EXPANDER

A System for Testing and Verifying
Functional-Logic Programs

April 6, 2002

Peter Padawitz
Fachbereich Informatik
Universität Dortmund, Germany
peter@ls5.informatik.uni-dortmund.de

Contents

1 A uniform proof method

Expander is a proof and test environment for algebraic data type specifications and functional-logic programs. Program design, test and verification are supported by tailor-made deduction rules for proving **Gentzen clauses** and **guarded Horn clauses** (cf. [Pad 91], [Pad 92], [Pad 96]). These are the most frequently used formulas for expressing requirements and queries to data types. Inductive proofs are built up from simplification, subsumption, resolution, rewriting and narrowing steps in a natural way that permits interaction on various levels. The overall proof/test procedure works as follows.

An initial conjecture $ini\_conc \Leftarrow ini\_prem$ is proved by constructing stepwise two lists of goal sets:

$$front = [conc,\ gs_k, \ldots,\ gs_1,\ ini\_conc] \quad \text{and} \quad rear = [prem,\ hs_n, \ldots,\ hs_1,\ ini\_prem]$$

front represents a backward proof of $ini\_conc \Leftarrow conc$, while *rear* yields a forward proof of $prem \Leftarrow ini\_prem$. The proof of $ini\_conc \Leftarrow ini\_prem$ is complete as soon as *prem* subsumes *conc*. Subsumption is a purely syntactical condition on *prem* and *conc* that implies $conc \Leftarrow prem$ (cf. 4.5(5)). If, in particular, if $conc = FALSE$ then $\neg ini\_conc$ is valid. Testing *ini_conc* means deriving solutions of *ini_conc*. Solutions are identified as the equations in **solved form** among the the goals of *conc* (cf. Section 4.4).

When building up *front* and *rear* one may interactively modify the underlying specification and add lemmas or induction orderings. The overall goal is to bring *prem* and *conc* "closer together" so that, eventually, *prem* subsumes *conc*. The construction of *front* and *rear* may be interrupted whereby $ini\_conc \Leftarrow conc$ and $prem \Leftarrow ini\_prem$ are identified as proved subtheorems and $conc \Leftarrow prem$ is memorized as the remaining subconjecture. The current induction ordering is transmitted to a later proof of the subconjecture. If the underlying specification is **functional** in the sense of [Pad 95] or **canonical** in the sense of [Pad 96], then a number of additional rules are applicable that often "speed up" the proof construction considerably.

Expander is written in Standard ML (cf. [Pau 91]). The source code should be translated by the New Jersey compiler, Version 0.93. After directories *Specifications, Proofs* and *Graphs* have been created, the ML function call *Install "Specifications" "Proofs" "Graphs" "EXPANDER"* exports the target code to the executable file *EXPANDER*.

Expander's specification language is defined in Section 2. The inference rules underlying the commands for building up proofs and tests are listed in Section 3. The commands themselves are described in Section 4. Automatic formula simplifications are presented in Section 5. Error messages are listed in Section 6. Sample proofs and tests of conjectures given in Section 2 are carried out in Section 7. Section 8 presents Expander's entire source code.

2 Design language

2.1 Syntax and semantics

Each specification to be processed by Expander is generated by the following context-free grammar in extended Backus-Naur form. *, | and ?[1] denote the usual operators of regular expressions. [and] are used for bracketing subexpressions. Keywords are written in `typewriter` style.

```
specification        ::= spec_name signature [ axioms clause* ]?
                         [ theorems clause* ]?
                         [ conjects clause* ]?
spec_name            ::= string
signature            ::= [ base spec_name* ]?
                         [ consts [ construct | in_construct ]* ]?
                         [ functs funct_def* ]?
                         [ preds pred_def* ]?
                         [ infixes [ in_funct | in_pred ]* ]?
                         [ vars var* ]?
funct_def            ::= [ funct | in_funct ] number*
pred_def             ::= [ pred | in_pred ] number*
clause               ::= ( number ) [ goal_set | goal_set <== goal ]
goal_set             ::= goal | goal \/ goal_set | FALSE
goal                 ::= { atoms }
atoms                ::= atom | atom, atoms
atom                 ::= pred tuple | term restatom
restatom             ::= in_pred term
terms                ::= term | term , terms
term                 ::= [ number | tuple | list | bag | set | map ] infix_rest |
                         [ construct | funct | var | var? | var! | var/\ ] restterm
tuple                ::= ( terms )
list                 ::= [ terms ][2]
bag                  ::= {{ terms }}
set                  ::= { terms }
map                  ::= << assignments >>
assignments          ::= assignment | 0 | assignments, map
assignment           ::= term < − term
restterm             ::= tuple restterm | infix_rest
infix_rest           ::= [ in_construct | in_funct ] term | ε
pred                 ::= string
in_pred              ::= standard_in_pred | string
standard_in_pred     ::= = | <> | < | > | <= | >= | in | notIn | sub
construct            ::= standard_construct | string
standard_construct   ::= true | false | 0 | s | nil | bag | set | upd | rect
in_construct         ::= :: | string
funct                ::= standard_funct | string
standard_funct       ::= ite | not | eq | neq | leq | geq | lt | gt | In | NotIn | Sub | min |
                         max | length | fn | cont | tup_C | apply_C | rewrite | rewriteAll
```

[1] the reflexive closure

in_funct	::=	standard_in_funct | string
standard_in_funct	::=	`and` | `or` | `+` | `-` | `*` | `/` | `//` | `mod` | `@` | `at` | >>
var	::=	string
comment	::=	`(*` string `*)`
number	::=	*any natural number*
string	::=	*any string*

All built-in and user-defined symbols are separated from each other by blanks and the following characters:

, ( ) { } [ ] < > = ~ + - * / ? ! & @ # % § $ ^ \n \t

The standard constructors, functions and predicates are explained in Section 2.2. >> always denotes an induction ordering used in an inductive proof. The axioms for >> depend on the actual proof and thus have to be given by the specifier. Non-standard constructors, functions or predicates occurring in axioms, theorems or conjectures, must be listed after the keywords **`consts`**, **`functs`** and **`preds`**, respectively. By default, they are prefix symbols. Those used as infix symbols must also be listed after the keyword **`infixes`**. Each symbol occurring in the *funct_def* or *pred_def* list is followed by the sequence of numbers denoting the axioms that define the respective function or predicate. Constructors are not allowed to have axioms. Expander handles standard types with **`non-free constructors`** such as sets, bags and maps in a particular way (cf. Sections 2.2 and 4).

The grammar rules

$$\begin{array}{lll} \text{term} & ::= & [\ \text{funct} \mid \text{var}\ ]\ \text{restterm} \\ \text{restterm} & ::= & \text{tuple restterm} \end{array}$$

admit **higher-order terms**. Expander translates them into first-order terms. For instance, the term $t = F(t_1,\dots,t_k)(u_1,\dots,u_n)$ is represented as

$$\texttt{APPLY}(F(t_1,\dots,t_k),u_1,\dots,u_n) \quad \text{or} \quad \texttt{APPLY(APPLY}(F,t_1,\dots,t_k),u_1,\dots,u_n)$$

if F is a function symbol or a variable, respectively. The term $F(t_1,\dots,t_k)$ can be unified with $\texttt{APPLY}(G,t_1,\dots,t_k)$ if G is a variable. This is Expander's only modification of first-order unification for handling higher-order terms. In general, constructors for function domains are forbidden and thus any unification of two higher-order terms reduces to matching one of them against the other.

A **λ-expression** $\lambda x.e$ is written as $fn(x,e)$. β-reduction is part of Expander's simplifier. Similarly, a term t is evaluated by **continuation passing** if t is embedded into the term $cont(t,fn(x,x))$ (cf. Section 5.3).

Let V and Z be sets of variables. A **goal** is a universally quantified conjunction $\forall Z(p_1\wedge\ldots\wedge p_n)$ of atom(ic formula)s. The empty conjunction is denoted by *TRUE*. A **goal set** is an existentially quantified disjunction $\exists V g_1\vee\ldots\vee g_k$ of goals. The empty conjunction is denoted by *FALSE*.[3] Given goals g and h, $g\Leftarrow h$ is a **Horn clause** with conclusion g and premise h. All axioms of a specification SP are Horn clauses. Given a goal set gs and a goal h, $gs\Leftarrow h$ is called a **Gentzen clause** with conclusion gs and premise h. All theorems and conjectures of SP are Gentzen clauses. Horn clauses are Gentzen clauses, but not vice versa. A Gentzen clause $FALSE\Leftarrow h$ stands for $\neg h$.

All variables used in a specification are listed after the keyword **`vars`**. Given a variable x, all occurrences of x, $x?$ and $x!$ are treated as the same variable. $x!$ (resp. $x\wedge$) denotes an existentially (resp. universally) quantified

[2] The brackets `[` and `]` are terminal symbols of the language defined.

[3] *FALSE* is a goal set, while `false` is a term (cf. 2.2).

variable whose scope is the goal where it occurs. $x!$ (resp. $x\wedge$) never occurs in the premise (resp. conclusion) of an Gentzen clause.

$x?$ denotes a **read-only variable**. $x?$ is substituted by a non-variable term only if that term is part of a unifier of two atoms or terms, which contain $x!$ or x, but not $x?$. If $x?$ occurs in a goal of a conjecture conclusion, then x is automatically existentially quantified, i.e. the same goal must also contain $x!$. Read-only variables may only occur in premises of axioms, in right-hand sides of equational axiom conclusions and in conjecture conclusions.

The axioms of SP are given by the clauses in the `axioms` list of SP and by the axioms of the specifications in the `base` list of SP (cf. Section 3).

A Gentzen clause is **SP-valid** if it holds true in all **initial SP-models over reachable SP'-models** where SP' is the subspecification of SP "generated" by the parameter sorts of SP (cf. [Pad 95a]). The `theorems` list SP may be used both to state **lemmas**, i.e. clauses whose SP-validity is already known, and to fix **constraints**, i.e. clauses that, besides the axioms, further restrict the class of models of SP: if constraints C are used in the proof of a conjecture c, the validity of c can certainly be guaranteed only for SP-models satisfying C.

Expander does neither check nor infer types of the functions and predicates of SP. For complying with the above described semantics all functions and predicates of SP are supposed to be implicitly well-typed.

2.2 Types and traits

The objects of a data type T are represented as ground terms over constructors. In general, different constructor terms stand for different objects of type T. This defines the equality relation `=` and its negation `<>`. These relations are also available as the Boolean functions `eq` and `neq`, respectively. Expander provides the following standard types and type constructors:

- Boolean constants `true` and `false`, natural numbers, tuples, finite lists, bags (multisets), sets and maps (functions with finite domain).

Standard Boolean functions are `not`, `and` and `or`, defined as usual. Arithmetic, list, bag and set comparators are available both as relations and Boolean functions:

- $<$ and `lt`, $>$ and `gt`, $<=$ and `leq`, $>=$ and `geq`, `in` and `In` (membership), `notIn` and `NotIn` (negation of membership), `sub` and `Sub` (set inclusion).

A Boolean term, i.e. a term with a Boolean function as the leftmost operator, may be used as the first argument of the ternary conditional `ite` (if-then-else).

The constructors for natural numbers are the constant `0` and the successor function `s`. Any natural number may be used in a specification. The standard functions on natural numbers are `+, -, *, min, max, mod` (remainder), / (natural number division), // (natural number division if the remainder is zero, otherwise // adds 1 to the quotient).

As in ML, the list constructors are the empty list `nil` and the append-left (or *cons*) operator `::`. A constructor term of the form `x1::x2::...::nil` may be written as `[x1,x2,...,nil]`. Besides the above-mentioned Boolean functions Expander provides the standard list function @ (concatenation).

For `cont`, `tup_C` and `apply_C`, cf. Section 5.3.

Let us compare Expander's types with specification constructs provided by the *Larch Shared Language* LSL (cf. [GH 93]). First we note that the *assertions* of a Larch trait T correspond to the axioms of an Expander specification SP, the *assumptions* of T are the theorems of SP and the *implications* are the conjectures of SP. LSL's *generated, converts* and *partitioned* clauses can be formulated in SP as follows.

s generated by $F_1,\ldots,F_n$ means that each ground term of the sort s is semantically equivalent to a ground term over the constructors $F_1,\ldots,F_n$. This requirement can be expressed as the conjecture *Gen(x)* where the sorted predicate *Gen* is specified by an axiom

$$Gen(F_i(x_1,\ldots,x_k)) \Leftarrow Gen(x_1) \wedge \ldots \wedge Gen(x_k)$$

for each $1 \leq i \leq n$.

converts F exempting F(t) means that all ground terms of the form *F(u)*, except those matching *F(t)*, are equivalent to ground terms over constructors[4]. We may express this clause as the conjecture *Gen(F(x))* after *Gen(F(t))* has been added to the axioms of *Gen* given above.

s partitioned by $G_1,\ldots,G_n$ means that two ground terms t and t' of sort s are considered to be equivalent unless there is an invalid ground equation of the form $G_i(u,t,v) = G_i(u,t',v)$. Hence different constructor terms become equivalent, i.e. s denotes a data type with *non-free constructors*. The most frequently used types with non-free constructors ones: sets, bags and maps (see above), are built into Expander as standard types. The equality and inequality relations on these types are handled by the system's simplifier (cf. Section 4.1). Since simplification may change the structure of a term involving set, bag or map constructors, the specifier must ensure that the axioms defining set, bag or map functions or predicates respect the equivalence of constructor terms that represent the same set, bag or map, respectively. The possible constructors read as follows:

- `nil` denotes the empty set. `set` turns a list into a set, i.e. if two lists L and L' contain the same elements, then `set`(L) is equivalent to `set`(L'). `set([x1,...,xn])` may be written as `{x1,...,xn}`.
- `nil` denotes the empty bag. `bag` turns a list into a bag, i.e. if a list L is a permutation of a list L', then `bag`(L) is equivalent to `bag`(L'). `bag([x1,...,xn])` may be written as `{{x1,...,xn}}`.
- `nil` denotes the map defined nowhere. `upd(x,e,f)` denotes the map obtained from the map f by changing its value at x into e. `Map` turns a map into its graph, i.e. $\mathtt{Map}(nil) =_{def} \emptyset$ and $\mathtt{Map}(upd(x,e,f)) =_{def} \{(x,e)\} \cup \{(y,e') \in \mathtt{Map}(f) \mid y \neq x\}$. Two map terms f and g are equivalent if $\mathtt{Map}(f) = \mathtt{Map}(g)$. If $\mathtt{Map}(f) = \{(\mathtt{x1},\mathtt{e1}),\ldots,(\mathtt{xn},\mathtt{en})\}$, then f may be written as `<<x1<-e1,...,xn<-en>>`.

The membership predicate `in`, its negation `notIn` and their functional counterparts `In` and `NotIn` (see above) are defined on sets, bags and maps as well as on lists. In the case of a map f the term x `in` f means that f is defined on x, i.e. $(x,e) \in \mathtt{Map}(f)$ for some e. `at` denotes the lookup function for maps, i.e. if $(x,e) \in \mathtt{Map(f)}$, then the term f `at` x is equivalent to e. Otherwise f `at` x reduces to the term `noEntryAt`(x). Set inclusion is denoted by the predicate `sub` and the Boolean function `Sub`.

The standard function `rewrite` is defined on a single list of elements of an arbitrary fixed sort. The standard function `rewriteAll` has a variable arity and cooperates with `rewrite` in a particular way: a term of the form `rewrite`$([t_1,\ldots,t_n])$ unifies with a term of the form `rewriteAll`$(u_1,\ldots,u_k)$ iff there are $1 \leq i_1,\ldots,i_k \leq n$ such that for all $1 \leq j \leq k$, t_{i_j} unifies with u_j. The following specification inspired by an example of [Mes 93] shows how these functions can be used for realizing associative-commutative rewriting:

```
ACCOUNT
        consts   obj credit debit transfer Peter Paul Mary
        functs   rewrite 1 2 3
        vars     A B k m n L
```

[4]LSL's additional requirement that these ground terms are unique up to semantic equivalence is always satisfied because equality in the initial model of a Horn clause specification *is* semantic equivalence.

```
axioms   (1) { rewriteAll(credit(A,m),obj(A,n)) = rewrite(obj(A,n+m)::REST) }
         (2) { rewriteAll(debit(A,m),obj(A,n)) = rewrite(obj(A,n-m)::REST) } <== { n >= m }
         (3) { rewriteAll(transfer(m,A,B),obj(A,k),obj(B,n))
                = rewrite(obj(A,k-m)::(obj(B,n+m)::REST)) } <== { k >= m }
conjects (1) { rewrite([obj(Peter,300),obj(Paul,250),debit(Peter,200),debit(Paul,50),
               credit(Paul,300),credit(Paul,300),obj(Mary,560),credit(Mary,250)]) = L }
```

When one of the above axioms is applied, the standard variable **REST** is instantiated by the list of all elements of the argument of **rewrite** that are not unifiable with arguments of **rewriteAll**. In terms of the above definition, these are all $t_i \in \{t_1, \ldots, t_n\}$ with $i \neq i_j$ for all $1 \leq j \leq k$. Using Expander's *Solve* command (cf. 4.4) Conjecture 1 of ACCOUNT can be reduced to the narrowed goal

$$\{ \texttt{rewrite([obj(Mary,810),obj(Peter,100),obj(Paul,800)]) = L} \}.$$

Types with non-free constructors as well as infinite types such as streams (cf. 2.3.8) can also be defined implicitly as **action types** (cf. [Pad 95]).

Besides handling *assumes, implies, generated, converts* and *partitioned* the *Larch prover* checks the consistency of a Larch trait T, i.e. tries to derive $true \neq false$ from T. In general, a specification is consistent if it has a model. Since the axioms of an Expander specification SP are Horn clauses, it always has the initial model. LSL's proof obligation of consistency stems from the fact that T is purely equational and thus all logical formulas of T are in fact Boolean terms. Expander admits Boolean functions as well as "real" predicates. SP can be inconsistent in the sense that $true = false$ is derivable from SP only if Boolean functions are involved. LSL's consistency check means proving the Gentzen clause $FALSE \Leftarrow true = false$ (cf. 2.1). This needs rarely be proved directly. Instead, if SP is functional (cf. [Pad 95]) or canonical (cf. [Pad 96]), then $true \neq false$ is automatically valid. Instead of turning a predicate P into a Boolean function (which may cause an inconsistency) one may specify the negation of P as a separate predicate $\neg P$. Then consistency amounts to the proof obligation that $P(x) \wedge \neg P(x)$ is SP-valid.

Let us illustrate the syntax of Expander by a couple of small examples. The following specifications are referred to in Section 9 where sample Expander sessions can be found.

2.3 Sample specifications

2.3.1 *Even and odd*

```
EVEN_ODD
       functs     double 1 2  >> 5
       preds      Def 3 4
       vars       x y ex ey
       axioms      (1) { double(0) = 0 }
                   (2) { double(s(x)) = s(s(double(x))) }
                   (3) { Def(0) }
                   (4) { Def(s(x)) } <== { Def(x) }
                   (5) { s(x) >> x = true }
       conjects    (1) { x = double(ex!) } \/ { x = s(double(ey!)) } <== { Def(x) }
```

Conjecture 1 says that each natural number is even or odd. Its inductive proof given in Example 7.1 is based on the induction ordering defined by Axiom 5.

2.3.2 *Less or equal*

```
LEQ
```

```
        functs      min 4 5  >> 3
        preds       <= 1 2
        vars        x y y' z z'
        axioms       (1) { 0 <= x }
                     (2) { s(x) <= s(y) } <== { x <= y }
                     (3) { (s(x),z) >> (x,z') = true }
        conjects     (1) { x <= z } <== { x <= y, y <= z }
                     (2) FALSE <== { x <= y, x > y }
```

The proof of Conjecture 1 given in Example 7.2 is based on the induction ordering defined by Axiom 3.

2.3.3 *Mergesort*

```
MSORT
        functs      mergesort 4 5 6  merge 7 8 9 10  split 11 12 13
        preds       sorted 1 2 3
        vars        x y w l r
        axioms       (1) { sorted(nil) }
                     (2) { sorted([x]) }
                     (3) { sorted(x::(y::w)) } <== { x <= y, sorted(y::w) }
                     (4) { mergesort(nil) = nil }
                     (5) { mergesort([x]) = [x] }
                     (6) { mergesort(x::(y::w)) = merge(mergesort(x::l),mergesort(y::r)) }
                           <== { split(w) = (l,r) }
                     (7) { merge(nil,w) = w }
                     (8) { merge(w,nil) = w }
                     (9) { merge(x::l,y::r) = x::merge(l,y::r) } <== { x <= y }
                    (10) { merge(x::l,y::r) = y::merge(x::l,r) } <== { x > y }
                    (11) { split(nil) = (nil,nil) }
                    (12) { split([x]) = ([x],nil) }
                    (13) { split(x::(y::w)) = (x::l,y::r) } <== { split(w) = (l,r) }
```

The correctness conditions on mergesort and the lemmas needed in their proof are listed in the following extension of MSORT:

```
MSORT_P
        base        MSORT
        preds       Q 1
        vars        l' r'
        axioms      (1) { Q(l,r) } <== { sorted(r), bag(l) = bag(r) }
        theorems    (1) { sorted(merge(l,r)) } <== { sorted(l), sorted(r) }
                    (2) { bag(merge(l,r)) = bag(l@r) }
                    (3) { bag(l@l') = bag(r@r') } <== { bag(l) = bag(r), bag(l') = bag(r') }
                    (4) { bag(l@r) = bag(w) } <==  { split(w) = (l,r) }
                    (5) { bag(x::y::w) = bag(y::x::w) }
        conjects    (1) { sorted(r), bag(l) = bag(r) } <== { mergesort(l) = r }
                    (2) { Q(nil,nil) }
                    (3) { Q([x],[x]) }
                    (4) { Q(x::y::w,merge(l',r')) } <== { split(w) = (l,r), Q(x::l,l'), Q(y::r,r') }
```

bag is a built-in constructor for multisets (cf. Section 2.2). Conjecture 1 is obtained from Conjectures 2 to 4 by applying the *fixpoint induction rule* of [Pad 95]. This rule transforms entire Gentzen clauses and not only goal sets and thus does not fit into the Expander's current proof schema described in Section 1. Gentzen clause rules will first be provided by the next version of Expander. Here Conjecture 1 has been turned into Conjectures 2 to 4 still "by hand". Example 7.3 presents the proofs of Conjectures 2 to 4.

2.3.4 *Replace by minimum*

```
REPMIN
        consts       % #
        functs       rep_and_min 1 2  rep_by_min 3
        infixes      #
        vars         t t1 t2 t3 t4 x m m1 m2
        axioms        (1) { rep_and_min(%(x),m) = (%(m),x) }
                      (2) { rep_and_min(t1#t2,m) = (t3#t4,m3) }
                            <== { rep_and_min(t1,m) = (t3,m1),
                                  rep_and_min(t2,m) = (t4,m2), min(m1,m2) = m3 }
                      (3) { rep_by_min(t) = t1 } <== { rep_and_min(t,m) = (t1,m) }
        conjects      (1) { rep_by_min((%(5)#%(4))#(%(3)#%(2))) = t }
```

`REPMIN` defines binary trees with natural number entries in the leafs. The function `rep_by_min` replaces all leaf entries of a tree by the minimum of these entries. Note that the premise equation of Axiom 3 contains a "feedback" variable `m`, i.e. `m` occurs on both sides of the equation. Here Conjecture 1 is not to be proved, but to be solved (in `t`). In this case the solution process amounts to pure rewriting (cf. 7.4).

2.3.5 *Find paths*

```
ROBOT
        preds        Path 1 2 3
        vars         p w
        axioms        (1) { Path(x,y,i,j,(x,y)::p,w) }
                            <== { x < i, (x,y) notIn w, Path(s(x),y,i,j,p,w) }
                      (2) { Path(x,y,i,j,(x,y)::p,w) }
                            <== { y < j, (x,y) notIn w, Path(x,s(y),i,j,p,w) }
                      (3) { Path(x,y,x,y,[(x,y)],w) } <== { (x,y) notIn w }
        conjects      (1) { Path(1,2,5,4,p,[(1,3),(2,4)]) }
```

As in the previous example, Conjecture 1 is to be solved, here in `p`. Each solution denotes a path from (1,2) to (5,4), which does not include the points (1,3) and (2,4). In contrast to Conjecture 1 of `REPMIN`, the solution process amounts to pure resolution, i.e. executing the logic program given by the axioms of `ROBOT`.

2.3.6 *Mutual exclusion*

```
MUTEX
        consts       alpha beta
        functs       act 1 2 3 4
        preds        invariant 5 6 7
        vars         i x y d d' x1 y1 d1 d1'
        axioms        (1) { act(1)(x,0,alpha,d') = (1,0,beta,d') }
                      (2) { act(1)(x,y,beta,d') = (0,y,alpha,d') }
                      (3) { act(2)(0,y,d,alpha) = (0,1,d,beta) }
                      (4) { act(2)(x,y,d,beta) = (x,0,d,alpha) }
                      (5) { invariant(0,0,alpha,alpha) }
                      (6) { invariant(1,0,beta,alpha) }
                      (7) { invariant(0,1,alpha,beta) }
        conjects      (1) { invariant(x1,y1,d1,d1') }
                            <== { invariant(x,y,d,d'), act(i)(x,y,d,d') = (x1,y1,d1,d1') }
```

`MUTEX` specifies the effect of two processes `act(1)` and `act(2)` on two dynamic variables x and y and two devices `alpha` and `beta`. Axioms 5 to 7 list the admissable "invariant" states of the system. Each state consists of the actual values of x and y and the devices d and d' actually used by `act(1)` and `act(2)`, respectively. Conjecture 1 says that invariant states remain invariant under each state transition performed by `act(1)` or `act(2)`.

Since `invariant` excludes the states where `act(1)` and `act(2)` use `beta` simultaneously, mutual exclusion is guaranteed.

2.3.7 *List partitions*

```
PARTITION
        functs      flatten 4 5
        preds       part 1 2 3  Q 6
        vars        x y L L' P
        axioms       (1) { part([x],[[x]]) }
                     (2) { part(x::y::L,[x]::P) } <== { part(y::L,P) }
                     (3) { part(x::y::L,(x::L')::P) } <== { part(y::L,L'::P) }
                     (4) { flatten(nil) = nil }
                     (5) { flatten(L::P) = L@flatten(P) }
                     (6) { Q(P,L) } <== { L = flatten(P) }
        conjects     (1) { L = flatten(P) } <== { part(L,P) }
                     (2) { Q([x],[[x]]) }
                     (3) { Q(x::y::L,[x]::P) } <== { Q(y::L,P) }
                     (4) { Q(x::y::L,(x::L')::P) } <== { Q(y::L,L'::P) }
```

The first argument of `part` is a nonempty list L. The second argument yields a partition of L into sublists. Axiom 1 defines the case where L consists of a single element x. Then the only partition of L has L as its single element. Axioms 2 and 3 deal with lists starting with two elements x and y. Here L denotes the rest of the list. Both clauses involve the same recursive call: remove x from the list and get a partition – P resp. $L' :: P$ – of the remaining list $y :: L$. Axiom 2 extends P to a partition of the entire list $x :: y :: L$ by appending the sublist $[x]$ to P. Axiom 3 extends $L' :: P$ to a partition of $x :: y :: L$ by appending the sublist $[x :: L']$ to P. Here L' stands for the first element of the partition of $y :: L$ that returns from the recursive call. Analogously to Example 2.3.3, Conjecture 1 is obtained from Conjectures 2 to 4 imply by applying the fixpoint induction rule. Example 7.7 presents a proof of Conjectures 2 to 4.

2.3.8 *Fair streams*

```
FAIR
         consts      OI IO &
         functs      head 6 8 10  tail 7 9 11  # 4 5
         preds       fair 1  fair' 2 3  Q 12 13
         infixes     & #
         vars        st st' i x L
         axioms       (1) { fair(st) } <== { fair'(i/\,st) }
                      (2) { fair'(0,st) }
                      (3) { fair'(s(i),st) } <== { st = L#(0&st'), fair'(i,st') }
                      (4) { nil#st = st }
                      (5) { (x::L)#st = x&(L#st) }
                      (6) { head(OI) = 0 }
                      (7) { tail(OI) = IO }
                      (8) { head(IO) = 1 }
                      (9) { tail(IO) = OI }
                     (10) { head(x&st) = x }
                     (11) { tail(x&st) = st }
                     (12) { Q(OI) }
                     (13) { Q(IO) }
         theorems     (1) { 0&IO = OI }
                      (2) { 1&OI = IO }
         conjects     (1) { fair(OI), fair(IO) }
                      (2) { st = L!#(0&st'!), Q(st'!) } <== { Q(st) }
```

& appends a number to the left of a stream. OI and IO are the two streams whose elements alternate between zeros and ones. # concatenates a list and a stream into a stream. `fair`(s) holds true if s contains infinitely many zeros. Conjecture 2 implies Conjecture 1. 1 is obtained from 2 by applying the *fixpoint coinduction rule* of [Pad 95]. Fixpoint coinduction is dual to fixpoint induction (cf. 2.3.3) and also defined on entire Gentzen clauses so that it is not provided by the current version of Expander. Example 7.8 presents a proof of Conjecture 2.

3 Rules

The commands described in Sections 4.2 to 4.5 use axioms, theorems or conjectures of SP to extend *front* or *rear* (cf. Section 1). In all cases a Gentzen clause

$$\exists X_1 u = u_1 \vee \ldots \vee \exists X_k u = u_k \vee \exists X_{k+1} u_{k+1} \vee \ldots \vee \exists X_l u_l \quad \Leftarrow \quad h \tag{c}$$

is applied to the head of *front* or *rear*. Note that $u, u_1, \ldots, u_k$ are terms, while $u_{k+1}, \ldots, u_l$ are goals. Section 4.2 deals with the case $k = 0$, i.e. the exclusive replacement of *atoms*. Section 4.3 handles the case $k = l$, i.e. the exclusive replacement of *terms*. For all $1 \leq i \neq j \leq l$, the sets X_i and X_j of variables are supposed to be disjoint. If c does not satisfy this condition, c can always be transformed into a clause that satisfies it. Expander carries out this transformation whenever c is actually applied. Moreover, the applicability conditions of each rule are checked automatically. $\Uparrow$ and $\Downarrow$ denote the direction in which each rule establishes an implication. $\Uparrow$-rules are called **expansions** or **generalizations**. $\Downarrow$-rules are called **contractions** or **specializations**. Further notations used in the rules are defined first.

For a substitution σ, $\mathbf{dom}(\sigma)$ is the set of variables x with $x\sigma \neq x$ and $\mathbf{EQ}(\sigma)$ is the set of equations $x = x\sigma$ such that $x \in dom(\sigma)$. Given a term or goal t, $x \in var(t)$ and σ with $dom(\sigma) \subseteq \{x\}$, we sometimes use the functional notations $t(x)$ and $t(x\sigma)$ instead of t and $t\sigma$, respectively. The restriction $\sigma_{\mathbf{V}}$ of σ to a set V of variables is defined by $x(\sigma_V) = x\sigma$ for $x \in V$ and $x(\sigma_V) = x$ otherwise. Given a further set Z of variables, σ is a **renaming of** V **away from** Z if $V\sigma$ is a set of variables with the cardinality of V and $V\sigma \cap Z = \emptyset$. A term t **matches** or **subsumes** an expression u, written $t \leq u$, if $t\sigma = u$ for some substitution σ. σ is **most general** or **minimal** w.r.t. some condition C if σ subsumes all substitutions satisfying C. t **unifies** with u if $t\sigma = u\sigma$ for some σ. For a goal set gs, $\mathbf{in}(\mathbf{gs})$ denotes the set of input (= universally quantified) variables of c.

With c as above, the rules applied by Expander commands read in their most general form as follows.

