ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

S. Renault, A. Pettorossi, M. Proietti

DESIGN, IMPLEMENTATION, AND USE OF THE MAP TRANSFORMATION SYSTEM

R. 491 Dicembre 1998

Sophie Renault - Département d'Informatique et Recherche Opérationnelle, Université de Montréal, 2920 chemin de la Tour, Montréal, Québec H3C 3J7, Canada.
E-mail : `renault@IRO.UMontreal.CA`.
URL : `http://www.iro.umontreal.ca/~renault`.

Maurizio Proietti - Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy. Email : `proietti@iasi.rm.cnr.it`.
URL : `http://www.iasi.rm.cnr.it/~proietti`.

Alberto Pettorossi - Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via di Tor Vergata, I-00133 Roma, Italy. Email : `adp@iasi.rm.cnr.it`.
URL : `http://www.iasi.rm.cnr.it/~adp`.

This work has been partially supported by MURST Progetto Cofinanziato 'Tecniche Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software' (Italy), Progetto Coordinato CNR 'Programmazione Logica' (Italy), Progetto Coordinato CNR 'Verifica, Analisi e Trasformazione dei Programmi Logici' (Italy), Programma Galileo CRUI (Italy), and INTAS Project 'Efficient Symbolic Computing' 93–1702. This work has been done while the first author was visiting IASI-CNR.

Istituto di Analisi dei Sistemi ed Informatica, CNR
viale Manzoni 30
00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.rm.cnr.it
URL: http://www.iasi.rm.cnr.it

Abstract

The MAP transformation system is a tool which supports the interactive derivation of logic programs using the unfold/fold transformation methodology.

A derivation consists of a sequence of applications of some predefined transformation rules, starting from a given initial program. When a rule is invoked, if the conditions of its application are satisfied, the system transforms the current program into a new program. Such transformation steps may be performed interactively, in a sequence, until a final program is derived.

The system provides a graphical user interface through which the transformation rules are applied by means of mouse clicks, button presses, menu selections, and dialog boxes. MAP also provides the user with many facilities for controlling and documenting program derivations, such as exploring alternative derivations, printing, saving, and restoring derivations.

Key words: Automatic program derivation, program transformation, logic programming, transformation rules, graphical user interface

1. Introduction

The MAP transformation system is a tool which is designed to support the interactive derivation of logic programs [2] using the unfold/fold transformation methodology [1, 4, 10]. It has been developed at the IASI Institute of the National Research Council, Rome (Italy) on the basis of an existing implementation described in [8]. The main novelty in the new implementation is a graphical user interface (GUI) through which transformations are specified and applied by means of mouse clicks, button presses, menu selections, and dialog boxes. The following set of predefined transformation rules is provided by the system: *definition, restriction, unfolding, folding, goal replacement, goal rearrangement, goal addition, goal deletion, clause replacement, clause rearrangement, clause addition, clause deletion, generalization*, and *case split*. (Notice that, some rules are particularly useful special cases of other rules.)

The program derivation process consists of a sequence of applications of those rules, starting from a given initial program, together with a *theory*, that is, a collection of *replacement laws* which are needed for applying the goal replacement rule.

When a rule is invoked, if suitable conditions for its application are satisfied, the system transforms the current program, say P_k, into a new program, say P_{k+1}. Several such derivation steps may be performed one after the other, starting from an initial program P_0, until a final program P_n is derived, and thus, a derivation can be seen as a sequence $P_0, \ldots, P_n$ of programs.

The system keeps track of the *history* of the derivation and it allows the user to backtrack to some previous programs in the derivation and explore alternative derivations. It is also possible to *undo* some transformation steps if the last part of the derivation which has been generated has to be discarded.

The MAP system also provides the user with various facilities for documenting the derivation of programs, such as printing, saving and restoring programs, theories, and derivations themselves.

The transformation methodology underlying MAP, suggests the definition of *strategies*, that is, sequences of applications of the rules which allow us to derive very efficient programs. Strategies can be generated in a semiautomatic way. Some examples of strategies for logic programs can be found in [5, 6, 7] and are partially implemented in the system [8]. At present, however, these strategies cannot be used via the GUI.

In Section 2, we recall the basic rules for logic programs transformation. In Section 3 we give an overview of the system by describing the main components of the GUI and how they interact with the underlying Prolog implementation of the transformation rules. In Section 4, we describe in detail how the user can apply the various transformation rules via the GUI, and we illustrate the use of the menus and dialog boxes for controlling and documenting the derivations. Finally, in Section 5 we list some enhancements of the system which are under development.

2. Logic Program Transformation via Unfold/Fold Rules

2.1. Programs, theories, and transformation sequences

In this report we consider *definite logic programs* [2]. We are developing some extensions of the system which will allow the user to deal with other languages (see Section 5).

For specifying the syntax of definite logic programs, which we here simply call *programs*, we take the usual definitions of variables, constants, function symbols, terms, predicate symbols, and atoms [2]. We assume that '$=$' and '$\neq$' are included in the set of predicate symbols. For goals, clauses, and programs we take the following conventions.

A goal is a (possibly empty) sequence of atoms. The concatenation operator for goals is denoted by comma ','. This operator is associative and has the empty sequence of atoms, called the *empty goal*, as neutral element. Thus, for instance, when we write 'a goal G_1, G_2, G_3' we mean 'a goal which is the concatenation of the (possibly empty) goals G_1, G_2, and G_3'.

A clause is an expression of the form $p(t_1, \ldots, t_k) \leftarrow G$, where $p(t_1, \ldots, t_k)$ is an atom with predicate symbol p distinct from '=' and '$\neq$' and G is a goal. The head and the body of a clause C will be denoted by $hd(C)$ and $bd(C)$, respectively.

A program is a (possibly empty) sequence of clauses.

The semantics of a logic program P is defined as the least Herbrand model of P [2], extended by the interpretations of '=' and '$\neq$'. The least Herbrand model of P is denoted by $M(P)$ and it is constructed by considering P as a set of formulas of first-order logic. The elements of this set are the clauses of P, where the sequences of atoms are to be read as conjunctions of atoms, the $\leftarrow$ operator is the reverse implication, and all variables occurring in a clause are assumed to be universally quantified. We assume that for all P, $M(P)$ includes the sets $\{t = t \mid t \text{ is a ground term}\}$, and $\{t_1 \neq t_2 \mid t_1 \text{ and } t_2 \text{ are different ground terms}\}$.

Two programs P_1 and P_2 are *equivalent* (w.r.t. the least Herbrand model semantics) iff $M(P_1) = M(P_2)$.

During program derivation we often have to make use of properties of predicates. In the MAP system such properties are exploited by means of the goal replacement rule (see below) which uses a replacement law, that is, an equivalence of the form $G_1 \leftrightarrow G_2$ where G_1 and G_2 are goals.

