Atelier B

Interactive Prover

Reference Manual

version 3.6

ATELIER B
Interactive Prover—Reference Manual
version 3.6

Document made by CLEARSY.

CLEARSY
ATELIER B Maintenance
Europarc de PICHAURY
1330 Av. J.R. Guilibert Gauthier de la Lauzière - Bât C2
13856 Aix-en-Provence Cedex 3
France

Tel: +33 (0)4 42 37 12 99
Fax: +33 (0)4 42 37 12 71
Email: maintenance.atelierb@clearsy.com

Contents

1 Introduction

This document constitutes the reference manual of the Atelier B interactive prover (PRI). It contains all the available commands, for the 3.6 version of Atelier B. The aim of this manual is not to give proof techniques, but to indicate to the user the syntax and the fields covered by the interactive proof commands.

For each function a description is given of the syntax to be used and the way to use it. An example will complete the description of the function.

This document also includes:

- the basic notions, necessary to carry out a proof (see chapter 3 page 7)
- the normalisation of proof obligations (PO), performed by the prover and the generator of proof obligations (GOP) (see chapter 4 page 17)
- the presentation of ways to enrich the prover's rule base (PatchProver (see chapter 8 page 145), PMM (see chapter 7 page 143))
- the list of available commands, classified by type and in alphabetical order (see chapter 12 page 155)

2 Changes since version 3.5

2.1 Filters for User_Pass

A filtering of the User_Pass rules allows to try the User_Pass commands on some proof obligations only (refer to chapter 10, page 149).

2.2 Customizing the prover using resources

Some of the prover functions can be customized using resources.

- Time_Out
  - Represents the time-out in seconds for satellite provers (predicate prover and mono-lemma prover) in User_Pass or Prove Replay automatic mode.
  - This resource enables to try in automatic mode (User_Pass) some proof tactics that use satellite provers massively.
  - This possibility of modulation enables to start proofs with small time-out (maximum calculus time allowed before the proof process stops) in order to quickly test the efficiency of such a tactic.
  - See also page 139.
- Keep_Non_Simplified_Hypothesis
  - This resource enables to choose keeping or not non-simplified predicates together with their simplified version.
  - In some cases, the simultaneous presence of the simplified and non-simplified forms of a predicate can prevent the application of some prover mechanisms. For example, the prover can't perform a *Modus ponens* on the following hypotheses: P, P' and (P and P') $\Rightarrow$ Q though it performs it on P and P' $\Rightarrow$ Q. It is in such cases that it is interesting to set the resource to **FALSE** (that means *do not keep the non-simplified predicates*).
  - See also page 140.
- Use_Rule_Package
  - Setting the resource to the `p1` value enables to use additional proof rules along with the automatic prover regular ones.

  - These rules enable to handle *B language* symbols that weren't fully handled by the rules of the prover native base. Thus, a B model in which the proof obligations contain the mod or / symbols are more likely to be automatically demonstrated using the p1 rules (referring to these symbols) than using only the prover regular rules.
  - See also page 141.

- Max_Number_Of_Universal_Hypothesis_Instanciation
  - The *GenAny* mechanism of the prover enables to particularize hypotheses of the form $\forall X \cdot (P(X) \Rightarrow Q)$, where $P(X)$ is a predicate verified by X, according to hypotheses $P(X_i)$, that is to say, to generate hypotheses $[X := X_i]\ Q$ for all X_i verifying P.
  - By giving a quadruplet of integers, the resource enables to set the maximum number of applications of the *GenAny* mechanism for each of the four prover forces (respectively 0, 1, 2 and 3).
  - It may be interesting to limit the number of instantiations of universally quantified formulas to avoid a combinative (combinatorial?) explosion, especially in cases where there are a lot of universally quantified hypotheses.
  - See also page 142.

2.3 New commands have been added

- AbstractExpression
  - The command ae(EX, ID) enables to generate the goal $ID = EX \ \Rightarrow\ G$ where G represents the current goal in which the occurrences of the EX expression have been substituted by the ID identifier.
  - A complex goal is simplified using this command and then becomes more readable.
  - The predicate prover becomes more efficient as it works on a simpler goal.
  - See also page 19.

- CurrentGoal
  - This command displays the current goal.
  - Enables a better readability of the goal when several interactive commands were tried.
  - See also page 42.

- ModelChecking
  - This command enables to perform demonstration by cases of a goal that involves variables whose values are taken from enumerated sets.
  - It is interesting to use this command on goals containing boolean variables: it is, indeed, generally easier to demonstrate some goals when values of some of their variables are known.

  - See also page 78.

- MonoLemma prover
  - The Mono-Lemma prover works almost the same as the automatic prover but with a little different processing of hypotheses. This command enables to start the Mono-Lemma prover on the current goal.
  - It has the significant advantage to be able to be started with a time-out and a proof force level: it is possible to locally change the proof force.
  - See also page 84.

- PreViousPo
  - Enables to move to the previous unproved PO (if there are some).
  - It allows another browsing mode among PO, in addition to NExtPO.
  - See also page 105.

- ShowProof
  - This command displays the saved interactive proof commands of a given proof obligation.
  - When two proof obligations are quite similar, it may be wise to use or to be inspired by the proof commands of one of the two proof obligations to prove the other one. One need only to display these commands using the **ShowProof**.
  - See also page 116.

- UserSimplification
  - Application of user-provided rewrite rules while limiting memory consumption.
  - Whereas the classical rewriting is very memory consuming, this command enables to perform the same work but in an optimized manner.
  - See also page 134 and page 147.

2.4 The predicate prover has been enhanced

The predicate prover is now available in version 6.1. It contains:

- the arithmetic solver (normaliser) of the prover,
- the arithmetic prover, for predicates and arithmetic expressions processing,
- the taking into account of sequences and of the symbols succ, pred, max, card.

The hypotheses selection for the predicate prover has been enhanced.
See also page 96.

2.5 Miscellaneous

- Dynamic display of the proof tree during interactive proof enables to get one's bearing in the current proof (*Graphical Trace* page 71).
- A formula logical analyser enables to improve the understanding of a complex formula by the user (*Logical Analysis* page 74, *Display Term* page 55).
- A textual on-line help allows to quickly get the syntax and effects of the various interactive commands (*Help* page 73).
- The *GlobalSituation* (page 68), *SearchRule* (page 118) and *TryEverywhere* (page 128) commands have new options.