Deductive expansion upon c

$$\frac{gs}{g_1(u)\sigma \wedge \ldots \wedge g_k(u)\sigma \wedge g_{k+1}\sigma \wedge \ldots \wedge g_l\sigma \wedge h\sigma \wedge EQ(\sigma_{in(gs)})} \Uparrow$$

if $gs = g_1(t_1) \vee \ldots \vee g_k(t_k) \vee (g_{k+1} \wedge t_{k+1}) \vee \ldots \vee (g_l \wedge t_l)$ and for all $1 \leq i \leq l$, σ is a renaming of X_i away from $var(c)$ and a minimal unifier of t_i and a subset of u_i and $X_i\sigma \cap (in(gs) \cup var(in(gs)\sigma \cup g_i(u)\sigma)) = \emptyset$

We say that c is applied to the **redex** $(t_1, \ldots, t_l)$.

Greedy expansion upon c is a variant of deductive expansion that is applicable even if t_i and u_i are not unifiable. More precisely, let $F(t)$ and $G(u)$ be the leftmost-outermost subterms with different roots at corresponding positions of t_i resp. u_i. Let σ be the substitution obtained from the first steps to unify t_i and u_i before $F(t)$ and $G(u)$ were reached. If F is a defined function, G is a constructor and $gs \neq gs\sigma$, then greedy expansion returns the instance of hs by σ:

$$\frac{gs}{gs\sigma \wedge EQ(\sigma_{in(gs)})} \Uparrow$$

In all other cases greedy expansion responds the same as deductive expansion.

Deductive expansion applies c backward to *conc*, i.e. expansion steps build up *front*. The counterpart, called *deductive contraction*, applies c forward to *prem*, i.e. contraction steps build up *rear*.

Deductive contraction upon c

$$\frac{g(t) \wedge f}{(g(t_1) \wedge t = t_1) \vee \ldots \vee (g(t_k) \wedge t = t_k) \vee (g(t) \wedge t_{k+1}) \vee \ldots \vee (g(t) \wedge t_l)} \Downarrow$$

if $t = u\sigma$, $f = h\sigma$ and for all $1 \leq i \leq l$, σ is a renaming of X_i away from $var(c)$ and $t_i = u_i\sigma$ and $X_i\sigma \cap var(g(t) \cup f) = \emptyset$

We say that c is applied to the **redex** (t, f). Note that (u, h) is supposed to *match* (t, f), while deductive expansion admits a proper *unifier* of u_i and t_i.

Induction steps along a given induction ordering are deductive expansion or contraction steps *upon the conjecture to be proved* (cf. [Pad 91], [Pad 92], [Pad 96]). Given a Horn clause specification SP, any binary predicate $\gg: ww \in SP$ whose interpretation in some SP-model is well-founded may serve as an induction ordering.

Suppose that CS is a set of Gentzen clauses, $c \in CS$ has the form (1), $\vec{x} = (x_1, \ldots, x_n) \in X^+$ and ρ is a renaming of $V = \{x_1, \ldots, x_n\}$ away from V with $dom(\rho) = V$. Given that the following rules are employed in a Gentzen proof of a clause c, then V is supposed to be the set of input variables of c. Since, in general, $c \in CS$, i.e. c may be applied in its own proof, we have to distinguish V from the input variables of the variants of c that are applied in the proof of c. This distinction is accomplished by the renaming ρ.

Inductive expansion upon c

$$\frac{gs}{g_1(u\rho)\sigma \wedge \ldots \wedge g_k(u\rho)\sigma \wedge g_{k+1}\sigma \wedge \ldots \wedge g_l\sigma \wedge h\rho\sigma \wedge EQ(\sigma_{in(gs)}) \wedge \vec{x} \gg \vec{x}\rho\sigma} \Uparrow$$

if $gs = g_1(t_1) \vee \ldots \vee g_k(t_k) \vee (g_{k+1} \wedge t_{k+1}) \vee \ldots \vee (g_l \wedge t_l)$ and for all $1 \leq i \leq l$, σ is a renaming of X_i away from $var(c)$ and a minimal unifier of t_i and a subset of $u_i\rho$ and $X_i\sigma \cap (in(gs) \cup var(in(gs)\sigma \cup g_i(u\rho)\sigma)) = \emptyset$

Inductive contraction upon c

$$\frac{g(t) \wedge f}{\begin{array}{c}(g(t_1) \wedge t = t_1) \vee \ldots \vee (g(t_k) \wedge t = t_k) \vee (g(t) \wedge t_{k+1}) \vee \ldots \vee (g(t) \wedge t_l) \\ \vee\ (g(t) \wedge f \wedge \vec{x} \not\gg \vec{x}\rho\sigma)\end{array}} \Downarrow$$

if $t = u\rho\sigma$, $f = h\rho\sigma$ and for all $1 \leq i \leq l$, σ is a renaming of X_i away from $var(c)$ and $t_i = u_i\rho\sigma$ and $X_i\sigma \cap var(g(t) \cup f) = \emptyset$

Inductive expansion applies c to the **redex** $(t_1, \ldots, t_l)$. Inductive contraction applies c to the **redex** (t, f). Both rules differ from *deductive* expansion/contraction insofar as the **descent condition** $\vec{x} \gg \vec{x}\rho\sigma$ resp. $\vec{x} \not\gg \vec{x}\rho\sigma$ has been added to the rule succedent. Inductive contraction can be used if axioms for the induction ordering $\gg$ are part of SP. For inductive contraction we need axioms for the complement of $\gg$. Hence $\gg$ must be specified as a Boolean function as in examples of Section 2.3.

4 Commands

In the sequel we present the command language of Expander. The figures illustrating the commands and their parameters are taken from a NeXTSTEP interface to the ML program.

Figure 1. The commands provided by Expander

ParseSpec and ***OpenSpec*** open the directory *Specifications*. The selected specification SP is displayed in its own window. ***ParseSpec*** parses SP according to the grammar given in Section 2.1 and enters SP into the file *PARSE*. If SP is syntactically correct, SP is also assigned to the specification component of the system state (see below). Otherwise *PARSE* is displayed together with error messages. All subspecifications of SP whose names have been listed behind the key word `base` are expanded, i.e. the signatures and axioms, but no theorems or conjectures, of these specifications are added to SP.

Each system state is a quintuple with the following components:

- the current specification ***SP***,
- the set ***IN*** of universally quantified ("input") variables of the conclusions of the current conjectures,
- the current proof, given by two lists of goal sets (cf. Section 1):

$$\boldsymbol{front} = [\boldsymbol{conc}, gs_k, \dots, gs_1, \boldsymbol{ini\_conc}] \quad \text{and} \quad \boldsymbol{rear} = [\boldsymbol{prem}, hs_n, \dots, hs_1, \boldsymbol{ini\_prem}],$$

- the set ***cpos*** of the positions of the current conjectures within the list of conjectures of SP,
- the proof ***direction*** indicating whether the subsequent command is applied to *conc* (*direction* = *backward*) or to *prem* (*direction* = *forward*). *direction* is set to *backward* by calling ***Conclusion*** and set to *forward* by calling ***Premise*** (cf. Fig. 1).

Given a directory *Proofs*, the command ***StoreProof*** generates the file *Proofs/SP cpos* and enters the current proof *(front,rear)* into it. ***StoreGraph*** generates the file *Graphs/SP cpos* and enters into it each solved goal (cf. 4.4) of *(front,rear)* that is either a list of pairs (x, y) of natural numbers or a list of terms of the form $rect(x, y, l, h)$. (x, y) is entered as a point in the 2-dim. plane. $rect(x, y, l, h)$ is entered as a rectangle with least coordinate (x, y), length l and height h (cf. Example 7.5). ***OpenProof*** and ***OpenGraph*** display the directories *Proofs* and *Graphs*, repectively. The selected proof or graph is displayed on the screen. ***Prove*** opens a submenu containing all commands for generating, modifying and accessing the current proof. ***NewShell*** opens a new specification window and thus allows us to run several proofs with respect to different specifications in parallel.

4.1 Starting a proof

Click ***Start***. Alternatively, one may prove (1) a single Gentzen clause or (2) several Horn clauses simultaneously:

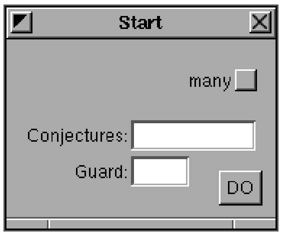

Figure 2. The start window

(1) Enter a number n at ***Conjectures***. Let $c = gs \Leftarrow h$ be the n-th conjecture of SP. The system state (see above) is changed as follows:

$$IN := \text{set of universally quantified variables of } gs,$$
$$\mathit{front} := [gs], \quad \mathit{rear} := [h], \quad \mathit{cpos} := \{n\}, \quad \mathit{direction} := \mathit{backward}.$$

(2) Click ***many*** and enter k numbers $i_1, \ldots, i_k$ at ***Conjectures*** and a number n at ***Guard***. For all $1 \leq i \leq k$, let c_i be the i-th conjecture of SP. $c_1, \ldots, c_k$ are supposed to be **guarded** Horn clauses whose first, say n, premise atoms are identical, i.e. there is a goal h consisting of n atoms such that c_i can be written as $g_i \Leftarrow h \wedge h_i$. Moreover, the goal set $h_1 \vee \ldots \vee h_k$ is supposed to be h-**minimal**:

Definition (Guarded clause, h-minimal goal set) (cf. [Pad 92], [Pad 96]) A Horn clause $c = g \Leftarrow h \wedge g'$ is **guarded** if $fresh(c) \subseteq var(g')$ where $fresh(c) = var(c) \setminus var(t)$ if g is an equation with right-hand side t and $fresh(c) = var(c) \setminus var(g)$ otherwise. A goal set gs is h-**minimal** if for all $g, g' \in gs$ and ground substitutions σ, τ such that $\sigma_{var(h)} = \tau_{var(h)}$ and $h\sigma \wedge g\sigma \wedge g'\tau$ is SP-valid, g is equal to g' and for all $x \in var(g)$, $x\sigma = x\tau$ is SP-valid. ❑

An h-minimal goal set gs represents a minimal case analysis insofar as two different goals $g, g' \in gs$ are not solvable by the same ground substitution. For instance, with respect to a specification of integers, $f(x) \equiv 0 \vee f(x) \equiv succ(y)$ is an $(f(x) \geq 0)$-minimal goal set.

The system state (see above) is changed as follows:

$$IN := \text{set of universally quantified variables of } \{c_1, \ldots, c_k\},$$
$$\mathit{front} := [(g_1 \wedge h_1) \vee \ldots \vee (g_k \wedge h_k)], \quad \mathit{rear} := [h], \quad \mathit{cpos} := \{i_1, \ldots, i_k\}, \quad \mathit{direction} := \mathit{backward}.$$

By [Pad 92], Thm. 5.5, or [Pad 96], Thm. 4.2(2), the above assumptions imply that a proof of $\mathit{front} \Leftarrow \mathit{rear}$ entails the validity of $c_1, \ldots, c_k$.

The current conjectures are supposed to be free of universal premise variables (cf. 2.1). If, in the course of a proof, clauses are applied, which have universal premise variables, these are added to IN.

*

For each function or predicate F of SP, **axioms(F)** denotes the list of all axioms of SP that define F and whose positions are given by the numbers following F in the *funct_def* list of SP (cf. 2.1).

We use the ML notation for lists and list operations. Given a list L with n elements and $1 \leq i \leq n$, $L(i)$ denotes the i-th element of L. Given $I \subseteq \{1, \ldots, n\}$,

$$L(I) =_{def} \{L(i) \mid i \in I\} \quad \text{and} \quad L(-I) =_{def} \{L(i) \mid i \in \{1, \ldots, n\} \setminus I\}.$$

Given an atom at and a number list $n_1, \ldots, n_k$, $at(n_1, \ldots, n_k)$ denotes the term at position $n_1, \ldots, n_k$ of at.

4.2 Predicate handling

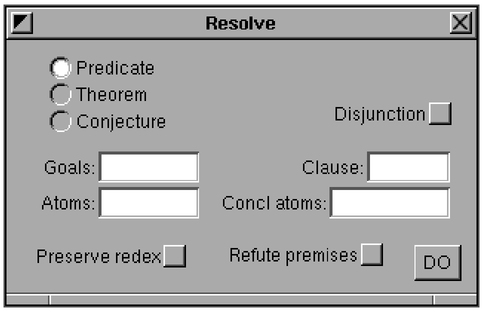

Figure 3. The resolve window

Click ***Resolve***. Alternatively, one may (1) expand a **conclusion or premise atom** upon all applicable axioms, (2) expand **conclusion atoms** of a single goal upon a theorem or conjecture, (3) expand **conclusion atoms** distributed over several goals upon a theorem or conjecture, (4) contract **premise atoms** of a single goal upon a theorem or conjecture.

(1) Enter a number n at ***Goals*** and a number k at ***Atoms***. Let $P(t) = conc(n)(k)$ if $direction = backward$ and $P(t) = prem(n)(k)$ if $direction = forward$. Then all clauses of $axioms(P)$ are applied successively to the redex $P(t)$ by greedy expansion steps (cf. Section 3). Suppose that $axioms(P)$ has l elements. The disjunction of corresponding rule succedents $rs_1, \ldots, rs_l$ extends the current proof as follows:

$$front := (rs_1 \vee \ldots \vee rs_l \vee conc(-n)) :: front \quad \text{if } direction = backward,$$
$$rear := (rs_1 \vee \ldots \vee rs_l \vee prem(-n)) :: rear \quad \text{if } direction = forward.$$

If $direction = forward$, then SP is supposed to be canonical (cf. Sect. 1) and the left-hand side of each equational axiom of SP is supposed to contain a defined function because then we can conclude from [Pad 96], Cor. 10.11, that the sequence of expansion steps is a *contraction*, i.e. yields an implication from $prem$ to $rs_1 \vee \ldots \vee rs_l \vee prem(-n)$ (cf. Section 3).

If you click ***Refute premises***, then only those axioms for P are applied to $P(t)$, which have a *feasible* premise g, i.e. the instance of g generated by the above expansion step cannot be proved unsolvable within a certain number of expansion steps (here: 5). Hence ***Refute premises*** may restrict the number of applicable axioms.

(2) $direction$ is supposed to be $backward$. Enter a number n at ***Goals***, click ***Theorem*** or ***Conjecture*** and enter a number c at ***Clause***. The theorem resp. conjecture of SP denoted by c is supposed to have the form $\exists V g \Leftarrow h$, i.e. the conclusion consists of a single goal. Enter $k \leq card(g)$ numbers $n_1, \ldots, n_k$ at ***Atoms*** and k numbers $m_1, \ldots, m_k$ at ***Concl atoms***. For all $1 \leq i \leq k$, let t_i be the conclusion atom denoted by n_i and u_i be the atom of g denoted by m_i. Then the clause $\exists V(u_1 \wedge \ldots \wedge u_k) \Leftarrow h$ is applied to the redex $t_1 \wedge \ldots \wedge t_k$ by deductive expansion if c is a theorem of SP and by inductive expansion if c is a conjecture of SP (cf. Section 3). The corresponding rule succedent rs extends the current proof as follows:

$$front := (rs \vee conc(-n)) :: front.$$

(3) $direction$ is supposed to be $backward$. Click ***Disjunction***, enter $k \leq card(conc)$ numbers $n_1, \ldots, n_k$ at ***Goals***, click ***Theorem*** or ***Conjecture*** and enter a number c at ***Clause***. The theorem resp. conjecture

of SP denoted by c is supposed to have the form $\exists X_1 g_1 \vee \ldots \vee \exists X_k g_k \Leftarrow h$, i.e. the conclusion consists of k goals. Enter k numbers $m_1, \ldots, m_k$ at ***Atoms*** and k numbers $c_1, \ldots, c_k$ at ***Concl atoms***. For all $1 \leq i \leq k$, let t_i be the conclusion atom denoted by m_i and u_i be the atom of g_i denoted by c_i. Then the clause $\exists X_1 u_1 \vee \ldots \vee \exists X_k u_k \Leftarrow h$ is applied to the redex $(t_1, \ldots, t_k)$ by deductive expansion if c is a theorem of SP and by inductive expansion if c is a conjecture of SP (cf. Section 3). The corresponding rule succedent rs extends the current proof as follows:

$$front := (rs \vee conc(-\{n_1, \ldots, n_k\})) :: front.$$

(4) *direction* is supposed to be *forward*. Enter a number n at ***Goals***, click ***Theorem*** or ***Conjecture*** and enter a number c at ***Clause***. Let $gs \Leftarrow h$ be the theorem resp. conjecture of SP denoted by c. Enter $k = card(h)$ numbers $n_1, \ldots, n_k$ at ***Atoms***. Let g be the goal consisting of the premise atoms denoted by $n_1, \ldots, n_k$. Then $gs \Leftarrow h$ is applied to the redex g by deductive contraction if c is a theorem of SP and by inductive contraction if c is a conjecture of SP (cf. Section 3). The corresponding rule succedent rs extends the current proof as follows:

$$rear := (rs \vee prem(-n)) :: rear.$$

Preserve redex saves *conc* resp. *prem* into the new goal set. Hence the four cases lead to the following updates of the current proof:

(1) $front := (rs_1 \vee \ldots \vee rs_l \vee conc) :: front$ if $direction = backward$,
$rear := (rs_1 \vee \ldots \vee rs_l \vee prem) :: rear$ if $direction = forward$.

(2/3) $front := (rs \vee conc) :: front.$

(4) $rear := (rs \vee rear) :: rear.$

4.3 Function handling

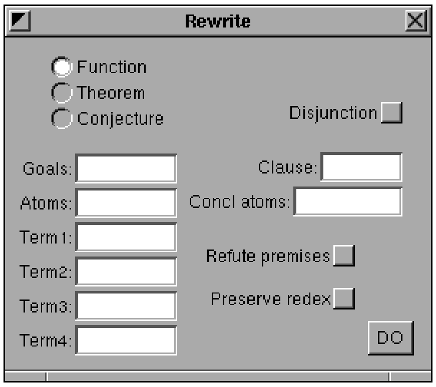

Figure 4. The rewrite window

Click ***Rewrite***. Alternatively, one may (1) expand a **conclusion or premise term** upon all applicable axioms, (2) expand a **conclusion term** upon a theorem or conjecture, (3) expand **conclusion terms** distributed over several goals upon a theorem or conjecture, (4) contract a **premise term and premise atoms** of a single goal upon a theorem or conjecture.

(1) Enter a number n at ***Goals*** and a term position (= number list) $n_1, \ldots, n_k$ at ***Term***$_1$. Let $F(t) = conc(n)(n_1)(n_2, \ldots, n_k)$ if $direction = backward$ and $F(t) = prem(n)(n_1)(n_2, \ldots, n_k)$ if $direction = forward$. Then all clauses of axioms(F) are applied successively to the redex $F(t)$ by greedy expansion steps (cf. Section 3). Suppose that $axioms(F)$ has l elements. The disjunction of corresponding rule succedents $rs_1, \ldots, rs_l$ extends the current proof as follows:

$$front := (rs_1 \vee \ldots \vee rs_l \vee conc(-n)) :: front \quad \text{if } direction = backward,$$
$$rear := (rs_1 \vee \ldots \vee rs_l \vee prem(-n)) :: rear \quad \text{if } direction = forward.$$

If $direction = forward$, then SP is supposed to be canonical (cf. Sect. 1) and the left-hand side of each equational axiom of SP is supposed to contain a defined function because then we can conclude from [Pad 96], Cor. 10.11, that the sequence of expansion steps is a *contraction*, i.e. yields an implication from $prem$ to $rs_1 \vee \ldots \vee rs_l \vee prem(-n)$ (cf. Section 3).

If you click ***Refute premises***, then only those axioms for P are applied to $P(t)$, which have a feasible premise (cf. 4.2(1)).

(2) *direction* is supposed to be *backward*. Enter a number n at ***Goals*** and a term position nL at ***Term***$_1$. Click ***Theorem*** or ***Conjecture*** and enter a number c at ***Clause***. The theorem resp. conjecture of SP denoted by c is supposed to have the form $\exists V g \Leftarrow h$, i.e. the conclusion consists of a single goal. Let t be the term at position nL of the n-th conclusion goal. Enter a number e at ***Concl atoms***. e is supposed to denote an equation of g, say $u = u'$. Then the clause $\exists V(u = u') \Leftarrow h$ is applied to the redex t by deductive expansion if c is a theorem of SP and by inductive expansion if c is a conjecture of SP (cf. Section 3). The corresponding rule succedent rs extends the current proof as follows:

$$front := (rs \vee conc(-n)) :: front.$$

(3) *direction* is supposed to be *backward*. Click ***Disjunction***, enter $k \leq card(conc)$ numbers $n_1, \ldots, n_k$ at ***Goals*** and for all k term positions $p_1, \ldots, p_k$ at ***Term***$_1, \ldots,$ ***Term***$_k$, respectively. Click ***Theorem*** or ***Conjecture*** and enter a number c at ***Clause***. The theorem resp. conjecture of SP denoted by c is supposed to have the form $\exists X_1 g_1 \vee \ldots \vee \exists X_k g_k \Leftarrow h$, i.e. the conclusion consists of k goals. For all $1 \leq i \leq k$, let t_i be the term at position p_i of the n$_i$-th conclusion goal. Enter k numbers $e_1, \ldots, e_k$ at ***Concl atoms***. For all $1 \leq i \leq k$, e_i is supposed to denote an equation of g_i with the same right-hand side, say $u_i = u$. Then the clause $\exists X_1(u_1 = u) \vee \ldots \vee \exists X_k(u_k = u) \Leftarrow h$ is applied to the redex $(t_1, \ldots, t_k)$ by deductive expansion if c is a theorem of SP and by inductive expansion if c is a conjecture of SP (cf. Section 3). The corresponding rule succedent rs extends *front* as follows:

$$front := (rs \vee conc(-\{n_1, \ldots, n_k\})) :: front.$$

(4) *direction* is supposed to be *forward*. Enter a number n at ***Goals*** and click ***Theorem*** or ***Conjecture*** and enter a number c at ***Clause***. Let $\exists X_1 g_1 \vee \ldots \vee \exists X_k g_k \Leftarrow h$ be the theorem resp. conjecture of SP denoted by c. Enter k term positions $p_1, \ldots, p_k$ at ***Term***$_1, \ldots,$ ***Term***$_k$, respectively, k numbers $e_1, \ldots, e_k$ at ***Concl atoms*** and $l = card(h)$ numbers $n_1, \ldots, n_l$ at ***Atoms***. For all $1 \leq i \leq k$, p_i is supposed to denote the same term of the n-th conclusion goal, say t, and e_i is supposed to denote an equation of g_i with the same left-hand side, say $u = u_i$. Let g be the goal consisting of the premise atoms denoted by $n_1, \ldots, n_l$. Then $\exists X_1(u = u_1) \vee \ldots \vee \exists X_k(u = u_k) \Leftarrow h$ is applied to the redex (t, g) by deductive contraction if c is a theorem of SP and by inductive contraction if c is a conjecture of SP (cf. Section 3). The corresponding rule succedent rs extends *rear* as follows:

$$rear := (rs \vee prem(-n)) :: rear.$$

Preserve redex saves *conc* resp. *prem* into the new goal set. Hence the four cases lead to the following updates of *front* resp. *rear*:

(1) $front := (rs_1 \vee \ldots \vee rs_l \vee conc) :: front$ if $direction = backward$,
$rear := (rs_1 \vee \ldots \vee rs_l \vee prem) :: rear$ if $direction = forward$.

(2/3) $front := (rs \vee conc) :: front$.

(4) $rear := (rs \vee rear) :: rear$.

4.4 Solving a goal

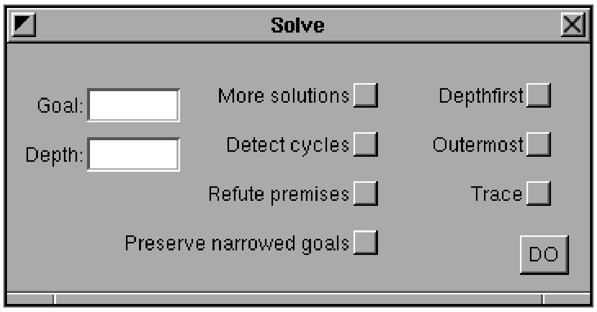

Figure 5. The solve window

Click ***Solve*** and enter a number n at ***Goal*** and a number d at ***Depth***. If $direction = backward$, then the n-th conclusion goal, otherwise the n-th premise goal, say g, is to be solved in at most d steps. For this purpose g is transformed by a sequence of greedy expansion steps upon axioms in order to derive solved goals from g (cf. Section 3). A **solved goal** is a set of equations $E = \{x_1 = t_1, \ldots, x_n = t_n\}$ such that $x_1, \ldots, x_n$ are universal variables and $t_1, \ldots, t_n$ are terms consisting of constructors. If E is obtained from $g(x_1, \ldots, x_n)$ by greedy expansion steps, then E represents a solution of g in $x_1, \ldots, x_n$, i.e. the instance $g(t_1, \ldots, t_n)$ is SP-valid (cf. 2.1).

Analogously to the construction of *rear* as an expansion of *ini_conc*, ***Solve*** builds up an expansion of g. Redices are selected from the current goal set gs by traversing gs either in depthfirst order – if ***Depthfirst*** is activated – or in breadthfirst order – if ***Depthfirst*** is not activated. Within each goal $h \in gs$, ***Solve*** searches for the leftmost-outermost – if ***Outermost*** is activated – or the leftmost-innermost – if ***Outermost*** is not activated – atom resp. subterm $F(t)$ of h such that $F(t)$ is a redex for a sequence of greedy expansion step according to 4.2(1) resp. 4.3(1) and this step does not produce a contradictory goal. Recall that this amounts of the subsequent application of all axioms for F to $F(t)$. The reducts (= expansion rule succedents) of individual expansion steps are displayed only if ***Trace*** has been activated. Among the reducts, solved goals as well as **narrowed goals**, i.e. unsolved goals without expansion redices, are not taken into account for further expansion steps. If ***Preserve narrowed goals*** is not active, then narrowed goals are deleted.

The derivation of further goals stops if (1) the total number of expansion steps exceeds d or (2) a solved goal has been derived and ***More solutions*** has not been activated. In both cases the final goal set derived from g, say gs, is embedded into the original goal set *conc* or *prem* where the g was taken from. Hence gs extends the current proof as follows:

$$front := (gs \vee conc(-n)) :: front \quad \text{if } direction = backward,$$
$$rear := (gs \vee prem(-n)) :: rear \quad \text{if } direction = forward.$$

If $direction = forward$, then SP is supposed to be canonical (cf. Sect. 1) and the left-hand side of each equational axiom of SP is supposed to contain a defined function because then we can conclude from [Pad 96], Cor. 10.11, that the sequence of expansion steps performed by ***Solve*** is a *contraction*, i.e. yields an implication from $prem$ to $gs \vee prem(-n)$ (cf. Section 3).

If you click ***StoreGraph*** in Expander's main menu, then all solved goals, which are lists of pairs of natural numbers or lists of terms of the form **rect**(x,y,l,h), are stored as graphs into the file *Graphs/SP cpos* (cf. Examples 7.5 and 7.7).

In general, if SP is functional or canonical (cf. Sect. 1), then narrowed goals are unsolvable. They can be safely removed from the set of goals derived from g because their expansion cannot lead to solutions of g. However, the use of certain features of Expander may result in the derivation of narrowed goal that are solvable even if SP is functional or canonical:

(1) If SP is built up stepwise, then a narrowed goal may become non-narrowed in a later stage of proof because axioms have been added to SP. The addition of axioms need not even have changed the semantics of SP. Certain *theorems* of SP acting as constraints (cf. 2.1) may follow from the new axioms. However, theorems are not applied by ***Solve***.

(2) ***Solve*** does not consider atoms or terms with read-only variables (cf. 2.1) as possible expansion redices. Atoms or terms are taken into account as redices only *after* their read-only variables have been instantiated in the course of other expansion steps. This allows us to realize other expansion strategies that the strict leftmost-innermost or -outermost order of redex search.

(3) An atom or term is also narrowed if it consists of variables and *standard* functions and predicates because these are evaluated by the simplifier (cf. 5.2) and thus have no axioms in SP to be expanded upon. Maybe with the help of (2) the user should choose a strategy that ensures that the simplifier has removed such atoms and terms before the solver can take them into consideration.

By default, *Solve* replaces the leftmost-innermost redex (see above). By [Pad 96], Lemmata 9.3 and 10.9, this strategy is complete, i.e. generates all solutions of g if SP is canonical (cf. Sect. 1) and the left-hand side of each equational axiom of SP contains a defined function.

Since ***Solve*** applies *greedy* expansion, the leftmost-outermost strategy amounts to the *needed strategy* defined in [AEH 94] and thus, by [Pad 96], Lemmata 9.4 and 10.9, this strategy is complete under the same assumptions. For instance, in the following example, this strategy computes a solution of the goal $f(f(x,y),z) = 0$ that would not be found by non-greedy expansion steps:

```
NEEDED
          functs    f 1 2 3
          vars      x y z
          axioms    (1)  { f(0,0) = 0 }
                    (2)  { f(s(x),0) = 1 }
                    (3)  { f(x,s(y)) = 2 }
          conjects  (1)  { f(f(x,y),z) = 0 }
       initial conclusion:
        (1) { f(f(x,y),z) = 0 }
       initial premise:
        (1) TRUE
       conclusion goal 1 is to be solved in 33 steps
       proceed breadthfirst and outermost
       goal to be solved:
        (1) { f(f(x,y),z) = 0 }
       term at position 1 1 replaced with axiom NEEDED1
       term at position 1 1 replaced with axiom NEEDED2
```

```
term at position 1 1 replaced with axiom NEEDED3
reduct:
 (1) { z = 0, f(f(x,y),0) = 0 }
 (2) { z = 0, f(f(x,y),0) = 0 }
goal to be solved:
 (1) { z = 0, f(f(x,y),0) = 0 }
term at position 2 1 1 replaced with axiom NEEDED1
term at position 2 1 1 replaced with axiom NEEDED2
term at position 2 1 1 replaced with axiom NEEDED3
reduct:
 (1) { y = s(y1), f(2,0) = 0, z = 0 }
 (2) { x = s(x1), y = 0, f(1,0) = 0, z = 0 }
 (3) { x = 0, y = 0, f(0,0) = 0, z = 0 }
goal to be solved:
 (1) { y = s(y1), f(2,0) = 0, z = 0 }
term at position 2 1 replaced with axiom NEEDED2
reduct:
 FALSE
goal to be solved:
 (1) { x = s(x1), y = 0, f(1,0) = 0, z = 0 }
term at position 3 1 replaced with axiom NEEDED2
reduct:
 FALSE
goal to be solved:
 (1) { x = 0, y = 0, f(0,0) = 0, z = 0 }
term at position 3 1 replaced with axiom NEEDED1
reduct:
 (1) { x = 0, y = 0, z = 0 }
conclusion:
 (1) { x = 0, y = 0, z = 0 }
solved goals: 1
narrowed goals have been deleted
```

A higher-order term $t(t_1)\dots(t_n)$ is represented as $ht = \mathtt{APPLY}(\dots(\mathtt{APPLY}(t,t_1)\dots),t_n)$ (cf. 2.1). Consequently, when ***Solve*** searches for a redex and encounters ht, the solver proceeds to t[5] and takes the minimal expansion redex u of ht that encloses t. Hence all axioms for the root symbol of t, say F, are checked for applicability even if there are axioms of different order such as

$$\begin{array}{lcl} F(\dots)(u_1)\dots(u_k) & = u & \Leftarrow \dots \\ F(\dots)(v_1)\dots(v_m) & = v & \Leftarrow \dots \end{array}$$

where $k \neq m$. ***Solve*** treats a λ-abstraction $t = \mathtt{fn}(x,e)$ (cf. 2.1) as a closure. Applications $t(u)$ are eliminated by outermost β-reduction (cf. Section 5.3) and thus do not occur as redices for ***Solve***.

If you click ***Refute premises***, then the solver applies only axioms with feasible premises (cf. 4.2). If ***Detect cycles*** is active, then certain loops in the derivations of g are detected and each goal h derived from a goal h' such that the solved part of h **subsumes** the solved part of h' and the unsolved parts of h and h' are **isomorphic** (cf. 4.5(5)) is not subjected to further expansion steps.