Let $vars(E)$ denote the set of variables occurring in a syntactic construct E. We say that replacement law $G_1 \leftrightarrow G_2$ is *valid in* a program P iff

$$M(P) \models \forall X_1, \ldots, X_k.\ [(\exists Y_1, \ldots, Y_m.\ G_1) \leftrightarrow (\exists Z_1, \ldots, Z_n.\ G_2)]$$

where (i) $\{X_1, \ldots, X_k\} = vars(G_1) \cap vars(G_2)$, (ii) $\{Y_1, \ldots, Y_m\} = vars(G_1) - \{X_1, \ldots, X_k\}$, and (iii) $\{Z_1, \ldots, Z_n\} = vars(G_2) - \{X_1, \ldots, X_k\}$. A *theory for* P is a set of replacement laws which are valid in P. Currently, MAP does not offer any support for proving the validity of replacement laws, and these laws should be proved off-line by the user.

Given an *initial program* P_0 and a theory T for P_0, we construct a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence* as follows: for $k = 0, \ldots, n-1$, program P_{k+1} is derived from program P_k by applying one of the transformation rules defined in Section 2.2 below, and possibly using a replacement law of the theory T. The MAP system allows the user to change T by adding new replacement laws and deleting replacement laws which are not used.

The programs P_0 and P_n can be shown to be equivalent if suitable conditions on the application of the transformation rules (in particular, on the folding and goal replacement rules [10]) are satisfied. The system does *not* check all these conditions, and thus, the equivalence between P_0 and P_n is left to the user's responsibility.

2.2. MAP Transformation Rules

In this section we present the transformation rules currently implemented in the MAP system. We first recall some notions needed for the presentation of these rules.

We say that a predicate p *depends on* a predicate q in a program P iff either there exists in P a clause of the form: $p(\ldots) \leftarrow B$ such that q occurs in the goal B or there exists in P a predicate r such that p depends on r in P and r depends on q in P. A predicate p is *defined* in P iff p

occurs in the head of a clause of P. An atom A is defined in P iff the predicate of A is defined in P.

As already mentioned, the semantics of a program does not depend on the variable names used in its clauses. Thus, for the application of the transformation rules we may consistently rename the variables occurring in the program clauses. In particular, variable renaming may be needed for applying the unfolding, folding, and goal replacement rules where we require that some clauses do not have variables in common with other clauses or replacement laws.

Let P_0 be a given initial program and T be a theory for P_0. Let $P_0, P_1, \ldots, P_k$ be a transformation sequence constructed at steps $1, \ldots, k$. At step $k+1$, in order to derive program P_{k+1} from program P_k, any of the following rules can be used:

1. *Definition*: Let $C_i: \ newp(\ldots) \leftarrow B_i$, with $i = 1, \ldots, n$ be a sequence of clauses such that the predicate *newp* does not occur in $P_0, \ldots, P_k$. By the definition rule, we get P_{k+1} by adding to P_k the sequence $C_1, \ldots, C_n$.

   The sequence of clauses introduced by the definition rule up to step k is denoted by Def_k.

2. *Restriction*: Let *pred* be a subset of the predicate symbols occurring in P_k. By *restricting* P_k to *pred* we derive program P_{k+1} by deleting from P_k all clauses of the form $p(\ldots) \leftarrow B$ such that $p \notin pred$ and no predicate in *pred* depends on p in P_k.

   The restriction rule can be viewed as an inverse of the definition rule and it coincides with the *definition elimination* rule of [4] when *pred* includes all predicates occurring in P_0.

   By $P_0 + Def_k$ we denote the sequence of clauses obtained by first concatenating P_0 and Def_k, and then restricting the resulting program to the set of predicates occurring in P_k.

3. *Unfolding*: Let $C: \ H \leftarrow Left, A, Right$ be a clause in P_k, such that (i) A is an atom defined in P_0 and (ii) $vars(C) \cap vars(P_0 + Def_k) = \emptyset$. Let $D_1, \ldots, D_n$ be all clauses in program $P_0 + Def_k$, such that A is unifiable with $hd(D_1), \ldots, hd(D_n)$, with most general unifier $\theta_1, \ldots, \theta_n$ respectively. By unfolding C w.r.t. A we derive the n new clauses $C_i: \ (H \leftarrow Left, bd(D_i), Right)\ \theta_i$, for $i = 1, \ldots, n$, and we get the new program P_{k+1} by replacing in P_k clause C by clauses $C_1, \ldots, C_n$.

   In particular, if there is no clause in $P_0 + Def_k$ whose head unifies with A, then we get P_{k+1} by deleting C from P_k.

   Let $C: \ H \leftarrow Left, t = u, Right$ be a clause in P_k. If t and u are unifiable via the most general unifier θ, then by unfolding C w.r.t. $t = u$ we derive the new clause $C_1: \ (H \leftarrow Left, Right)\ \theta$, and we get the new program P_{k+1} by replacing C by C_1 in P_k. Otherwise, t and u are not unifiable and we get the new program P_{k+1} by deleting C from P_k.

   Let $C: \ H \leftarrow Left, t \neq u, Right$ be a clause in P_k. If t and u are non-unifiable terms, then by unfolding C w.r.t. $t \neq u$ we derive the new clause $C_1: \ H \leftarrow Left, Right$, and we get the new program P_{k+1} by replacing C by C_1 in P_k. Otherwise, if t and u are identical terms then we get the new program P_{k+1} by deleting C from P_k.

4. *Folding*: Let $C_1, \ldots, C_n$ and $D_1, \ldots, D_n$ be two subsequences of (possibly non-contiguous) clauses in P_k and $P_0 + Def_k$ respectively, such that $vars(C_1, \ldots, C_n) \cap vars(D_1, \ldots, D_n) = \emptyset$. Suppose that there exist a clause $C: \ H \leftarrow Left, A, Right$ and a sequence of substitutions $\theta_1, \ldots, \theta_n$, such that, for $i = 1, \ldots, n$:

   (a) the domain of θ_i is $bd(D_i)$,

(b) C_i is a variant of the clause $H \leftarrow \mathit{Left}, bd(D_i)\theta_i, \mathit{Right}$,

(c) $A = hd(D_i)\theta_i$,

(d) for every clause D of $P_0 + \mathit{Def}_k$ which is not in the sequence $D_1, \ldots, D_n$, the atom $hd(D)$ is not unifiable with any variant of the atom A, and

(e) for every variable X in the set $vars(bd(D_i)) - vars(hd(D_i))$, we have that
   - $X\theta_i$ is a variable which does not occur in $(H, \mathit{Left}, \mathit{Right})$, and
   - for any variable Y occurring in $bd(D_i)$ and different from X, the variable $X\theta_i$ does not occur in the term $Y\theta_i$.

By folding $C_1, \ldots, C_n$ w.r.t. the goals $bd(D_1)\theta_1, \ldots, bd(D_n)\theta_n$ using $D_1, \ldots, D_n$, we derive clause C and we get the new program P_{k+1} by replacing in P_k clauses $C_1, \ldots, C_n$ by clause C.

