar := Define : newDefine(ClauseVar,PnewVar);
   focus root;
   focus := LabeledProgram : strip-label(NewVar);
```

```
   "LogA" . focus := Log : l-appendN(DuoVar,NewPnewVar,NewVar);
   "LogA" . focus previous , next , next
  od;
   "LogB" . focus root;
   "LogB" . focus := Log : l-dummy()
doc : "Continuation of Fold / Define transformation"

button Previous
when ButtonConditions : e-previous(focus , "LogA" . focus root)
enable
  LogVar := "LogA" . focus;
  ProgVar := "LogA" . focus previous , down , next ;
  focus := LabeledProgram : strip-label(ProgVar);
  "LogB" . focus root , down;
  while "LogB" . focus next is DUOPROGRAM do "LogB" . focus next od;
  LogBVar := "LogB" . focus;
  "LogB" . focus := Log : l-appendB(LogBVar,LogVar);
  "LogB" . focus root;
  "LogA" . focus root;
  LogAVar := "LogA" . focus;
  "LogA" . focus := Log : l-delete(LogAVar);
  "LogA" . focus down;
  while "LogA" . focus next is DUOPROGRAM do "LogA" . focus next od
doc: "One step backwards in transformation sequence."

button Next
when ButtonConditions : e-next("LogB" . focus)
enable
  focus root;
  "LogB" . focus root , down;
  while "LogB" . focus next is DUOPROGRAM do "LogB" . focus next od;
  DuoVar := "LogB" . focus;
  ProgVar := "LogB" . focus down , next;
  focus := LabeledProgram : strip-label(ProgVar);
  LogVar := "LogA" . focus;
  "LogA" . focus :=  Log : l-appendA(LogVar,DuoVar);
  "LogA" . focus previous , next , next;
  "LogB" . focus root;
  LogBVar := "LogB" . focus;
  "LogB" . focus :=  Log : l-delete(LogBVar)
doc: "One step forward in transformation sequence."

button Initialize
when ButtonConditions : e-init(focus root) and PrologFunctions : syntaxCheck(focus)
enable
  FocusVar := focus ;
  create("LogA", Log : l-initialize(FocusVar));
  create("LogB", Log : l-dummy());
  "LogA" . focus down , next
doc : "Initialize log windows."
```

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 (CWI). Thanks to Eelco Visser (University of Amsterdam) for commenting the TransLog specification.

References

[Bru95] J.J. Brunekreef. TransLog, an Interactive Tool for Transformation of Logic Programs. Technical Report P9512, Programming Research Group, University of Amsterdam, 1995.

[Bru96] J.J. Brunekreef. A transformation tool for pure Prolog programs. In *LOPSTR '96 – Sixth International Workshop on Logic Program Synthesis and Transformation*, 1996. To appear.

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

[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.

[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.

[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.