4.5 Interactive simplification

Click ***Simplify***. One may (1) **factor** two atoms of a goal, (2) **unify** the sides of an equation, (3) **replace** terms in a goal according to an equation in the same goal, (4) **delete** atoms or goals, (5) replace a goal set gs (*conc* or *prem*) by a subset of gs that is **subsumed** by gs, (6) **evaluate** goals by applying Expander's simplifier.

[5] supposed that t is a maximal *first-order* subterm of ht

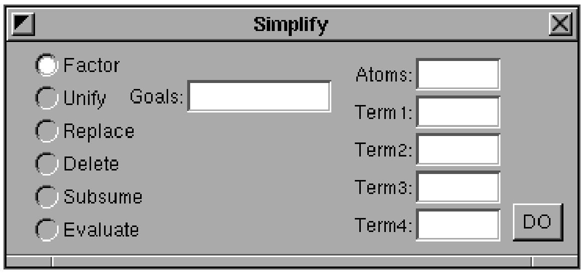

Figure 6. The simplify window

(1) *direction* is supposed to be *backward*. Click ***Factor*** and enter a number n at ***Goals*** and two numbers k_1 and k_2 at ***Atoms***. If possible, $conc(n)(k_1)$ is unified with $conc(n)(k_2)$ and $conc(n)$ is instantiated by the resulting unifier σ. The current proof is extended as follows:

$$front := ((conc(n)(-k_1)\sigma \wedge EQ(\sigma_{IN})) \vee conc(-n)) :: front.$$

(2) *direction* is supposed to be *backward*. Click ***Unify*** and enter a number n at ***Goals*** and a number k at ***Atoms***. If $conc(n)(k)$ is an equation with unifiable sides, then these sides are unified and $conc(n)$ is instantiated by the resulting unifier σ. The current proof is extended as follows:

$$front := ((conc(n)(-k)\sigma \wedge EQ(\sigma_{IN})) \vee conc(-n)) :: front.$$

(3) Click ***Replace*** and enter a number n at ***Goals***, a number k at ***Atoms*** and l term positions $p_1, \dots, p_l$ at $\boldsymbol{Term}_1, \dots, \boldsymbol{Term}_l$, respectively. Let $g = conc(n)$ if $direction = backward$ and $g = prem(n)$ if $direction = forward$. If $g(k)$ is an equation, then each term t at one of the positions $p_1, \dots, p_l$ of g is replaced by that side of $g(k)$ that differs from t.[6] The resulting goal, say h, replaces g, i.e.

$$\begin{aligned} front &:= (h \vee conc(-n)) :: front \quad \text{if } direction = backward, \\ rear &:= (h \vee prem(-n)) :: rear \quad \text{if } direction = forward. \end{aligned}$$

(4a) Click ***Delete*** and enter a number n at ***Goals*** and a non-empty number list nL at ***Atoms***. Let $g = conc(n)$ if $direction = backward$ and $g = prem(n)$ if $direction = forward$. $g(nL)$ is removed from g, i.e.

$$\begin{aligned} front &:= (g(-nL) \vee conc(-n)) :: front \quad \text{if } direction = backward, \\ rear &:= (g(-nL) \vee prem(-n)) :: rear \quad \text{if } direction = forward. \end{aligned}$$

If $direction = backward$, then $g(nL)$ is supposed to be unsolvable. Otherwise $g(-nL) \vee conc(-n)$ may not imply $conc$.

(4b) Click ***Delete*** and enter a number list nL at ***Goals***. Make sure that nothing has been entered at ***Atoms***. Let $gs = conc(nL)$ if $direction = backward$ and $gs = prem(nL)$ if $direction = forward$. gs, i.e.

$$\begin{aligned} front &:= conc(-nL) :: front \quad \text{if } direction = backward, \\ rear &:= prem(-nL) :: rear \quad \text{if } direction = forward. \end{aligned}$$

If $direction = forward$, then $prem(nL)$ is supposed to be unsolvable. Otherwise $prem$ may not imply $prem(-nL)$.

[6] If both sides of $g(k)$ differ from t, then the message `different_terms` is delivered (cf. Section 5.2).

(5) Click ***Subsume***. Subsumption is a syntactical criterion for implication that is defined as follows:

Definition (Goal set subsumption) A goal set $hs = (h_1 \vee \ldots \vee h_m)$ **subsumes** a goal set $gs = (g_1 \vee \ldots \vee g_n)$ if for all $1 \leq i \leq m$ there is $1 \leq j \leq n$ such that h_i subsumes g_j. A goal h **subsumes** a goal g if $h = FALSE$ (cf. 2.1) or $g = \emptyset$ or $g = \exists V g'$ and there is a substitution σ such that $g'\sigma_V \subseteq h$.[7] Two goal sets are **isomorphic** if they subsume each other (cf. 4.4). ❑

For instance, $x < 4 \wedge y > 0$ subsumes $\exists z : x < z$, and $\exists y, z : x = s(y) \wedge y > z$ subsumes $\exists y, y', z : x = s(y) \wedge y' > z$, but not conversely: $\exists y, y', z : x = s(y) \wedge y' > z$ does not subsume $\exists y, z : x = s(y) \wedge y > z$. In general, if hs subsumes gs, then $hs \Leftarrow gs$ is SP-valid (cf. 2.1). If gs and hs are isomorphic, then $gs \Leftrightarrow hs$ is SP-valid.

Let $hs = conc$ if $direction = backward$ and $hs = prem$ if $direction = forward$. Let gs be a minimal subset of hs such that hs subsumes gs. gs replaces hs, i.e.

$$\begin{aligned} front &:= gs :: front \quad \text{if } direction = backward,\\ rear &:= gs :: rear \quad \text{if } direction = forward. \end{aligned}$$

Since $gs \subseteq hs$, gs subsumes hs. Hence gs and hs are in fact isomorphic, i.e. *Subsume* replaces a goal set by an equivalent, but often simpler one.

(6) Click ***Evaluate*** and enter a number list nL at ***Goals***. Let $gs = conc(nL)$ if $direction = backward$ and $gs = prem(nL)$ if $direction = forward$. gs is simplified by Expander's automatic simplifier into, say, hs (cf. Section 4). hs replaces gs, i.e.

$$\begin{aligned} front &:= hs :: front \quad \text{if } direction = backward,\\ rear &:= hs :: rear \quad \text{if } direction = forward. \end{aligned}$$

4.6 Backtracking and stopping a proof

One may (1) check if the proof is complete, i.e. whether or not *prem* subsumes *conc* (cf. 4.5(5)), (2) backtrack to previous elements of *front* or *rear*, (3) stop the proof by constructing the remaining conjectures and add these to the `conjects` list of SP, (4) add the actual conjectures to the `theorems` list of SP, (5) store the actual "flat" version of SP under a new name.

(1) Click ***Prove*** and then ***Proved?***. If *prem* subsumes *conc*, then Expander answers with:

`conjectures` *cpos* `have been proved.`

Otherwise the message is:

`conjectures` *cpos* `have not yet been proved.`

Recall that subsumption of *conc* by *prem* indeed implies that the conjectures at positions *cpos* of the `conjects` list of SP are SP-valid.

(2) Click ***Prove*** and then ***Backtrack***. *conc* resp. *prem* is removed from *front* resp. *rear*:

$$\begin{aligned} front &:= tail(front) \quad \text{if } direction = backward,\\ rear &:= tail(rear) \quad \text{if } direction = forward. \end{aligned}$$

[7] Goal set subsumption generalizes to goal sets Plotkin's θ-subsumption of disjunctions of literals (cf. [Plo 70]).

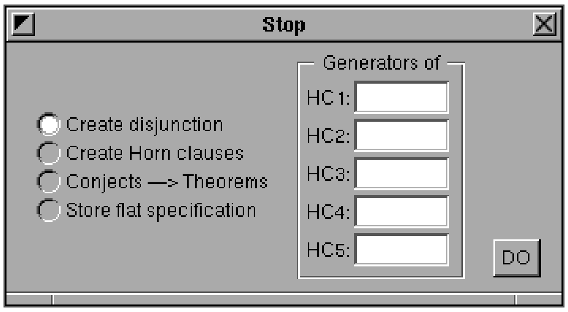

Figure 7. The stop window

(3a) Click ***Stop*** and then ***Create disjunction***. Let $conc = \exists X_1 g_1 \vee \ldots \vee \exists X_k g_k$, $prem = \exists Y_1 h_1 \vee \ldots \vee \exists Y_n h_n$ and $V = var(conc) \setminus var(prem)$. The Gentzen clauses

$$\exists V g_1 \vee \ldots \vee \exists V g_k \;\Leftarrow\; h_1$$

$$\ldots$$

$$\exists V g_1 \vee \ldots \vee \exists V g_k \;\Leftarrow\; h_n$$

are added to the list of conjectures of SP. If these clauses are SP-valid, then the original conjecture $ini\_conc \Leftarrow ini\_prem$ is also valid.

(3b) Click ***Stop*** and then ***Create Horn clauses***. Let $conc = \exists X_1 g_1 \vee \ldots \vee \exists X_k g_k$ and $prem = \exists Y h$, i.e. $prem$ is supposed to consist of a single goal. Enter k number lists $nL_1, \ldots, nL_k$ at $\boldsymbol{HC}_1, \ldots, \boldsymbol{HC}_k$, respectively. Suppose that the Horn clauses

$$g_1(-nL_1) \;\Leftarrow\; h \wedge g_1(nL_1)$$

$$\ldots$$

$$g_k(-nL_k) \;\Leftarrow\; h \wedge g_1(nL_k)$$

are guarded (cf. 4.1(2)) and $\exists X_1 g_1(nL_1) \vee \ldots \vee \exists X_k g_k(nL_k)$ is SP-valid. By [Pad 96], Lemma 3.8(3), these assumptions imply that the original conjecture $ini\_conc \Leftarrow ini\_prem$ is valid if the above Horn clauses are valid. Hence they are added to the list of conjectures of SP.

(4) Click ***Conjects→Theorems***. The clauses at positions *cpos* of the `conjects` list of SP are moved to `theorems` list of SP.

(5) Click ***Store flat specification***. The flat version of SP as created by ***ParseSpec*** is entered into the file *Specifications/SPflat*.

5 Built-in simplifications

With the commands of Section 4.5 certain goal (set) simplifications are performed *interactively*. Other simplifications are carried out *automatically*. These can be regarded as a *partial evaluation* or *normalization* of goals, similar to normalization in functional-logic language implementations (cf. [Han 93]). Simplifications transform a goal set into a semantically equivalent one. Usually each simplification step corresponds to a long sequence of "primitive" rule applications, which are performed all at once by Expander's built-in simplifier. This does not only speed up the generation of a proof, it also makes the proof shorter and more readable because primitive derivation steps are no longer visible. The simplifier also provides the interface between Expander as a proof checker on the one hand and Expander as an executable specification language on the other hand. In fact, the simplifier first compiles a goal g into an ML expression (usually including logical variables), then (partially) evaluates the expression using built-in ML functions and finally retranslates the evaluated expression into a goal that is equivalent to g. The ML functions used in this process *implement* hidden axioms and theorems of SP. Their application simulates particular sequences of expansions and contractions upon these axioms and theorems in a very efficient way. For instance, the ML function for natural number multiplication (cf. 2.2) reads as follows:

```
fun     times(zero,_) = zero
        |       times(succ(x),y) = plus(times(x,y),y)
        |       times(_,zero) = zero
        |       times(x,succ(y)) = plus(x,times(x,y))
        |       times(x,y) = Op("*",[x,y],fui)
```

The last equation covers all patterns of a product term that cannot be simplified: the compiled term is retranslated into the original term.[8]

The simplifier is applied to the goals generated by a *Start* command (cf. 4.1) and on the result of each expansion, contraction or interactive simplification step (cf. 4.2 to 4.5) except expansion and contractions upon theorems (cf. 4.2/3). The exception is necessary because the purpose of applying a theorem is to generate a particular goal structure, which might be changed by the simplifier. For instance, simplifying the results of the theorem applications of 7.7 would invert these applications.

A goal set containing the empty goal *TRUE* represents a tautology and is thus replaced by *TRUE*. A goal set $gs \vee FALSE$ is reduced to gs (cf. 2.1). An additional simplification is applied to *ini_conc*, the initial conclusion created by a ***Start*** command (cf. 4.1): *ini_conc* is subjected to an automatic call of ***Subsume*** (cf. 4.5(5)).

The simplification of a goal will always result in a single goal. This means that the simplifier does not generate disjunctions such as case analyses or sets of solutions. The way how a single goal is to be splitted into subgoals depends on the actual proof and is not determined by the simplifier. For producing case analyses or solution sets one should carry out expansion or contraction steps explicitly by calling ***Resolve***, ***Rewrite*** and *Solve* commands (cf. 4.2 to 4.4).

A single atom is simplified into a goal or into the formula $False$. An atom at that cannot be simplified is transformed into the goal $\{at\}$. A simplified goal of the form $g \wedge False$ is further reduced to $False$.

5.1 Equations and inequations

An equation between identical terms is deleted from a goal. Let $t = F(t_1, \ldots, t_k)$ and $u = G(u_1, \ldots, u_n)$ be terms such that F and G are constructors except `set`, `bag` or `upd` (cf. 2.2). If $F \neq G$, then the equation $t = u$ is reduced to $False$. If $F = G$, then $t = u$ is transformed into the goal $t_1 = u_1 \wedge \ldots \wedge t_k = u_k$, which is then further simplified. Equations of the form $set(t) = set(u)$, $bag(t) = bag(u)$ or $upd(i, e, t) = upd(j, f, u)$ are simplified by suitable equality interpretations, which regard t and u as representations of sets, bags or maps, respectively.

A goal of the form $\exists x, \ldots g(x) \wedge x = t$ where t consists of constructors and variables is reduced to $\exists \ldots g(t)$ and then simplified. These two steps: removing constructor equations and simplifying instances of the rest goal, are repeated until the resulting goal has no constructor equations. On the one hand, constructor equations often decrease the size of a goal because they avoid multiple occurrences of the same term. On the other hand, the generation of a constructor equation $x = t$ usually splits a term u into a term v and the constructor term t, which may prevent certain simplifications of u. In fact, the simplifier restores u from v and t in order to carry out these simplifications.

An inequation $t <> u$ simplifies to the negation of the simplification of $t = u$, i.e. to $False$ if $t = u$ reduces to $\emptyset$, and to $\emptyset$ if $t = u$ reduces to $False$.

[8] `Op("*",[x,y],fui)` is the internal representation of the term $x * y$.

5.2 Standard functions and predicates

Terms and atoms starting with a standard function or a standard predicate, respectively, are simplified by a sort of *partial evaluation* that realizes the semantics of the function or predicate as defined in Section 2.2. Here are some examples of terms and atoms and their respective results of the simplification process:[9]

```
(5+x)-3                                 -->  s(s(x))
        (5+x)-3 = 2                             -->  { x = 0 }
        (5+x)-3 = 1                             -->  False
        s(x+y) < 5+6                            -->  { x+y < 10 }
        s(2+4) = 6                              -->  False
        [2,x,3,y,5,3] = [2,3,x,5,z,3]           -->  { x = 3, y = 5, 5 = z }
        {{2,x,3,y,5,3}} = {{5,3,y,2,x,3}}       -->  { }
        {{2,x,3,y,5,3}} = {{5,3,z,2,x,3}}       -->  { y = z }
        {{2,x,3,5,z',3,y}} = {{z,5,3,z,2,x,3}}  -->  { {{y,z'}} = {{z,z}} }
        {{2,x,3,[y,y'],5,3}} = {{5,3,[z,z'],2,x,3}} -->  { y = z, y' = z' }
        bag(x::(y::w)) = bag(y::(x::w))         -->  { }
        bag(x::(x::w)) = bag(x::w)              -->  { bag(x::w) = bag(w) }
        {2,3,4} = {3,5,2,3}                     -->  False
        {2,3,5} = {3,5,2,3}                     -->  { }
        s(s(s(s(x)))) in [0,1,2,3]              -->  False
        {2,s(s(s(s(x))))} sub {0,1,2,3}         -->  False
        {2,x,3} sub {3,2,3}                     -->  { x in [3,2,3] }
        set(x::(x::w)) = set(x::w)              -->  { }
        <<2<-1,4<-3,2<-5>> = <<2<-5,4<-3>>      -->  { }
        <<2<-1,4<-3,2<-5>> = upd(4,3,f)         -->  { <<(2<-1),(2<-5)>> = f }
        (2+(<<2<-1,4<-x,2<-5>> at 4))+7 = 12    -->  { x = 3 }
        (2+(<<2<-1,4<-x,2<-5>> at 3))+7 = 12    -->  { noEntryAt(3) = 3 }
        upd(x,e,upd(x,e',f)) = upd(x,e",f)      -->  { e = e" }
        upd(x,e,upd(y,e',f)) = upd(y,e',upd(x,e,f)) -->  { e = upd(y,e',upd(x,e,f)) at x }
```

5.3 Higher-order transformations

λ-expressions $fn(x,e)(t)$ (cf. 2.1) are simplified by applying leftmost-outermost β-reduction as far as possible[10]. Before t is substituted for the occurrences of x in e, each bounded variable y of e is renamed into yB and thus made distinct from the variables of t.

Terms of the form $cont(t,c)$ are simplified by applying the following rules that shall enforce the strict "stack-like" leftmost-innermost evaluation of t (cf. e.g. [BJM 91], Section 6.2):

$cont(x,c) \longrightarrow c(x)$ for all variables x

$cont(F,c) \longrightarrow c(F)$ for all constants F of SP

$cont(F(e_1,\dots,e_k),c)$

$\longrightarrow cont(e_1, cont(e_2,\dots,cont(e_k,F_C(c))\dots))$ for all non-constant functions F of SP

where the **continuation counterpart** F_C of F is supposed to be specified by the following axioms that are derived from the axioms defining F:

$F_C(c)(x_k)\dots(x_1) = c(F(x_1,\dots,x_k))$ if F is a constructors

$F_C(c)(e_k)\dots(e_1) = cont(t,c)$ for all axioms $F(e_1,\dots,e_k) = t \Leftarrow P(t_1,\dots,t_n)\wedge\dots$

$\Leftarrow P(cont(t_1,c),\dots,cont(t_n,c))\wedge\dots$ if F is a defined function

For instance, the factorial function and its continuation counterpart could be specified as follows:

[9] {} stands for the empty goal.

[10] Make sure that your λ-expressions have normal forms!

```
FACT
        functs    fact 1 2  fact_C 3 4
        vars      x y c
        axioms    (1)   { fact(0) = 1 }
                  (2)   { fact(s(x)) = s(x)*fact(x) }
                  (3)   { fact_C(c)(0) = c(1) }
                  (4)   { fact_C(c)(s(x)) = cont(s(x)*fact(x),c) }
        conjects  (1)   { fact(4) = x }
                  (2)   { cont(fact(4),fn(y,y)) = x }
```

The continuation counterparts of the tupling constructor and the apply function (cf. 2.1) are included in the system as the standard functions `tup_C` and `apply_C`, respectively. They are generated only if a term $cont(t, c)$ containing tuples or higher-order functions is simplified. `tup_C` and `apply_C` are not evaluated by the simplifier. Axioms for `tup_C` and `apply_C` that cover the actual arities of tuples resp. higher-order functions of SP must be added to SP explicitly. For instance, if tupling occurs only as pairing and all higher-order functions are unary, then the axioms `tup_C(c)(x)(y)=c((y,x))` and `apply_C(c)(x)(f)=cont(f(x),c)` are sufficient.

6 Error messages

- `wrong_lhs (wrong_rhs)` indicates that a command tries to access the left-hand (right-hand) side of an atom, which is not an equation.
- `occur_check_failed` is raised if a variable x occurs in a term where x is not supposed to occur.
- `wrong_predicate, wrong_function` or `wrong_application` is returned by the simplifier if a standard operator is applied to incompatible arguments.
- `no_unifier` indicates that (1) a command attempts to unify two terms or atoms that are not unifiable or (2) an expansion command instantiates existential variables of a clause by universal variables or non-variable terms or (3) the ***Refute premises*** option of 4.2(1), 4.3(1) and 4.4 detects that all applicable axiom instances have an unfeasible premise (cf. 4.2(1)). Case (2) violates the applicability condition of the expansion rule to be applied to *conc* (cf. Section 3).
- `no_matching` is raised by a contraction command if the clause to be applied to *prem* does not match the redex.
- `different_terms` indicates that (1) – in Case 4.3(4) – the terms at positions $p_1, \ldots, p_k$ are not identical or (2) – in Case 4.5(3) – the term at some position p_i, $1 \leq i \leq m$, differs from both sides of the selected equation $g(k)$.
- `clause_not_found, goal_not_found, atom_not_found, term_not_found` or `list_elem_not_found` is returned if a command is called with a non-existing clause, goal, atom, term or list element position, respectively.
- `exVars_is_ambiguous` is raised if an expansion command instantiates different existential variables of a clause by the same variable. This violates the applicability condition of the expansion rule to be applied to *conc* (cf. Section 3).
- `exVars_contains_reductVars` is delivered if an expansion command instantiates existential variables of a clause by variables that occur *not only* in the redex. Again, this violates the applicability condition of the expansion rule to be applied to *conc*.
- `not_applicable_to_premise` is returned if $direction = forward$ and a command is called that is only applicable *front*. This may happen in Cases 4.3(3), 4.5(1) and 4.5(2).

7 Sample proofs and tests

We present the Expander output of proofs or tests of conjectures taken from the sample specifications of Section 2.3. The numbers at the right margin denote the sections where the commands performing the respective proof steps have been described.

7.1 *Even and odd*

Proof of Conjecture 1 of EVEN_ODD (cf. 2.3.1):

```
initial conclusion:                                                                     4.1(1)
 (1) { double(ex) = x }
 (2) { s(double(ey)) = x }
initial premise:
 (1) { Def(x) }

term at position 1 1 in conclusion goal 1 replaced with axiom EVEN_ODD1                 4.3(1)
term at position 1 1 in conclusion goal 1 replaced with axiom EVEN_ODD2
conclusion:
 (1) { s(s(double(x1))) = x }
 (2) { 0 = x }
 (3) { s(double(ey)) = x }
terms at positions 1 1 1 , 1 1 1 in conclusion goals 3 1 replaced with conjecture 1     4.3(3)
conclusion:
 (1) { x >> x1 = true, s(x1) = x, Def(x1) }
 (2) { 0 = x }
term at position 1 1 in conclusion goal 1 replaced with axiom EVEN_ODD5                 4.3(1)
conclusion:
 (1) { x = s(x1), Def(x1) }
 (2) { 0 = x }

premise:
 (1) { Def(x) }
atom 1 in premise goal 1 replaced with axiom EVEN_ODD3                                  4.2(1)
atom 1 in premise goal 1 replaced with axiom EVEN_ODD4
premise:
 (1) { x = s(x1), Def(x1) }
 (2) { x = 0 }

conjecture 1 has been proved                                                            4.6(1)
```

7.2 *Less or equal*

Proof of Conjecture 1 of LEQ (cf. 2.3.2):

```
initial conclusion:                                                                     4.1(1)
 (1) { x <= z }
initial premise:
 (1) { x <= y, y <= z }

atom 1 in conclusion goal 1 replaced with axiom NAT1                                    4.2(1)
atom 1 in conclusion goal 1 replaced with axiom NAT2
conclusion:
 (1) { x = s(x1), z = s(y1), x1 <= y1 }
 (2) { x = 0 }
atom 3 in conclusion goal 1 replaced with conjecture 1                                  4.2(2)
conclusion:
```

```
 (1) { (x,z) >> (x1,y1) = true, x1 <= y2, y2 <= y1, x = s(x1), z = s(y1) }
 (2) { x = 0 }
term at position 1 1 in conclusion goal 1 replaced with axiom NAT3          4.3(1)
conclusion:
 (1) { x = s(x1), x1 <= y2, y2 <= y1, z = s(y1)}
 (2) { x = 0 }

premise:
 (1) { x <= y, y <= z }
atom 1 in premise goal 1 replaced with axiom NAT1                            4.2(1)
atom 1 in premise goal 1 replaced with axiom NAT2
premise:
 (1) { x = s(x1), y = s(y1), x1 <= y1, s(y1) <= z }
 (2) { x = 0, y <= z }
atom 4 in premise goal 1 replaced with axiom NAT2                            4.2(1)
premise:
 (1) { z = s(y2), y1 <= y2, x = s(x1), y = s(y1), x1 <= y1 }
 (2) { x = 0, y <= z }

conjecture 1 has been proved                                                 4.6(1)
```

Proof of Conjecture 2 of LEQ (cf. 2.3.2):

```
initial conclusion:                                                          4.1(1)
 (1) FALSE
initial premise:
 (1) { x <= y, y < x }

premise goal 1 is to be solved in 10 steps                                   4.4
proceed breadthfirst and innermost and detect loops
goal to be solved:
 (1) { x <= y, y < x }
atom 1 replaced with axiom NAT1
atom 1 replaced with axiom NAT2
reduct:
 (1) { x = s(x1), y = s(y1), x1 <= y1, y1 < x1 }
atom 3 replaced with axiom NAT1
atom 3 replaced with axiom NAT2
reduct:
 (1) { x1 <= y1, x = s(s(x1)), y = s(s(y1)), y1 < x1 }
premise:
 (1) { x1 <= y1, x = s(s(x1)), y = s(s(y1)), y1 < x1 }
solved goals:
narrowed goals: 1
goal 1 deleted

premise:
 FALSE

conjecture 2 has been proved                                                 4.6(1)
```

7.3 *Mergesort*

Proof of Conjecture 2 of MSORT_P (cf. 2.3.3):

```
initial conclusion:                                                          4.1(1)
 (1) { Q(nil,nil) }
initial premise:
 (1) TRUE
```

```
atom 1 in conclusion goal 1 replaced with axiom MSORT_P1                    4.2(1)
conclusion:
 (1) { sorted(nil) }

atom 1 in conclusion goal 1 replaced with axiom MSORT1                      4.2(1)
conclusion:
 (1) TRUE

conjecture 2 has been proved                                                4.6(1)
```

Proof of Conjecture 3 of MSORT_P:

```
initial conclusion:                                                         4.1(1)
 (1) { Q([x],[x]) }
initial premise:
 (1) TRUE

atom 1 in conclusion goal 1 replaced with axiom MSORT_P1                    4.2(1)
conclusion:
 (1) { sorted([x]) }

atom 1 in conclusion goal 1 replaced with axiom MSORT2                      4.2(1)
conclusion:
 (1) TRUE

conjecture 3 has been proved                                                4.6(1)
```

Proof of Conjecture 4 of MSORT_P:

```
initial conclusion:                                                         4.1(1)
 (1) { Q(x::(y::w),merge(l',r')) }
initial premise:
 (1) { split(w) = (l,r), Q(x::l,l'), Q(y::r,r') }

atom 1 in conclusion goal 1 replaced with axiom MSORT_P1                    4.2(1)
conclusion:
 (1) { sorted(merge(l',r')), bag(y::(x::w)) = bag(merge(l',r')) }

atom 1 in conclusion goal 1 replaced with theorem 1                         4.2(2)
conclusion:
 (1) { sorted(l'), sorted(r'), bag(y::(x::w)) = bag(merge(l',r')) }

term at position 3 1 in conclusion goal 1 replaced with theorem 5           4.3(2)
conclusion:
 (1) { bag(x::(y::w)) = bag(merge(l',r')), sorted(l'), sorted(r') }

term at position 1 2 in conclusion goal 1 replaced with theorem 2           4.3(2)
conclusion:
 (1) { bag(l'@r') = bag(x::(y::w)), sorted(l'), sorted(r') }

term at position 1 1 in conclusion goal 1 replaced with theorem 3           4.3(2)
conclusion:
 (1) { bag(l') = bag(r), bag(r') = bag(r'1),
       bag(r@r'1) = bag(x::(y::w)), sorted(l'), sorted(r') }

atom 3 in conclusion goal 1 replaced with theorem 4                         4.2(2)
conclusion:
 (1) { split(x::(y::w)) = (r,r'1), bag(l') = bag(r), bag(r') = bag(r'1),
       sorted(l'), sorted(r') }
```

```
term at position 1 1 in conclusion goal 1 replaced with axiom MSORT13                  4.3(1)
conclusion:
 (1) { split(w) = (L1,L2), bag(l') = bag(x::L1), bag(r') = bag(y::L2),
       sorted(l'), sorted(r') }

premise:
 (1) { split(w) = (l,r), Q(x::l,l'), Q(y::r,r') }

atom 2 in premise goal 1 replaced with axiom MSORT_P1                                  4.2(1)
premise:
 (1) { sorted(l'), bag(x::l) = bag(l'), split(w) = (l,r), Q(y::r,r') }

atom 4 in premise goal 1 replaced with axiom MSORT_P1                                  4.2(1)
premise:
 (1) { sorted(r'), bag(y::r) = bag(r'), sorted(l'), bag(x::l) = bag(l'),
       split(w) = (l,r) }

conjecture 4 has been proved                                                           4.6(1)
```

7.4 *Replace by minimum*

Computation of a solution of Conjecture 1 of REPMIN (cf. 2.3.4):

```
initial conclusion:                                                                    4.1(1)
 (1) { rep_by_min((%(5)#%(4))#(%(3)#%(2))) = t }
initial premise:
 (1) TRUE

conclusion goal 1 is to be solved in 18 steps                                          4.4
proceed depthfirst and innermost
goal to be solved:
 (1) { rep_by_min((%(5)#%(4))#(%(2)#%(3))) = t }
term at position 1 1 replaced with axiom REPMIN3
reduct:
 (1) { rep_and_min((%(5)#%(4))#(%(2)#%(3)),m) = (t,m) }
term at position 1 1 replaced with axiom REPMIN2
reduct:
 (1) { rep_and_min(%(5)#%(4),m) = (t3,m1), rep_and_min(%(2)#%(3),m) = (t4,m2), min(m1,m2) = m,
       t3#t4 = t }
term at position 1 1 replaced with axiom REPMIN2
reduct:
 (1) { rep_and_min(%(5),m) = (t31,m11), rep_and_min(%(4),m) = (t41,m21), min(m11,m21) = m1,
       rep_and_min(%(2)#%(3),m) = (t4,m2), min(m1,m2) = m, (t31#t41)#t4 = t }
term at position 1 1 replaced with axiom REPMIN1
reduct:
 (1) { rep_and_min(%(4),m) = (t41,m21), min(5,m21) = m1, rep_and_min(%(2)#%(3),m) = (t4,m2),
       min(m1,m2) = m, (%(m)#t41)#t4 = t }
term at position 1 1 replaced with axiom REPMIN1
reduct:
 (1) { rep_and_min(%(2)#%(3),m) = (t4,m2), min(4,m2) = m, (%(m)#%(m))#t4 = t }
term at position 1 1 replaced with axiom REPMIN2
reduct:
 (1) { rep_and_min(%(2),m) = (t3,m1), rep_and_min(%(3),m) = (t41,m21), min(m1,m21) = m2,
       min(4,m2) = m, (%(m)#%(m))#(t3#t41) = t }
term at position 1 1 replaced with axiom REPMIN1
reduct:
 (1) { rep_and_min(%(3),m) = (t41,m21), min(2,m21) = m2, min(4,m2) = m,
       (%(m)#%(m))#(%(m)#t41) = t }
term at position 1 1 replaced with axiom REPMIN1
```

```
reduct:
 (1) { (%(2)#%(2))#(%(2)#%(2)) = t }

conclusion:
 (1) { (%(2)#%(2))#(%(2)#%(2)) = t }
solved goals: 1
narrowed goals have been deleted
```