3 Basic Notions

3.1 What is a proof obligation?

A proof obligation (PO) consists of a goal G and a set of hypotheses H. Demonstrating a PO means demonstrating this goal G, under the assumption that all the hypotheses of H are verified.

The hypotheses H are said to be *contextual hypotheses*.

If the current goal is in the form $P \Rightarrow Q$, by application of the deduction principle, P will become a hypothesis and the new goal becomes Q. P is then called *a derived hypothesis*.

The *current hypotheses* are composed of the contextual hypotheses and the derived hypotheses.

The *hypotheses stack* contains all the contextual hypotheses and the derived hypotheses, in the order they appeared.

3.2 What is a well-typed predicate?

When expressions are entered into the interactive prover (as command parameters), take care to not introduce typing mistakes.

For example:

- introduction of the predicate $i = E$ where i and E are typed as follows: $i \in \mathbb{Z}$ and $E \in \mathbb{P}(\mathbb{Z})$

  The following typing is correct: $i \in \mathbb{Z}$ et $E \in \mathbb{Z}$.

- introduction of the predicate $E \leq F$ where E and F are typed as follows: $E \in \mathbb{P}(\mathbb{Z})$ and $F \in \mathbb{P}(\mathbb{Z})$

  The following typing is correct: $E, F \in \mathbb{Z} \times \mathbb{Z}$.

- introduction of the predicate $b = e_1 + e_2$ where b, e_1 and e_2 are typed as follows: $b \in \mathsf{BOOL}$, e_1 and e_2 belong to an enumerated set.

  The following typing is correct: $b, e_1, e_2 \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.

In general, a typing error will produce a ill-typed goal which will not be provable.

3.3 What is a well-defined expression?

3.3.1 Presentation

A mathematical expression is well-defined when it can be assigned a meaning (refer to Mdelta User Manual, version 1.0). On the opposite case, the expression would be said as *meaningless*. Any expression requiring conditions to avoid being meaningless is defined as *potentially meaningless*.

For example, given the expression:

$$y = \frac{x}{\frac{x+8}{c}} \tag{3.1}$$

This expression can be true or false, provided that it is well-defined. If this expression is not well-defined, it is then impossible to associate it a true or false value. This ill-definedness means that at least one operator of the expression has at least one operand that does not belong to its domain. Expression (3.1) is obviously arithmetic by nature. We consider that this expression is has been type-checked (operation performed by the type checker) so that y, x, and c are integers.

The operators appearing in the expression (3.1) are:

- equality

  An equality a=b is well-defined on the condition that a and b are well-defined

- addition

  An addition a+b is well-defined on the condition that a and b are well-defined

- integer division

  An integer division a/b is well-defined on the condition that a and b are well-defined and b is non-zero.

We will then have to check that:

- the denominator of $x/(x + 8/c)$ is non-zero
- the denominator of $8/c$ is non-zero

To establish that expression (3.1) is well-defined, the user has to prove the following predicates:

$$(x + 8)/c \neq 0 \tag{3.2}$$

$$c \neq 0 \tag{3.3}$$

The expression context has to contain these predicates in the form of hypotheses or to enable to deduce them.

If it is not the case, the expression (3.1) is potentially ill-defined. Refer to table (1) to see the list of the expressions that can be ill-defined.

3.3.2 Conditions of well-definedness

The well-definedness conditions are listed in the table below.

Expression	Condition of well-definedness
a^b	$a \in \mathbb{N} \land b \in \mathbb{N}$
$a \bmod b$	$b \in \mathbb{N}_1 \land a \in \mathbb{N}$
a/b	$b \in \mathbb{Z}_1$
$\Pi(x).(P\|E)$	$\{x\|P\} \in FIN(\{x\|P\})$
$\Sigma(x).(P\|E)$	$\{x\|P\} \in FIN(\{x\|P\})$
$\mathsf{max}(S)$	$S \cap \mathbb{N} \in FIN(\mathbb{N}) \land S \neq \emptyset$
$\mathsf{min}(S)$	$S \cap (\mathbb{Z} - \mathbb{N}) \in FIN(\mathbb{Z}) \land S \neq \emptyset$
$\mathsf{card}(S)$	$S \in FIN(S)$
$\mathsf{inter}(U)$	$U \neq \emptyset$
$\bigcap(x).(P\|E)$	$\{x\|P\} \neq \emptyset$
r^n	$n \in \mathbb{N}$
$f(x)$	$x \in \mathsf{dom}(f) \land f \in \mathsf{dom}(f) \ \nrightarrow \ \mathsf{ran}(f)$
$\mathsf{perm}(S)$	$S \in FIN(S)$
$\mathsf{conc}(s)$	$s \in \mathsf{seq}(\mathsf{ran}(s)) \land \forall x.(x \in \mathsf{dom}(s) \ \Rightarrow \ s(x) \in \mathsf{seq}(\mathsf{ran}(s(x))))$
$s \frown t$	$s \in \mathsf{seq}(\mathsf{ran}(s)) \land t \in \mathsf{seq}(\mathsf{ran}(t))$
$\mathsf{size}(s)$	$s \in \mathsf{seq}(\mathsf{ran}(s))$
$\mathsf{rev}(s)$	$s \in \mathsf{seq}(\mathsf{ran}(s))$
$s \leftarrow e$	$s \in \mathsf{seq}(\mathsf{ran}(s))$
$e \rightarrow s$	$s \in \mathsf{seq}(\mathsf{ran}(s))$
$\mathsf{tail}(s)$	$\mathsf{size}(s) \geq 1 \land s \in \mathsf{seq}(\mathsf{ran}(s))$
$\mathsf{first}(s)$	$\mathsf{size}(s) \geq 1 \land s \in \mathsf{seq}(\mathsf{ran}(s))$
$\mathsf{front}(s)$	$\mathsf{size}(s) \geq 1 \land s \in \mathsf{seq}(\mathsf{ran}(s))$
$\mathsf{last}(s)$	$\mathsf{size}(s) \geq 1 \land s \in \mathsf{seq}(\mathsf{ran}(s))$
$s \uparrow n$	$n \in 0 \,..\, \mathsf{size}(s) \land s \in \mathsf{seq}(\mathsf{ran}(s))$
$s \downarrow n$	$n \in 0 \,..\, \mathsf{size}(s) \land s \in \mathsf{seq}(\mathsf{ran}(s))$

Table (1): Potentially meaningless expressions

3.4 What is a rule?

A rule is a formula of the following form: A $\Rightarrow$ B.
A is called the antecedent of the rule.
B is called the consequent of the rule.
A and B can be predicate conjunctions.
A can be omitted. In this case, the rule is said to be atomic.
A rule can be:

- inductive (backward)
  If the current goal is B, then to prove B, it is sufficient to prove A.
  A is supposed to be more simple than B or, at least, more easily provable than B.