5. *Goal Replacement*: Let $C: \; H \leftarrow \mathit{Left}, G_1\theta, \mathit{Right}$ be a clause in P_k and let $G_1 \leftrightarrow G_2$ be a replacement law in T such that:

   (a) the predicates occurring in G_2 are defined in P_k,

   (b) the domain of θ is $vars(G_1)$,

   (c) $vars(C) \cap vars(G_1 \leftrightarrow G_2) = \emptyset$,

   (d) for every variable X in the set $vars(G_1) - vars(G_2)$, we have that
   - $X\theta$ is a variable which does not occur in $(H, \mathit{Left}, \mathit{Right})$, and
   - for any variable Y occurring in G_1 and different from X, the variable $X\theta$ does not occur in the term $Y\theta$.

   By replacement of $G_1\theta$ in C using the law $G_1 \leftrightarrow G_2$, we derive the new clause $D: \; H \leftarrow \mathit{Left}, G_2\theta, \mathit{Right}$ and we get P_{k+1} by replacing in P_k clause C by clause D.

6. *Goal Rearrangement*: We derive P_{k+1} by reordering the atoms in the body of a clause in P_k.

7. *Goal Addition and Deletion*: By goal addition we derive the new program P_{k+1} by replacing the clause $H \leftarrow \mathit{Left}, \mathit{Right}$ in P_k by the clause $H \leftarrow \mathit{Left}, G, \mathit{Right}$. By goal deletion we derive the new program P_{k+1} by replacing the clause $H \leftarrow \mathit{Left}, G, \mathit{Right}$ in P_k by the clause $H \leftarrow \mathit{Left}, \mathit{Right}$. The goal addition and deletion rules are applied in the following cases:

   (a) *Addition and Deletion of Duplicate Goals*: G occurs in $(\mathit{Left}, \mathit{Right})$.

   (b) *Addition and Deletion of True Goals*:
   - either G is an atom of the form $t = t$
   - or G is an atom of the form $t_1 \neq t_2$, where t_1 and t_2 are not unifiable.

8. *Clause Replacement*: We derive P_{k+1} by replacing in P_k the sequence of clauses $C_1, \ldots, C_n$ by the sequence of clauses $D_1, \ldots, D_m$.

   In MAP, clause replacement is always possible without any check performed by the system. A warning message informs the user that semantics presevation is not ensured, and that the correctness of the corresponding transformation step is left to his responsibility.

   The following rules 9–12 are particular instances of clause replacement.

9. *Clause Rearrangement*: We derive P_{k+1} by reordering the clauses in P_k.

10. *Clause Addition and Deletion*: We derive P_{k+1} either by adding to P_k or by deleting from P_k a clause C in the cases listed below.

    (a) *Addition and Deletion of Subsumed Clauses*: C is subsumed by a clause D in P_k. C and D may be equal, but for clause deletion they should occur at different positions in the sequence of clauses which constitutes program P_k.
    In particular we may duplicate a clause and also delete one of the occurrences of a clause with multiple occurrences.

    (b) *Addition and Deletion of Tautologies*: C is of one of the following forms:
    - $H \leftarrow \mathit{Left}, t_1 = t_2, \mathit{Right}$, where t_1 and t_2 are not unifiable, or
    - $H \leftarrow \mathit{Left}, t \neq t, \mathit{Right}$, or
    - $H \leftarrow \mathit{Left}, H, \mathit{Right}$.

11. *Generalization*: We derive P_{k+1} by replacing a clause C in P_k by the clause $D: \ H \leftarrow X = t, \mathit{Body}$, where X is a variable not occurring in C and C is equal to $(H \leftarrow \mathit{Body})\ \{X/t\}$.

12. *Case Split*: Let C: $H \leftarrow \mathit{Body}$ be a clause in P_k. Let X be a variable occurring in H and t a term without occurrences of X. By case split of C w.r.t. $\{X/t\}$, we derive the following two clauses $C_1 : \ (H \leftarrow \mathit{Body})\{X/t\}$ and $C_2 : \ H \leftarrow X \neq t, \mathit{Body}$, and we get P_{k+1} by replacing C by (C_1, C_2) .

2.3. Program Derivation History

MAP keeps track of the history of a program derivation, that is, the sequence of programs which have been generated to derive the current program and the corresponding sequence of transformation rules which have been applied. The system provides the user with facilities to show, save, and print the history of a derivation. The process of constructing a derivation is nondeterministic, because many transformation rules may be applied to a given program. The MAP transformation system stores the various programs in a tree, called the *transformation tree*. With each node of the transformation tree we associate a program and with each arc we associate the transformation rule used for deriving the son program from the father program. Thus, the history of the current program corresponds to the branch of the transformation tree starting from the root (associated with the initial program) and ending with the node associated with the current program. MAP provides a *goto* facility which allows the user to backtrack to any ancestor program in the transformation tree and to explore alternative branches. It also provides an *undo* facility to discard the current program from the transformation tree.

3. Overview of the MAP system

3.1. Implementation

The MAP system has two main components: the *transformation engine* and the *graphical user interface* (GUI). The transformation engine is implemented in SICStus Prolog [9] and consists of modules for the transformation rules, the transformation strategies (not yet available through the user interface), the theories, and the transformation tree. The GUI is a window system through which the user may interact with the transformation engine. The GUI is implemented in the Tcl language and uses the Tk toolkit [3]. Tcl is an interpreted string based scripting language with many extension packages, in particular the graphical interface toolkit Tk. The main Tcl/Tk program is *wish* (*wi*ndowing *sh*ell) with which one creates graphical applications. Tk provides a set of Tcl commands that are used to construct GUI's, by allowing the creation and manipulation of *widgets*. A widget is a window in a GUI that has a particular appearance and behaviour which can be modified by using configuration options (font, size, visibility, highlighting, etc.). Widget types include buttons, scrollbars, menus, labels, and text windows.

The interface between the MAP transformation engine and the GUI is implemented by means of SICStus Prolog's `tcltk` library package which provides a bidirectional interface to Tcl/Tk.

Figure 1 describes the main components of the MAP system.

3.2. Window organization

The MAP system has a main window, called the *MAP window*, with menus and buttons which allow the user to perform program derivations and also to make use of other system utilities which we will describe later on. From the MAP window the user may activate three other subsidiary windows, which provide additional information related to the program derivation: the *Theory*, the *Init+Defs*, and the *History* windows.

The *MAP window* includes a text area where at each step of the derivation the current program is displayed. This area is called the *Program window*. At the beginning of a derivation, an initial program can be selected from the list of the available ones. Clauses and atoms in the Program window are selectable objects, that is, the user can select a clause or an atom in a clause by clicking on it. This allows him to specify the arguments of a transformation rule to be applied. Sometimes, the arguments of a transformation rule are not clauses or atoms of the current program. For instance the folding rule uses definition clauses which might belong to the initial program, and the goal replacement rule uses replacement laws which belong to a theory. These clauses and replacement laws can be selected from the Init+Defs and Theory windows, respectively.