7.5 *Find paths*

Computation of a solution of Conjecture 1 of ROBOT (cf. 2.3.5):

```
initial conclusion:                                                       4.1(1)
 (1) { Path(1,2,5,4,p,[(1,3),(2,4)]) }
initial premise:
 (1) TRUE

conclusion goal 1 is to be solved in 12 steps                            4.4
proceed depthfirst and outermost
goal to be solved:
 (1) { Path(1,2,5,4,p,[(1,3),(2,4)]) }
atom 1 replaced with axiom ROBOT1
atom 1 replaced with axiom ROBOT2
reduct:
 (1) { p = (1,2)::p1, Path(1,3,5,4, p1,[(1,3),(2,4)]) }
 (2) { p = (1,2)::p1, Path(2,2,5,4,p1,[(1,3),(2,4)]) }
goal to be solved:
 (1) { p = (1,2)::p1, Path(1,3,5,4,p1,[(1,3),(2,4)]) }
goal to be solved:
 (1) { p = (1,2)::p1, Path(2,2,5,4,p1,[(1,3),(2,4)]) }
atom 2 replaced with axiom ROBOT1
atom 2 replaced with axiom ROBOT2
reduct:
 (1) { Path(2,3,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::p1) }
 (2) { Path(3,2,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::p1) }
goal to be solved:
 (1) { Path(2,3,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::p1) }
atom 1 replaced with axiom ROBOT1
atom 1 replaced with axiom ROBOT2
reduct:
 (1) { Path(2,4,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::p1)) }
 (2) { Path(3,3,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::p1)) }
goal to be solved:
 (1) { Path(2,4,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::p1)) }
goal to be solved:
 (1) { Path(3,3,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::p1)) }
atom 1 replaced with axiom ROBOT1
atom 1 replaced with axiom ROBOT2
reduct:
 (1) { Path(3,4,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::((3,3)::p1))) }
 (2) { Path(4,3,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::((3,3)::p1))) }
goal to be solved:
 (1) { Path(3,4,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::((3,3)::p1))) }
atom 1 replaced with axiom ROBOT1
reduct:
 (1) { Path(4,4,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::((3,3)::((3,4)::p1)))) }
atom 1 replaced with axiom ROBOT1
reduct:
 (1) { Path(5,4,5,4,p1,[(1,3),(2,4)]),
       p = (1,2)::((2,2)::((2,3)::((3,3)::((3,4)::((4,4)::p1))))) }
```

```
atom 1 replaced with axiom ROBOT3
reduct:
 (1) { p = [(1,2),(2,2),(2,3),(3,3),(3,4),(4,4),(5,4)] }

conclusion:
 (1) { p = [(1,2),(2,2),(2,3),(3,3),(3,4),(4,4),(5,4)] }
 (2) { Path(4,3,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::((2,3)::((3,3)::p1))) }
 (3) { Path(3,2,5,4,p1,[(1,3),(2,4)]), p = (1,2)::((2,2)::p1) }
solved goals: 1
narrowed goals have been deleted
```

StoreGraph (cf. Sect. 4) enters the following ASCII diagram into the file *Graphs/ROBOT 1*:

```
Solutions of
{ Path(1,2,5,4,p,[(1,3),(2,4)]) }

  4 |        *  *  *
  3 |     *  *
  2 |  *  *
  1 |
     ----------------
       1  2  3  4  5
```

7.6 *Mutual exclusion*

Proof of Conjecture 1 of MUTEX (cf. 2.3.6):

```
initial conclusion:                                                     4.1(1)
 (1) { invariant(x1,y1,d1,d1') }
initial premise:
 (1) { invariant(x,y,d,d'), act(i)(x,y,d,d') = (x1,y1,d1,d1') }

conclusion goal 1 is to be solved in 12 steps                           4.4
get all solutions
proceed depthfirst and innermost
goal to be solved:
 (1) { invariant(x1,y1,d1,d1') }
atom 1 in goal 1 replaced with axiom MUTEX5
atom 1 in goal 1 replaced with axiom MUTEX6
atom 1 in goal 1 replaced with axiom MUTEX7
reduct:
 (1) { x1 = 0, y1 = 1, d1 = alpha, d1' = beta }
 (2) { x1 = 1, y1 = 0, d1 = beta, d1' = alpha }
 (3) { x1 = 0, y1 = 0, d1 = alpha, d1' = alpha }

conclusion:
 (1) { x1 = 0, y1 = 1, d1 = alpha, d1' = beta }
 (2) { x1 = 1, y1 = 0, d1 = beta, d1' = alpha }
 (3) { x1 = 0, y1 = 0, d1 = alpha, d1' = alpha }
solved goals: 1 2 3
narrowed goals:

premise:
 (1) { invariant(x,y,d,d'), act(i)(x,y,d,d') = (x1,y1,d1,d1') }

premise goal 1 is to be solved in 12 steps                              4.4
get all solutions
proceed depthfirst and innermost
goal to be solved:
```

```
 (1) { invariant(x,y,d,d'), act(i)(x,y,d,d') = (x1,y1,d1,d1') }
atom 1 replaced with axiom MUTEX5
atom 1 replaced with axiom MUTEX6
atom 1 replaced with axiom MUTEX7
reduct:
 (1) { x = 0, y = 1, d = alpha, d' = beta, act(i)(0,1,alpha,beta) = (x1,y1,d1,d1') }
 (2) { x = 1, y = 0, d = beta, d' = alpha, act(i)(1,0,beta,alpha) = (x1,y1,d1,d1') }
 (3) { x = 0, y = 0, d = alpha, d' = alpha, act(i)(0,0,alpha,alpha) = (x1,y1,d1,d1') }
goal to be solved:
 (1) { x = 0, y = 1, d = alpha, d' = beta, act(i)(0,1,alpha,beta) = (x1,y1,d1,d1') }
term at position 5 1 replaced with axiom MUTEX4
reduct:
 (1) { i = 2, 0 = x1, 0 = y1, alpha = d1, alpha = d1', x = 0, y = 1, d = alpha, d' = beta }
goal to be solved:
 (1) { x = 1, y = 0, d = beta, d' = alpha, act(i)(1,0,beta,alpha) = (x1,y1,d1,d1') }
term at position 5 1 replaced with axiom MUTEX2
reduct:
 (1) { i = 1, 0 = x1, 0 = y1, alpha = d1, alpha = d1', x = 1, y = 0, d = beta, d' = alpha }
goal to be solved:
 (1) { x = 0, y = 0, d = alpha, d' = alpha, act(i)(0,0,alpha,alpha) = (x1,y1,d1,d1') }
term at position 5 1 replaced with axiom MUTEX1
term at position 5 1 replaced with axiom MUTEX3
reduct:
 (1) { i = 2, 0 = x1, 1 = y1, alpha = d1, beta = d1', x = 0, y = 0, d = alpha, d' = alpha }
 (2) { i = 1, 1 = x1, 0 = y1, beta = d1, alpha = d1', x = 0, y = 0, d = alpha, d' = alpha }

premise:
 (1) { i = 2, 0 = x1, 0 = y1, alpha = d1, alpha = d1', x = 0, y = 1, d = alpha, d' = beta }
 (2) { i = 1, 0 = x1, 0 = y1, alpha = d1, alpha = d1', x = 1, y = 0, d = beta, d' = alpha }
 (3) { i = 2, 0 = x1, 1 = y1, alpha = d1, beta = d1', x = 0, y = 0, d = alpha, d' = alpha }
 (4) { i = 1, 1 = x1, 0 = y1, beta = d1, alpha = d1', x = 0, y = 0, d = alpha, d' = alpha }
solved goals: 1 2 3 4
narrowed goals:

conjecture 1 has been proved                                                   4.6(1)
```

Due to the finite data domains of MUTEX we have obtained a simple non-inductive proof by computing all solutions of the conjecture's conclusion and premise, respectively. Since the premise solutions subsume the conclusion solutions, the proof is complete (cf. 4.5(5)).

7.7 *List partitions*

Proof of Conjecture 2 of PARTITION (cf. 2.3.7):

```
initial conclusion:                                                            4.1(1)
 (1) { Q([x],[[x]]) }
initial premise:
 (1) TRUE

atom 1 in conclusion goal 1 replaced with axiom PARTITION6                     4.2(1)
conclusion:
 (1) { [x] = flatten([[x]]) }

term at position 1 2 in conclusion goal 1 replaced with axiom PARTITION5       4.3(1)
conclusion:
 (1) { flatten(nil) = nil }

term at position 1 1 in conclusion goal 1 replaced with axiom PARTITION4       4.3(1)
conclusion:
```

```
 (1) TRUE

conjecture 2 has been proved                                           4.6(1)
```

Proof of Conjecture 3 of PARTITION:

```
initial conclusion:                                                    4.1(1)
 (1) { Q(x::(y::L),([x])::P) }
initial premise:
 (1) { Q(y::L,P) }

atom 1 in conclusion goal 1 replaced with axiom PARTITION6             4.2(1)
conclusion:
 (1) { x::(y::L) = flatten(([x])::P) }

term at position 1 2 in conclusion goal 1 replaced with axiom PARTITION5   4.3(1)
conclusion:
 (1) { flatten(P) = y::L }

premise:
 (1) { Q(y::L,P) }

atom 1 in premise goal 1 replaced with axiom PARTITION6                4.2(1)
premise:
 (1) { y::L = flatten(P) }

conjecture 3 has been proved                                           4.6(1)
```

Proof of Conjecture 4 of PARTITION:

```
initial conclusion:                                                    4.1(1)
 (1) { Q(x::(y::L),(x::L')::P) }
initial premise:
 (1) { Q(y::L,L'::P) }

atom 1 in conclusion goal 1 replaced with axiom PARTITION6             4.2(1)
conclusion:
 (1) { x::(y::L) = flatten((x::L')::P) }

term at position 1 2 in conclusion goal 1 replaced with axiom PARTITION5   4.3(1)
conclusion:
 (1) { L'@flatten(P) = y::L }

premise:
 (1) { Q(y::L,L'::P) }

atom 1 in premise goal 1 replaced with axiom PARTITION6                4.2(1)
premise:
 (1) { y::L = flatten(L'::P) }

term at position 1 2 in premise goal 1 replaced with axiom PARTITION5  4.3(1)
premise:
 (1) { L'@flatten(P) = y::L }

conjecture 4 has been proved                                           4.6(1)
```

7.8 *Fair streams*

Proof of Conjecture 1 of FAIR (cf. 2.3.8):

```
initial conclusion:                                                               4.1(1)
 (1) { st = L#(0&st'), Q(st') }
initial premise:
 (1) { Q(st) }

term at position 1 2 in conclusion goal 1 replaced with axiom FAIR4               4.3(1)
term at position 1 2 in conclusion goal 1 replaced with axiom FAIR5
conclusion:
 (1) { x&(L#(0&st')) = st, Q(st') }
 (2) { 0&st' = st, Q(st') }

atom 2 in conclusion goal 1 replaced with axiom FAIR12                            4.2(1)
atom 2 in conclusion goal 1 replaced with axiom FAIR13
conclusion:
 (1) { x&(L#(0&IO)) = st }
 (2) { x&(L#(0&OI)) = st }
 (3) { 0&st' = st, Q(st') }

goal 2 deleted                                                                   4.5(4b)
conclusion:
 (1) { x&(L#(0&IO)) = st }
 (2) { 0&st' = st, Q(st') }

term at position 1 1 2 in conclusion goal 1 replaced with axiom FAIR4             4.3(1)
term at position 1 1 2 in conclusion goal 1 replaced with axiom FAIR5
conclusion:
 (1) { x&(x1&(L#(0&IO))) = st }
 (2) { x&(0&IO) = st }
 (3) { 0&st' = st, Q(st') }

goal 1 deleted                                                                   4.5(4b)
conclusion:
 (1) { x&(0&IO) = st }
 (2) { 0&st' = st, Q(st') }

term at position 1 1 2 in conclusion goal 1 replaced with theorem 1               4.3(2)
conclusion:
 (1) { x&OI = st }
 (2) { 0&st' = st, Q(st') }

atom 1 in conclusion goal 1 replaced with theorem 2                               4.2(2)
conclusion:
 (1) { st = IO }
 (2) { 0&st' = st, Q(st') }

atom 1 in conclusion goal 2 replaced with theorem 1                               4.2(2)
conclusion:
 (1) { st = IO }
 (2) { st = OI, Q(IO) }

atom 2 in conclusion goal 2 replaced with axiom FAIR13                            4.2(1)
conclusion:
 (1) { st = IO }
 (2) { st = OI }

premise:
 (1) { Q(st) }

atom 1 in premise goal 1 replaced with axiom FAIR12                               4.2(1)
atom 1 in premise goal 1 replaced with axiom FAIR13
premise:
 (1) { st = IO }
```

```
 (2) { st = OI }

conjecture 2 has been proved                                                            4.6(1)
```

8 Source code

8.1 Commands as ML function calls

The following table lists all commands described in Section 4 along with the equivalent ML function call of Expander's source code. *this* denotes the number entry at the text field *this* of Expander's NeXTSTEP interface. For instance, the command

Prove Rewrite Theorem Disjunction Goals Term1 Term2 Clause Concl atoms Preserve redex

is equivalent to the function call

rewThmOrPres [Goals] [[Term1],[Term2]] Clause [Concl atoms];

{ and } enclose optional parts. List parameters of ML functions are enclosed into *[* and *]*. Non-bracketed parameters indicate that the corresponding text fields admits only a single (number) entry.

Command	**ML function call**
ParseSpec file	*parse"file";*
StoreProof	*StoreProof();*
StoreGraph	*StoreGraph();*
Prove Start Conjectures	*start Conjectures;*
Prove Start many Conjectures Guard	*startM [Conjectures] Guard;*
Prove Resolve Predicate Goals Atoms {Preserve redex} {Refute premises}	
	resPred{Pres}{Refute} Goals Atoms;
Prove Resolve Theorem Goals Atoms Clause Concl atoms {Preserve redex}	
	resThm{Pres} Goals [Atoms] Clause [Concl atoms];
Prove Resolve Conjecture Goals Atoms Clause Concl atoms {Preserve redex}	
	resConj{Pres} Goals [Atoms] Clause [Concl atoms];
Prove Resolve Theorem Disjunction Goals Atoms Clause Concl atoms {Preserve redex}	
	resThmOr{Pres} [Goals] [Atoms] Clause [Concl atoms];
Prove Resolve Conjecture Disjunction Goals Atoms Clause Concl atoms {Preserve redex}	
	resConjOr{Pres} [Goals] [Atoms] Clause [Concl atoms];
Prove Rewrite Function Goals Term1 {Preserve redex} {Refute premises}	
	rewFunc{Pres}{Refute} Goals [Term1];
Prove Rewrite Theorem Goals Atoms Term1 Term2 ... Clause Concl atoms {Preserve redex}	
	rewThm{Pres} Goals [[Term1],[Term2]...] [Atoms] Clause [Concl atoms];
Prove Rewrite Conjecture Goals Atoms Term1 Term2 ... Clause Concl atoms {Preserve redex}	
	rewConj{Pres} Goals [[Term1],[Term2]...] [Atoms] Clause [Concl atoms];

Command	**ML function call**
Prove Rewrite Theorem Disjunction Goals Term1 Term2 ... Clause Concl atoms {*Preserve redex*}	
	rewThmOr{*Pres*} *[Goals] [[Term1],[Term2]...]*
	Clause [Concl atoms];
Prove Rewrite Conjecture Disjunction Goals Term1 Term2 ... Clause Concl atoms {*Preserve redex*}	
	rewConjOr{*Pres*} *[Goals][[Term1],[Term2]...]*
	Clause [Concl atoms];
Prove Solve Goal Depth {*Refute premises*}	*solve*{*Refute*} *Goal Depth dc m o df p t;*
	dc = if *Detect cycles* then detectCycles else no
	m = if *More solutions* then moreSols else no
	o = if *Outermost* then outermost else no
	df = if *Depthfirst* then depthfirst else no
	p = if *Preserve narrowed goals* then presNarrowed else no
	t = if *Trace* then trace else no
Prove Simplify Factor Goals Atoms	*factor Goals* $Atoms_1$ $Atoms_2$*;*
Prove Simplify Unify Goals Atoms	*unify Goals Atoms;*
Prove Simplify Replace Goals Atoms Term1 Term2 ...	
	replace Goals Atoms [[Term1],[Term2]...];
Prove Simplify Delete Goals Atoms	*delAtoms unGoals [Atoms];*
Prove Simplify Delete Goals	*delGoals [unGoals];*
Prove Simplify Subsume	*subsume();*
Prove Simplify Evaluate Goals	*eval Goals;*
Prove Conclusion	*conc();*
Prove Premise	*prem();*
Prove Proved?	*proved();*
Prove Backtrack	*back();*
Prove Stop Create disjunction	*createDis();*
Prove Stop Create Horn clauses HC1 HC2 ...	*createHCs [HC1,HC2...];*
Prove Stop Conjects→Theorems	*conjs2thms();*
Prove Stop Store flat specification	*storeSpec();*

All these commands may also be called from the NeXTSTEP interface by typing them into the session window. Further commands may be used here, namely `sp`, `dc` and `dp`. `sp()` displays the current specification *SP* in the session window. `dc n scale` and `dp n scale` enter a tree representation of the n-th goal of the current conclusion ***conc*** or premise ***prem*** into the PostScript file *CONCn.eps* or *PREMn.eps*, respectively. The size of the coordinate system's units is multiplied with the real number `scale`.

8.2 Modules

The source code is divided into the main program (cf. 8.21) and the following ML structures.

Aux	auxiliary functions for list, number and string processing
Scan	specification scanner
AbstractSyntax	internal data types and formula simplification
Parse	specification parser
Unparse	specification and proof unparser
Substitute	substitution and unification
Search	functions for extracting subformulas
InferAux	auxiliary functions for proof manipulation, variable renaming, subsumption, goal and goal set simplification
Display	display functions
Start	commands for starting a proof (cf. 4.1)
Unify	unification commands (cf. 4.5)
Replace	term replacement commands (cf. 4.5)
Delete	atom and goal deletion commands (cf. 4.5)
Derive	resolving and rewriting commands (cf. 4.2 and 4.3). *Derive* is an ML functor, parameterized by a command for solving the premises of applied clauses
Draw	functions for visualizing point and rectangle solutions
Solve	goal solving commands (cf. 4.4). *Solve* is a functor, parameterized by commands for the individual resolving and rewriting steps performed by the solver
More	backtrack, simplification, subsumption, load and store commands (cf. 4.5 and 4.6)
Finish	commands for stopping a proof (cf. 4.6)

8.3 Aux

```
structure Aux = struct

  fun   up(k,0) = nil | up(k,n) = k::up(k+1,n-1)
  fun   first(s,0) = nil | first(x::s,n) = x::first(s,n-1)
  fun   front[x] = (nil,x) | front(x::s) = let val (s',z) = front(s) in (x::s',z) end
  fun   Max(s) = fold(max)(s)(~1)
  fun   mapconc(f)(s) = fold(op @)(map(f)(s))(nil)
  val   implod = mapconc(fn(x)=>x)

  fun   perms(nil) = [nil] | perms(x::L) = mapconc(insert(x))(perms(L))
  and   insert(x)(L) = let fun shift(L,x,nil) = [L@[x]]
```

```
                    |   shift(L,x,y::R) = (L@x::y::R)::shift(L@[y],x,R)
                in shift(nil,x,L) end

fun   map21(f)(x::s,y::s') = f(x,y)::map21(f)(s,s') | map21 _ _ = nil
fun   mapconc21(f)(s,s') = fold(op @)(map21(f)(s,s'))(nil)

fun   map22(f)(x::s,y::s') = let val (fst,snd) = f(x,y)
                                 val (rest1,rest2) = map22(f)(s,s')
                             in (fst::rest1,snd::rest2) end
|     map22 _ _ = (nil,nil)

datatype 'X Correct_or_Error = C of 'X | E

fun   for(s,x::s')(p) = if p(x) then C(s,x,s') else for(s@[x],s')(p) | for(_,nil) _ = E

fun   For(s)(p) = case for(nil,s)(p) of C(s,x,s') => (true,length(s)) | E => (false,0)

fun   filter(p)(x::s) = let val (filtered,rest) = filter(p)(s)
                        in if p(x) then (x::filtered,rest)  else (filtered,x::rest) end
|     filter _ _ = (nil,nil)

fun   filter2(f,p)(x::s,y::s') = if p(x,y) then filter2(f,p)(s,s')
                                           else f(x,y)::filter2(f,p)(s,s')
|     filter2 _ _ = nil

fun   filterall(p)(x::s,nil,acc,f) = if f(x) = nil then E else filterall(p)(s,acc,nil,f)
|     filterall(p)(x::s,y::s',acc,f)  = if p(x,y) then let fun g(z) = if z=x then y::f(x) else f(z)
                                                       in filterall(p)(x::s,s',y::acc,g) end
                                        else filterall(p)(x::s,s',y::acc,f)
|     filterall(p)(nil,_,_,f) = C(f)

fun   makeFuns([w]) = map(fn(x)=>[x])(w)
|     makeFuns(w::s) = let fun f(x) = map(fn(v)=>x::v)(makeFuns(s))
                       in mapconc(f)(w) end

fun   Exists(p)(x,s) = exists(fn(y)=>p(x,y))(s)

fun   checkallFuns(p,q)(nil,s) = false
|     checkallFuns(p,q)(s,s') = case filterall(p)(s,s',nil,fn(x)=>nil) of
                                     C(f) => Exists(q)(s,makeFuns(map(f)(s)))
                                |    E => false

fun   forall(p)(x::s) = p(x) andalso forall(p)(s) | forall _ _ = true

fun   forall2(p)(nil,nil) = true
|     forall2(p)(x::s,y::s') = p(x,y) andalso forall2(p)(s,s')
|     forall2 _ _ = false

fun   Equal(x,y) = x = y
val   member = Exists(Equal)

fun   forallExists(p)(s,s') = forall(fn(x)=>Exists(p)(x,s'))(s)

fun   disjoint(x::s,s') = not(member(x,s')) andalso disjoint(s,s') | disjoint(nil,_) = true

fun   disjointAcc(x::s,s') = not(member(x,s')) andalso disjointAcc(s,x::s')
|     disjointAcc(nil,_) = true

fun   is_set(s) = disjointAcc(s,nil)

fun   Subtract(p)(x::s,s') = if Exists(p)(x,s') then Subtract(p)(s,s') else x::Subtract(p)(s,s')
|     Subtract _ (nil,_) = nil
```

```
   fun    SubtractAcc(p)(x::s,s') = if Exists(p)(x,s')  then SubtractAcc(p)(s,s')
                                                        else x::SubtractAcc(p)(s,x::s')
   |      SubtractAcc _ (nil,_) = nil

   val    subtract = Subtract(Equal)
   fun    Makeset(p)(s) = SubtractAcc(p)(s,nil)
   val    makeset = Makeset(Equal)
   fun    Maximize(p)(s) = Reflect(p)(s,s)

   and    Reflect(p)(x::s,y::s') = if Exists(p)(x,s') then Reflect(p)(s,s') else Reflect(p)(s,s'@[x])
   |      Reflect _ (nil,s) = s

   fun    makeint("") = 0 | makeint(s) = let val i = size(s)-1
                                  in makeint(substring(s,0,i))*10+ordof(s,i)-48 end

   fun    number(s) = forall(digit)(explode(s))

   and    digit(s) = ord(s)>47 andalso ord(s)<58

   fun    name(x,0) = x | name(x,n) = x^makestring(n:int)

   fun    mapstring(x::s) = " "^makestring(x:int)^mapstring(s) | mapstring _ = " "

   fun    mapstrings[x] = mapstring(x)
   |      mapstrings(x::s) = mapstring(x)^","^mapstrings(s)
   |      mapstrings _ = " "

   fun    instring(file,str) = if end_of_stream(file) then str else instring(file,str^input_line(file))

   fun    Output(file,str) = (output(std_out,str); output(file,str); close_out(file))

end (*Aux*)
```

8.4 Scan

```
structure Scan = struct

   fun    syms(s) = symsLoop("",s,nil)

   and    symsLoop(w,"("::"*"::s,acc) = comment(w,s,acc)
   |      symsLoop(w,"<"::"="::"="::s,acc) = symbols(w,"<==",s,acc)
   |      symsLoop(w,"\\"::"/"::s,acc) = symbols(w,"\\/",s,acc)
   |      symsLoop(w,"/"::"\\"::s,acc) = symbols(w,"/\\",s,acc)
   |      symsLoop(w,"{"::"{"::s,acc) = symbols(w,"{{",s,acc)
   |      symsLoop(w,"}"::"}"::s,acc) = symbols(w,"}}",s,acc)
   |      symsLoop(w,"<"::"<"::s,acc) = symbols(w,"<<",s,acc)
   |      symsLoop(w,">"::">"::s,acc) = symbols(w,">>",s,acc)
   |      symsLoop(w,"<"::">"::s,acc) = symbols(w,"<>",s,acc)
   |      symsLoop(w,"<"::"="::s,acc) = symbols(w,"<=",s,acc)
   |      symsLoop(w,">"::"="::s,acc) = symbols(w,">=",s,acc)
   |      symsLoop(w,"<"::"-"::s,acc) = symbols(w,"<-",s,acc)
   |      symsLoop(w,":"::":"::s,acc) = symbols(w,"::",s,acc)
   |      symsLoop(w,"/"::"/"::s,acc) = symbols(w,"//",s,acc)
   |      symsLoop(w,x::"_"::"C"::s,acc) = if delimiter(x) then symbols(w,x^"_C",s,acc)
                                                           else symsLoop(w^x^"_C",s,acc)
   |      symsLoop(w,x::s,acc) = if delimiter(x) then symbols(w,x,s,acc)
                                                 else symsLoop(w^x,s,acc)
   |      symsLoop("",nil,acc) = acc
   |      symsLoop(w,nil,acc) = acc@[w]
```

```
   and   symbols("",w,s,acc) = if empty_space(w) then symsLoop("",s,acc)
                                                 else symsLoop("",s,acc@[w])
   |     symbols(v,w,s,acc) = if empty_space(w) then symsLoop("",s,acc@[v])
                                                else symsLoop("",s,acc@[v,w])

   and   comment(w,"*"::")"::s,acc) = symsLoop(w,s,acc)
   |     comment(w,_::s,acc) = comment(w,s,acc)

   and   delimiter(x) = empty_space(x) orelse
                      = Aux.member(x,[",","(",")","{","}","[","]","<",">","=","~","/",
                                      "+","-","*","?","!","&","@","#","%","","$","^"])

   and   empty_space(x) = Aux.member(x,[" ","\n","\t"])

end (*Scan*)
```