- deductive (forward)
  If hypotheses A appear in the hypotheses stack, then hypotheses B are generated and loaded in the stack, if they do not already exist.

- for rewriting
  In this case, B has the form C == D.
  If A is verified then C is rewritten as D.
  This type of rule only applies to sub formulas of the current goal or to the current goal itself.

  For example, the rule SimplifyIntMaxXY.3:

```
btest(p<=q)
=>
max{{p}\/{q}} == q
```

  can be applied to goal

  $$0 \leq \mathsf{max}(\{3\} \cup \{5\}) - \mathsf{min}(1..4)$$

  transforming it into

  $$0 \leq 5 - \mathsf{min}(1..4)$$

Rules, contrary to hypotheses and goals, contain wildcards. A wildcard is a variable, which can take any value (literal, expression, ...) If it is assigned a value, it is said to be instantiated. A wildcard is represented by a letter of the alphabet: thus no more than 52 wildcards can appear inside a rule (lower case and upper case letters).

3.5 What is a theory?

A theory consists in a container of rules, written in theory language [1].

Rules are named $t.n$, with

- t: name of the theory,
- n: index of the rule in the theory [2].

Example:

```
THEORY th1 IS

    binhyp(a: B) &
    binhyp(B<: C)
    =>
    a: C;


    btest(0<=-t)
    =>
    0<=t**2 - 4*t + 1

END
```

The prover always tries to apply higher index rules before lower index rules (from the " bottom" of the theory to the "top").

3.6 What is a tactic?

A tactic is an ordered list of theories, that determines the traversal of a rule base [3] to determine if one or several rules will be applied.

Theories are classified in two categories:

- Backward The goal to be handled is divided into sub-goals or discharged.
- Forward New hypotheses are generated.

[1] see *Reference Manual of the Logic Solver*

[2] the rule put at the beginning of the theory corresponds to index 1

[3] A rule base is composed of a set of theories.

A full tactic is presented as a combination of a backward tactic and a forward tactic:

```
<tactic>          ::= <backward tactic> , <forward tactic>
<backward tactic> ::= <backward tactic> ; <backward tactic>
                      <backward tactic> ~
                      theory name

<forward tactic>  ::= <forward tactic> ; <forward tactic>
                      <forward tactic> ~
                      theory name
```

The forward tactic is optional.

The traversal of a tactic thus consists in searching in each backward theory a rule that can be applied. The search order corresponds to the order of the listed theories.

If a rule can be applied, it is then applied and the search continues with the next theory. This process is repeated until no more backward theories remain.

If, during the search through the backward theories, at least one hypothesis has been generated, the process described for the backward theories will apply for the list of forward theories (if there are any), and this is repeated each time a hypothesis is produced.

A theory or a group of theories can be "tilded". In this case, there will be an attempt to apply a rule of this theory as long as the rules of this theory are applied. When no more rules are applied, the next theory will be examined.

Example:

```
((t1,t2)~,t3,t4~)
```

Implicitly, the most external list of theories is "tilded", which means that:

```
(t1; t2; t3;...;tn)
```

is equivalent to

```
(t1; t2; t3;...;tn)~
```

3.7 What is a proof?

The Atelier B prover is composed of an automatic prover and an interactive prover. The automatic prover enables to attempt to demonstrate automatically a set of proof obligations, by applying a given set of general mechanisms. The prover is parameterizable according to its force (Fast, force 0 to force 3). The higher the force is, the longer the proof takes.

The interactive prover enables the user to assist the automatic prover in its demonstration task, by carrying out the proof (addition of hypotheses, proof by contradiction, proof by cases, ...). The prover is carried out using interactive commands . These commands are applied for a given proof obligation and are saved for this PO. They constitute the command line.

In spite of all our efforts, the prover may sometimes loop (see chapter 3.9 page 14), i.e, it has a highly diverging, uninterrupted, iterative behaviour . The probabilities of such a phenomenon occurring increase with the force used.

3.8 The prover

The Atelier B automatic prover is composed of:

- mechanisms

  These mechanisms orientate the proof, in order to demonstrate the goals of the proof obligations. They enable triggering off the rules of the rule base.

- rules

  These rules are those of the prover rule base (see chapter 3.4 page 9). They may also have been added by the user (manual rules) using:

  – a **pmm** file for a given component (see chapter 7 page 143)
  – a **PatchProver** file for a full project (see chapter 8 page 145).

  Beware!! The use of manual rules calls the proof validity into question (false rules may lead to prove false proof obligations). Thus, when rules are added, they must be rigorously demonstrated and inspected by a third party.

- Resources

  The prover can be customized by setting the following resources in a resource file:

  – **Keep_Non_Simplified_Hypothesis** if set to **TRUE**, it enables to keep in hypotheses, the simplified predicates (by the prover) along with their non simplified version. If set at **FALSE**, it tells the prover to keep only the simplified version of the hypotheses. Default is **TRUE**.
  – **Time_Out** whose value is an integer expressing the time-out of satellite provers (predicate prover and mono-lemma prover) in *User_Pass* and *Replay* mode, i.e. the time after which we decide to stop the action of satellites provers. Default is 300 seconds.
  – **Use_Rule_Package** when set at the p1 value enables to use additional mathematical rules.
  – **Max_Number_Of_Universal_Hypothesis_Instanciation** can be set to a nuplet of four natural integers whose values correspond to the maximum number of instantiations of universally quantified hypotheses for respectively the 0, 1, 2 and 3 proof force.