When a transformation step is performed, the system issues a message indicating which transformation rule has been applied, and the derived clauses, if any, are displayed in the Program window. Clauses are numbered and the numbers of the clauses which belong to the current program are in boldface style, so that the clauses belonging to the current program can be distinguished from those belonging to old programs. The clauses appearing in the Program window, and not belonging to the current program cannot be selected for applying a transformation rule. It is possible to refresh the Program window so that the system clears everything but the clauses belonging to the current program.

The *Theory window* displays the replacement laws, if any, which have been loaded by the user. This window is shown under the user's request via a menu item, and it is updated each time a law is added or deleted during a derivation. The replacement laws needed for the application of the goal replacement rule can only be selected from this window.

At the beginning of a derivation the *Init+Defs window* displays the clauses of the initial program. This window is updated by adding the clauses introduced by the definition rule and by deleting the clauses defining predicates discarded by the restriction rule. The Init+Defs window is shown under the user's request via a menu item and it must be used for selecting the clauses needed for an application of the folding rule.

The *History window* displays the history of the current program, i.e. the summary of all transformation steps performed from the initial program to the current one. This window is

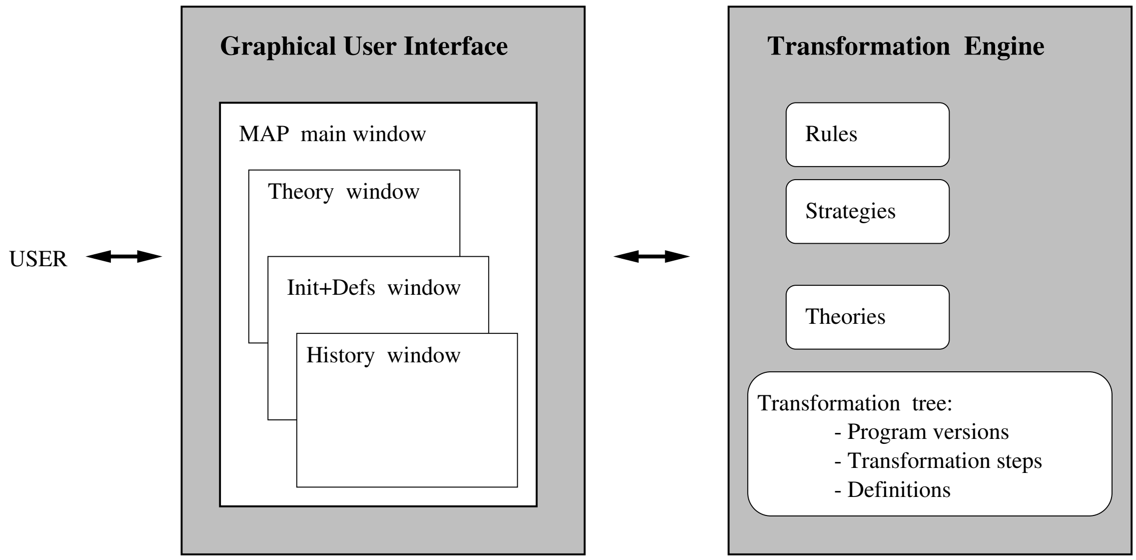


Figure 1: Components of the MAP system.

shown under the user's request via a menu item. There is no selectable widget in the History window and in particular, it is never used to specify the arguments of a transformation rule.

3.3. Interaction for Applying Transformation Rules

The predicate which realizes the application of the transformation rules can be viewed as a function of type: $Rule \times Term^* \longrightarrow Prog$ where

- *Rule* is the set of transformation rules described in Section 2.2, and
- $Term^*$ is a set of tuples of terms which may be clauses, atoms or laws selected from the current program, the theory, the initial program, or the set of clauses introduced by the definition rule,
- *Prog* is the set of programs.

A transformation step consists of an application of a transformation rule taken from *Rule* to a tuple of arguments taken from $Term^*$.

In order to issue a command to the transformation engine for realizing a transformation step, the user interacts with the interface objects. The style of interaction used in MAP is first to select the rule arguments, if any, and then to select the rule. Rule arguments are specified by means of mouse clicks on the Program window and possibly, on the Init+Defs window or the Theory window. Rules are selected by means of button presses. When a rule is selected, if the conditions for its application together with the selected arguments are satisfied, the corresponding transformation step is performed. Then the Program window and the Init+Defs window (in case of a definition or restriction transformation) are updated. Sometimes the selection of a rule activates a dialog box or a warning box for requesting some additional information before the corresponding command can be issued to the transformation engine. This additional information can be supplied by the user either by a mouse click or by typing on the keyboard.

3.4. Transformation Engine

A sequence of interaction events occurring in the GUI is mapped to a term which is interpreted by the transformation engine as a call of a transformation rule with suitable arguments. When this call is passed to the transformation engine, the conditions for the applicability of the selected rule are checked. If they are not satisfied, no further computation is done and no changes affect the displayed objects in the GUI, except that an error message is shown in the main window. Otherwise, the transformation engine computes the new program which derives from the current one by the specified application of the selected rule. Also the transformation tree is changed by adding an arc and a node corresponding to the application of the transformation rule. The computation of the new program forms the core of the system and it is independent of the GUI. The SICStus Prolog code implementing the transformation engine includes the modules listed below.

Modules for the transformation rules:

- `definition_rule.pl` which contains predicates for the definition and restriction rules.
- `unfolding_rule.pl` which contains predicates for the unfolding rule.
- `folding_rule.pl` which contains predicates for the folding rule.
- `goal_replacement.pl` which contains predicates for the goal replacement, goal addition, goal deletion, and goal rearrangement rules.
- `clause_replacement.pl` which contains predicates for the clause replacement, clause addition, clause deletion, and clause rearrangement rules.
- `case_split.pl` which contains predicates for the case split rule.
- `generalize.pl` which contains predicates for the generalization rule.

Modules for the transformation strategies (not yet available through the GUI):

- `uve.pl` which implements the strategy for the elimination of unnecessary variables described in [7].
- `lap.pl` which implements the Loop Absorption strategy defined in [6].
- `determ_prog.pl` which implements the strategy for reducing nondeterminism described in [5].

Modules for the transformation tree and the theories:

- `environment.pl` which contains predicates that are responsible for updating the transformation tree when a transformation step is performed. It also contains the predicates which manipulate theories.
- `struct.pl` which contains operations on data structures representing the transformation tree.

The mapping of interaction events occurring in the GUI to calls of Prolog predicates in the transformation engine and vice versa, is realized by the following modules, which use SICStus Prolog's `tcltk` library:

- `general_calls.pl` which contains predicates which realize the mapping from the GUI to the transformation engine. In particular, it contains the predicate

  $apply\_transformation\_rule(Rule, Parameters, Success)$

  which, depending on the instantiation of *Rule* and *Parameters* provides a call to the appropriate transformation rule implemented in one of the above described modules.

- `interface.pl` which contains predicates which realize the mapping from the transformation engine to the GUI. In particular, it contains the predicate

  $update\_proofText(Label, OldClauses, NewClauses)$

  which, at the end of a transformation step, updates the current program displayed in the Program window.