8.5 AbstractSyntax

```
structure AbstractSyntax = struct

   datatype symbol_mode = upvar | svar | srvar | sevar | ivar | ovar of int | orvar of int |
                          cop | coi | fup | fui | rep | rei | suspend |

            (* upvar = universal premise variable,
               svar = universal spec variable, srvar = read-only spec variable,
               sevar = existential spec variable, ivar = input variable,
               ovar = output variable, orvar = read-only output variable,
               cop = prefixed constructor, coi = infixed constructor,
               fup = prefixed function, fui = infixed function,
               rep = prefixed relation, rei = infixed relation,
               suspend: cf. Substitute.unify(n)

       The following constructors are only used in Parse: *)

       Bool | Num | Tuple | List of string * string | Assignment | Error

            (* Bool is true or false, Num is a number, List("[","]") = list,
               List("{","}") = set, List("{{","}}") = bag = multiset,
               List("<<",">>") = map with finite domain, Assignment = map update *)

   datatype Term = Var of string * symbol_mode | Op of string * Term list * symbol_mode |
                   Eq of Term * Term | FALSE | zero |

               (* The following constructors are only used inside this structure: *)

                  True | False | Not of Term | And of Term * Term | Or of Term * Term |
                  succ of Term | tuple of Term | empty | append of Term * Term | bag of Term |
                  set of Term | apply of Term list | update of Term * Term * Term | noEntry of Term


   type Signature = string list * (string list * (string (* = specname *) * int) list list)
                                * (string list * (string (* = specname *) * int) list list)
                                * string list * string list

   type Specification = string * Signature * (Term list list * Term list) list
                                           * (Term list list * Term list) list
                                           * (Term list list * Term list) list

   type State = Specification * string list * Term list list
                              * (Term list list list * Term list list list)
                              * Term list * int list
```

```
val emptyState = (("",(nil,(nil,nil),(nil,nil),nil,nil),nil,nil,nil),nil,nil,(nil,nil),nil,nil)
                 : State

datatype direction = backward | forward | test | next
datatype display = short | long
datatype clause_mode = axm | thm | conj

fun   mapVar(f)(Var(x)) = f(x) |
      mapVar(f)(Eq(l,r)) = mapVar(f)(l)@mapVar(f)(r) |
      mapVar(f)(Op(_,tlist,_)) = Aux.mapconc(mapVar(f))(tlist) |
      mapVar(f) _ = nil

val   VarsModes = mapVar(fn(x)=>[x])
val   Vars = mapVar(fn(x,_)=>[x])
val   iVars = mapVar(fn(x,mode)=>if mode = ivar then [x] else nil)
val   upVars = mapVar(fn(x,mode)=>if mode = upvar then [x] else nil)

fun   readOnly(Var(_,orvar _)) = true |
      readOnly(Eq(l,r)) = readOnly(l) orelse readOnly(r) |
      readOnly(Op(_,tlist,_)) = exists(readOnly)(tlist) |
      readOnly _ = false

fun   constructor(Var _ ) = true |
      constructor(Op(_,tlist,cop)) = Aux.forall(constructor)(tlist) |
      constructor(Op(_,tlist,coi)) = Aux.forall(constructor)(tlist) |
      constructor _ = false

exception wrong_lhs and wrong_rhs

fun  lhs(Eq(l,_)) = l | lhs _ = raise wrong_lhs

fun  rhs(Eq(_,r)) = r | rhs _ = raise wrong_rhs

fun   check_disjunct(gs) = if Aux.member([],gs) then [[]]
                           else let val gs = Aux.subtract(gs,[[FALSE]])
                                in if gs = nil then [[FALSE]] else gs end

fun   check_conjunct(g) = if Aux.member(FALSE,g) then [FALSE] else g

(* Syntactic equality *)

fun   eq_term(Var(x,ovar(m)),Var(y,orvar(n))) = x = y andalso m = n
|     eq_term(Var(x,orvar(m)),Var(y,ovar(n))) = x = y andalso m = n
|     eq_term(Eq(l1,r1),Eq(l2,r2)) = (eq_term(l1,l2) andalso eq_term(r1,r2)) orelse
                                     (eq_term(l1,r2) andalso eq_term(r1,l2))
|     eq_term(Op(F,tlist,_),Op(G,tlist',_)) = F = G andalso Aux.forall2(eq_term)(tlist,tlist')
|     eq_term(x,y) = x = y

val   Member = Aux.Exists(eq_term)

exception occur_check_failed

fun  occurs(x)(Var(y)) = eq_term(Var(x),Var(y)) |
     occurs(x)(Op(_,tlist,_)) = exists(occurs(x))(tlist) |
     occurs _ _ = raise occur_check_failed

(* Compilation of atoms and terms *)

exception wrong_predicate and wrong_function and wrong_application

fun   eval_atom(Op(P,tL,mode)) = (evalP(P,mode) o eval_termL)(tL)
```

```
|     eval_atom(Eq(l,r)) = eq(eval_term(l),eval_term(r))
|     eval_atom(at) = [at]

and  eval_term(Op("APPLY",[Op("fn",[Var(x,_),e],_),t],_))
       = (eval_term o beta(t,x) o alpha(nil))(e)
|    eval_term(Op("APPLY",[Op("fn",_,_),_],_)) = raise wrong_application
|    eval_term(Op(F,tL,mode)) = (evalF(F,mode) o eval_termL)(tL)
|    eval_term(t) = t

and   eval_termL(tL) = map(eval_term)(tL)

and   evalP("<>",_)[x,y] = neq(x,y)
|     evalP("<",_)[x,y] = less(x,y)
|     evalP(">",_)[x,y] = greater(x,y)
|     evalP("<=",_)[x,y] = leq(x,y)
|     evalP(">=",_)[x,y] = geq(x,y)
|     evalP("in",_)[x,w] = isIn(x,w)
|     evalP("notIn",_)[x,w] = notIn(x,w)
|     evalP("sub",_)[v,w] = subset(v,w)
|     evalP(P,mode)(tL) = if Aux.member(P,["<>","<",">","<=",">=","in","notIn","sub"])
                          then raise wrong_predicate else [Op(P,tL,mode)]

and   evalF("true",_)[] = True
|     evalF("false",_)[] = False
|     evalF("eq",_)[x,y] = Eqf(x,y)
|     evalF("neq",_)[x,y] = Neq(x,y)
|     evalF("lt",_)[x,y] = Less(x,y)
|     evalF("gt",_)[x,y] = Greater(x,y)
|     evalF("leq",_)[x,y] = Leq(x,y)
|     evalF("geq",_)[x,y] = Geq(x,y)
|     evalF("In",_)[x,w] = IsIn(x,w)
|     evalF("NotIn",_)[x,w] = NotIn(x,w)
|     evalF("Sub",_)[v,w] = Subset(v,w)
|     evalF("not",_)[b] = Not(b)
|     evalF("and",_)[b,c] = And(b,c)
|     evalF("or",_)[b,c] = Or(b,c)
|     evalF("ite",_)[b,x,y] = Ite(b,x,y)
|     evalF("0",_)[] = zero
|     evalF("s",_)[x] = succ(x)
|     evalF("+",_)[x,y] = plus(x,y)
|     evalF("-",_)[x,y] = minus(x,y)
|     evalF("*",_)[x,y] = times(x,y)
|     evalF("/",_)[x,y] = divF(x,y)
|     evalF("mod",_)[x,y] = MOD(x,y)
|     evalF("//",_)[x,y] = divC(x,y)
|     evalF("min",_)[x,y] = Min(x,y)
|     evalF("max",_)[x,y] = Max(x,y)
|     evalF("",_)(tL) = tuple(tL)
|     evalF("nil",_)[] = empty
|     evalF("::",_)[x,w] = append(x,w)
|     evalF("bag",_)[w] = bag(w)
|     evalF("set",_)[w] = set(w)
|     evalF("@",_)[v,w] = conc(v,w)
|     evalF("length",_)[w] = Length(w)
|     evalF("upd",_)[x,e,f] = update(x,e,f)
|     evalF("at",_)[f,x] = get(x,f)
|     evalF("cont",_)[t,c] = cont(t,c)
|     evalF("APPLY",_)(tL) = (evalA o apply)(tL)
|     evalF(F,mode)(tL)
       = if Aux.member(F,["true","false","eq","neq","lt","gt","leq","geq","In",
                          "NotIn","Sub","not","and","or","ite","0","s","+","-",
                          "*","/","//","min","max","nil","::","bag","set","@",
```

```
                             "length","upd","at","cont"])
            then raise wrong_function elseOp(F,tL,mode)

(* Application evaluation *)

   and    evalA(apply[apply[Op("eq_C",[c],_),x],y]) = (evalA o apply)[c,Eqf(y,x)]
   |      evalA(apply[apply[Op("neq_C",[c],_),x],y]) = (evalA o apply)[c,Neq(y,x)]
   |      evalA(apply[apply[Op("leq_C",[c],_),x],y]) = (evalA o apply)[c,Leq(y,x)]
   |      evalA(apply[apply[Op("geq_C",[c],_),x],y]) = (evalA o apply)[c,Geq(y,x)]
   |      evalA(apply[apply[Op("lt_C",[c],_),x],y]) = (evalA o apply)[c,Less(y,x)]
   |      evalA(apply[apply[Op("gt_C",[c],_),x],y]) = (evalA o apply)[c,Greater(y,x)]
   |      evalA(apply[apply[Op("In_C",[c],_),w],x]) = (evalA o apply)[c,IsIn(x,w)]
   |      evalA(apply[apply[Op("NotIn_C",[c],_),w],x]) = (evalA o apply)[c,NotIn(x,w)]
   |      evalA(apply[apply[Op("Sub_C",[c],_),v],w]) = (evalA o apply)[c,Subset(w,v)]
   |      evalA(apply[Op("not_C",[c],_),x]) = (evalA o apply)[c,Not(x)]
   |      evalA(apply[apply[Op("and_C",[c],_),x],y]) = (evalA o apply)[c,And(y,x)]
   |      evalA(apply[apply[Op("or_C",[c],_),x],y]) = (evalA o apply)[c,Or(y,x)]
   |      evalA(apply[apply[apply[Op("ite_C",[c],_),y],x],b]) = (evalA o apply)[c,Ite(b,x,y)]

   |      evalA(apply[Op("s_C",[c],_),x]) = (evalA o apply)[c,succ(x)]
   |      evalA(apply[apply[Op("+_C",[c],_),x],y]) = (evalA o apply)[c,plus(y,x)]
   |      evalA(apply[apply[Op("-_C",[c],_),x],y]) = (evalA o apply)[c,minus(y,x)]
   |      evalA(apply[apply[Op("*_C",[c],_),x],y]) = (evalA o apply)[c,times(y,x)]
   |      evalA(apply[apply[Op("/_C",[c],_),x],y]) = (evalA o apply)[c,divF(y,x)]
   |      evalA(apply[apply[Op("mod_C",[c],_),x],y]) = (evalA o apply)[c,MOD(y,x)]
   |      evalA(apply[apply[Op("//_C",[c],_),x],y]) = (evalA o apply)[c,divC(y,x)]
   |      evalA(apply[apply[Op("min_C",[c],_),x],y]) = (evalA o apply)[c,Min(y,x)]
   |      evalA(apply[apply[Op("max_C",[c],_),x],y]) = (evalA o apply)[c,Max(y,x)]
   |      evalA(apply[apply[Op("::_C",[c],_),w],x]) = (evalA o apply)[c,append(x,w)]
   |      evalA(apply[Op("bag_C",[c],_),w]) = (evalA o apply)[c,bag(w)]
   |      evalA(apply[Op("set_C",[c],_),w]) = (evalA o apply)[c,set(w)]
   |      evalA(apply[apply[Op("@_C",[c],_),w],v]) = (evalA o apply)[c,conc(v,w)]
   |      evalA(apply[Op("length_C",[c],_),w]) = (evalA o apply)[c,Length(w)]
   |      evalA(apply[apply[apply[Op("upd_C",[c],_),f],e],x]) = (evalA o apply)[c,update(x,e,f)]
   |      evalA(apply[apply[Op("at_C",[c],_),x],f]) = (evalA o apply)[c,get(x,f)]

   |      evalA(t) = t

(* alpha- and beta-reduction *)

        and      alpha(xL)(Op("fn",[Var(x,mode),e],_))
                   = Op("fn",[Var(x^"B",mode),alpha(x::xL)(e)],fup)
        |        alpha(xL)(Op(F,eL,mode)) = Op(F,map(alpha(xL))(eL),mode)
        |        alpha(xL)(Var(x,mode))
                   = if Aux.member(x,xL) then Var(x^"B",mode) else Var(x,mode)
        |        alpha _ (t) = t

        and      beta(t,x)(Op(F,eL,mode)) = Op(F,map(beta(t,x))(eL),mode)
        |        beta(t,x)(Var(y,mode)) = if x = y then t else Var(y,mode)
        |        beta _ (t) = t

(* Equality evaluation *)

   and    eq(t,u) = if eq_term(t,u) then []
                    else case (t,u) of (True,False) => [FALSE]
                                    |  (False,True) => [FALSE]
                                    |  (zero,succ _) => [FALSE]
                                    |  (succ _,zero) => [FALSE]
                                    |  (succ(x),succ(y)) => eq(x,y)
                                    |  (tuple(x),tuple(y)) => tupEq(x,y)
                                    |  (empty,w) => isEmpty(w)
                                    |  (w,empty) => isEmpty(w)
```

```
                              |  (append(x),w) => listEq(append(x),w)
                              |  (bag(v),bag(w)) => bagEq(empty,v,w)
                              |  (set(v),set(w)) => setEq(v,w)
                              |  (update(z),f) => mapEq(empty,update(z),f)
                              |  _ => constrEq(t,u)

   and   isEmpty(append _) = [FALSE]
   |     isEmpty(update _) = [FALSE]
   |     isEmpty(w) = [Eq(w,empty)]

   and   listEq(append(x,v),append(y,w)) = check_conjunct(eq(x,y)@eq(v,w))
   |     listEq(v,w) = [Eq(v,w)]

   and   bagEq(empty,empty,append _) = [FALSE]
   |     bagEq(empty,append _,empty) = [FALSE]
   |     bagEq(u,append(x,v),w) = (case isIn(x,w) of [] => bagEq(u,v,delete(x,w))
                                              |  [FALSE] => [FALSE]
                                              |  _ => bagEq(append(x,u),v,w))
   |     bagEq(u,v,w) = BagEq(conc(u,v),w)

   and   BagEq(append(x,empty),append(y,empty)) = eq(x,y)
   |     BagEq(v,w) = if eq_term(v,w) then [] else [Eq(bag(v),bag(w))]

   and   setEq(v,w) = check_conjunct(subset(v,w)@subset(w,v))

   and   mapEq(empty,empty,empty) = []
   |     mapEq(empty,empty,update _) = [FALSE]
   |     mapEq(empty,update _,empty) = [FALSE]
   |     mapEq(f,update(x,e,g),h)
           = (case isIn(x,h) of [] => check_conjunct(eq(e,get(x,h))@mapEq(f,delete(x,g),delete(x,h)))
                             |  [FALSE] => [FALSE]
                             |  _ => mapEq(conc(f,update(x,e,empty)),g,h))
   |     mapEq(f,g,h) = MapEq(conc(f,g),h)

   and   MapEq(update(x,e,empty),update(y,e',empty)) = check_conjunct(eq(x,y)@eq(e,e'))
   |     MapEq(f,g) = if eq_term(f,g) then [] else [Eq(f,g)]

   and   delete(x,append(y,w)) = if eq(x,y) = [] then w else append(y,delete(x,w))
   |     delete(x,update(y,e,f)) = if eq(x,y) = [] then delete(x,f) else update(y,e,delete(x,f))
   |     delete(x,w) = w

   and   tupEq(x,y) = (check_conjunct o Aux.mapconc21(eq))(x,y)

   and   constrEq(Op(F,x,mode1),Op(G,y,mode2)) = if constr_mode(mode1) andalso constr_mode(mode2)
                                             then if F <> G then [FALSE]
                                                  else (check_conjunct o Aux.mapconc21(eq))(x,y)
                                             else [Eq(Op(F,x,mode1),Op(G,y,mode2))]
   |     constrEq(t,u) = [Eq(t,u)]

   and   constr_mode(mode) = mode = cop orelse mode = coi

(* Predicate evaluation *)

   and  neq(t,u) = case eq(t,u) of [] => [FALSE]  |  [FALSE] => []  |  _ => [Op("<>",[t,u],rei)]

   and  less(_,zero) = [FALSE]
   |    less(zero,succ _) = []
   |    less(succ(x),succ(y)) = less(x,y)
   |    less(x,y) = [Op("<",[x,y],rei)]

   and  greater(x,y) = less(y,x)
```

```
   and  leq(x,y) = case eq(x,y) of [] => []  |  [FALSE] => less(x,y)
                                             |  _ => case less(x,y) of [] => []
                                                                   |  [FALSE] => eq(x,y)
                                                                   |  _ => [Op("<=",[x,y],rei)]
   and  geq(x,y) = leq(y,x)

   and  isIn(_,empty) = [FALSE]
   |    isIn(x,bag(w)) = isIn(x,w)
   |    isIn(x,set(w)) = isIn(x,w)
   |    isIn(x,append(y,w)) = (case eq(x,y) of [] => []
                                            |  [FALSE] => isIn(x,w)
                                            |  _ => case isIn(x,w) of
                                                         [] => []
                                                    |    [FALSE] => eq(x,y)
                                                    |    _ => [Op("in",[x,append(y,w)],rei)])
   |    isIn(x,update(y,e,f)) = (case eq(x,y) of [] => []
                                              |  [FALSE] => isIn(x,f)
                                              |  _ => case isIn(x,f) of
                                                           [] => []
                                                      |    [FALSE] => eq(x,y)
                                                      |    _ => [Op("in",[x,update(y,e,f)],rei)])
   |    isIn(x,w) = [Op("in",[x,w],rei)]

   and  notIn(x,w) = case isIn(x,w) of [] => [FALSE]  |  [FALSE] => []
                                                      |  _ => [Op("notIn",[x,w],rei)]

   and  subset(empty,_) = []
   |    subset(append(x,v),w) = check_conjunct(isIn(x,w)@subset(v,w))
   |    subset(v,append(x,w)) = if subset(v,w) = [] then []
                                else [Op("sub",[set(v),set(append(x,w))],rei)]
   |    subset(set(v),set(w)) = subset(v,w)
   |    subset(v,w) = if eq_term(v,w) then [] else [Op("sub",[set(v),set(w)],rei)]

(* Function evaluation *)

   and  Eqf(x,y) = case eq(x,y) of [] => True  |  [FALSE] => False  |  _ => Op("eq",[x,y],fup)
   and  Neq(x,y) = case eq(x,y) of [] => False  |  [FALSE] => True  |  _ => Op("neq",[x,y],fup)
   and  Less(x,y) = case less(x,y) of [] => True  |  [FALSE] => False  |  _ => Op("lt",[x,y],fup)
   and  Greater(x,y) = case less(y,x) of [] => True  |  [FALSE] => False  |  _ => Op("gt",[x,y],fup)
   and  Leq(x,y) = case leq(x,y) of [] => True  |  [FALSE] => False  |  _ => Op("leq",[x,y],fup)
   and  Geq(x,y) = case leq(y,x) of [] => True  |  [FALSE] => False  |  _ => Op("geq",[x,y],fup)
   and  IsIn(x,w) = case isIn(x,w) of [] => True  |  [FALSE] => False  |  _ => Op("In",[x,w],fup)
   and  NotIn(x,w) = case isIn(x,w) of [] => False  |  [FALSE] => True  |  _ => Op("NotIn",[x,w],fup)
   and  Subset(v,w) = case subset(v,w) of [] => True  |  [FALSE] => False  |  _ => Op("Sub",[v,w],fup)

        and     Not(True) = False
        |       Not(False) = True
        |       Not(b) = Op("not",[b],fup)

        and     And(True,b) = b
        |       And(b,True) = b
        |       And(False,b) = False
        |       And(b,False) = False
        |       And(b,c) = Op("and",[b,c],fui)

        and     Or(True,b) = True
        |       Or(b,True) = True
        |       Or(False,b) = b
        |       Or(b,False) = b
        |       Or(b,c) = Op("or",[b,c],fui)

        and     Ite(True,x,y) = x
```

```
|       Ite(False,x,y) = y
|       Ite(b,x,y) = Op("ite",[b,x,y],fup)

and     plus(zero,x) = x
|       plus(x,zero) = x
|       plus(succ(x),y) = succ(plus(x,y))
|       plus(x,succ(y)) = succ(plus(x,y))
|       plus(x,y) = Op("+",[x,y],fui)

and     minus(zero,_) = zero
|       minus(x,zero) = x
|       minus(succ(x),succ(y)) = minus(x,y)
|       minus(x,y) = Op("-",[x,y],fui)

and     times(zero,_) = zero
|       times(succ(x),y) = plus(times(x,y),y)
|       times(_,zero) = zero
|       times(x,succ(y)) = plus(x,times(x,y))
|       times(x,y) = Op("*",[x,y],fui)

and     divF(x,y) = case less(x,y) of [] => zero  |  [FALSE] => succ(divF(minus(x,y),y))
                                                  |  _ => Op("/",[x,y],fui)

and     MOD(x,y) = case less(x,y) of [] => x  |  [FALSE] => MOD(minus(x,y),y)
                                              |  _ => Op("mod",[x,y],fui)

and     divC(zero,y) = zero
|       divC(succ(x),y) = let val x = succ(x)
                          in case less(x,y) of [] => succ(zero)
                                            |  [FALSE] => succ(divC(minus(x,y),y))
                                            |  _ => Op("//",[x,y],fui) end

and     Min(x,y) = case less(x,y) of [] => x  |  [FALSE] => y  |  _ => Op("min",[x,y],fup)

and     Max(x,y) = case less(x,y) of [] => y  |  [FALSE] => x  |  _ => Op("max",[x,y],fup)

and     conc(empty,w) = w
|       conc(w,empty) = w
|       conc(append(x,v),w) = append(x,conc(v,w))
|       conc(update(x,e,f),g) = update(x,e,conc(f,g))
|       conc(v,w) = Op("@",[v,w],fui)

and     Length(empty) = zero
|       Length(append(x,w)) = succ(Length(w))
|       Length(w) = Op("length",[w],fup)

and     get(x,empty) = noEntry(x)
|       get(x,update(y,e,f)) = (case eq(x,y) of [] => e  |  [FALSE] => get(x,f)
                                                         |  _ => Op("at",[update(y,e,f),x],fui))
|       get(x,f) = Op("at",[f,x],fui)

(* Continuation passing *)

and     cont(Var(x),c) = (evalA o apply)[c,Var(x)]
|       cont(Op(F,nil,mode),c) = (evalA o apply)[c,Op(F,nil,mode)]
|       cont(Op(F,tL,_),c) = let val c = Op(F^"_C",[c],fup) in fold(cont)(tL)(c) end
|       cont(True,c) = (evalA o apply)[c,True]
|       cont(False,c) = (evalA o apply)[c,False]
|       cont(zero,c) = (evalA o apply)[c,zero]
|       cont(succ(x),c) = cont(x,Op("s_C",[c],fup))
|       cont(tuple(tL),c) = let val c = Op("tup_C",[c],fup) in fold(cont)(tL)(c) end
|       cont(empty,c) = (evalA o apply)[c,empty]
```

```
        |       cont(append(x,w),c) = cont(x,cont(w,Op("::_C",[c],fup)))
        |       cont(bag(w),c) = cont(w,Op("bag_C",[c],fup))
        |       cont(set(w),c) = cont(w,Op("set_C",[c],fup))
        |       cont(update(x,e,f),c) = cont(x,cont(e,cont(f,Op("upd_C",[c],fup))))
        |       cont(apply(tL),c) = let val c = Op("apply_C",[c],fup) in fold(cont)(tL)(c) end
        |       cont(t,c) = Op("cont",[t,c],fup)

(* Recompilation of atoms and terms *)

        fun     quote_term(Op(F,tL,mode)) = Op(F,quote_termL(tL),mode)
        |       quote_term(True) = Op("true",nil,cop)
        |       quote_term(False) = Op("false",nil,cop)
        |       quote_term(zero) = Op("0",nil,cop)
        |       quote_term(succ(x)) = Op("s",[quote_term(x)],cop)
        |       quote_term(tuple(tL)) = Op("",quote_termL(tL),cop)
        |       quote_term(empty) = Op("nil",nil,cop)
        |       quote_term(append(x,w)) = Op("::",quote_termL[x,w],coi)
        |       quote_term(bag(w)) = Op("bag",[quote_term(w)],cop)
        |       quote_term(set(w)) = Op("set",[quote_term(w)],cop)
        |       quote_term(update(x,e,f)) = Op("upd",quote_termL[x,e,f],cop)
        |       quote_term(noEntry(x)) = Op("noEntryAt",[quote_term(x)],cop)
        |       quote_term(apply(tL)) = Op("APPLY",quote_termL(tL),fup)
        |       quote_term(t) = t

        and     quote_termL(tL) = map(quote_term)(tL)

        fun     quote_atom(Op(P,tL,mode)) = Op(P,quote_termL(tL),mode)
        |       quote_atom(Eq(l,r)) = Eq(quote_term(l),quote_term(r))
        |       quote_atom(at) = at

        val     eval_goal = map(quote_atom) o check_conjunct o Aux.mapconc(eval_atom)

end (*AbstractSyntax*)
```

8.6 Parse

```
structure Parse = struct  open  AbstractSyntax

   val  member = Aux.member

   fun  get_mode(x::s,sign)
          = let val (cs,(fs,_),(ps,_),is,vs) = sign
            in if Aux.number(x) then (x,s,Num)
               else if x = "(" then (x,s,Tuple)
               else if x = "[" then (x,s,List(x,"]"))
               else if x = "{" then (x,s,List(x,"}"))
               else if x = "{{" then (x,s,List(x,"}}"))
               else if x = "<<" then (x,s,List(x,">>"))
               else if member(x,["true","false","0","s","rect","nil","bag","set","upd"])
                    then (x,s,cop)
               else if x = "::" then (x,s,coi)
               else if x = "<-" then (x,s,Assignment)
               else if member(x,["not","fn","cont","tup_C","apply_C","eq","neq","lt","gt",
                                 "leq","geq","min","max","In","NotIn","Sub","ite","length",
                                 "rewrite","rewriteAll"]) then (x,s,fup)
               else if member(x,["and","or","+","-","*","/","mod","//","@","at",">>"])
                    then (x,s,fui)
               else if member(x,["=","<",">","<=",">=","<>","in","notIn","sub"])
                    then (x,s,rei)
               else if member(x,cs) then if member(x,is) then (x,s,coi) else (x,s,cop)
               else if member(x,fs) then if member(x,is) then (x,s,fui) else (x,s,fup)
```

```
      else if member(x,ps) then if member(x,is) then (x,s,rei) else (x,s,rep)
      else if x = "REST" then (x,s,svar)
      else if member(x,vs) then let val next_sym = hd(s)
                                in if next_sym = "?" then (x,tl(s),srvar)
                                   else if next_sym = "!" then (x,tl(s),sevar)
                                   else if next_sym = "/\\" then (x,tl(s),upvar)
                                   else (x,s,svar) end
      else (x,s,Error) end

fun      inspec(s,dir) = let val file = open_in(dir^s)
                             val charlist = explode(Aux.instring(file,""))
                             val symlist = Scan.syms(charlist)
                             val (spec,parse,error) = spec(symlist)
                             val _ = close_in(file)
                             val check = open_out"PARSE"
                         in output(check,parse); close_out(check); (spec,error) end

and      spec(s) = let val (sign,rest) = signat(s)
                       val (name,bs,cs,(fs,fAxioms),(ps,pAxioms),is,vs) = sign
                       val specs = flatten(bs)
                       val (axms,rest,parse,error) = axioms(rest,sign)
                       val (no_signs,baseAxms,n) = collectAxms(specs,0)
                       fun f1(x) = (name^makestring(x:int),x)
                       val f2 = (map o map)(f1)
                       val fs = (fs,f2(fAxioms))
                       val ps = (ps,f2(pAxioms))
                       val no_sign = (n,cs,fs,ps,is,vs)
                       val sign = changeSign(no_sign::no_signs)
                       val (thms,rest,parse,error) = theorems(rest,sign,parse,error)
                       val (conjs,rest,parse,error) = conjects(rest,sign,parse,error)
                   in ((name,sign,baseAxms@axms,thms,conjs),parse,error) end

and      signat(s) = let val name::s = s
                         val (bs,rest) = base(s)
                         val (cs,rest) = consts(rest)
                         val (fs,fAxioms,rest) = functs(rest)
                         val (ps,pAxioms,rest) = preds(rest)
                         val (is,rest) = infixes(rest)
                         val (vs,rest) = vars(rest)
                     in ((name,bs,cs,(fs,fAxioms),(ps,pAxioms),is,vs),rest) end

and      flatten(bs) = let fun f(b) = let val (spec,error) = inspec(b,(!DIRS))
                                          val _ = read(b,error)
                                          val (_,sign,axms,_,_,_) = spec
                                          val (cs,fs,ps,is,vs) = sign
                                      in (cs,fs,ps,is,vs,axms) end
                       in map(f)(bs) end

and      read(spec,true) = let val file = open_in"PARSE"
                               val parse = Aux.instring(file,"")^"\n"
                               val str = " is syntactically incorrect:\n"
                           in close_in(file); output(std_out,spec^str^parse) end
|        read(spec,false) = output(std_out,spec^" is syntactically correct\n")

and      collectAxms(nil,k) = (nil,nil,k)
|        collectAxms((cs,fs,ps,is,vs,axms)::specs,k)
           = let val (no_signs,axms1,n) = collectAxms(specs,k+length(axms))
                 val no_sign = (k,cs,fs,ps,is,vs)
             in (no_sign::no_signs,axms@axms1,n) end

and      changeSign(nil) = (nil,(nil,nil),(nil,nil),nil,nil)
|        changeSign((n,cs,(fs,fAxioms),(ps,pAxioms),is,vs)::no_signs)
```

```
              = let val (cs',fs',ps',is',vs') = changeSign(no_signs)
                    val (fs',fAxioms') = fs'
                    val (ps',pAxioms') = ps'
                    fun f1(name,x) = (name,n+x)
                    val f2 = (map o map)(f1)
                    val fs = (fs@fs',f2(fAxioms)@fAxioms')
                    val ps = (ps@ps',f2(pAxioms)@pAxioms')
                    val cs = Aux.makeset(cs@cs')
                    val is = Aux.makeset(is@is')
                    val vs = Aux.makeset(vs@vs')
                in (cs,fs,ps,is,vs) end

and     base("base"::s) = symbols(s,nil)  |  base(s) = (nil,s)
and     consts("consts"::s) = symbols(s,nil)  |  consts(s) = (nil,s)
and     functs("functs"::s) = symbolsAndAxioms(s,nil,nil)  |  functs(s) = (nil,nil,s)
and     preds("preds"::s) = symbolsAndAxioms(s,nil,nil)  |  preds(s) = (nil,nil,s)
and     infixes("infixes"::s) = symbols(s,nil)  |  infixes(s) = (nil,s)
and     vars("vars"::s) = symbols(s,nil)  |  vars(s) = (nil,s)

and     symbols(x::s,syms)
          = if member(x,["consts","functs","preds","infixes","vars","axioms","theorems","conjects"])
            then (syms,x::s) else symbols(s,syms@[x])
|       symbols(nil,syms) = (syms,nil)

and     symbolsAndAxioms(x::s,syms,axms)
          = if member(x,["preds","infixes","vars","axioms","theorems","conjects"])
            then (syms,axms,x::s)
            else if Aux.number(x)
                 then let val (rest,axms) = Aux.front(axms)
                              val axms = rest@[axms@[Aux.makeint(x)]]
                          in symbolsAndAxioms(s,syms,axms) end
                 else symbolsAndAxioms(s,syms@[x],axms@[nil])
|       symbolsAndAxioms(nil,syms,axms) = (syms,axms,nil)

and     axioms("axioms"::s,sign) = clauses(s,sign,nil,"axioms",false)
|       axioms(s,_) = (nil,s,"",false)

and     theorems("theorems"::s,sign,parse,error) = clauses(s,sign,nil,parse^"\ntheorems",error)
|       theorems(s,_,parse,error) = (nil,s,parse,error)

and     conjects("conjects"::s,sign,parse,error) = clauses(s,sign,nil,parse^"\nconjects",error)
|       conjects(s,_,parse,error) = (nil,s,parse,error)

and     clauses("("::x::")"::s,sign,acc,parse,error)
          = let val tab = if size(x) = 1 then " " else ""
                    val (cl,rest,parse,error) = clause(s,sign,parse^"\n"^tab^"("^x^")",error)
                in clauses(rest,sign,acc@[cl],parse,error) end
|       clauses(x::s,sign,acc,parse,error)
          = if member(x,["theorems","conjects"])
            then (acc,x::s,parse,error)
            else clauses(s,sign,acc,parse^x,error)
|       clauses(nil,_,acc,parse,error) = (acc,nil,parse,error)

and     clause(s,sign,parse,error)
          = case goals(s,sign,parse,error) of
            (concl,"<=="::rest,parse,error)
               => let val (prem,rest,parse,error) = goal(rest,sign,parse^"<==",error)
                  in ((concl,prem),rest,parse,error) end |
            (concl,x::rest,parse,error)
               => if member(x,["(","theorems","conjects"])
                  then ((concl,nil),x::rest,parse,error)
                  else clause_error(rest,parse^x) |
```

```
            (concl,nil,parse,error) => ((concl,nil),nil,parse,error)

and     clause_error(s,parse) = ((nil,nil),s,parse^" NO CLAUSE ",true)

and     goals("FALSE"::s,sign,parse,error) = ([[FALSE]],s,parse^"FALSE",error)
|       goals(s,sign,parse,error) = list_goals(s,sign,nil,parse,error)

and     list_goals(s,sign,acc,parse,error)
          = case goal(s,sign,parse,error) of
                 (g,"\\/"::rest,parse,error)
                    => list_goals(rest,sign,acc@[g],parse^"\\/",error) |
                 (g,x::rest,parse,error)
                    => if member(x,["<==","(","theorems","conjects"])
                       then (acc@[g],x::rest,parse,error)
                       else goals_error(rest,parse^x) |
                 (g,nil,parse,error) => (acc@[g],nil,parse,error)

and     goals_error(s,parse) = (nil,s,parse^" NO GOALS ",true)

and     goal("{"::s,sign,parse,error) = goallist(s,sign,nil,parse^" { ",error)
|       goal(s,_,parse,error) = goal_error(s,parse)

and     goallist(s,sign,acc,parse,error)
          = case atom(s,sign,parse,error) of
                 (at,","::rest,parse,error)
                    => goallist(rest,sign,acc@[at],parse^", ",error) |
                 (at,"}"::rest,parse,error)
                    => (acc@[at],rest,parse^" } ",error) |
                 (at,rest,parse,error) => goal_error(rest,parse)

and     goal_error(s,parse) = (nil,s,parse^" NO GOAL ",true)

and     atom(s,sign,parse,error)
          = case get_mode(s,sign) of
            (P,"("::rest,rep)
              => let val (tlist,rest,parse,error)
                             = tuple(rest,sign,parse^P,error)
                 in (Op(P,tlist,rep),rest,parse,error) end |
            _ => let val (t,rest,parse,error)
                             = term(s,sign,parse,error)
                 in rest_atom(t,rest,sign,parse,error) end

and     rest_atom(t,nil,_,parse,error) = atom_error(nil,parse)
|       rest_atom(t,s,sign,parse,error)
          = case get_mode(s,sign) of
            ("=",rest,rei)
              => let val (t',rest,parse,error) = term(rest,sign,parse^" = ",error)
                 in (Eq(t,t'),rest,parse,error) end |
            (P,rest,rei)
              => let val (t',rest,parse,error) = term(rest,sign,parse^" "^P^" ",error)
                 in (Op(P,[t,t'],rei),rest,parse,error) end |
            (x,rest,_) => atom_error(rest,parse^x)

and     atom_error(s,parse) = (zero,s,parse^" NO ATOM ",true)

and     assignment(s,sign,parse,error)
          = let val (t,rest,parse,error) = term(s,sign,parse,error)
            in case get_mode(rest,sign) of
                    (_,rest,Assignment)
                      => let val (t',rest,parse,error) = term(rest,sign,parse^"<-",error)
                               in (Eq(t,t'),rest,parse,error) end |
                    (x,rest,_) => assignment_error(rest,parse^x) end
```

```
and     assignment_error(s,parse) = (zero,s,parse^" NO ASSIGNMENT ",true)

and     term(s,sign,parse,error)
          = case get_mode(s,sign) of
            (x,rest,Error) => term_error(rest,parse^x) |
            (x,rest,Num)
              => let val n = num_term(Aux.makeint(x))
                 in infix_rest(n,rest,sign,parse^x,error) end |
            (_,rest,Tuple)
              => (case tuple(rest,sign,parse,error) of
                        ([t],rest,parse,error) => infix_rest(t,rest,sign,parse,error) |
                        (tlist,rest,parse,error) => infix_rest(Op("",tlist,cop),rest,
                                                               sign,parse,error)) |
            (_,rest,List(br1,br2))
              => let val (tlist,rest,parse,error) = termlist(rest,sign,parse^br1,error,br2)
                       val t = seq_term(br1)(tlist)
                 in infix_rest(t,rest,sign,parse,error) end |
            (x,"("::rest,mode)
              => let val (tlist,rest,parse,error) = tuple(rest,sign,parse^x,error)
                       val t = if member(mode,[upvar,svar,srvar,sevar])
                               then Op("APPLY",Var(x,mode)::tlist,fup)
                               else if member(mode,[cop,fup])
                                    then Op(x,tlist,mode) else zero
                 in if t = zero then term_error(rest,parse^x^"(")
                              else rest_term(t,rest,sign,parse,error)  end |
            (x,rest,mode)
              => let val t = if member(mode,[upvar,svar,srvar,sevar])
                               then Var(x,mode)
                               else if member(mode,[cop,fup])
                                    then Op(x,nil,mode) else zero
                 in if t = zero then term_error(rest,parse^x)
                    else infix_rest(t,rest,sign,parse^x,error) end

and     term_error(s,parse) = (zero,s,parse^" NO TERM ",true)

and     infix_rest(t,nil,_,parse,error) = (t,nil,parse,error)
|       infix_rest(t,s,sign,parse,error)
          = case get_mode(s,sign) of
            (F,rest,mode)
              => if member(mode,[coi,fui])
                 then let val (t',rest,parse,error) = term(rest,sign,parse^F,error)
                      in (Op(F,[t,t'],mode),rest,parse,error) end
                 else (t,s,parse,error)

and     rest_term(t,nil,_,parse,error) = (t,nil,parse,error)
|       rest_term(t,s,sign,parse,error)
          = case get_mode(s,sign) of
            ("(",rest,_)
              => let val (tlist,rest,parse,error) = tuple(rest,sign,parse,error)
                 in rest_term(Op("APPLY",t::tlist,fup),rest,sign,parse,error) end |
            _ => infix_rest(t,s,sign,parse,error)

and     tuple(s,sign,parse,error) = termlist(s,sign,parse^"(",error,")")

and     termlist(s) = termlistLoop(s)(nil)

and     termlistLoop(s,sign,parse,error,">>")(acc) = mapLoop(s,sign,parse,error)(acc)
|       termlistLoop(s,sign,parse,error,br)(acc)
          = case term(s,sign,parse,error) of
                 (t,","::rest,parse,error)
                   => termlistLoop(rest,sign,parse^",",error,br)(acc@[t]) |
```

```
                (t,br2::rest,parse,error) => (acc@[t],rest,parse^br2,error) |
                (t,rest,parse,_) => (nil,rest,parse^" NO TERMLIST ",true)

and     mapLoop(s,sign,parse,error)(acc)
          = case assignment(s,sign,parse,error) of
                (t,",":rest,parse,error) => mapLoop(rest,sign,parse^",",error)(acc@[t]) |
                (t,">>"::rest,parse,error) => (acc@[t],rest,parse^">>",error) |
                (t,rest,parse,_) => (nil,rest,parse^" NO MAP ",true)

and     num_term(0) = Op("0",nil,cop) |
        num_term(n) = Op("s",[num_term(n-1)],cop)

and     seq_term"["(tL) = list_term(tL)
|       seq_term"{{"(tL) = Op("bag",[list_term(tL)],cop)
|       seq_term"{"(tL) = Op("set",[list_term(tL)],cop)
|       seq_term"<<"(tL) = (map_term o rev)(tL)

and     list_term(nil) = Op("nil",nil,cop)
|       list_term(t::tL) = Op("::",[t,list_term(tL)],coi)

and     map_term(nil) = Op("nil",nil,cop)
|       map_term(Eq(t,t')::tL) = Op("upd",[t,t',map_term(tL)],cop)

end (*Parse*)
```