3.9 What is a prover loop?

For example, if we use the rule:

```
a*a == a*a*a/a
```

on the goal to be proved

```
cc(vv) = vv*vv
```

We will obtain successively

```
cc(vv) = vv*vv*vv/vv
cc(vv) = (vv*vv*vv/vv)*vv/vv
...
```

The proof kernel produces the following messages:

```
krt: sequence  memory short
krt: asking for 1500000 int, waiting for system reply...
krt: OK, memory obtained, we continue.

krt: sequence  memory short
krt: asking for 2249997 int, waiting for system reply...
krt: OK, memory obtained, we continue.

krt: sequence  memory short
krt: asking for 3374992 int, waiting for system reply...
krt: OK, memory obtained, we continue.

...
```

The messages `krt: sequence  memory short` are generated by the kernel which dynamically allocates the lacking memory.

This example is simple. Loops may occur for groups of rules and may be a priori more difficult to detect.

3.10 What is a command?

A command can be:

- simple

  It is composed of a single Interactive command. The simple commands are listed at the end of the document (see chapter 12 page 155).

- mixed

  A mixed command is a list of simple commands separated by &. The various commands from the list will be applied successively. when we use the **rr** command (see

chapter 5.39 page 110) and we have just entered a mixed command, this mixed command will be entirely replayed.

Example of a mixed command:

```
ah(aa+bb<=100) & pr & dd & pr
```

4 Normalisation of proof obligations

Hypotheses and goals are normalised by the Proof Obligation Generator and the prover. This normalisation enables the transformation of expressions into expressions in normal form, which will afterwards be used by all the rules related to this expression.

This limits polymorphism of the rules of the prover rule base, and thus their number.

The normal forms selected are:

Expression	Normal Form
$n > m$	$m + 1 <= n$
$m < n$	$m + 1 <= n$
$a \ <=> \ b$	$(a => b)\&(b => a)$
$a \ <: \ b$	$a : POW(b)$
$a \ <<: \ b$	$a : POW(b)\&not(a = b)$
$a \ / : \ b$	$not(a : b)$
$a \ / = \ b$	$not(a = b)$
$a \ / <: \ b$	$not(a : POW(b))$
$a \ / <<: \ b$	$a : POW(b) => a = b$
$a : NATURAL$	$a : INTEGER\&0 <= a$
$NATURAL1$	$NATURAL - \{0\}$
$NAT1$	$NAT - \{0\}$
$FIN1(A)$	$FIN(A) - \{\{\}\}$
$POW1(A)$	$POW(A) - \{\{\}\}$
$seq1(A)$	$seq(A) - \{\{\}\}$
$iseq1(A)$	$iseq(A) - \{\{\}\}$
$perm(E)$	$iseq(E)/\backslash(NATURAL - \{0\} + - >> E)$
$<>$	$\{\}$
$\{x, y\}$	$\{x\}\backslash/\{y\}$
$\{x \mid P\}$	$SET(x).P$

It is advised during a rule writing, to check that this rule is normalised. If not, the rule will be normalised when loading and may not be applied anymore.

For example, the following rule:

```
btest(0<x)
=>
0<=x**2-1
```

is normalised into

```
btest(0+1<=x)
=>
0<=x**2-1
```

But the **btest** only accepts parameters with the form `a op b`, where `a` and `b` are literal integers. This rule will never be applied. It should have rather been written:

```
btest(1<=x)
=>
0<=x**2-1
```

5 Interactive commands

5.1 Abstract Expression

Abstracting an expression

Syntax

ae(EX,ID)
with:

- EX is a mathematical expression with at least one occurrence appearing in the current goal,
- ID is an identifier (in the B language sense) which is neither free in the goal nor in the hypotheses.

Use

This command enables to generate the goal $\Delta_e(EX) \wedge (EX = ID => [EX := ID]G)$ where $\Delta_e(EX)$ represents the well-definedness lemma of the expression EX (see chapter 3.3 page 8) and $[EX := ID]G$ the current goal in which every occurrence of expression EX (appearing in the goal) has been replaced by identifier ID.

It is required to check the well-definedness of expression EX as it might be ill-defined, once taken out of context (i.e., out of the current goal). Let's assume that under the hypothesis $xx + 1 <= 1$ we have the goal $xx + 1 <= 1 \vee \mathsf{max}(1..xx) = xx$. This goal is well-defined because $xx + 1 <= 1$ is well-defined and $not(xx + 1 <= 1) \Rightarrow \Delta_p(\mathsf{max}(1..xx) = xx)$ *i.e.,* $not(xx + 1 <= 1) => \Delta_e(\mathsf{max}(1..xx))$, from the well-definedness of $\vee$ and max this yields $not(xx + 1 <= 1) => 1..x \cap \mathsf{NAT} \in FIN(\mathsf{NAT}) \wedge not(1..xx) = \emptyset$. If we perform `ae(max(1..xx),MAX)` without caring about well-definedness, we get the goal $\mathsf{max}(1..xx) = MAX => xx + 1 <= 1 \vee MAX = xx$. Then a deduction (by `dd`) raises in hypotheses the formula $\mathsf{max}(1..xx) = MAX$ which is now meaningless as we have the hypothesis $xx + 1 <= 1$ (and thus $1..xx = \emptyset$).

Let us finally remark that the prover attempts an automatic proof of the lemma of well-definedness of the expression EX, and if the proof fails, the lemma of well-definedness will have to be interactively demonstrated.