4. Description of the GUI

In this section we illustrate the main functionalities of MAP's user interface.

When the transformation system is started up, the MAP main window is activated. As we have already mentioned in the previous section, the MAP window has menus, buttons, and a text window where the current program is displayed (initially this window is empty). The reader may find a snapshot of the MAP window in Figure 2.

In this section we first describe the menus available in the MAP window, and we then see how the user can interact with the various interface objects to perform program transformations.

When performing 'Load' and 'Save' actions the system uses some default subdirectories of the main MAP directory. These directories are created at installation time and they are:

`Histories/   Programs/   Sessions/   Theories/`

4.1. Menus

4.1.1. Options Menu

The 'Options' menu allows the user to change the font size and also to print programs, theories, and histories.

4.1.2. Derivation Menu

Initially, the Derivation pull-down menu has only its first two entries enabled. The other ones are disabled (dimmed) until a derivation is initialized. A derivation can be initialized by selecting one of the first two items:

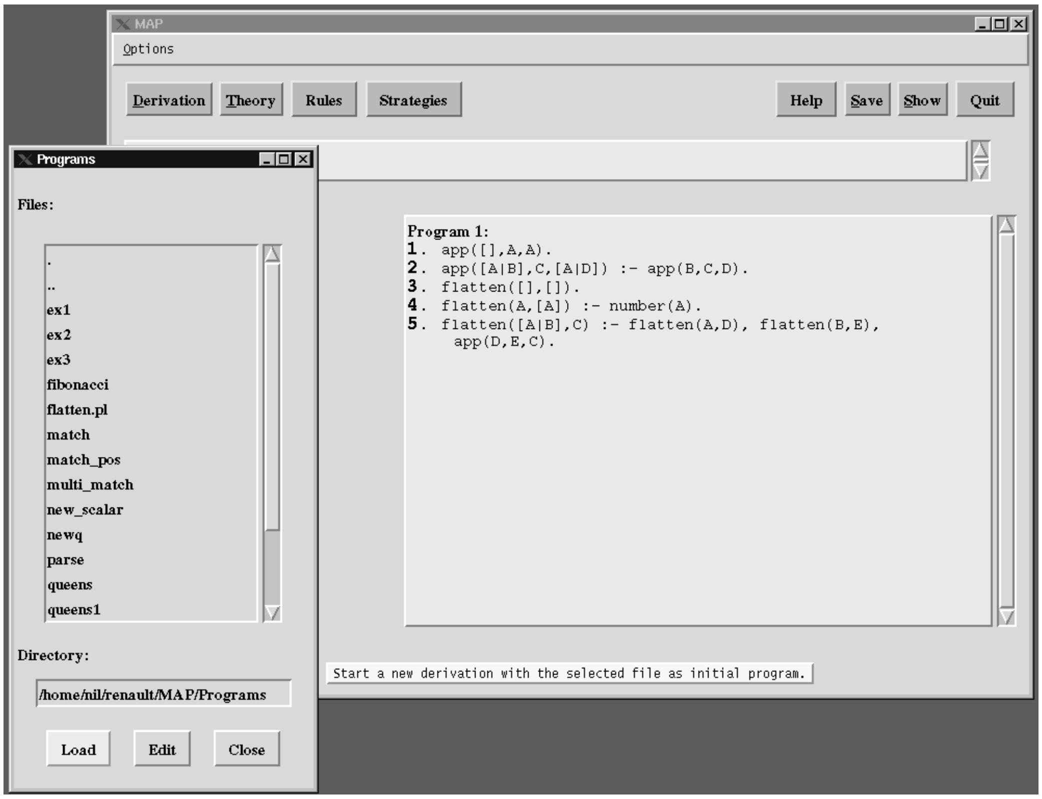


Figure 2: Loading an initial program and starting a new derivation.

- 'New' opens a dialog box for file selection. It allows the user to browse in his directories and to select a file for editing and/or starting a new derivation using the selected file as initial program. The default location for editing, loading, and saving programs is the subdirectory `Programs/`. Figure 2 shows the system after the user has selected the program 'flatten.pl' (double-clicked on that item) and pressed the button 'Load'.

- 'Load' opens a dialog box for file selection and restores the derivation corresponding to the selection. The selected file must correspond to a derivation saved in a previous session via the menu entry 'Save/Derivation'. By default this file is located in the subdirectory `Sessions/`.

When a transformation is started, the selected initial program is displayed in the Program window, and the other entries of the 'Derivation' menu are enabled. They provide the following functionalities:

- 'Restart' allows the user to start a new derivation with the current program as initial program. It is a shortcut for the sequence of actions consisting in saving the current program and then starting a new derivation from the saved program.

- 'Undo' deletes the tranformation step which has been performed to derive the current program. (If the user has done a 'Goto' action described below, the current program may be not the most recently derived program.) This operation consists in discarding from the transformation tree the node corresponding to the current program, and discarding also the ingoing arc. The father of the current program becomes the new current program. An 'Undo' action is allowed only if the current program is a leaf.

- 'Goto' allows the user to move to any node in the transformation tree and to continue the derivation from that node. In particular, the user may backtrack to an ancestor node. Thus, by a 'Goto' the user may make any program in the transformation tree to be the current program without deleting any branches in the tree.

- 'Show' is a cascade menu with the following entries:
  - 'Program' refreshes the Program window and it clears everything but the clauses belonging to the current program.
  - 'History' activates the History window where the history of the current program is displayed (see Figure 3).
  - 'Theory' activates the Theory window where the current theory is displayed. The current theory consists of all replacement laws that have been loaded via the actions 'Load' or 'Add law' and that have not been deleted by the 'Delete law' action (see pull down menu 'Theory' below).
  - 'Init+Defs' activates the Init+Defs window. Recall that at the start of a derivation, the Init+Defs window lists the clauses of the initial program, and it is updated by adding the clauses introduced by a definition step and by discarding the clauses defining the predicates deleted by a restriction step. This window is necessary for selecting clauses for applying the folding rule.

- 'Save' is a cascade menu with the following entries:
  - 'Program' saves the current program into a file in the subdirectory `Programs/`.
  - 'History' saves the history of the current program into a file in the subdirectory `Histories/`.
  - 'Theory' saves the current theory into a file in the subdirectory `Theories/`.
  - 'Derivation' saves the current transformation tree into a file in the subdirectory `Sessions/`.

  For each of the above four items, a dialog box asks the user for a file name to be used in the save operation.

4.1.3. Theory Menu

A theory is made out of a sequence of expressions of the form $L_1, \ldots, L_n$ `->` $R_1, \ldots, R_m$ where $L_1, \ldots, L_n, R_1, \ldots, R_m$ are atoms. The intended meaning is that $L_1, \ldots, L_n \leftrightarrow R_1, \ldots, R_m$ is a replacement law which is valid in the initial program, and the goal replacement rule may be used for replacing an instance of $L_1, \ldots, L_n$ by the corresponding instance of $R_1, \ldots, R_m$ (with the restrictions indicated in Section 2). Thus, our representation of the replacement laws indicates a left-to-right direction for the replacement to be performed. Theories are given by the user and the system does not check that the replacement laws are valid in the initial program.