8.7 Unparse

```
structure Unparse = struct  open  AbstractSyntax

        val     numbers = let fun f(_,n) = n in Aux.mapstring o map(f) end
        val     Size = 66

        fun     outspec(sp,s) = let val file = open_out(s)
                in output(file,spec(sp)); close_out(file) end

        and     spec(spec_name,(cs,(fs,ms),(ps,ns),is,vs),axms,thms,conjs)
                  = "\n"^spec_name^"\nconsts  "^symbols(cs,Size)^
                    "\nfuncts  "^symbolsAndAxioms(fs,ms,Size)^
                    "\npreds   "^symbolsAndAxioms(ps,ns,Size)^
                    "\ninfixes "^symbols(is,Size)^
                    "\nvars    "^symbols(vs,Size)^"\naxioms"^clauses(axms)^
                    "\ntheorems"^clauses(thms)^"\nconjects"^clauses(conjs)

        and     symbols(nil,_) = "" |
                symbols([x],n) = if size(x) <= n then x else blanks(8)^x |
                symbols(x::L,n) = let val k = size(x)+1
                                  in if k <= n then x^" "^symbols(L,n-k)
                                     else blanks(8)^x^" "^symbols(L,Size-k) end

        and     symbolsAndAxioms(nil,_,_) = "" |
                symbolsAndAxioms([x],[ax_nos],n) = let val s = x^numbers(ax_nos)
                                                   in if size(s) <= n then s else blanks(8)^s end |
                symbolsAndAxioms(x::L,ax_nos::ns,n)
                  = let val s = x^numbers(ax_nos)
                        val k = size(s)+2
                    in if k <= n then s^"  "^symbolsAndAxioms(L,ns,n-k)
                       else blanks(8)^s^"  "^symbolsAndAxioms(L,ns,Size-k) end

        and     clauses(cls) = list_clauses(cls,1)

        and     list_clauses(nil,_) = "" |
```

```
        list_clauses((concl,prem)::axms,n)
          = let val premise = if prem = nil then "" else " <== "^goal(prem,11)
                val tab = if n < 10 then "\n " else "\n"
            in tab^"("^makestring(n:int)^") "^conclusion(concl)^premise^list_clauses(axms,n+1) end

and     conclusion(nil) = "" |
        conclusion[g] = goal(g,7) |
        conclusion(g::gs) = goal(g,7)^"\n  \\/"^conclusion(gs)

and     goal(nil,_) = "TRUE" |
        goal([FALSE],_) = "FALSE" |
        goal(g,tab) = "{ "^atomlist(g,tab,Size,true)^" }"

and     atomlist(nil,_,_,_) = "" |
        atomlist([at],tab,n,b) = let val s = term(at)
                                     val k = size(s)
                                     val s = (implode o break(tab+1) o explode)(s)
                                 in if k <= n orelse b then s else blanks(tab)^s end |
        atomlist(at::g,tab,n,b) = let val s = term(at)
                                      val k = size(s)+2
                                      val s = (implode o break(tab+1) o explode)(s)
                                  in if k <= n orelse b
                                     then s^", "^atomlist(g,tab,n-k,false)
                                     else blanks(tab)^s^", "^atomlist(g,tab,Size-k,false) end

and     blanks(0) = "\n"  |  blanks(n) = blanks(n-1)^" "

and     break(tab)(L) = if length(L) <= Size then L
                        else Aux.first(L,Size)@blanks(tab)::break(tab)(nthtail(L,Size))

and     term(Var(x,upvar)) = x^"/\\" |
        term(Var(x,sevar)) = x^"!" |
        term(Var(x,srvar)) = x^"?" |
        term(Var(x,ovar(n))) = Aux.name(x,n) |
        term(Var(x,orvar(n))) = Aux.name(x,n)^"?" |
        term(Var(x,_)) = x |
        term(Eq(l,r)) = term(l)^" = "^term(r) |
        term(Op(P,tL,rep)) = P^"("^termlist(tL)^")" |
        term(Op(P,[t,u],rei)) = term(t)^" "^P^" "^term(u) |
        term(Op(F,nil,_)) = F |
|       term(Op("s",[t],_)) = let val (n,bv) = (num(t),true) handle no_num => (0,false)
                              in if bv then makestring(n+1) else "s("^term(t)^")" end
|       term(Op("::",[t,u],_)) = let val (tL,bv) = (seq(u),true)
                                                   handle no_seq => (nil,false)
                                 in if bv then "["^terms(t::tL)^"]"
                                         else enclosed(t)^"::"^enclosed(u) end
|       term(Op("bag",[Op("nil",_,_)],_)) = "nil"
|       term(Op("bag",[t],_)) = let val (tL,bv) = (seq(t),true)
                                                  handle no_seq => (nil,false)
                                in if bv then "{{"^terms(tL)^"}}"
                                        else "bag("^term(t)^")" end
|       term(Op("set",[Op("nil",_,_)],_)) = "nil"
|       term(Op("set",[t],_)) = let val (tL,bv) = (seq(t),true)
                                                  handle no_seq => (nil,false)
                                in if bv then "{"^terms(tL)^"}"
                                        else "set("^term(t)^")" end
|       term(Op("upd",[t,u,v],_)) = let val (tL,bv) = (assoc_list(v),true)
                                                      handle no_map => (nil,false)
                                    in if bv then "<<"^terms(tL@[Op("<-",[t,u],fui)])^">>"
                                       else "upd("^term(t)^","^term(u)^","^term(v)^")" end
|       term(Op(F,[t,u],fui)) = if size(F) < 3 then enclosed(t)^F^enclosed(u)
                                           else enclosed(t)^" "^F^" "^enclosed(u)
```

```
|       term(Op(F,[t,u],coi)) = term(Op(F,[t,u],fui))
|       term(Op("APPLY",t::tL,_)) = term(t)^"("^terms(tL)^")"
|       term(Op(F,tL,_)) = F^"("^terms(tL)^")"

and     terms(nil) = ""
|       terms[t] = term(t)
|       terms(t::tL) = term(t)^","^terms(tL)

and     enclosed(Op(F,tL,fui)) = "("^term(Op(F,tL,fui))^")"
|       enclosed(Op(F,tL,coi)) = "("^term(Op(F,tL,coi))^")"
|       enclosed(t) = term(t)

and     num(Op("0",_,_)) = 0
|       num(Op("s",[t],_)) = num(t) + 1
|       num _ = raise no_num

and     seq(Op("nil",_,_)) = nil
|       seq(Op("::",[t,tL],_)) = t::seq(tL)
|       seq _ = raise no_seq

and     assoc_list(Op("nil",_,_)) = nil
|       assoc_list(Op("upd",[t,u,tL],_)) = assoc_list(tL)@[Op("<-",[t,u],fui)]
|       assoc_list _ = raise no_map

fun     goals[[FALSE]] = "\n FALSE"
|       goals(gs) = list_goals(gs,1)

and     list_goals(g::gs,n) = let val tab = if n < 10 then "\n " else "\n"
                              in tab^"("^makestring(n:int)^") "^goal(g,7)^
                                 list_goals(gs,n+1) end
|       list_goals _ = ""

end (*Unparse*)
```

8.8 PostScript output

```
structure PS_output = struct

        open    AbstractSyntax

        infix   %
        datatype 'a tree = % of 'a * ('a tree list)

        fun     leaf(a) = a%nil

        fun     last[x] = x
        |       last(x::L) = last(L)

(* compute coordinates *)

        fun     strlg(a) = real(String.length(a))*7.0

        fun     leaf(a,x,y) = ((a,x,y)%nil,strlg(a))

        fun     newTree(a,y)(ctL,b) = let val (_,x,_)%_ = hd(ctL)
                                          val (_,z,_)%_ = last(ctL)
                                          val x = x*0.5+z*0.5
                                          val r = strlg(a)
                                          val b = if b >= r then b else r
                                      in ((a,x,y)%ctL,b) end
```

```
and     goal(nil)(x,y) = (leaf("TRUE",x,y),20.0)
|       goal[FALSE](x,y) = (leaf("FALSE",x,y),20.0)
|       goal[at](xy) = term(at)(xy)
|       goal(g)(x,y) = newTree("AND",y)(terms(g)(x,y-30.0))

and     term(Var(x,upvar))(i,j) = leaf("all "^x,i,j)
|       term(Var(x,sevar))(i,j) = leaf(x^"!",i,j)
|       term(Var(x,srvar))(i,j) = leaf(x^"?",i,j)
|       term(Var(x,ovar(n)))(i,j) = leaf(Aux.name(x,n),i,j)
|       term(Var(x,orvar(n)))(i,j) = leaf(Aux.name(x,n)^"?",i,j)
|       term(Var(x,_))(i,j) = leaf(x,i,j)
|       term(Eq(l,r))(xy) = term(Op("=",[l,r],rep))(xy)
|       term(Op(P,tL,rep))(x,y) = newTree(P,y)(terms(tL)(x,y-30.0))
|       term(Op(P,tL,rei))(xy) = term(Op(P,tL,rep))(xy)
|       term(Op(F,nil,_))(x,y) = leaf(F,x,y)
|       term(Op("",tL,_))(xy) = term(Op("tup",tL,coi))(xy)
|       term(Op("s",[t],_))(x,y) = let val (n,bv) = (Unparse.num(t),true)
                                            handle Unparse.no_num => (0,false)
                                   in if bv then leaf(makestring(n+1),x,y)
                                      else let val (ctL,b) = terms[t](x,y-30.0)
                                               val (_,x,_)%_ = hd(ctL)
                                           in (("s",x,y)%ctL,b) end end
|       term(Op("::",[t,u],_))(x,y) = let val (tL,bv) = (Unparse.seq(u),true)
                                                handle Unparse.no_seq => (nil,false)
                                          val F = if bv then "list" else "::"
                                          val tL = if bv then t::tL else [t,u]
                                      in newTree(F,y)(terms(tL)(x,y-30.0)) end
|       term(Op("bag",[Op("nil",_,_)],_))(x,y) = leaf("nil",x,y)
|       term(Op("bag",[t],_))(x,y) = let val (tL,bv) = (Unparse.seq(t),true)
                                               handle Unparse.no_seq => (nil,false)
                                         val tL = if bv then tL else [t]
                                     in newTree("bag",y)(terms(tL)(x,y-30.0)) end
|       term(Op("set",[Op("nil",_,_)],_))(x,y) = leaf("nil",x,y)
|       term(Op("set",[t],_))(x,y) = let val (tL,bv) = (Unparse.seq(t),true)
                                               handle Unparse.no_seq => (nil,false)
                                         val tL = if bv then tL else [t]
                                     in newTree("set",y)(terms(tL)(x,y-30.0)) end
|       term(Op("upd",[t,u,v],_))(x,y) = let val (tL,bv) = (Unparse.assoc_list(v),true)
                                                   handle Unparse.no_map => (nil,false)
                                             val F = if bv then "map" else "upd"
                                             val tL = if bv then tL@[Op("<-",[t,u],fui)]
                                                           else [t,u,v]
                                         in newTree(F,y)(terms(tL)(x,y-30.0)) end
|       term(Op("fn",[Var(x,_),e],_))(ij) = term(Op("fn "^x,[e],fup))(ij)
|       term(Op(F,tL,_))(x,y) = newTree(F,y)(terms(tL)(x,y-30.0))

and     terms(nil) _ = (nil,0.0)
|       terms(t::tL)(x,y) = let val (ct,b) = term(t)(x,y)
                                val (ctL,b') = terms(tL)(x+b,y)
                            in (ct::ctL,b+b') end

(* draw edges *)

fun     moveto(x:real,y:real) = makestring(x)^" "^makestring(y)^" moveto\n"

fun     lineto(a,x:real,y:real) = makestring(x)^" "^makestring(y+10.0)^
                                  " lineto ("^a^") center stroke\n"

fun     gotoRoot((a,x,y)%ctL) = moveto(x,y+10.0)^" ("^a^") center\n"^
                                traverse(ctL)(moveto(x,y))

and     traverse(nil) _ = ""
```

```
        |       traverse(((a,x,y)%ctL)::ctL')(move)
                  = move^lineto(a,x,y)^traverse(ctL)(moveto(x,y))^traverse(ctL')(move)

(* draw tree *)

        fun     goalheight(nil) = 1
        |       goalheight[FALSE] = 1
        |       goalheight[at] = termheight(at)
        |       goalheight(g) = fold(max)(map(termheight)(g))(0)+1

        and     termheight(Var _) = 1
        |       termheight(Eq(l,r)) = max(termheight(l),termheight(r))+1
        |       termheight(Op(_,nil,_)) = 1
        |       termheight(Op("s",[t],_)) = let val (n,bv) = (Unparse.num(t),true)
                                                      handle Unparse.no_num => (0,false)
                                        in if bv then 1 else termheight(t)+1 end
        |       termheight(Op("::",[t,u],_)) = let val (tL,bv) = (Unparse.seq(u),true)
                                                          handle Unparse.no_seq => (nil,false)
                                               val tL = if bv then t::tL else [t,u]
                                           in fold(max)(map(termheight)(tL))(0)+1 end
        |       termheight(Op("bag",[t],_)) = let val (tL,bv) = (Unparse.seq(t),true)
                                                         handle Unparse.no_seq => (nil,false)
                                              val tL = if bv then tL else [t]
                                          in fold(max)(map(termheight)(tL))(0)+1 end
        |       termheight(Op("set",[t],_)) = let val (tL,bv) = (Unparse.seq(t),true)
                                                         handle Unparse.no_seq => (nil,false)
                                              val tL = if bv then tL else [t]
                                          in fold(max)(map(termheight)(tL))(0)+1 end
        |       termheight(Op("upd",[t,u,v],_)) = let val (tL,bv) = (Unparse.assoc_list(v),true)
                                                             handle Unparse.no_map => (nil,false)
                                                  val tL = if bv then tL@[Op("<-",[t,u],fui)]
                                                                 else [t,u,v]
                                              in fold(max)(map(termheight)(tL))(0)+1 end
        |       termheight(Op(F,tL,_)) = fold(max)(map(termheight)(tL))(0)+1

        fun     maxX((_,x,_)%nil) = x
        |       maxX((_,x,_)%ctL) = maxXL(ctL)

        and     maxXL[ct] = maxX(ct)
        |       maxXL(ct::ctL) = maxXL(ctL)

        fun     draw(gs,n,dir,scale:real) = let val str = makestring(n)
                                                val file = if dir = backward then "CONC"^str^".eps"
                                                                               else "PREM"^str^".eps"
                                                val g = get_goal(gs,n)
                                                val file = open_out(file)
                                                val h = real(goalheight(g))*30.0
                                                val scalestr = makestring(scale)
                                                val (ct,_) = goal(g)(40.0,h)
                                                val code = "%!PS-Adobe-3.0 EPSF-3.0\n"^
                                                           "%%BoundingBox: 10 10 "^
                                                           makestring(scale*40.0+scale*maxX(ct))^" "^
                                                           makestring(scale*30.0+scale*h)^"\n"^
                                                           scalestr^" "^scalestr^" scale\n"^
                                                           "/Helvetica 9 selectfont\n"^
                                                           "/center { dup stringwidth"^
                                                           " pop -2 div -8 rmoveto show } def\n"^
                                                           gotoRoot(ct)^"showpage"
                                            in output(file,code); close_out(file) end

end (*PS_output*)
```

8.9 Substitute

```
structure Substitute = struct  open  AbstractSyntax

       val     yy = ("YY",ivar)
       val     YY = Var(yy)

       type    Variable = string * symbol_mode
       type    Substitution = Variable -> Term
       type    Subst2 = Substitution * Variable list
       type    Subst2L = Substitution * Variable list list
       type    GoalE = Term list * Subst2L * Specification * string list

       fun     id(x) = Var(x)

       infix   /
       fun     (t/x)(y) = if y = x then t else Var(y)

       fun     substitute(f)(Var(x)) = f(x)
       |       substitute(f)(Eq(l,r)) = Eq(substitute(f)(l),substitute(f)(r))
       |       substitute(f)(Op("APPLY",[t,Var(x)],_))
                 = (case f(x) of Op("",tlist,_) => substituteAPPLY(f)(t::tlist)
                              |               _ => substituteAPPLY(f)[t,Var(x)])
       |       substitute(f)(Op("APPLY",tlist,_)) = substituteAPPLY(f)(tlist)
       |       substitute(f)(Op(F,[Var(x)],mode))
                 = (case f(x) of Op("",tlist,_) => Op(F,tlist,mode)  |  _ => Op(F,[f(x)],mode))
       |       substitute(f)(Op(F,tlist,mode)) = Op(F,map(substitute(f))(tlist),mode)
       |       substitute _ (t) = t

       and     substituteAPPLY(f)(Var(x)::tlist)
                 = (case f(x) of Op(F,nil,mode) => Op(F,map(substitute(f))(tlist),mode)
                              |               _ => Op("APPLY",f(x)::map(substitute(f))(tlist),fup))
       |       substituteAPPLY(f)(tlist) = Op("APPLY",map(substitute(f))(tlist),fup)

       infix   &
       fun     f&g = substitute(g) o f

       infix   //
       fun     t//(x,svar) = (t/(x,svar))&(t/(x,srvar)) |
               t//(x,srvar) = t//(x,svar) |
               t//(x,ovar(n)) = (t/(x,ovar(n)))&(t/(x,orvar(n))) |
               t//(x,orvar(n)) = t//(x,ovar(n)) |
               t//x = t/x

       datatype Subst = C of Subst2 | CL of Subst2L | E

       fun     noSubst(t,u) = if eq_term(t,u) then C(id,nil) else E

       fun     unify(1)(Var(_,ivar),Var(_,sevar)) = E |
               unify(1)(Var(_,sevar),Var(_,ivar)) = E |
               unify(1)(Var(x),Var(y,sevar)) = C(Var(x)/(y,sevar),[x]) |
               unify(1)(Var(x,sevar),Var(y)) = C(Var(y)/(x,sevar),[y]) |
               unify(1)(_,Var(_,sevar)) = E |
               unify(1)(Var(_,sevar),_) = E |

                 (* unify(1) substitutes only one existential redex variable
                    for an existential spec conclusion variable *)

               unify(1)(Var(x,orvar(n)),Var(y,svar)) = C(Var(x,orvar(n))//(y,svar),nil) |
               unify(1)(Var(x,orvar(n)),Var(y,srvar)) = C(Var(x,orvar(n))//(y,svar),nil) |
               unify(1)(Var(x,orvar(n)),t) = noSubst(t,Var(x,ovar(n))) |
               unify(1)(t,Var(x,orvar(n))) = unify(1)(Var(x,orvar(n)),t) |
```

```
          (* unify(1) does not instantiate read-only existential redex variables *)

        unify(2)(Var(_,ivar),Var(_,sevar)) = E |
        unify(2)(Var(_,sevar),Var(_,ivar)) = E |
        unify(2)(Var(x),Var(y,sevar)) = C(Var(x)/(y,sevar),[x]) |
        unify(2)(Var(x,sevar),Var(y)) = C(Var(y)/(x,sevar),[y]) |
        unify(2)(_,Var(_,sevar)) = E |
        unify(2)(Var(_,sevar),_) = E |

          (* unify(2) substitutes only one existential redex variable
             for an existential spec variable *)

        unify(2)(Var(x,ivar),Var(y,svar)) = C(Var(x,ivar)//(y,svar),nil) |
        unify(2)(Var(x,ivar),Var(y,srvar)) = C(Var(x,ivar)//(y,svar),nil) |
        unify(2)(Var(x,ivar),t) = noSubst(t,Var(x,ivar)) |
        unify(2)(Var(x,ovar(n)),Var(y,svar)) = C(Var(x,ovar(n))//(y,svar),nil) |
        unify(2)(Var(x,ovar(n)),Var(y,srvar)) = C(Var(x,ovar(n))//(y,svar),nil) |
        unify(2)(Var(x,ovar(n)),t) = noSubst(t,Var(x,ovar(n))) |
        unify(2)(Var(x,orvar(n)),Var(y,svar)) = C(Var(x,orvar(n))//(y,svar),nil) |
        unify(2)(Var(x,orvar(n)),Var(y,srvar)) = C(Var(x,orvar(n))//(y,svar),nil) |
        unify(2)(Var(x,orvar(n)),t) = noSubst(t,Var(x,ovar(n))) |
        unify(2)(t,Var(x,ivar)) = unify(2)(Var(x,ivar),t) |
        unify(2)(t,Var(x,ovar(n))) = unify(2)(Var(x,ovar(n)),t) |
        unify(2)(t,Var(x,orvar(n))) = unify(2)(Var(x,orvar(n)),t) |

          (* unify(2) does not instantiate redex variables *)

        unify(3)(Var(x,ivar),Var(y,ivar)) = if x = y then C(id,nil) else E |
        unify(3)(Var(_,ivar),_) = E |
        unify(3)(_,Var(_,ivar)) = E |
        unify(3)(Var(x,ovar(n)),Var(y,sevar)) = C(Var(x,ovar(n))//(y,sevar),nil) |
        unify(3)(Var(x,ovar(n)),t) = noSubst(t,Var(x,ovar(n))) |
        unify(3)(Var(x,orvar(n)),Var(y,sevar)) = C(Var(x,orvar(n))//(y,sevar),nil) |
        unify(3)(Var(x,orvar(n)),t) = noSubst(t,Var(x,ovar(n))) |
        unify(3)(t,Var(x,ovar(n))) = unify(3)(Var(x,ovar(n)),t) |
        unify(3)(t,Var(x,orvar(n))) = unify(3)(Var(x,orvar(n)),t) |

          (* unify(3) does not instantiate neither universal nor premise variables *)

        unify(n)(Var(x),Var(y)) = C(Var(x)//y,nil) |
        unify(n)(t,Var(x)) = if occurs(x)(t) then E else C(t//x,nil) |
        unify(n)(Var(x),t) = if occurs(x)(t) then E else C(t//x,nil) |

        unify(n)(Eq(l1,r1),Eq(l2,r2))
          = (case unifyComm(n)([l1,r1],[l2,r2]) of CL(f,exVars) => C(f,Aux.implod(exVars))
                                                 |  E => E) |

        unify(n)(Op(F,tL1,mode1),Op(G,tL2,mode2))
          = if F = G
            then case unifyall(n)(tL1,tL2) of CL(f,exVars) => C(f,Aux.implod(exVars))
                                            |  E => E
            else if F = "rewriteAll" andalso G = "rewrite" then unifyDistr(n)(tL1,hd(tL2))
            else if F = "rewrite" andalso G = "rewriteAll" then unifyDistr(n)(tL2,hd(tL1))
            else if n = 1 andalso suspendable(mode1,mode2) then C(id,[("",suspend)]) else E |
        unify _ _ = E

and     suspendable(mode1,mode2) = (constr_mode(mode1) andalso not(constr_mode(mode2))) orelse
                                   (constr_mode(mode2) andalso not(constr_mode(mode1)))

and     suspended(_,exVars) = let fun f(_,mode) = mode in Aux.member(suspend,map(f)(exVars)) end
and     suspendedL(_,exVars) = let fun f(_,mode) = mode
```

```
                              in Aux.member(suspend,map(f)(Aux.implod(exVars))) end

and     unifyDistr _ (nil,redex) = C(redex/("REST",svar),nil) |
        unifyDistr(n)(t::tL,redex)
          = (case search(n)(t,redex,nil) of
             (C(f,exVars),redex) => (case unifyDistr(n)(map(substitute(f))(tL),
                                          substitute(f)(redex)) of
                                 C(g,exVars') => C(f&g,exVars@exVars') |
                         E => E) |
             (E,FALSE) => E) |
        unifyDistr _ _ = E

and     search(n)(t1,Op("::",[t2,t3],_),tL)
          = (case unify(n)(t1,t2) of C(f,exVars) => (C(f,exVars),concat(t3,tL)) |
                                     E => search(n)(t1,t3,t2::tL)) |
        search _ _ = (E,FALSE)

and     concat(t,nil) = t |
        concat(t,x::L) = Op("::",[x,concat(t,L)],cop)

and     unifyall(n)(nil,nil) = CL(id,nil) |
        unifyall(n)(t1::tL1,t2::tL2)
          = (case unify(n)(t1,t2) of
                 C(f,exVars) => let val tL1 = map(substitute(f))(tL1)
                                    val tL2 = map(substitute(f))(tL2)
                                in case unifyall(n)(tL1,tL2) of
                                        CL(g,exVars') => CL(f&g,exVars::exVars') |
                                        E => E end |
                 E => E) |
        unifyall _ _ = E

and     unifyComm(n)(tL1,tL2) = (get(nil) o map(fn(tL)=>unifyall(n)(tL1,tL)) o Aux.perms)(tL2)

and     get(nil)(nil) = E
|       get(L)(nil) = CL(hd(L))
|       get(L)(CL(f,exVars)::L') = if Aux.member(suspend,map(#2)(Aux.implod(exVars)))
                                   then get((f,exVars)::L)(L') else CL(f,exVars)
|       get(L)(E::L') = get(L)(L')

exception no_unifier and no_matching

fun     unifies(t1,t2) = case unify(1)(t1,t2) of C(f,exVars) => (f,exVars) |
                                                 E => raise no_unifier

fun     unifyTerms(tL1,tL2) = case unifyall(1)(tL1,tL2) of CL(f,exVars) => (f,exVars) |
                                                           E => raise no_unifier

fun     unifyAtoms(tL1,tL2) = case unifyComm(1)(tL1,tL2) of CL(f,exVars) => (f,exVars) |
                                                            E => raise no_unifier

fun     matchTerms(tL1,tL2) = case unifyall(2)(tL1,tL2) of CL(f,exVars) => (f,exVars) |
                                                           E => raise no_matching

fun     matchAtoms(tL1,tL2) = case unifyComm(2)(tL1,tL2) of CL(f,exVars) => (f,exVars) |
                                                            E => raise no_matching

fun     subMatch(tL1,tL2) = case unifyall(3)(tL1,tL2) of E => false | _ => true

end (*Substitute*)
```

8.10 Search

```
structure Search = struct  open  Substitute

    fun     axioms(sym,(sym'::symbols,axms::rest))
              = if sym = sym' then axms else axioms(sym,(symbols,rest)) |
            axioms(sym,(nil,_)) = nil

    exception error and clause_not_found and goal_not_found and
              atom_not_found and term_not_found and list_elem_not_found

    fun     single(s,n) = case Aux.for(nil,s)(fn(x) => x = nth(s,n-1)) of
                            Aux.C(w) => w | _ => raise error

    fun     clause(cls,n) = #2(single(cls,n)) handle error => raise clause_not_found

    fun     goal(gs,n) = single(gs,n) handle error => raise goal_not_found

    fun     atom(g,n) = single(g,n) handle error => raise atom_not_found

    fun     sublist(xs,ns) = subLoop(xs,nil,ns,nil)

    and     subLoop(xs,ys,n::ns,acc)
              = let val (xs,x,ys) = single(xs@ys,n)
                                      handle error => raise list_elem_not_found
                    fun inc(k) = if k > n then k-1 else k
                    val (acc,xs) = subLoop(xs@ys,nil,map(inc)(ns),acc)
                in (x::acc,xs) end |
            subLoop(xs,ys,nil,acc) = (acc,xs@ys)

    fun     disjunct(gs,ns) = Loop(nil,gs,ns,nil)

    and     Loop(rest,g::gs,n::ns,acc)
              = let val (g1,at,g2) = atom(g,n) in Loop(rest@g1@g2,gs,ns,acc@[at]) end |
            Loop(rest,_,nil,acc) = (acc,rest) |
            Loop _ = raise atom_not_found

    fun     term(Eq(l,r),1::ns,context) = term(l,ns,substitute(Eq(YY,r)/yy)(context)) |
            term(Eq(l,r),2::ns,context) = term(Eq(r,l),1::ns,context) |
            term(Op(F,tlist,mode),n::ns,context) = termlist(nil,tlist,1,n,ns,context,F,mode) |
            term(t,nil,context) = (context,t) |
            term _ = raise term_not_found

    and     termlist(xs,t::ys,k,n,ns,context,F,mode)
              = if k = n then let val c = Op(F,xs@(YY::ys),mode)
                                in term(t,ns,substitute(c/yy)(context)) end
                         else termlist(xs@[t],ys,k+1,n,ns,context,F,mode) |
            termlist _ = raise term_not_found

    fun     function(Op("APPLY",t::_,_)) = function(t) |
            function(Op(F,_,_)) = F

    fun     solved_eq(Eq(Var(_,ivar),r)) = constructor(r) |
            solved_eq(Eq(l,Var(_,ivar))) = constructor(l) |
            solved_eq _ = false

    fun     solution(gs) = Aux.for(nil,gs)(Aux.forall(solved_eq))

end (*Search*)
```