Example 1

Consider the following goal to be demonstrated:

```
1+wr<=p2 => p2+1<=p1 or p1<=wr &
1+p2<=wr => p2+1<=p1 & p1<=wr
=>
p2+1<=p1 or p1<=p3
```

The user wants to replace the expression $p2 + 1$ with $pp2$:

```
PRI> ae(p2+1,pp2)
```

The following goal is obtained (the well-definedness lemma of $p2 + 1$, reduced to **btrue**, is automatically discharged by the prover):

```
p2+1=pp2
=>
        (1+wr<=p2 => pp2<=p1 or p1<=wr &
        1+p2<=wr => pp2<=p1 & p1<=wr
        =>
        pp2<=p1 or p1<=p3)
```

We can then raise in hypothesis the equality $pp2 = p2 + 1$ (using the **deduction** `dd` command) and this is checked with the **Search Hypothesis** command:

```
PRI> sh(p2)

Searching all Hypothesis that
  contain p2
  match with a
Starting search...
Found hypothesis List is
    pp2=p2+1 &
    3<=p2 &
    p2<=2147483647 &
    0<=p2 &
    p2: INTEGER
End of found hypothesis
```

Example 2

Given the following goal to be demonstrated:

```
Hypothesis
  ...
  0<=aa &
  aa <= bb &
  ...
Goal
  max(aa..bb) = {bb}
```

The user wants to replace expression $max(aa..bb)$ by $MAXI$:

```
ae(max(aa..bb),MAXI)
```

The well-definedness lemma associated to expression $max(aa..bb)$ is $not(aa..bb = \emptyset) \wedge aa..bb \cap NATURAL : FIN(NATURAL)$. The automatic prover fails to demonstrate it, it then generates the first following sub-goal:

```
not(aa..bb = {})
```

Once this goal has been demonstrated (for example by `ah(aa<=bb) & pp(rp.0)`), the prover generates the following sub-goal:

```
aa..bb /\ NATURAL: FIN(NATURAL)
```

The `pr` command is sufficient to demonstrate this goal. Then, the well-definedness of expression $max(aa..bb)$ is proved. The prover can perform the substitution of $max(aa..bb)$ by $MAXI$ in the starting goal and generate the new goal:

```
max(aa..bb) = MAXI => MAXI = {bb}
```

5.2 Add hypothesis

Addition of a hypothesis

Syntax

ah(P)

with:

- P is a predicate

Use

This command enables to add the P predicate in the hypotheses stack.

In the current hypotheses, the P predicate must:

- be well-typed (refer to chapter 3.2 page 7),
- be well-defined (see chapter 3.3 page 8),
- be deducable from current hypotheses.

If the current proof was
prove G under the hypotheses h_1, ..., h_n
the prover will try successively
to prove P under the hypotheses h_1, ..., h_n
then
to prove G under the hypotheses h_1, ..., h_n, P

These proofs are carried out with the current proof force.

Once the command **ah(P)** is executed, the current goal becomes P. If after the command **pr** (see chapter 5.34 page 99), the current goal is still P, the hypothesis P has not been proved.

When the hypothesis P has been proved, the goal becomes $P \Rightarrow G$. The user can then, either add the hypothesis P directly (command **dd** (see chapter 5.13 page 53)), or activate the automatic prover which will add the hypothesis P', obtained after operations on P.

If P cannot be proved with the current force, we can try a higher force. The whole command line will then be executed again with the new force.

As this command is not protected against ill-typing and ill-definedness, the user has to check that the added hypothesis is well-typed and well-defined.

This can be checked, *with hindsight*, using tool **mdelta** (refer to its User manual version 1.0).

Example

Given the following situation:

```
    Hypothesis
        xx: 1..10 &
        yy: 1..10 &
        zz: 1..100
    Goal
        xx+yy-1: 1..100
```

The user wishes to add the hypothesis $xx + yy :$ 2..20. He executes the command **ah**:

```
PRI> ah(xx+yy: 2..20)
 Starting Add Hypothesis
```

The new goal becomes:

```
    Goal
        xx+yy: 2..20
```

This hypothesis must be proved in order to be able to continue. The user starts the automatic prover:

```
PRI> pr
Starting Prover Call
```

The hypothesis $xx + yy :$ 2..20 has been proved: the goal becomes $xx + yy :$ 2..20 => *current goal*.

```
    Goal
        xx+yy: 2..20 => xx+yy-1: 1..100
```

By using the command **pr** (see chapter 5.34 page 99) or **dd** (see chapter 5.13 page 53), the proof can then go on with the new hypothesis.

5.3 Arithmetical proof

Calling the Arithmetical Prover

Syntax

ap
ap(n)

with:

- n is a numeric value which permits to limit the functioning of the mechanism. If this value is not specified, the value of 400 is taken.

Use

The arithmetical prover is a mechanism whose aim is to search for a contradiction in a set of inequations. This contradiction is sought by creating new inequations by linear combination. The number of inequations is limited so as to avoid a loop in the mechanism.

The **ap** command permits the calling of the mechanism on the current proof obligation. The mechanism will work on the inequations contained in the hypotheses stack. If the current goal is an inequation of the form $a \leq b$, then the inequation $a > b$ is added to the list of inequations on which the mechanism is going to work.

Example 1

Given the following proof obligation:

```
Hypothesis
    xx: INTEGER &
    0<=xx &
    xx<=10 &
    yy: INTEGER &
    0 = 1+yy-xx &
    xx-1 = yy &
    btrue &
    0<=9 &
    9: INTEGER
Goal
    xx-1<=9
```

Given the form of the goal and the number of inequations in hypotheses, the use of the **ap** command is advised.

```
PRI> ap
Begin Arithmetic Proof
```

The current proof is therefore discharged.

Example 2

This example shows the behaviour of the command when the mechanism fails in its work. Given the following proof obligation:

```
    Hypothesis
        xx: INTEGER &
        0<=xx &
        xx<=10 &
        btrue &
        0<=9 &
        9: INTEGER
    Goal
        xx<=9
```

The proof obligation being false, the command doesn't discharge the goal.

```
PRI> ap
Begin Arithmetic Proof
This Command gives nothing new
```

The **ap** command is not saved, the current goal is not modified.

5.4 Apply rule

Application of a rule or of a user theory

Syntax

ar(T)
ar(r, M)

with

- T is a theory or tactic identifier (see chapter 3.5 page 11),
- r is a rule identifier (Theory.number),
- M is a mode which depends on the type of rule:
  - Rules performing no rewriting:
    - M=**Once**: a single application of the rule
    - M=**Multi**: application of the rule, as long as applicable
    - M=**Fwd**: application of the rule forward. The rule should then take the form $h_1 \wedge \ldots \wedge h_n \Rightarrow g$
      For all the sets of the n valid hypotheses which matches $h_1, \ldots, h_n$, the corresponding hypothesis g is generated.
    - M=**Fwd(P)**: Idem previous case, but we impose in addition that one of the hypotheses $h_1, \ldots, h_n$ matches [1] P.
  - Rewrite rules:
    M must be in the form A or A.B, with:
    - A = **AllHyp**: rewriting of all the hypotheses.
    - A = **Goal**: rewriting of the goal.
    - A = **Hyp(f)**: rewriting of the hypotheses that coincide with f.
    - B can be **Part(g)**. In this case, the rewriting is restricted to the sub formulas of the selected formulas which match g.