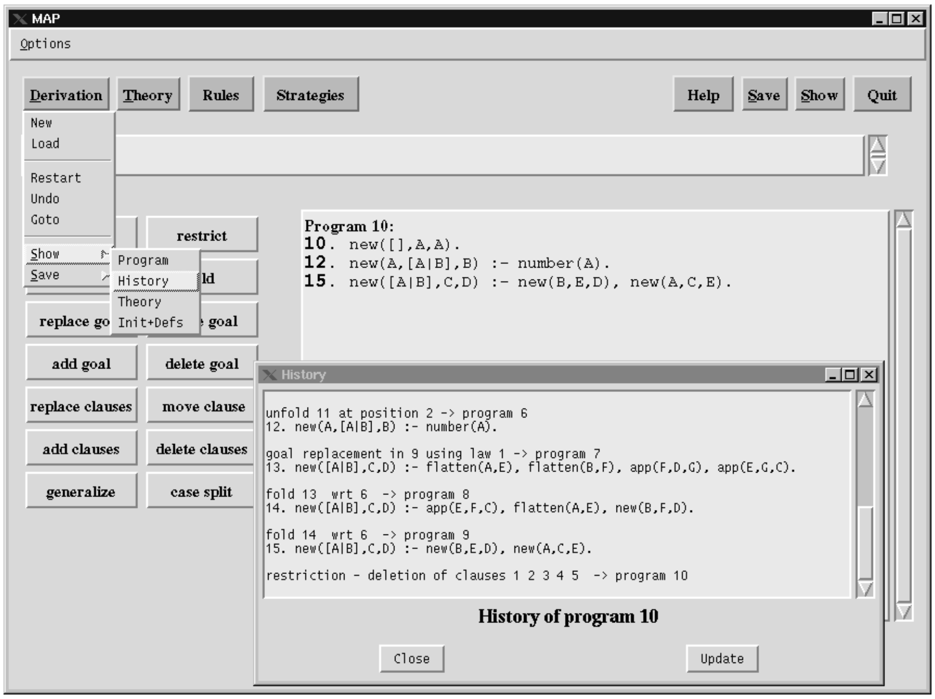
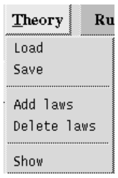


Figure 3: Showing the history of the current program.

The entries of the ‘Theory’ pull-down menu correspond to actions as follows:



- ‘Load’ opens a dialog box for file selection. It allows the user to browse in his directories and to select a file for editing and/or loading a new theory. When the user loads a theory, the replacement laws of this theory are added to the current theory, that is, to the set of laws which are available for the goal replacement rule.
- ‘Save’ saves the current theory into a file in the subdirectory `Theories/`. The name of the file for the theory to be saved is given by the user via a dialog box. It is a shortcut for the entry ‘Show/Theory’ in the ‘Derivation’ menu.
- ‘Add laws’ opens a text window in which the user can edit laws. These laws are added to the current theory.
- ‘Delete laws’ allows the user to delete some laws in the current theory. The deletion of a law is allowed only if the law has not been used so far in the derivation.
- ‘Show’ activates the Theory window and displays the current theory. It is a shortcut for the entry ‘Show/Theory’ in the ‘Derivation’ menu.

4.1.4. Rules Menu

The 'Rules' button displays a set of buttons in a panel on the left-hand side of the MAP main window. Each button is associated with a transformation rule which the user may apply to the current program. In general, rules require some arguments which may be clauses, atoms, or replacement laws to be selected from the Program, Theory, and Init+Defs windows. Arguments are selected before rule application. In order to select an atom in the body of a clause in the Program window, the user has to click on that atom with the left button of the mouse. In order to select a clause from the Program window or from the Init+Defs window, the user has to click on the head of the clause with the left button of the mouse. In order to select a replacement law from the Theory window, the user has to click on the law with the left button of the mouse. For multiple selections, one has to press the 'Control' key while clicking. Below we describe the use of each button corresponding to a transformation rule.

define opens a text window, called the 'define' window, in which the user can edit the new definition clauses. If the user selects some clauses in the current program before pressing the 'define' button of the main window, and all these clauses have the same head predicate, then the system displays in the 'define' window a list of new clauses which have the same bodies as the clauses selected in the current program and have a new head predicate name. This new name is of the form *newk* for some $k \geq 1$. The user is allowed to edit the new clauses proposed by the system. This behaviour of the system often saves a considerable amount of typing, because the new clauses one wants to introduce by definition are often similar to existing clauses in the current program (in particular the new definitions may be used to fold clauses occurring in the current program). The 'define' button in the 'define' window applies the definition rule while the 'close' button cancels the operation.

restrict restricts the current program to a predicate selected by the user. In order to select a predicate, the user should select, from the Program window, an atom where this predicate occurs. Restrictions to more that one predicate at a time are also allowed.

unfold unfolds a clause of the current program w.r.t. an atom selected in its body. Figure 4 shows the MAP window when the user presses the button 'unfold' after having selected an atom in the body of a clause appearing in the Program window.

fold folds one or more clauses selected in the current program using the same number of clauses selected in the Init+Defs window. Figure 5 shows the MAP and the Init+Defs windows when the user presses the button 'fold' after the selection of a clause in the current program and a definition clause in the Init+Defs window. The Help subwindow within the MAP window is also active in this figure, and this fact indicates that the user has pressed the 'Help' button before applying the folding step. (For more information on the Help window see Section 4.1.5.)

replace goal replaces a goal selected in the body of a clause in the Program window using a replacement law selected in the Theory window. Figure 6 shows the MAP and the Theory windows when the user presses the button 'replace goal' after the selection of a goal and a replacement law. The message 'Theory updated' in the Program window indicates that before applying the goal replacement rule the user has changed the current theory (for instance, by loading a new theory).