8.11 InferAux

```
structure InferAux = struct  open  Substitute

(* Expansion operators *)

       fun     current_goal_set(state,dir) = let val (_,_,_,(front,rear),_,_) = state
                                             in if dir = forward then hd(rear) else hd(front) end

       fun     new_inVars(state,inVars) = let val (spec,_,ini,proof,fin,conj_nos) = state
                                          in (spec,inVars,ini,proof,fin,conj_nos) end

       fun     extend_proof(state,gs,dir)
                 = let val (spec,inVars,ini,(front,rear),fin,conj_nos) = state
                       val proof = if dir = forward then (front,gs::rear) else (gs::front,rear)
                   in (spec,inVars,ini,proof,fin,conj_nos) end

       fun     undo(state,n,dir)
                 = let val (spec,inVars,ini,(front,rear),fin,conj_nos) = state
                       val proof = if dir = forward then nthtail(rear,n) else nthtail(front,n)
                       val proof = if dir = forward then (front,proof) else (proof,rear)
                   in (spec,inVars,ini,proof,fin,conj_nos) end

       fun     show(gs,dir) = let val file = open_append"EXPANSION"
                                  val str1 = case dir of backward => "conclusion:" |
                                                         forward => "premise:" |
                                                         test => "reduct:" |
                                                         next => "goal to be solved:"
                                  val gs = if dir <> test then gs
                                           else let val gs = tl(gs)
                                                in if gs = nil then [[FALSE]] else gs end
                                  val str2 = "\n"^str1^Unparse.goals(gs)^"\n"
                              in Aux.Output(file,str2) end

(* Renaming of variables *)

       exception Initialize and Finalize

       fun     initializeAll(gs) = let fun f(x,svar) = Var(x,ivar) |
                                           f(x,sevar) = Var(x,ovar(0)) |
                                           f(x,srvar) = Var(x,orvar(0)) |
                                           f _ = raise Initialize
                                   in (map o map o substitute)(f)(gs) end

       fun     initialize(gs,inVars) = let fun f(x,svar) = if Aux.member(x,inVars)
                                                           then Var(x,ivar) else Var(x,ovar(0)) |
                                               f(x,sevar) = Var(x,ovar(0)) |
                                               f(x,srvar) = Var(x,orvar(0)) |
                                               f _ = raise Initialize
                                       in (map o map o substitute)(f)(gs) end

       fun     rename_goal(g,inVars) = let fun index(x) = new_no(x,g,inVars)
                                           fun f(x,upvar) = if Aux.member(x,inVars)
                                                            then f(x^"i",upvar) else Var(x,ivar) |
                                               f(x,svar) = Var(x,ovar(index(x))) |
                                               f(x,srvar) = Var(x,orvar(index(x))) |
                                               f(x,sevar) = Var(x,ovar(index(x))) |
                                               f(x) = Var(x)
                                       in (map o substitute)(f)(g) end

       and     new_no(x,g,inVars) = let val nlist = Aux.mapconc(old_nos(x))(g)
                                        val n = if Aux.member(x,inVars) then 0 else ~1
```

```
                          in Aux.Max(n::nlist)+1 end

    and     old_nos(x)(Var(y,ovar(n))) = if x = y then [n] else nil |
            old_nos(x)(Eq(l,r)) = old_nos(x)(l)@old_nos(x)(r) |
            old_nos(x)(Op(_,tlist,_)) = Aux.mapconc(old_nos(x))(tlist) |
            old_nos _ _ = nil

    fun     rename_disjunct(nil,g) = nil |
            rename_disjunct(g::gs,g') = let fun index(x) = new_no(x,g',nil)
                                            fun f(x,ovar(n)) = Var(x,ovar(index(x)+n)) |
                                                f(x,orvar(n)) = Var(x,orvar(index(x)+n)) |
                                                f(x) = Var(x)
                                            val g = map(substitute(f))(g)
                                        in g::rename_disjunct(gs,g@g') end

    fun     finalize(gs) = map(finalize_goal)(gs)
    and     finalize_goal(g) = map(finalize_atom)(g)
    and     finalize_atom(at) = let fun f(x,ivar) = Var(x,svar) |
                                        f(x,ovar(n)) = Var(Aux.name(x,n),sevar) |
                                        f(x,orvar(n)) = Var(Aux.name(x,n),sevar) |
                                        f _ = raise Finalize
                                in substitute(f)(at) end

    fun     rename_concl(g) = map(rename_atom)(g)
    and     rename_atom(at) = let fun f(x,ovar(n)) = Var(Aux.name(x,n),sevar) |
                                      f(x,orvar(n)) = Var(Aux.name(x,n),sevar) |
                                      f(x) = Var(x)
                              in substitute(f)(at) end

    fun     change_mode(gs,vars,new_mode)
              = let fun f(x,mode) = if Aux.member(x,vars) then Var(x,new_mode) else Var(x,mode)
                in (map o map o substitute)(f)(gs) end

    fun     intoUniVars(gs,vars) = change_mode(gs,vars,svar)

    fun     intoExVars(g,vars) = (hd o change_mode)([g],vars,sevar)

(* Subsumption *)

    fun     subsumes(g1,g2) = g1 = [FALSE] orelse g2 = [] orelse
                              let fun p(at1,at2) = subMatch([rename_atom(at1)],[at2])
                                  fun q(g1,g2) = subMatch(rename_concl(g1),g2)
                              in Aux.checkallFuns(p,q)(g2,g1) end

    fun     isomorphic(g1,g2) = subsumes(g1,g2) andalso subsumes(g2,g1)

    val     isoMember = Aux.Exists(isomorphic)

    val     implies = Aux.forallExists(subsumes)

    val     implicates = Aux.Maximize(subsumes)

(* Simplification *)

    infix //

    fun     del_eqs(g) = case Aux.for(nil,g)(constr_eq) of
                              Aux.E => g |
                              Aux.C(rest1,eq,rest2) => let val f = (map o substitute o make_sub)(eq)
                                                       in (del_eqs o f)(rest1@rest2) end

    and     constr_eq(Eq(Var(_,ivar),Var(_,ivar))) = false |
```

```
        constr_eq(Eq(Var(_,ivar),Var _)) = true |
        constr_eq(Eq(Var _,Var(_,ivar))) = true |
        constr_eq(Eq(Var(_,ivar),_)) = false |
        constr_eq(Eq(_,Var(_,ivar))) = false |
        constr_eq(Eq(Var _,r)) = constructor(r) |
        constr_eq(Eq(l,Var _)) = constructor(l) |
        constr_eq _ = false

and     make_sub(Eq(Var(x,ivar),Var(y))) = Var(x,ivar) // y |
        make_sub(Eq(Var(x),Var(y,ivar))) = Var(y,ivar) // x |
        make_sub(Eq(Var(x),r)) = r // x |
        make_sub(Eq(l,Var(x))) = l // x

fun     simplifyLoop(g) = let val g = eval_goal(g)
                              val n = length(g)
                              val g = del_eqs(g)
                          in if length(g) = n then g else simplifyLoop(g) end

val     simplify = Aux.Makeset(eq_term) o simplifyLoop

val     simplify_disjunct = implicates o check_disjunct o map(simplify)

(* Derivation auxiliaries *)

fun     SUBST(g,(sub,_),inVars) = let val inSub = map(fn(x)=>sub(x,ivar))(inVars)
                                      val EQsub = make_EQsub(inVars,inSub)
                                  in EQsub @ (map o substitute)(sub)(g) end

and     make_EQsub(inVS) = let fun f(x,t) = Eq(Var(x,ivar),t)
                               fun p(x,t) = (Var(x,ivar) = t)
                           in Aux.filter2(f,p)(inVS) end

fun     DESCENT((sub,_),inVars,b) = let val in1Sub = map(fn(x)=>sub(x,ivar))(inVars)
                                        val in2Sub = map(fn(x)=>sub(x,svar))(inVars)
                                    in Eq(make_descent(in1Sub,in2Sub),Op(b,nil,cop)) end

and     make_descent([t1],[t2]) = Op(">>",[t1,t2],fui) |
        make_descent(tL1,tL2) = Op(">>",[Op("",tL1,fup),Op("",tL2,fup)],fui)

exception exVars_is_ambiguous and exVars_contains_reductVars

open    Aux

fun     CHECK(reduct,(_,exVars)) = let val reductVars = mapconc(VarsModes)(reduct)
                                   in if forall(is_set)(exVars)
                                      then if disjoint(implod(exVars),reductVars)
                                           then () else raise exVars_contains_reductVars
                                      else raise exVars_is_ambiguous end

fun     REWRITE(spec,inVars,gs,t_poss,concl,prem,cl_mode,Solve)
          = let val at_nos = map(hd)(t_poss)
                val t_poss = map(tl)(t_poss)
                val gs = rename_disjunct(gs,nil)
                val (atoms,rest) = Search.disjunct(gs,at_nos)
                fun f(at,t_pos) = Search.term(at,t_pos,YY)
                val (contexts,terms) = map22(f)(atoms,t_poss)
                val lefts = map(lhs)(concl)
                val sub = unifyTerms(terms,lefts)
                val descent = DESCENT(sub,inVars,"true")
                val Prem = if cl_mode = conj then descent::prem else prem
                val _ = if Prem <> nil andalso Solve(Prem,sub,spec,inVars) = [[FALSE]]
                        then raise no_unifier else ()
```

```
              val reduct = if suspendedL(sub)
                           then if cl_mode = axm
                                then let val redex = hd(gs)
                                         val instance = SUBST(redex,sub,inVars)
                                     in if subsumes(redex,instance)
                                        then raise no_unifier else instance end
                                else raise no_unifier
                           else let val Eq(_,right)::_ = concl
                                    val f = (map o substitute)(right/yy)
                                    val reduct = prem@f(contexts)@rest
                                    val reduct = SUBST(reduct,sub,inVars)
                                    val _ = if cl_mode = axm then ()
                                                            else CHECK(reduct,sub)
                                                    (* check variable condition *)
                                in if cl_mode = conj then descent::reduct else reduct end
              val reduct = rename_goal(reduct,inVars)
              val inVars = makeset(inVars@mapconc(iVars)(reduct))
              val reduct = if cl_mode <> thm then simplify(reduct)
                                             else Makeset(eq_term)(reduct)
          in (reduct,inVars) end

fun     LOOP(Delete,Derive)(pos,funct)(state,g_no,redex_lost,dir,display)
          = let val gs = current_goal_set(state,dir)
              val (_,g,_) = Search.goal(gs,g_no)
              val (_,at,_) = Search.atom(g,hd(pos))
              val t = if funct then #2(Search.term(at,tl(pos),YY)) else at
              val F = Search.function(t)
              val (spec,_,_,_,_,_) = state
              val (_,(_,fs,ps,_,_),_,_,_)= spec
              val axioms = if funct then fs else ps
              val ax_nos = Search.axioms(F,axioms)
              val (state,r,fr)  (* r (fr) = number of (false) reducts *)
                     = Loop(ax_nos,state)(0,0,g_no,pos,dir,display,Derive)
              val (state,proof_incr) = if r > 0 andalso redex_lost
                                       then (Delete(state,[g_no],dir,short),r+1)
                                       else (state,r)
              val gs = current_goal_set(state,dir)
              val state = undo(state,proof_incr,dir)
              val gs = check_disjunct(gs)
              val state = extend_proof(state,gs,dir)
          in if r = 0 orelse display = short then (state,r,fr)
                                             else (show(gs,dir); (state,r,fr)) end

and     Loop(nil,state) _ = (state,0,0) |

        Loop(ax_no::ax_nos,state)(r,fr,g_no,pos,dir,display,Derive)
          = let val (applicable, (state,reduct_is_false))
                    = (true, Derive(state,g_no,pos,ax_no,axm,false,dir,short,display))
                      handle no_unifier => (false, (state,false))
              val r = if applicable then r+1 else r
              val fr = if reduct_is_false then fr+1 else fr
          in if ax_nos = nil then (state,r,fr)
             else Loop(ax_nos,state)(r,fr,g_no,pos,dir,display,Derive) end

end (*InferAux*)
```

8.12 Display

```
structure Display = struct  open  AbstractSyntax

        fun     start(conc,prem) = let val file = open_out"EXPANSION"
```

```
                          val str = "\ninitial conclusion:"^Unparse.goals(conc)^
                                    "\ninitial premise:"^Unparse.goals(prem)^"\n"
                      in Aux.Output(file,str) end

fun     start_solve(g_no,dir,depth,Dontsubsume,OneSol,Innermost,Breadthfirst)
          = let val file = open_append"EXPANSION"
                val str1 = if dir = backward then "\nconclusion" else "\npremise"
                val str2 = if OneSol then "\n" else "\nget all solutions\n"
                val str3 = if Breadthfirst then "proceed breadthfirst"
                                           else "proceed depthfirst"
                val str4 = if Innermost then " and innermost" else " and outermost"
                val str5 = if Dontsubsume then "\n" else " and detect loops\n"
                val str = str1^" goal "^makestring(g_no:int)^
                          " is to be solved in "^makestring(depth:int)^
                          " steps"^str2^str3^str4^str5
            in Aux.Output(file,str) end

fun     finish_solve(solved,narrowed)
          = let val file = open_append"EXPANSION"
                val str1 = if lg = 1 then " has" else "s have"
                val str = "\nsolved goals:"^Aux.mapstring(solved)^
                          "\nnarrowed goals"^
                          (if DelNarrowed then " have been deleted"
                                          else ":"^Aux.mapstring(narrowed))^
                          "\n"^makestring(lg:int)^" step"^str1^" been executed\n"
            in Aux.Output(file,str^"\n") end

fun     factor(gs,g_no,at1_no,at2_no)
          = let val file = open_append"EXPANSION"
                val str = "\natom "^makestring(at1_no:int)^" factored with atom "^
                          makestring(at2_no:int)^" in conclusion goal "^
                          makestring(g_no:int)^"\nconclusion:"^Unparse.goals(gs)^"\n"
            in Aux.Output(file,str) end

fun     unify(gs,g_no,eq_pos)
          = let val file = open_append"EXPANSION"
                val str = "\nequation "^makestring(eq_pos:int)^" in conclusion goal "^
                          makestring(g_no:int)^" unified\nconclusion:"^Unparse.goals(gs)
            in Aux.Output(file,str^"\n") end

fun     replace(gs,g_no,eq_pos,t_poss,dir)
          = let val file = open_append"EXPANSION"
                val str1 = if dir = backward then "conclusion" else "premise"
                val str2 = case t_poss of [_] => "" | _ => "s"
                val str3 = "\nterm"^str2^" at position"^str2^Aux.mapstrings(t_poss)^
                           "replaced with equation "^makestring(eq_pos:int)^" in "^str1^
                           " goal "^makestring(g_no:int)^"\n"^str1^":"^Unparse.goals(gs)^"\n"
            in Aux.Output(file,str3) end

fun     delete_goal(gs,g_nos,dir)
          = let val file = open_append"EXPANSION"
                val str1 = if dir = backward then "conclusion" else "premise"
                val str2 = case g_nos of [_] => "" | _ => "s"
                val str3 = "\ngoal"^str2^Aux.mapstring(g_nos)^
                           "deleted\n"^str1^":"^Unparse.goals(gs)^"\n"
            in Aux.Output(file,str3) end

fun     delete_atoms(gs,g_no,at_nos,dir)
          = let val file = open_append"EXPANSION"
                val str1 = if dir = backward then "conclusion" else "premise"
                val str2 = case at_nos of [_] => "" | _ => "s"
                val str3 = "\natom"^str2^Aux.mapstring(at_nos)^"in "^str1^" goal "^
```

```
                        makestring(g_no:int)^" deleted\n"^str1^":"^Unparse.goals(gs)^"\n"
                 in Aux.Output(file,str3) end

        fun     resolve(g_nos,at_nos,cl_no,cl_mode,dir,display,gs)
                  = let val file = open_append"EXPANSION"
                        val str1 = case at_nos of nil => "\n" |
                                                  [n] => "\natom "^makestring(n:int)^" in " |
                                                  _ => "\natoms"^Aux.mapstring(at_nos)^"in "
                        val str2 = if dir = forward then "premise" else "conclusion"
                        val str3 = case g_nos of [_] => "" | _ => "s"
                        val str4 = case cl_mode of axm => "axiom " | thm => "theorem " |
                                                   conj => "conjecture "
                        val str5 = if dir = test then "\natom"^Aux.mapstring(at_nos)^"replaced with axiom "
                                   else str1^str2^" goal"^str3^Aux.mapstring(g_nos)^"replaced with "^str4
                        val str6 = if display = short then "" else "\n"^str2^":"^gs^"\n"
                    in Aux.Output(file,str5^cl_no^str6) end

        fun     paramodulate(g_nos,t_poss,cl_no,cl_mode,dir,display,gs)
                  = let val file = open_append"EXPANSION"
                        val str0 = case g_nos of [_] => "" | _ => "s"
                        val str1 = case t_poss of [_] => "" | _ => "s"
                        val str2 = if dir = forward then "premise" else "conclusion"
                        val str3 = case cl_mode of axm => "axiom " | thm => "theorem " |
                                                   conj => "conjecture "
                        val str4 = if dir = test then "\nterm at position"^Aux.mapstrings(t_poss)^
                                                      "replaced with axiom "
                                   else "\nterm"^str1^" at position"^str1^Aux.mapstrings(t_poss)^
                                        "in "^str2^" goal"^str0^Aux.mapstring(g_nos)^"replaced with "^str3
                        val str5 = if display = short then "" else "\n"^str2^":"^gs^"\n"
                    in Aux.Output(file,str4^cl_no^str5) end

        fun     proved(conjs,proved)
                  = let val file = open_append"EXPANSION"
                        val str1 = case conjs of [_] => "" | _ => "s"
                        val str2 = case conjs of [_] => "has" | _ => "have"
                        val str3 = case proved of true => ""| false => " not yet"
                        val str4 = "\nconjecture"^str1^Aux.mapstring(conjs)^str2^str3^" been proved\n"
                    in Aux.Output(file,str4) end

        fun     change_spec(cls,n,str)
                  = let val str1 = case cls of [_] => " " | _ => "s "
                        val str2 = case cls of [_] => "is" | _ => "are"
                        val str3 = "\nthe "^str^str1^Unparse.list_clauses(cls,n)^
                                   "\n"^str2^" added to the specification\n"
                    in output(std_out,str3) end

end (*Display*)
```

8.13 Start

```
structure Start = struct  open  InferAux

        fun     read(state,s)
                  = let val (spec,syntax_error) = Parse.inspec(s,(!DIRS))
                        val (spec_name,_,_,_,_) = spec
                        val (_,inVars,ini,proof,fin,conj_nos) = state
                        val _ = Parse.read(spec_name,syntax_error)
                    in (spec,inVars,ini,proof,fin,conj_nos) end

        fun     proof(state,conj_nos,guard_no)
                  = let val (spec,_,_,_,_,_) = state
```

```
                 val (_,_,_,_,conjs) = spec
                 val (iniAndInVars,fin) = new_conj(conjs,conj_nos,guard_no)
                 val ini = map(#1)(iniAndInVars)
                 val ini = simplify_disjunct(initialize(ini))
                 val fin = simplify(hd(initialize[fin]))
                 val inVars = Aux.makeset(Aux.mapconc(#2)(iniAndInVars))
                 val proof = ([ini],[[fin]])
                 val _ = Display.start(ini,[fin])
             in (spec,inVars,ini,proof,fin,conj_nos) end

    and     new_conj(cls,ms,n) = new_conjLoop(cls,ms,n,1,nil)

    and     new_conjLoop(cl::cls,m::ms,n,k,acc)
              = if m = k then new_conjLoop(cls,ms,n,k+1,acc@[cl])
                         else new_conjLoop(cls,m::ms,n,k+1,acc) |
            new_conjLoop(_,nil,n,_,acc) = let val (_,prem) = hd(acc)
                                          in (new_concl(acc,n),Aux.first(prem,n)) end

    and     new_concl(([concl],prem)::rest,n)
              = let val allVars = Aux.makeset(Aux.mapconc(Vars)(concl@prem))
                    val gens = nthtail(prem,n)
                    val outVars = Aux.makeset(Aux.mapconc(Vars)(map(rhs)(gens)))
                    val g = intoExVars(concl@gens,outVars)
                    val inVars = Aux.subtract(allVars,outVars)
                in (g,inVars)::new_concl(rest,n) end |
            new_concl(nil,_) = nil

    fun     proof_dis(state,conj_no)
              = let val (spec,_,_,_,_,_) = state
                    val (_,_,_,_,conjs) = spec
                    val (ini,fin) = Search.clause(conjs,conj_no)
                    val ini = simplify_disjunct(initialize(ini))
                    val inVars = Aux.makeset(Aux.mapconc(Aux.mapconc(iVars))(ini))
                    val fin = simplify(hd(initialize([fin],inVars)))
                    val proof = ([ini],[[fin]])
                    val _ = Display.start(ini,[fin])
                    val file = open_out"SOLUTIONS"
                    val str = "Solutions of\n"^Unparse.goal(hd(ini),7)^"\n\n"
                    val _ = (output(file,str); close_out(file))
                in (spec,inVars,ini,proof,fin,[conj_no]) end

end (*Start*)
```

8.14 Unify

```
structure Unify = struct  open  InferAux

    fun     atoms(state,g_no,at1_no,at2_no)
              = let val gs = current_goal_set(state,backward)
                    val (rest1,redex,rest2) = Search.goal(gs,g_no)
                    val (_,at1,_) = Search.atom(redex,at1_no)
                    val (rest3,at2,rest4) = Search.atom(redex,at2_no)
                    val sub = unifies(at1,at2)
                    val _ = if suspended(sub) then raise no_unifier else ()
                    val (_,inVars,_,_,_,_) = state
                    val reduct = simplify(SUBST(rest3@rest4,sub,inVars))
                    val gs = check_disjunct(rest1@reduct::rest2)
                    val _ = Display.factor(gs,g_no,at1_no,at2_no)
                in extend_proof(state,gs,backward) end

    fun     terms(state,g_no,eq_pos)
```

```
              = let val gs = current_goal_set(state,backward)
                    val (rest1,redex,rest2) = Search.goal(gs,g_no)
                    val (rest3,Eq(l,r),rest4) = Search.atom(redex,eq_pos)
                    val sub = unifies(l,r)
                    val _ = if suspended(sub) then raise no_unifier else ()
                    val (_,inVars,_,_,_,_) = state
                    val reduct = simplify(SUBST(rest3@rest4,sub,inVars))
                    val gs = check_disjunct(rest1@reduct::rest2)
                    val _ = Display.unify(gs,g_no,eq_pos)
                in extend_proof(state,gs,backward) end

end (*Unify*)
```

8.15 Replace

```
structure Replace = struct  open  InferAux

        exception different_terms

        fun     terms(state,g_no,eq_pos,t_poss,dir)
              = let val gs = current_goal_set(state,dir)
                    val (rest1,redex,rest2) = Search.goal(gs,g_no)
                    val (_,eq,_) = Search.atom(redex,eq_pos)
                    val reduct = sequence(t_poss,redex,eq)
                    val reduct = simplify(reduct)
                    val gs = check_disjunct(rest1@reduct::rest2)
                    val _ = Display.replace(gs,g_no,eq_pos,t_poss,dir)
                in extend_proof(state,gs,dir) end

        and     sequence(t_pos::t_poss,redex,eq) = let val reduct = single(t_pos,redex,eq)
                                                   in sequence(t_poss,reduct,eq) end |
                sequence(nil,redex,_) = redex

        and     single(at_pos::t_pos,redex,Eq(l,r))
              = let val (rest1,atom,rest2) = Search.atom(redex,at_pos)
                    val (context,t) = Search.term(atom,t_pos,YY)
                    val reduct = if eq_term(l,t) then substitute(r/yy)(context)
                                                 else if eq_term(r,t)
                                 then substitute(l/yy)(context) else raise different_terms
                in rest1@reduct::rest2 end

end (*Replace*)
```

8.16 Delete

```
structure Delete = struct  open  InferAux

        fun     goals(state,g_nos,dir,display)
              = let val gs = current_goal_set(state,dir)
                    val (_,rest) = Search.sublist(gs,g_nos)
                    val gs = check_disjunct(rest)
                    val _ = if display = short then () else Display.delete_goals(gs,g_nos,dir)
                in extend_proof(state,gs,dir) end

        fun     atoms(state,g_no,at_nos,dir)
              = let val gs = current_goal_set(state,dir)
                    val (rest1,redex,rest2) = Search.goal(gs,g_no)
                    val (_,rest) = Search.sublist(redex,at_nos)
                    val gs = check_disjunct(rest1@rest::rest2)
```

```
                  val _ = Display.delete_atoms(gs,g_no,at_nos)
              in extend_proof(state,gs,dir) end

end (*Delete*)
```

8.17 Derive

```
local  open  Substitute  in

functor Derive(val Solve : GoalE -> Term list list) = struct  open  Aux InferAux

        fun     rconjunct(conc_nos)(state,g_no,at_nos,cl_no,cl_mode,redex_lost,dir,display1,display2)
                  = let val (spec,inVars,_,_,_,_) = state
                        val (_,_,axms,thms,conjs) = spec
                        val clauses = case cl_mode of axm => axms | thm => thms | conj => conjs
                        val ([concl],prem) = Search.clause(clauses,#2(cl_no))
                        val (concl,_) = Search.sublist(concl,conc_nos)
                        val gs = current_goal_set(state,dir)
                        val (rest1,redex,rest2) = Search.goal(gs,g_no)
                        val (atoms,rest) = Search.sublist(redex,at_nos)
                        val sub = unifyAtoms(atoms,concl)
                        val descent = DESCENT(sub,inVars,"true")
                        val Prem = if cl_mode = conj then descent::prem else prem
                        val _ = if Prem <> nil andalso Solve(Prem,sub,spec,inVars) = [[FALSE]]
                                then raise no_unifier else ()
                        val reduct = if suspendedL(sub)
                                     then if cl_mode = axm
                                          then let val instance = SUBST(redex,sub,inVars)
                                               in if subsumes(redex,instance)
                                                  then raise no_unifier else instance end
                                          else raise no_unifier
                                     else let val reduct = SUBST(prem@rest,sub,inVars)
                                              val _ = if cl_mode = axm then () else CHECK(reduct,sub)
                                                                     (* check variable condition *)
                                          in if cl_mode = conj then descent::reduct else reduct end
                        val reduct = rename_goal(reduct,inVars)
                        val inVars = makeset(inVars@mapconc(iVars)(reduct))
                        val reduct
                               = if cl_mode <> thm then simplify(reduct) else Makeset(eq_term)(reduct)
                        val gs = if redex_lost then rest1@reduct::rest2 else rest1@redex::reduct::rest2
                        val state = extend_proof(state,gs,dir)
                        val state = new_inVars(state,inVars)
                        val (str,_) = cl_no   (* str = specname *)
                        val _ = if display2 = short then ()
                                else Display.resolve([g_no],at_nos,str,cl_mode,dir,display1,
                                                     Unparse.goals(gs))
                    in (state, reduct = [FALSE]) end

        fun     Rforward(state,g_no,at_nos,cl_no,cl_mode,redex_lost)
                  = let val (spec,inVars,_,_,_,_) = state
                        val (_,_,_,thms,conjs) = spec
                        val clauses = case cl_mode of thm => thms | conj => conjs
                        val (concl,prem) = Search.clause(clauses,cl_no)
                        val gs = current_goal_set(state,forward)
                        val (rest1,redex,rest2) = Search.goal(gs,g_no)
                        val (atoms,rest) = Search.sublist(redex,at_nos)
                        val sub = matchAtoms(atoms,prem)
                        fun f1(g) = (map o substitute)(#1(sub))(g@rest)
                        fun f2(g) = rename_goal(g,inVars)
                        val f3 = if cl_mode <> thm then simplify else Makeset(eq_term) o check_conjunct
                        val reduct = map(f3 o f2 o f1)(concl)
```

```
                val reduct
                      = if cl_mode <> conj then reduct else addDescent(sub,inVars,redex)::reduct
                val reduct = if redex_lost then reduct else map(fn(g)=>redex@g)(reduct)
                val gs = check_disjunct(rest1@reduct@rest2)
                val state = extend_proof(state,gs,forward)
                val str = makestring(cl_no:int)
                val _ = Display.resolve([g_no],at_nos,str,cl_mode,forward,long,Unparse.goals(gs))
            in state end

and     addDescent(sub,inVars,redex) = let val descent = DESCENT(sub,inVars,"false")
                                       in (simplify o rename_goal)(descent::redex,inVars) end

fun     Rdisjunct(state,g_nos,at_nos,cl_no,conc_nos,cl_mode,redex_lost)
          = let val (spec,inVars,_,_,_,_) = state
                val (_,_,_,thms,conjs) = spec
                val clauses = case cl_mode of thm => thms | conj => conjs
                val (concl,prem) = Search.clause(clauses,cl_no)
                val (concls,_) = Search.disjunct(concl,conc_nos)
                val gs = current_goal_set(state,backward)
                val (redex,rest1) = Search.sublist(gs,g_nos)
                val redex = rename_disjunct(redex,nil)
                val (atoms,rest) = Search.disjunct(redex,at_nos)
                val sub = unifyAtoms(atoms,concls)
                val _ = if suspendedL(sub) then raise no_unifier else ()
                val descent = DESCENT(sub,inVars,"true")
                val Prem = if cl_mode = conj then descent::prem else prem
                val _ = if Prem <> nil andalso Solve(Prem,sub,spec,inVars) = [[FALSE]]
                        then raise no_unifier else ()
                val reduct = SUBST(prem@rest,sub,inVars)
                val _ = CHECK(reduct,sub)
                val reduct = if cl_mode = conj then descent::reduct else reduct
                val reduct = rename_goal(reduct,inVars)
                val inVars = makeset(inVars@mapconc(iVars)(reduct))
                val reduct = if cl_mode <> thm then simplify(reduct) else Makeset(eq_term)(reduct)
                val gs = if redex_lost then reduct::rest1 else redex@reduct::rest1
                val gs = check_disjunct(gs)
                val state = extend_proof(state,gs,backward)
                val state = new_inVars(state,inVars)
                val str = makestring(cl_no:int)
                val _ = Display.resolve(g_nos,at_nos,str,cl_mode,backward,long,Unparse.goals(gs))
            in state end

fun     patom(conc_no)(state,g_no,t_pos,cl_no,cl_mode,redex_lost,dir,display1,display2)
          = let val (spec,inVars,_,_,_,_) = state
                val (_,_,axms,thms,conjs) = spec
                val clauses = case cl_mode of axm => axms | thm => thms | conj => conjs
                val ([concl],prem) = Search.clause(clauses,#2(cl_no))
                val (_,concl,_) = Search.atom(concl,conc_no)
                val gs = current_goal_set(state,dir)
                val (rest1,redex,rest2) = Search.goal(gs,g_no)
                val (reduct,inVars) = REWRITE(spec,inVars,[redex],[t_pos],[concl],prem,cl_mode,Solve)
                val gs = if redex_lost then rest1@reduct::rest2 else rest1@redex::reduct::rest2
                val state = extend_proof(state,gs,dir)
                val state = new_inVars(state,inVars)
                val (str,_) = cl_no   (* str = specname *)
                val _ = if display2 = short then ()
                        else Display.paramodulate([g_no],[t_pos],str,cl_mode,dir,display1,
                                                  Unparse.goals(gs))
            in (state, reduct = [FALSE]) end

fun     Pforward(state,g_no,t_poss,at_nos,cl_no,conc_nos,cl_mode,redex_lost)
          = let val (spec,inVars,_,_,_,_) = state
```

```
                  val (_,_,_,thms,conjs) = spec
                  val clauses = case cl_mode of thm => thms | conj => conjs
                  val (concl,prem) = Search.clause(clauses,cl_no)
                  val (concl,_) = Search.disjunct(concl,conc_nos)
                  val Eq(left,_) = hd(concl)
                  val gs = current_goal_set(state,forward)
                  val (rest1,redex,rest2) = Search.goal(gs,g_no)
                  val at_nos' = map(hd)(t_poss)
                  val t_poss' = map(tl)(t_poss)
                  val (atoms,rest) = Search.sublist(redex,at_nos@at_nos')
                  val n = length(at_nos)
                  val atoms' = nthtail(atoms,n)
                  val atoms = first(atoms,n)
                  fun f(at,t_pos) = Search.term(at,t_pos,YY)
                  val (contexts,terms) = map22(f)(atoms',t_poss')
                  val term = hd(terms)
                  fun g(t) = eq_term(t,term) orelse raise different_terms
                  val _ = map(g)(tl(terms))
                  val sub = matchAtoms(term::atoms,left::prem)
                  fun f1(eq) = let val Eq(_,right) = eq
                                   val subRight = substitute(right/yy)
                               in eq::map(subRight)(contexts) end
                  fun f2(g) = (map o substitute)(#1(sub))(g@rest)
                  fun f3(g) = rename_goal(g,inVars)
                  val f4 = if cl_mode <> thm then simplify else Makeset(eq_term)
                  val reduct = map(f4 o f3 o f2 o f1)(concl)
                  val reduct
                        = if cl_mode <> conj then reduct else addDescent(sub,inVars,redex)::reduct
                  val reduct = if redex_lost then reduct else map(fn(g)=>redex@g)(reduct)
                  val gs = check_disjunct(rest1@reduct@rest2)
                  val state = extend_proof(state,gs,forward)
                  val str = makestring(cl_no:int)
                  val _ = Display.paramodulate([g_no],t_poss,str,cl_mode,forward,long,
                                               Unparse.goals(gs))
              in state end

fun     Pdisjunct(state,g_nos,t_poss,cl_no,conc_nos,cl_mode,redex_lost)
          = let val (spec,inVars,_,_,_,_) = state
                  val (_,_,_,thms,conjs) = spec
                  val clauses = case cl_mode of thm => thms | conj => conjs
                  val (concl,prem) = Search.clause(clauses,cl_no)
                  val (concl,_) = Search.disjunct(concl,conc_nos)
                  val gs = current_goal_set(state,backward)
                  val (redex,rest) = Search.sublist(gs,g_nos)
                  val (reduct,inVars) = REWRITE(spec,inVars,redex,t_poss,concl,prem,cl_mode,Solve)
                  val gs = if redex_lost then reduct::rest else redex@reduct::rest
                  val gs = check_disjunct(gs)
                  val state = extend_proof(state,gs,backward)
                  val state = new_inVars(state,inVars)
                  val str = makestring(cl_no:int)
                  val _ = Display.paramodulate(g_nos,t_poss,str,cl_mode,backward,long,
                                               Unparse.goals(gs))
              in state end

fun     resolve(pos) = LOOP(Delete.goals,rconjunct[1])([pos],false)

fun     paramodulate(pos) = LOOP(Delete.goals,patom(1))(pos,true)

fun     Rconjunct(pos)(x) = let val (state,_) = rconjunct(pos)(x)
                                val gs = current_goal_set(state,backward)
                                val state = undo(state,1,backward)
                                val gs = check_disjunct(gs)
```

```
                                  val state = extend_proof(state,gs,backward)
                              in state end

        fun     Patom(pos)(x) = let val (state,_) = patom(pos)(x)
                                    val gs = current_goal_set(state,backward)
                                    val state = undo(state,1,backward)
                                    val gs = check_disjunct(gs)
                                    val state = extend_proof(state,gs,backward)
                                in state end

end (*Derive*) end
```