Use

ar is a command enabling the application of a rule or a theory on different parts of the lemma to prove.

For rewrite rules, the argument M enables controlled action on any part of the proof obligation, goal or hypothesis .

[1] We say that H matches P if H and P have the same form (i.e., that it is possible to instantiate the wildcards of P to obtain exactly H).
For instance, $xx = yy + 3 - \mathsf{min}(4\,..\,7)$ is matched by:
· $a = b$
taking xx for a and $yy + 3 - \mathsf{min}(4\,..\,7)$ for b
· $c = d - e$
taking xx for c, $yy + 3$ for d and $\mathsf{min}(4\,..\,7)$ for e

The rules used are those of the prover rule base, any rules contained in the pmm file (see chapter 7 page 143) associated with the component, and also the rules contained in the PatchProver file (see chapter 8 page 145).

To access the rules base, click on the "Display/print" menu and then, on the "Display Rules Database" button, of the window **INTERACTIVE PROOF**of the interactive prover.

Application in backward mode (M=Once or M=Multi):

- if the rule is in the form
  $a_1 \wedge \ldots \wedge a_n \Rightarrow c$
  and the current goal is in the form c, the proof is divided into n sub-goals
  $a_1, \ldots, a_n$
- If the rule is in the form
  $a_1 \wedge \ldots \wedge a_n \Rightarrow c == d$
  and c is a sub-formula [2] of the current goal, the proof is divided into n sub-goals
  $a_1, \ldots, a_n$
  If the n sub-goals are proved, then c is rewritten in d.

Application in forward mode(M=Fwd):

- if the rule is in the form
  $a_1 \wedge \ldots \wedge a_n \Rightarrow c$
  the antecedents $a_1, \ldots, a_n$ are sought in the hypotheses to generate the new hypothesis c.
  The procedure can easily loop (for example: the rule loops because it is re-applied on its consequent), this is why, the option Fwd(P) enables us to impose that one of the hypotheses a_i found coincides with P.

Application of a tactic:

- if we use only *Backward* theories The command will be:

```
ar(tb1;tb2;...;tbn)
```

- if we use *Backward* theories *Forward* The command will be:

```
ar((tb1;tb2;...;tbn;DED),(tf1;tf2;...;tfp))
```

[2] For example, xx + yy is a sub-formula of $0 \leq min(1..xx + yy)$

The DED theory (native theory of the kernel raising the hypotheses in the stack) is obligatory. The $Forward$ theories are only called when a hypothesis is raised. The $Backward$ theories generate derived goals $P => Q$ but do not raise the hypotheses P. It is thus necessary to associate a theory to them enabling a direct deduction (DED). When the Command **ar** has finished, the hypotheses will be associated to the current goal, which will then be:

$$\text{Hypotheses generated} \Rightarrow \text{current goal}$$

Of course, the theories can be "tilded". For example:

```
ar(((tb1;tb2~;)~;tbn;DED),(tf1;tf2~;...;tfp~))
```

Implicitly

```
ar((tb1;...;tbn),(tf1;...;tfp))
```

is equivalent to

```
ar((tb1;...;tbn)~,(tf1;...;tfp)~)
```

After the application of the **ar** command, if new hypotheses are generated, the goal is in the form $H \Rightarrow G$ (the user can see the new hypotheses generated). It is necessary to perform the **pr** command (see chapter 5.34 page 99) to reactivate the proof and raise these hypotheses.

Warning! If the user-provided rules (pmm, PatchProver) are used, the validity of the proof can be questioned. We must then perform a mathematical demonstration for each of these rules.

Example 1

Let us consider the following situation:

```
Hypothesis
    xx: 1..10 &
    yy: 1..10 &
    zz: 1..100
Goal
    xx+yy-1: 1..100
```

The user uses the *pmm* file as follows:

```
THEORY test IS

        a: 1..d &
        b: 1..d
        =>
        a+b: 2..2*d;

        d <= a-c &
        a-c <= e
        =>
        a-c: d..e

END
```

The *test* theory is read and compiled, using the **pc** command (see chapter 5.31 page 89).

```
PRI> pc
Loading theory test
```

The rule *test*.1 is then applied in $forward$ mode(generation of hypotheses).

```
PRI> ar(test.1,Fwd)
Starting Apply Rule
```

5 new hypotheses have been generated. The goal becomes:

```
    Goal
        xx+xx: 2..20 &
        xx+yy: 2..20 &
        yy+xx: 2..20 &
        yy+yy: 2..20 &
        zz+zz: 2..200
        =>
       xx+yy-1: 1..100
```

Using the **dd** command (see chapter 5.13 page 53).

```
PRI> dd
Starting Deduction
```

the hypotheses are then raised in the hypotheses stack.

```
 New Hypothesis since last command
    xx+xx: 2..20 &
    xx+yy: 2..20 &
    yy+xx: 2..20 &
    yy+yy: 2..20 &
    zz+zz: 2..200
Goal
    xx+yy-1: 1..100
```

The rule *test*.2 is then used in the *backward* mode.

```
PRI> ar(test.2, Once)
Starting Apply Rule
```

The rule is applied (we must check that the command line contains $ar(test.2, Once)$) and the two sub-goals $1 \leq xx + yy - 1$ and $xx + yy - 1 \leq 100$ will be processed. The first sub-goal is to be proved:

```
Goal
    1<=xx+yy-1
```

The automatic prover is called for the first time:

```
PRI> pr
Starting Prover Call
```

The first sub-goal is discharged. The second sub-goal becomes the current goal.

```
Goal
    xx+yy-1<=100
```

By calling the automatic prover

```
PRI> pr
Starting Prover Call
```

the second sub-goal is discharged and the proof obligation is proved, provided that the rules contained in the pmm file are accurate.