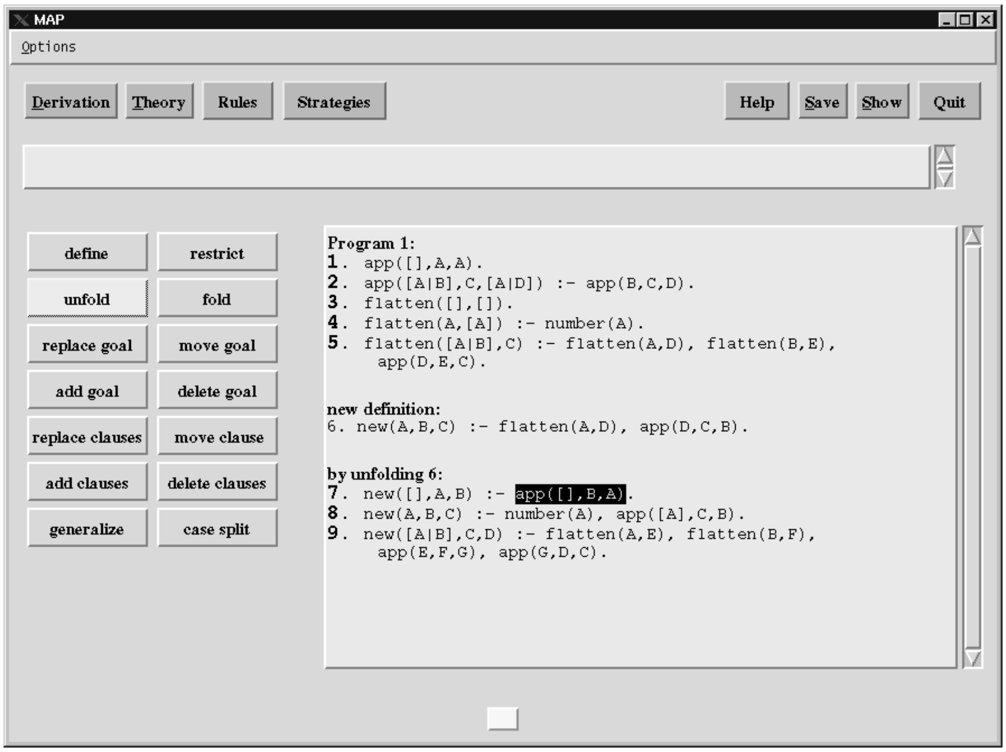


Figure 4: Unfold clause 7 w.r.t. the highlighted atom.

move goal moves the selected atoms within the body of a clause. The position where to move the selected atoms is specified by clicking on a 'reference atom' with the middle button of the mouse. The atoms selected for the 'move goal' rule are placed to the leftmost positions to the right of the reference atom. If the reference atom is the head of the clause, then the selected atoms are moved to leftmost positions of its body. When several atoms are moved their order is preserved.

add goal duplicates the atoms selected by the user in the body of a clause. This operation partially implements the goal addition rule described at Point 7 of Section 2.

delete goal deletes the atoms selected by the user in the body of a clause. This operation is allowed only in the cases indicated in the goal deletion rule at Point 7 of Section 2.

replace clauses opens a text window in which the user can edit new clauses which should replace one or more clauses selected by the user in the current program. No check is performed by the system to ensure that semantics is preserved. A warning message informs the user that the correctness of the corresponding transformation step is left to his responsibility.

move clause moves the clause selected by the user to a new position in the current program. The new position is specified by clicking on another clause, called the 'reference clause', with the middle button of the mouse. The selected clause is moved after the reference

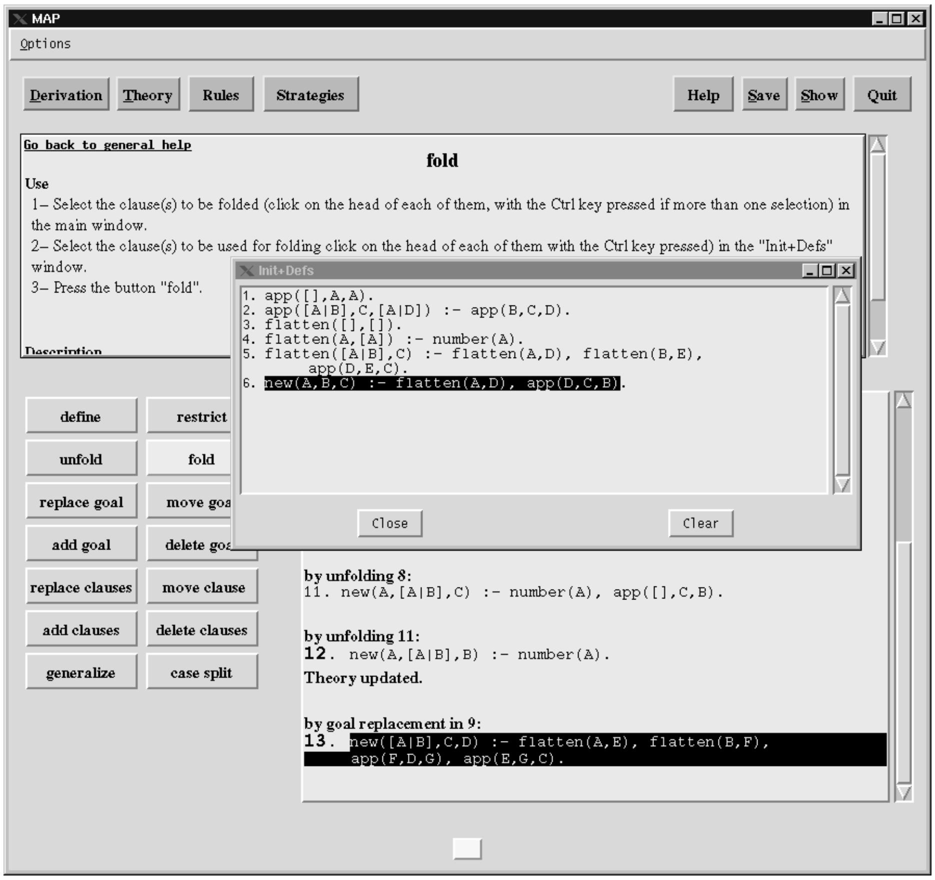


Figure 5: Fold clause 13 using definition clause 6.

clause. If no new position is specified, the selected clause is moved to the first position in the current program.

add clauses opens a text window in which the user can edit one or more clauses to be added to the current program. The system checks whether these clauses are subsumed by other clauses in the current program. If this is not the case, a warning message informs the user that the correctness of the corresponding transformation step is left to his responsibility.

delete clauses deletes the selected clauses from the current program. If the selected clauses are not subsumed by other clauses in the current program and they are not tautologies (see Section 2, Points 10.a and 10.b), then a warning message informs the user that the correctness of the corresponding transformation step is left to his responsibility.

generalize opens a text window, called the 'generalize' window, containing a clause selected by the user from the current program. The user may edit this clause and produce a *generalized* clause by replacing one or more occurrences of a term t by a new variable X. The 'generalize' button in the 'generalize' window applies the generalization rule by replacing in the current program the selected clause by the generalized clause with the equality atom $X = t$ added to its body. The 'close' button cancels the operation.