8.18 Draw

```
structure Draw = struct  open  AbstractSyntax

        fun     iterate _ (0) = ""  |  iterate(str)(n) = str^iterate(str)(n-1)

        val     Blanks = iterate" "

        val     space = 3

        fun     rest(str) = space-size(str)

        fun     StringL(str)(ch) = str^iterate(ch)(rest(str))

        fun     StringR(ch)(str) = iterate(ch)(rest(str))^str

        fun     StringM(str)(ch) = let val back = rest(str) div 2
                                       val even = rest(str) mod 2 = 0
                                       val front = if even then back else back+1
                                   in iterate(ch)(front)^str^iterate(ch)(back) end

        fun     Ints2Strings(n) = let fun f(i:int,s) = (StringM(makestring(i))" ")^s
                                  in fold(f)(Aux.up(1,n))"" end

        fun     newline(i:int) = "\n"^StringR" "(makestring(i))^" | "

        fun     columns(l) = "\n"^Blanks(space+2)^""^iterate(iterate""(space))(l)^
                             "\n"^Blanks(space+3)^Ints2Strings(l)

        fun     forward(f)(i)(x) = if i > x then () else (f(i); forward(f)(i+1)(x))

        fun     down(f)(i,j)(x,y) = if y < j then ()
                                          else (forward(fn(i)=>f(i,y))(i)(x); down(f)(i,j)(x,y-1))

        fun     grid(f)(l,h)(file) = let fun g(1,j) = output(file,newline(j)^f(1,j))
                                         |   g(i,j) = output(file,f(i,j))
                                     in down(g)(1,1)(l,h); output(file,columns(l)^"\n\n") end

        exception no_graph

        fun     Graphs[eq](f) = Graph(eq)(f)
        |       Graphs _ _ = raise no_graph

        and     Graph(Eq(Var _,Op(x)))(f) = List(Op(x))(f)
        |       Graph(Eq(Op(x),Var _))(f) = List(Op(x))(f)
        |       Graph _ _ = raise no_graph

        and     List(Op("::",[t,tL],_))(f,l,h) = let val points = makePoints(t)
                                                     val f = add(points)(f)
```

```
                                             val (l,h) = extendCart(points)(l,h)
                                          in List(tL)(f,l,h) end
     |       List(Op("nil",_,_))(f)  = f
     |       List _ _ = raise no_graph

     and     makePoints(Op("",[x,y],_)) = [(Num(x),Num(y))]
     |       makePoints(Op("rect",[x,y,l,h],_))
               = let val [x,y,l,h] = map(Num)[x,y,l,h]
                     val (x1,y1) = (x+x,y+y)
                     val (x2,y2) = (x1+l+l,y1+h+h)
                 in horizontal(y1,x1,x2)@horizontal(y2,x1,x2)@vertical(x1,y1,y2)@vertical(x2,y1,y2) end
     |       makePoints _ = raise no_graph

     and     Num(Op("s",[t],_)) = Num(t)+1
     |       Num(Op("0",_,_)) = 0
     |       Num _ = raise no_graph

     and     horizontal(y,x1,x2) = if x1 > x2 then nil else (x1,y)::horizontal(y,x1+1,x2)

     and     vertical(x,y1,y2) = if y1 > y2 then nil else (x,y1)::vertical(x,y1+1,y2)

     and     add(p::points)(f)(q) = if p = q then " * " else add(points)(f)(q)
     |       add _ (f)(q) = f(q)

     and     extendCart(points)(l,h)
               = (Aux.Max(l::map(#1)(points)),Aux.Max(h::map(#2)(points)))

     fun     solution(g) = let val f = (fn _ => "   ",1,1)
                                val (f,isGraph) = (Graphs(g)(f),true)
                                                  handle no_graph => (f,false)
                           in if isGraph
                              then let val (f,l,h) = f
                                           val file = open_append"SOLUTIONS"
                                       in grid(f)(l,h)(file); close_out(file) end
                              else () end

end (*Draw*)
```

8.19 Solve

```
local  open  AbstractSyntax
       type  stateE = State * int * bool * direction * display
in
functor Solve(val Resolve : int -> StateE -> State * int * int
              val Paramodulate : int list -> StateE -> State * int * int)

      = struct  open  InferAux

        datatype redex = found of int | resolve of int |
                         paramodulate of redex * int list * int | apply of redex * int list

        val     STATE = ref(emptyState)
        val     PREDS = ref(nil:Term list list)
        val     DONTSUBSUME = ref(true)
        val     ONESOLUTION = ref(true)
        val     INNERMOST = ref(true)
        val     BREADTHFIRST = ref(true)
        val     DEL_NARROWED = ref(true)
        val     DISPLAY = ref(short)

        fun     goal_no(state,g_no,dir,depth,Dontsubsume,Onesol,Innermost,Breadthfirst,
```

```
              DelNarrowed,display)
        = let val (spec,inVars,_,_,_,_) = state
              val gs = current_goal_set(state,dir)
              val (gs1,g,gs2) = Search.goal(gs,g_no)
              val _ = Display.start_solve(g_no,dir,depth,Dontsubsume,Onesol,
                                          Innermost,Breadthfirst)
              val (gs,sol,nar,lg) = goal(g,spec,inVars,depth,Dontsubsume,Onesol,Innermost,
                                         Breadthfirst,DelNarrowed,display)
              fun f(n) = n+length(gs1)
              val (sol,nar) = (map(f)(sol),map(f)(nar))
              val gs = check_disjunct(gs1@gs@gs2)
          in show(gs,dir); Display.finish_solve(sol,nar,DelNarrowed,lg);
             extend_proof(state,gs,dir) end

and     goal(g,spec,inVars,depth,Dontsubsume,OneSol,Innermost,Breadthfirst,DelNarrowed,Display)
        = (STATE := (spec,inVars,nil,(nil,nil),nil,nil); PREDS := nil;
           DONTSUBSUME := Dontsubsume; ONESOLUTION := OneSol;
           INNERMOST := Innermost; BREADTHFIRST := Breadthfirst;
           DEL_NARROWED := DelNarrowed; DISPLAY := Display;
           Loop([g],depth,1,nil,0))

and     Loop(gs,0,_,_,lg) = (gs,nil,nil,lg) |
        Loop(nil,_,_,_,lg) = (nil,nil,nil,lg) |
        Loop([[FALSE]],_,_,_,lg) = ([[FALSE]],nil,nil,lg) |
        Loop(gs,depth,n,sols,lg)
        = case Search.solution(gs) of
          Aux.C(gs1,g,gs2)    (* g is solved *)
            => let val gs = gs1@gs2
                   val oldsol = isoMember(g,sols)
                   val _ = if oldsol then () else Draw.solution(g)
                   val (gs,sol,nar) = if !ONESOLUTION then (gs,nil,nil,lg)
                                      else if oldsol then Loop(gs,depth-1,n,sols,lg+1)
                                           else Loop(gs,depth-1,n+1,g::sols,lg+1)
               in if oldsol then (gs,sol,nar,lg) else (g::gs,n::sol,nar,lg) end |
          _ => let val (redex,rest) = if !BREADTHFIRST then (gs,nil) else ([hd(gs)],tl(gs))
                   val _ = PREDS := !PREDS@redex
                   val (narrowed,reduct) = doStep(redex)
               in if reduct = [[]] then ([[]],nil,nil,lg)
                  else let val gs = implicates(reduct@rest)
                           val (subsumed,gs) = if !DONTSUBSUME then (nil,gs)
                                               else detect_loops(gs,!PREDS)
                           val narrowed = if !DEL_NARROWED then nil
                                          else narrowed@subsumed
                           val k = length(narrowed)
                           val (gs,sol,nar,lg) = Loop(gs,depth-1,n+k,sols,lg+1)
                       in (narrowed@gs,sol,Aux.up(n,k)@nar,lg) end end

and     doStep(nil) = (nil,nil) |
        doStep(g::gs)
        = (STATE := extend_proof(!STATE,[g],test);
           if !DISPLAY = long then show([g],next) else ();
           let val found(k) = redex_in_goal(g,1)
               val redex::rest = current_goal_set(!STATE,test)
           in if redex = [] then (nil,[[]])
              else let val (narrowed,reduct) = doStep(gs)
                   in if k > 0 then (narrowed,rest@reduct)
                            else (redex::narrowed,reduct) end end)

and     detect_loops(gs,preds)
        = let val preds = map(Aux.filter(Search.solved_eq))(preds)
              fun f(g) = Aux.filter(Search.solved_eq)(g)
              fun subiso((s1,r1),(s2,r2)) = subsumes(s1,s2) andalso isomorphic(r1,r2)
```

```
              fun p(g) = Aux.Exists(subiso)(f(g),preds)
          in Aux.filter(p)(gs) end

and     redex_in_goal(nil,_) = found(0) |
        redex_in_goal(at::g,n) = if Search.solved_eq(at) orelse readOnly(at)
                                 then redex_in_goal(g,n+1) else redex_in_atom(at,n)

and     redex_in_atom(Eq(l,r),n) = redex_in_terms(found(0),[l,r],[n],1) |
        redex_in_atom(Op(_,tlist,_),n)
          = if !INNERMOST then redex_in_terms(resolve(n),tlist,[n],1)
            else let val found(k) = make_step(resolve(n))
                 in if k > 0 then found(k) else redex_in_terms(found(0),tlist,[n],1) end

and     redex_in_terms(redex,nil,_,_) = make_step(redex) |
        redex_in_terms(redex,t::tlist,pos,n)
          = let val found(k) = redex_in_term(redex,t,pos@[n])
            in if k > 0 then found(k)
               else case redex of apply(red,_) => redex_in_terms(red,tlist,pos,n+1) |
                                  _ => redex_in_terms(redex,tlist,pos,n+1) end

and     redex_in_term(_,Var _,_) = found(0) |
        redex_in_term(_,Op("fn",_,_),_) = found(0) |
                                (*  no substitution into bounded variables  *)
        redex_in_term(redex,Op("APPLY",tlist,_),pos)
          = (case redex of apply _ => redex_in_terms(redex,tlist,pos,1) |
                           _ => redex_in_terms(apply(redex,pos),tlist,pos,1)) |
        redex_in_term(_,Op(_,tlist,cop),pos) = redex_in_terms(found(0),tlist,pos,1) |
        redex_in_term(_,Op(_,tlist,coi),pos) = redex_in_terms(found(0),tlist,pos,1) |
        redex_in_term(redex,Op(_,tlist,_),pos)
          = let val (redex,n)   (*  n = number of preceding APPLYs  *)
                    = case redex of apply(r,top) => (r,length(pos)-length(top)) |
                                    _ => (redex,0)
                val redex = paramodulate(redex,pos,n)
            in if !INNERMOST then redex_in_terms(redex,tlist,pos,1)
               else let val found(k) = make_step(redex)
                    in if k > 0 then found(k) else redex_in_terms(found(0),tlist,pos,1) end
            end

and     make_step(apply(redex,_)) = make_step(redex) |
        make_step(resolve(n))
          = let val (state,r,fr) = Resolve(n)(!STATE,1,false,test,!DISPLAY)
            in STATE := state; found(r-fr) end |
        make_step(paramodulate(redex,pos,n))
          = let val (state,r,fr) = Paramodulate(pos)(!STATE,1,false,test,!DISPLAY)
            in STATE := state;
               if r > fr then found(r-fr)
               else if n > 0 then let val (pos,_) = Aux.front(pos)
                                  in make_step(paramodulate(redex,pos,n-1)) end
                    else make_step(redex) end |
        make_step(redex) = redex

end (*Solve*) end
```

8.20 More

```
structure More = struct  open  InferAux

fun     backtrack(state,n,dir)
          = let val state = undo(state,n,dir)
                val gs = current_goal_set(state,dir)
                val _ = show(gs,dir)
```

```
              in state end

    fun     evaluate(state,g_nos,dir)
              = let val gs = current_goal_set(state,dir)
                    val (gs,rest) = Search.sublist(gs,g_nos)
                    val gs = map(simplify)(gs)@rest
                    val gs = check_disjunct(gs)
                    val _ = show(gs,dir)
                in extend_proof(state,gs,dir) end

    fun     subsume(state,dir)
              = let val gs = current_goal_set(state,dir)
                    val gs = implicates(gs)
                    val _ = show(gs,dir)
                in extend_proof(state,gs,dir) end

    fun     proved(state)
              = let val (_,_,_,(gs::front,gs'::rear),_,conj_nos) = state
                    val proved = implies(gs',gs)
                    val _ = Display.proved(conj_nos,proved)
                in state end

    fun     show_spec(state) = let val (spec,_,_,_,_,_) = state
                               in output(std_out,Unparse.spec(spec)^"\n") end

    fun     write_spec(state)
              = let val (spec,_,_,_,_,_) = state
                    val (spec_name,_,_,_,_) = spec
                    val file = (!DIRS)^"/"^spec_name^"flat"
                in Unparse.outspec(spec,file) end

    fun     write_proof(state)
              = let val (spec,_,_,_,_,conjs) = state
                    val (spec_name,_,_,_,_) = spec
                    val file = open_in"EXPANSION"
                    val str = Aux.instring(file,"")
                    val _ = close_in(file)
                    val file_name = (!DIRP)^"/"^spec_name^Aux.mapstring(conjs)
                    val file = open_out(file_name)
                    val _ = output(file,str)
                in close_out(file) end

    fun     write_graph(state)
              = let val (spec,_,_,_,_,conjs) = state
                    val (spec_name,_,_,_,_) = spec
                    val file = open_in"SOLUTIONS"
                    val str = Aux.instring(file,"")
                    val _ = close_in(file)
                    val file_name = (!DIRG)^"/"^spec_name^Aux.mapstring(conjs)
                    val file = open_out(file_name)
                    val _ = output(file,str)
                in close_out(file) end

end (*More*)
```

8.21 Finish

```
structure Finish = struct  open  InferAux

    fun     getVars(gs) = Aux.makeset(Aux.mapconc(Aux.mapconc(Vars))(gs))
```

```
fun     proof(state,gens)
          = let val (spec,_,_,proof,_,conj_nos) = state
                val (spec_name,sign,axms,thms,conjs) = spec
                val (gs::front,gs'::rear) = proof
                val conj_concl = finalize(gs)
                val conj_prem = finalize(gs')
                val uniVars = getVars(conj_concl@conj_prem)
                val conj_concl = intoUniVars(conj_concl,uniVars)
                val [guard] = intoUniVars(conj_prem,uniVars)
                val new_conjs = shift_gens(conj_concl,gens)(guard)
                val (cs,fs,ps,is,vs) = sign
                val sign = (cs,fs,ps,is,Aux.makeset(vs@uniVars))
                val spec = (spec_name,sign,axms,thms,conjs@new_conjs)
                val _ = Display.change_spec(new_conjs,length(conjs)+1,"conjecture")
            in (spec,nil,nil,(nil,nil),nil,nil) end

and     shift_gens(g::gs,ns::nss)(prem) = let val (gens,concl) = Search.atoms(g,ns)
                                          in ([concl],prem@gens)::shift_gens(gs,nss)(prem) end |
        shift_gens(nil,_) _ = nil

fun     proof_dis(state)
          = let val (spec,_,_,proof,_,conj_nos) = state
                val (spec_name,sign,axms,thms,conjs) = spec
                val (gs::front,gs'::rear) = proof
                val conj_concl = finalize(gs)
                val conj_prem = finalize(gs')
                val allVars = getVars(conj_concl@conj_prem)
                val uniVars = getVars(conj_prem)
                val conj_concl = intoUniVars(conj_concl,uniVars)
                val conj_prem = intoUniVars(conj_prem,uniVars)
                val new_conjs = map(fn(g)=>(conj_concl,g))(conj_prem)
                val (cs,fs,ps,is,vs) = sign
                val sign = (cs,fs,ps,is,Aux.makeset(vs@allVars))
                val spec = (spec_name,sign,axms,thms,conjs@new_conjs)
                val _ = Display.change_spec(new_conjs,length(conjs)+1,"conjecture")
            in (spec,nil,nil,(nil,nil),nil,nil) end

fun     conjects_to_thms(state)
          = let val (spec,_,_,_,_,conj_nos) = state
                val (spec_name,sign,axms,thms,conjs) = spec
                val (new_thms,rest) = Search.sublist(conjs,conj_nos)
                val spec = (spec_name,sign,axms,thms@new_thms,rest)
                val _ = Display.change_spec(new_thms,length(thms)+1,"theorem")
            in (spec,nil,nil,(nil,nil),nil,nil) end

end (*Finish*)
```

8.22 Main program

```
open    AbstractSyntax

        exception not_applicable_to_premise

val     STATE = ref(emptyState)
val     DIRECT = ref(backward)

fun     parse s = STATE := Start.read(!STATE,s)

fun     storeProof() = More.write_proof(!STATE)

fun     storeGraph() = More.write_graph(!STATE)
```

```
fun     start conj_no = (STATE := Start.proof_dis(!STATE,conj_no); DIRECT := backward)

fun     startM conj_nos n = (STATE := Start.proof(!STATE,conj_nos,n); DIRECT := backward)

fun     proved() = STATE := More.proved(!STATE)

structure InferAll = Derive(fun Solve _ = [nil])

structure SolveAll = Solve(val Resolve = InferAll.resolve
                           val Paramodulate = InferAll.paramodulate)

structure InferSome
  = Derive(fun Solve(g,(sub,_),spec,inVars)
                = let open InferAux
                      val g = (map o substitute)(sub)(g)
                      val g = (simplify o rename_goal)(g,inVars)
                      val (gs,_,_,_)
                          = SolveAll.goal(g,spec,inVars,5,true,true,true,true,true,short)
                  in gs end)

structure SolveSome = Solve(val Resolve = InferSome.resolve
                            val Paramodulate = InferSome.paramodulate)

local

fun     resolveAll(pos)(x) = STATE := #1(InferAll.resolve(pos)(x))
fun     resolveSome(pos)(x) = STATE := #1(InferSome.resolve(pos)(x))
fun     rewriteAll(pos)(x) = STATE := #1(InferAll.paramodulate(pos)(x))
fun     rewriteSome(pos)(x) = STATE := #1(InferSome.paramodulate(pos)(x))
fun     resolveConjunct(conc_nos,g_no,at_nos,cl_no,cl_mode,redex_lost)
          = STATE := InferAll.Rconjunct(conc_nos)(!STATE,g_no,at_nos,(makestring(cl_no:int),cl_no),
                                        cl_mode,redex_lost,backward,long,long)
fun     resolveDisjunct(conc_nos,g_nos,at_nos,cl_no,cl_mode,redex_lost)
          = STATE := InferAll.Rdisjunct(!STATE,g_nos,at_nos,cl_no,conc_nos,cl_mode,redex_lost)
fun     rewriteAtom(conc_no,g_no,t_pos,cl_no,cl_mode,redex_lost)
          = STATE := InferAll.Patom(conc_no)(!STATE,g_no,t_pos,(makestring(cl_no:int),cl_no),
                                   cl_mode,redex_lost,backward,long,long)
fun     rewriteDisjunct(conc_nos,g_nos,t_poss,cl_no,cl_mode,redex_lost)
          = STATE := InferAll.Pdisjunct(!STATE,g_nos,t_poss,cl_no,conc_nos,cl_mode,redex_lost)

in

fun     resPred g_no pos = resolveAll(pos)(!STATE,g_no,true,!DIRECT,long)

fun     resPredPres g_no pos = resolveAll(pos)(!STATE,g_no,false,!DIRECT,long)

fun     resPredRefute g_no pos = resolveSome(pos)(!STATE,g_no,true,!DIRECT,long)

fun     resPredPresRefute g_no pos = resolveSome(pos)(!STATE,g_no,false,!DIRECT,long)

fun     resThm g_no at_nos cl_no conc_nos
          = if !DIRECT = backward then resolveConjunct(conc_nos,g_no,at_nos,cl_no,thm,true)
            else STATE := InferAll.Rforward(!STATE,g_no,at_nos,cl_no,thm,true)

fun     resThmOr g_nos at_nos cl_no conc_nos
          = if !DIRECT = backward then resolveDisjunct(conc_nos,g_nos,at_nos,cl_no,thm,true)
            else STATE := InferAll.Rforward(!STATE,hd(g_nos),at_nos,cl_no,thm,true)

fun     resThmPres g_no at_nos cl_no conc_nos
          = if !DIRECT = backward then resolveConjunct(conc_nos,g_no,at_nos,cl_no,thm,false)
            else STATE := InferAll.Rforward(!STATE,g_no,at_nos,cl_no,thm,false)
```

```
fun     resThmOrPres g_nos at_nos cl_no conc_nos
          = if !DIRECT = backward then resolveDisjunct(conc_nos,g_nos,at_nos,cl_no,thm,false)
            else STATE := InferAll.Rforward(!STATE,hd(g_nos),at_nos,cl_no,thm,false)

fun     resConj g_no at_nos cl_no conc_nos
          = if !DIRECT = backward then resolveConjunct(conc_nos,g_no,at_nos,cl_no,conj,true)
            else STATE := InferAll.Rforward(!STATE,g_no,at_nos,cl_no,conj,true)

fun     resConjOr g_nos at_nos cl_no conc_nos
          = if !DIRECT = backward then resolveDisjunct(conc_nos,g_nos,at_nos,cl_no,conj,true)
            else STATE := InferAll.Rforward(!STATE,hd(g_nos),at_nos,cl_no,conj,true)

fun     resConjPres g_no at_nos cl_no conc_nos
          = if !DIRECT = backward then resolveConjunct(conc_nos,g_no,at_nos,cl_no,conj,false)
          else STATE := InferAll.Rforward(!STATE,g_no,at_nos,
                             cl_no,conj,false)

fun     resConjOrPres g_nos at_nos cl_no conc_nos
          = if !DIRECT = backward then resolveDisjunct(conc_nos,g_nos,at_nos,cl_no,conj,false)
            else STATE := InferAll.Rforward(!STATE,hd(g_nos),at_nos,cl_no,conj,false)

fun     rewFunc g_no pos = rewriteAll(pos)(!STATE,g_no,true,!DIRECT,long)

fun     rewFuncPres g_no pos = rewriteAll(pos)(!STATE,g_no,false,!DIRECT,long)

fun     rewFuncRefute g_no pos = rewriteSome(pos)(!STATE,g_no,true,!DIRECT,long)

fun     rewFuncPresRefute g_no pos = rewriteSome(pos)(!STATE,g_no,false,!DIRECT,long)

fun     rewThm g_no t_poss at_nos cl_no conc_nos
          = if !DIRECT = backward then rewriteAtom(hd(conc_nos),g_no,hd(t_poss),cl_no,thm,true)
            else STATE := InferAll.Pforward(!STATE,g_no,t_poss,at_nos,cl_no,conc_nos,thm,true)

fun     rewThmOr g_nos t_poss cl_no conc_nos
          = if !DIRECT = forward then raise not_applicable_to_premise
            else rewriteDisjunct(conc_nos,g_nos,t_poss,cl_no,thm,true)

fun     rewThmPres g_no t_poss at_nos cl_no conc_nos
          = if !DIRECT = backward then rewriteAtom(hd(conc_nos),g_no,hd(t_poss),cl_no,thm,false)
            else STATE := InferAll.Pforward(!STATE,g_no,t_poss,at_nos,cl_no,conc_nos,thm,false)

fun     rewThmOrPres g_nos t_poss cl_no conc_nos
          = if !DIRECT = forward then raise not_applicable_to_premise
            else rewriteDisjunct(conc_nos,g_nos,t_poss,cl_no,thm,false)

fun     rewConj g_no t_poss at_nos cl_no conc_nos
          = if !DIRECT = backward then rewriteAtom(hd(conc_nos),g_no,hd(t_poss),cl_no,conj,true)
            else STATE := InferAll.Pforward(!STATE,g_no,t_poss,at_nos,cl_no,conc_nos,conj,true)

fun     rewConjOr g_nos t_poss cl_no conc_nos
          = if !DIRECT = forward then raise not_applicable_to_premise
            else rewriteDisjunct(conc_nos,g_nos,t_poss,cl_no,conj,true)

fun     rewConjPres g_no t_poss at_nos cl_no conc_nos
          = if !DIRECT = backward then rewriteAtom(hd(conc_nos),g_no,hd(t_poss),cl_no,conj,false)
            else STATE := InferAll.Pforward(!STATE,g_no,t_poss,at_nos,cl_no,conc_nos,conj,false)

fun     rewConjOrPres g_nos t_poss cl_no conc_nos
          = if !DIRECT = forward then raise not_applicable_to_premise
            else rewriteDisjunct(conc_nos,g_nos,t_poss,cl_no,conj,false)
```

```
end    (* local *)

datatype parameters = detectCycles | moreSols | outermost | depthfirst | presNarrowed | trace | no

fun    solve g_no depth p1 p2 p3 p4 p5 p6
         = STATE := let val p1 = p1 <> detectCycles
                        val p2 = p2 <> moreSols
                        val p3 = p3 <> outermost
                        val p4 = p4 <> depthfirst
                        val p5 = p5 <> presNarrowed
                        val p6 = if p6 = trace then long else short
                    in SolveAll.goal_no(!STATE,g_no,!DIRECT,depth,p1,p2,p3,p4,p5,p6) end

fun    solveRefute g_no depth p1 p2 p3 p4 p5 p6
         = STATE := let val p1 = p1 <> detectCycles
                        val p2 = p2 <> moreSols
                        val p3 = p3 <> outermost
                        val p4 = p4 <> depthfirst
                        val p5 = p5 <> presNarrowed
                        val p6 = if p6 = trace then long else short
                    in SolveSome.goal_no(!STATE,g_no,!DIRECT,depth,p1,p2,p3,p4,p5,p6) end

fun    factor g_no at1_no at2_no = if !DIRECT = forward then raise not_applicable_to_premise
                                   else STATE := Unify.atoms(!STATE,g_no,at1_no,at2_no)

fun    unify g_no eq_pos = if !DIRECT = forward then raise not_applicable_to_premise
                           else STATE := Unify.terms(!STATE,g_no,eq_pos)

fun    replace g_no eq_pos t_poss = STATE := Replace.terms(!STATE,g_no,eq_pos,t_poss,!DIRECT)

fun    delGoals g_nos = STATE := Delete.goals(!STATE,g_nos,!DIRECT,long)

fun    delAtoms g_no at_nos = STATE := Delete.atoms(!STATE,g_no,at_nos,!DIRECT)

fun    eval g_nos = STATE := More.evaluate(!STATE,g_nos,!DIRECT)

fun    subsume() = STATE := More.subsume(!STATE,!DIRECT)

fun    createDis() = STATE := Finish.proof_dis(!STATE)

fun    createHCs gens = STATE := Finish.proof(!STATE,gens)

fun    conjs2thms() = STATE := Finish.conjects_to_thms(!STATE)

fun    storeSpec() = More.write_spec(!STATE)

fun    conc() = let val gs = InferAux.current_goal_set(!STATE,backward)
                in InferAux.show(gs,backward); DIRECT:=backward end

fun    prem() = let val gs = InferAux.current_goal_set(!STATE,forward)
                in InferAux.show(gs,forward); DIRECT:=forward end

fun    back() = STATE := More.backtrack(!STATE,1,!DIRECT)

fun    sp() = More.show_spec(!STATE)

fun    dc g_no scale = let val gs = InferAux.current_goal_set(!STATE,backward)
                       in PS_output.draw(gs,g_no,backward,scale) end

fun    dp g_no scale = let val gs = InferAux.current_goal_set(!STATE,forward)
                       in PS_output.draw(gs,g_no,forward,scale) end
```

```
val     DIRS = ref""
val     DIRP = ref""
val     DIRG = ref""
fun     Install specs proofs graphs exec
          = (DIRS:=specs; DIRP:=proofs; DIRG:=graphs; exportML(exec))
val     _ = (System.Control.Print.printDepth:=0; System.Control.Print.signatures:=0)
```

References

[AEH 94] S. Antoy, R. Echahed, M. Hanus, *A Needed Narrowing Strategy*, Proc. ACM Principles of Programming Languages (1994)

[BJM 91] J.-P. Banâtre, S.B. Jones, D. Le Métayer, *Prospects for Functional Programming in Software Engineering*, Springer 1991

[GH 93] J.V. Guttag, J.J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer 1993

[Han 94] M. Hanus, *The Integration of Functions into Logic Programming: From Theory to Practice*, J. Logic Programming 19&20 (1994) 583-628

[Mes 93] J. Meseguer, *A Logical theory of Concurrent Objects and its Realization in the Maude Language*, in: G. Agha, P. Wegner, A. Yonezawa, eds., Research Directions in Object-Based Concurrency, MIT Press (1993) 313-389

[Pad 88] P. Padawitz, *Computing in Horn Clause Theories*, Springer 1988

[Pad 91] P. Padawitz, *Inductive Expansion: A Calculus for Verifying and Synthesizing Functional and Logic Programs*, J. Automated Reasoning 7 (1991) 27-103

[Pad 92] P. Padawitz, *Deduction and Declarative Programming*, Cambridge University Press 1992

[Pad 95] P. Padawitz, *Swinging Data Types: The Dielectic between Actions and Constructors*, Report, FB Informatik, Universität Dortmund 1995, available from `http://ls5-www.informatik.uni-dortmund.de/user/peter`

[Pad 95a] P. Padawitz, *Specification and Deduction*, Report, FB Informatik, Universität Dortmund

[Pad 96] P. Padawitz, *Inductive Theorem Proving for Design Specifications*, J. Symbolic Computation 21 (1996) 41-99

[Pau 91] L.C. Paulson, *ML for the Working Programmer*, Cambridge University Press 1991

[Pau 94] L.C. Paulson, *Isabelle: A Generic Theorem Prover*, Springer LNCS 828 (1994)

[Plo 70] G.D. Plotkin, *A Note on Inductive Generalization*, in: B. Meltzer, D. Michie, eds., Machine Intelligence 5, Elsevier (1970) 153-163

[Ull 94] J.D. Ullman, *Elements of ML Programming*, Prentice-Hall 1994