Finally, the command line is:

```
        Force(0) &
          ar(test.1,Fwd) &
            dd &
              dd &
                ar(test.2,Once) &
                  pr &
                  pr &
          Next
```

Example 2

Given the following situation:

```
    Hypothesis
        tt: {e1,e2,e3,e4,e5} => zz = e5 &
        zz = e5 => tt: {e1,e2,e3,e4} &
        tt = e5 => zz = e1 &
        zz = e1 => tt = e5
    Goal
        tt = e5 or zz = e2
```

associated to the following *pmm* file:

```
THEORY test IS

        bguard(WRITE: bwritef("Application of test.1\n")) &
        (B=>not(A))
        =>
        (A or B)

END
&
THEORY testbis IS

        a = b &
        b: E
        =>
        a: E
END
```

The user attempts to apply the backward theory *test* to the current goal. The hypotheses will be raised by the DED theory. If hypotheses are generated, the forward theory *testbis* will then be tried.

```
PRI> ar((test;DED),testbis)
Application of test.1
Starting Apply Rule
```

The rule *test*.1 contains a guard enabling to print a message indicating its activation (*Application of test.1*). The rule *testbis*.1 was activated when hypothesis $e2 = zz$ was raised and enabled the generation of hypothesis $zz \in \{e1, e2, e3, e4, e5\}$. All the hypotheses generated are finally put as antecedents of the current goal.

```
    Goal
        btrue & zz=e2 & zz: {e1,e2,e3,e4,e5} => not(tt = e5)
```

5.5 Back

Backstep on a proof

Syntax

ba
ba(n)

with:

- n equals **Node** to go back to the previous node in the proof tree .
- n takes a numerical value when we know how many commands we want to backstep.

Use

This command induces a backstep in the proof: the effect of the last command is canceled. The current goal, the hypotheses and the command line go back to their previous state. The **ba** command may not produce a change, if we are at the beginning of a command line (no command performed).
This command has an effect on the proof obligation status: if this has just been proved and we apply the **ba** command, we will find ourselves just before the last command which finishes the demonstration and the proof obligation becomes unproved.

The Back command can make the number of backsteps specified when the command was called.

The **Node** parameter backsteps the state of the proof obligation to the previous level of indentation in the command line. This parameter gives a very fast Back because no command is added.

ba does not apply to a **ff** command(see chapter 5.16 page 61).

Example 1

Given the proof obligation whose command line is as follows:

```
Force(0) &
  ar(test.1,Fwd) &
    dd &
      dd &
        ar(test.2,Once) &
          pr &
          pr &
  Next
```

If we apply the command line **ba**, we check that the last (**pr**) command has been suppressed:

```
Force(0) &
  ar(test.1,Fwd) &
    dd &
      dd &
        ar(test.2,Once) &
          pr &
          Next
```

By beginning the same operation again, we check that the last (**pr**) command has been suppressed:

```
Force(0) &
  ar(test.1,Fwd) &
    dd &
      dd &
        ar(test.2,Once) &
          Next
```

We can thus take several backsteps through a single command:

```
PRI> ba(3)
```

In this case, the command line becomes:

```
Force(0) &
  ar(test.1,Fwd) &
    Next
```

Example 2

If the command line is:

```
ah(not(xx = e1)) &
  dc(zz,ENUM) &
    pr &
    pr &
      ah(ww = 5) &
        pr &
        pr &
    Next
```

After application of the **ba(Node)** command, the command line becomes:

```
ah(not(xx = e1)) &
  Next
```

5.6 Loop

Application of interactive commands during a proof division

Syntax

bb(f)

with:

- f is a sequence of commands of form:
  `c1 & c2 & .. & cn & ll((d1 & d2 & ... & dm),...,(z1 & z2 & ... & zp))`

Use

This command enables the application of a sequence of commands on all the branches of a part of the proof tree, when there is division of the proof. Thus the user does not need to enter and execute the same sequence of commands several times. If the command sequence does not succeed in proving one of the branches, the loop stops.

The **bb** command also applies when there is no proof division but the advantage of **bb** is in this case much reduced.

The presence of **ll** inside of the loop enables the application of interactive commands, when a proof branch cannot be automatically proved. In this case, the commands `(d1 & d2 & ... & dm)` will be tried. If the proof of the branch succeeds, we change branches and re-apply `c1 & c2 & .. & cn`. If the proof does not succeed, the following commands will be tried. This process is iterated until the proof of the branch succeeds or the commands `z1 & z2 & ... & zp` have been tried unsuccessfully.

Example 1

Given the following proof obligation:

```
ENS = {e1, e2 e3, e4, e5} &
ENS: FIN(NATURAL*{ENS.enum}) &
not (ENS={}) &
xx: ENS &
not(xx = e5)
=>
xx = e1 or xx = e2 or xx = e3 or xx = e4
```

We can perform one proof by cases:

```
dc(xx, ENS)
```

To avoid entering 5 times the **pr** command, we can enter:

```
bb(pr)
```

We then obtain the messages:

```
Starting Prover Call
Starting Prover Call
Starting Prover Call
Starting Prover Call
Starting Prover Call
```

and the proof obligation is demonstrated.

The Commands window (Executed / Next) now contains:

```
dc(xx,ENS) &
  fc(1) &
    pr &
  fc(1) &
    pr &
  fc(1) &
    pr &
  fc(1) &
    pr &
  fc(1) &
    pr &
Next
```

Each fc(1) [3] indicates the start of the execution of the **pr** command (parameter of the loop command). The number (here 1) indicates the number of nested loops.

We find the 5 cases of the proof by cases (`fc(1)`).

[3] Flag Command

Example 2

Given the following proof obligation:

```
ENS = {e1, e2 e3, e4, e5} &
ENS: FIN(NATURAL*{ENS.enum}) &
not(ENS = {}) &
xx: ENS &
not(xx = e5)
=>
xx = e1 or xx = e2 or xx = e3 or xx = e4
```

we can perform a proof by cases:

```
dc(xx, ENS)
```

If we execute the command:

```
bb(ch & bb(dd & bb(pr)))
```

we obtain:

```
Starting creating hyp
Starting Deduction
Starting Prover Call
Starting creating hyp
Starting Deduction
Starting Prover Call
Starting creating hyp
Starting Deduction
Starting Prover Call
Starting creating hyp
Starting Deduction
Starting Prover Call
Starting creating hyp
Starting Deduction
Starting Prover Call
```

and the proof obligation is demonstrated.