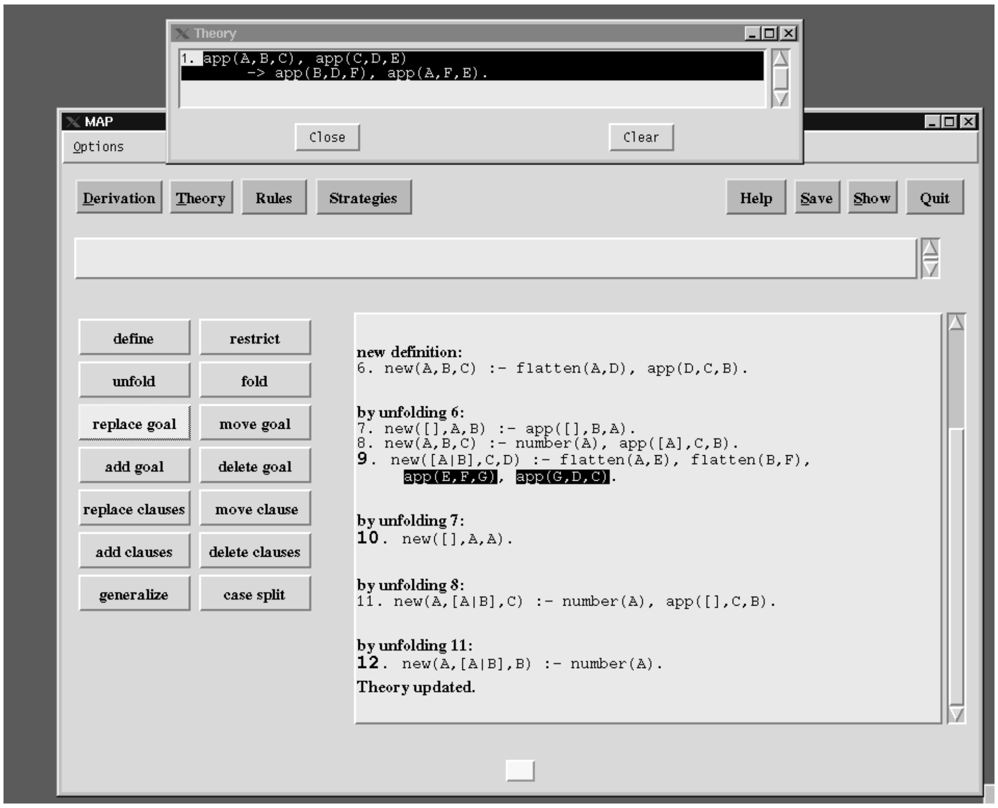


Figure 6: Replace highlighted atoms in clause 9 according to replacement law 1.

case split opens a text window, called 'case split' window, containing the head of a clause selected by the user from the current program. The user is requested to provide an instance of this atom by replacing a variable X by a non variable term t. The 'case split' button in the 'case split' window applies the case split rule to the selected clause w.r.t. the substitution $\{X/t\}$ while the 'close' button cancels the operation.

4.1.5. Help Window

MAP provides the user with a help facility. Help is given in a window which can be active (open) as in Figure 5 or inactive (closed) as in Figure 4. To activate and disactivate the Help window, one has to press the 'Help' button. By default the Help window is closed, and its first activation gives a general help on the use of the system, with links to access some more specific information. If the Help window is open when the user performs a transformation step, then it displays information related to the transformation rules which are applied. Notice also that the Help window may be activated automatically when the user tries to apply a transformation rule with missing or unsuitable paramaters.

4.1.6. Save and Show Menus

'Save' and 'Show' are two pull-down menus whose items gather together the various commands for saving and showing information which are available in other menus, namely the 'Derivation' and 'Theory' menus.

4.1.7. Quit Button

This button exits from the MAP transformation system after a confirmation message.

5. Future Work

We are planning several improvements to the graphical user interface for the MAP system. In particular, it could be interesting to address the following issues:

- Error handling: Before loading a program or a theory, the system should parse the corresponding user's file for preventing Prolog from failing on a syntax error when reading this file. The system should also be able to handle and recover properly from errors due to the failure of a Prolog goal, like for instance, the failure in the application of a transformation rule.

- Graphical view of a derivation: It would be useful to have a graphical representation of the transformation tree under construction, with the possibility of navigating, reading the history, and expanding new branches by means of mouse clicks.

- User guidance: The system could help the user to specify a transformation step, by constraining further choices on the basis of former selections and some knowledge of inappropriate dependencies between parameters. For instance, the button 'unfold' could be disabled if the object selected by the user in the current program is a clause and not an atom.

- Drag and drop transformations: It could be helpful to have the possibility of applying the rules 'move goal' and 'move clause' by means of drag and drop actions. The user would press a mouse button on the goal/clause to be moved, drag it, and drop it to the desired new position.

- Extraction of proof scripts: The system should be able to extract a proof script corresponding to a derivation accomplished via the GUI, and this script should allow the user to replay the transformation for similar initial programs.

- Application of strategies: In the command line system described in [8] three predefined strategies are available: `uve` which implements the strategy for eliminating the so-called unnecessary variables [7], `lap` which implements the Loop Absorption strategy defined in [6], and `determ` which implements the strategy for reducing nondeterminism described in [5]. We are currently working to make these three strategies available through the GUI of the MAP system. This requires, in particular, to provide a convenient graphical way for setting the parameters for these strategies.

More substantial work is needed to enhance the system so that it may offer: 1) the possibility of making use of several sets of transformation rules and strategies (not just one predefined set), 2) the possibility of choosing among several languages and semantics, and 3) some theorem proving capabilities.

For Point 1) we plan to develop libraries which allow the user to load several sets of predefined rules, strategies, and theories into the system. We also plan to design meta-languages that enable the users to define their own rules and strategies, so that the system may work as a generic, programmable program transformer.

For Point 2) we are considering various programming languages, and in particular: (i) general logic programs with negation, (ii) constraint logic programs, and (iii) functional programs.

For Point 3) we intend to include modules that perform automated theorem proving and program analysis. The future MAP system should be able to exploit the information produced by these modules for performing very powerful program transformations whose applicability conditions may depend on the specific properties of the programs at hand.

References

[1] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM*, vol. 24, pp. 44–67, January 1977.

[2] J. W. Lloyd, *Foundations of Logic Programming.* Berlin: Springer-Verlag, 1987. Second Edition.

[3] J. Ousterhout, *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[4] A. Pettorossi and M. Proietti, "Transformation of logic programs: Foundations and techniques," *Journal of Logic Programming*, vol. 19,20, pp. 261–320, 1994.

[5] A. Pettorossi, M. Proietti, and S. Renault, "Reducing nondeterminism while specializing logic programs," in *Proc. 24-th ACM Symposium on Principles of Programming Languages, Paris, France*, pp. 414–427, ACM Press, 1997.

[6] M. Proietti and A. Pettorossi, "The loop absorption and the generalization strategies for the development of logic programs and partial deduction," *Journal of Logic Programming*, vol. 16, no. 1–2, pp. 123–161, 1993.

[7] M. Proietti and A. Pettorossi, "Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs," *Theoretical Computer Science*, vol. 142, no. 1, pp. 89–124, 1995.

[8] S. Renault, "A system for transforming logic programs," R 97–04, Department of Computer Science, University of Rome Tor Vergata, Rome, Italy, 1997.

[9] SICS Programming Systems Group, *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, 1995.

[10] H. Tamaki and T. Sato, "Unfold/fold transformation of logic programs," in *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden* (S.-Å. Tärnlund, ed.), pp. 127–138, Uppsala University, 1984.