The Commands window (Executed / Next) now contains:

```
dc(xx,ENS) &
  fc(3) &
    ch &
      fc(2) &
        dd &
          fc(1) &
            pr &
  fc(3) &
    ch &
      fc(2) &
        dd &
          fc(1) &
            pr &
  fc(3) &
    ch &
      fc(2) &
        dd &
          fc(1) &
            pr &
  fc(3) &
    ch &
      fc(2) &
        dd &
          fc(1) &
            pr &
  fc(3) &
    ch &
      fc(2) &
        dd &
          fc(1) &
            pr &
Next
```

Each fc() [4] indicates the start of the execution of a sequence of commands (parameter of the loop command). The number (1, 2 ou 3) indicates the number of nested loops.

We find the 5 cases of the proof by case (`fc(3)`).

After saving, the command line saved is:

```
dc(xx,ENS) &
bb(ch & bb(dd & bb(pr & ll(?)) & ll(?)) & ll(?))
```

The command **ll()** has been added to the entries saved and contains any interactive commands added by the user, during the proof and inside the loops. In this case, no commands have been added and we thus obtain **ll(?)**.

This command line can in fact be entered directly by the user: this leads to the same

[4] Flag Command

result. However, the command **ll()** must be at the end of the command line contained in **bb()**.

For example, the following command is incorrect:

```
bb(ch & ll(dd) & pr)
```

and gives when executed:

```
Starting creating hyp
Unknown command ll(dd)
Starting Prover Call
Starting creating hyp
Unknown command ll(dd)
Starting Prover Call
Starting creating hyp
Unknown command ll(dd)
Starting Prover Call
Starting creating hyp
Unknown command ll(dd)
Starting Prover Call
Starting creating hyp
Unknown command ll(dd)
Starting Prover Call
```

However, the following command line is correct:

```
bb(ch & pr & ll(dd))
```

and gives when executed:

```
Starting creating hyp
Starting Prover Call
Starting creating hyp
Starting Prover Call
Starting creating hyp
Starting Prover Call
Starting creating hyp
Starting Prover Call
Starting creating hyp
Starting Prover Call
```

We are aware that the **dd** command contained in **ll** has not been executed.

5.7 CurrentGoal

Display the current goal

Syntax

cg

Use

This command displays the current proof goal. It thus corresponds to the "Current Goal" window of the Motif interface.

Example

Given the following goal to demonstrate:

```
foo = plus(nn)
```

The user then searches the hypotheses refering to foo, using **sh(foo)**:

```
...
foo: INTEGER &
foo: dom(ff) &
...
```

To display the current goal, the user has two possibilities: use the GUI or use the **CurrentGoal**. It makes the following information be displayed:

```
Goal
   foo = plus(nn)
```

5.8 CreateHyp

Creation of hypotheses

Syntax

ch

Use

This mechanism initiates the creation of hypotheses in relation to the form of the goal. After its application, the goal is presented in the form of an implication, the generated hypotheses are added as antecedents of the implication.

Example

Let us consider a proof obligation whose current goal is:

```
        foo = plus(nn)
```

The application of the command **ch**

```
PRI> ch
```

gives the following goal:

```
        btrue &
        plus(nn): ran(plus) &
        plus(nn): INTEGER +-> INTEGER
        =>
        foo  = plus(nn)
```

The antecedent of this implication contains the hypotheses generated by the command. The user can then process these hypotheses as he wants to.

5.9 Contradiction

Proof attempt by contradiction

Syntax

ct

Use

This command enables to attempt a proof by contradiction.
If the current goal is G, it is then transformed into:
$\neg G \Rightarrow bfalse$
It is then necessary that the hypotheses, completed by $\neg G$, enable the generation of bfalse. In this case, we obtain:
$bfalse \Rightarrow bfalse$
which is true.

Proof by contradiction can be used especially:

- if the goal is in the form $\neg P$
- if there are several contradictory hypotheses

Example

Let us consider the following proof obligation:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    tt: ENS &
    uu: ENS &
    not(uu = tt) &
    uu: {e1,e2,e3,e4} => tt = e5 &
    uu = e5 => tt = e1
Goal
    not(e2 = e5)
```

We attempt a proof by contradiction, given the form of the goal.

```
PRI> ct
Starting Contradiction
```

$\neg\neg(e2 = e5)$ is simplified in $e2 = e5$ then becomes a hypothesis. The goal becomes $bfalse$.

```
    New Hypothesis since last command
        e2 = e5
    Goal
        bfalse
```

The automatic prover is then called.

```
PRI> pr
Starting Prover Call
```

the command line then becomes:

```
        Force(0) &
          dd &
            ct &
              pr &
          Next
```

5.10 Special Contradiction

Attempt to prove by contradiction without automatic raising of hypotheses

Syntax

cts

Use

This command enables to attempt a proof by contradiction, without automatic raising of hypotheses.
If the current goal is G, it is then transformed into:
$\neg G \Rightarrow bfalse$
The hypotheses, completed by $\neg G$, must be able to generate bfalse.
This command is identical to **ct**, except that the generated hypotheses are not automatically raised.

Example

Let us consider the following proof obligation:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    tt: ENS &
    uu: ENS &
    not(uu = tt) &
    uu: {e1,e2,e3,e4} => tt = e5 &
    uu = e5 => tt = e1
Goal
    not(e2 = e5)
```

We attempt a proof by contradiction, given the form of the goal.

```
PRI> cts
Starting Contradiction
```

$\neg\neg e2 = e5$ is simplified in $e2 = e5$. The goal becomes:

```
Goal
    e2=e5 => bfalse
```

5.11 Do cases

PROOF ATTEMPT BY CASE

Syntax

dc(f)
dc(v,E)

with:

- f is a predicate well-typed according to the context of the present hypotheses,
- v is a variable and E is the name of a set in extension (maximum 50 elements)

Use

This command enables the starting up of a proof by cases.
If the goal to be proved is G and the parameter supplied to dc is X:

- if X is a predicate
  the proof is decomposed into 2 sub-goals:
  $X \Rightarrow G$
  and
  $\neg X \Rightarrow G$
- if X is in the form v,E
  the goal becomes a conjunction of n goals corresponding to all the possible values of v in E. If the hypothesis $v \in E$ does not exist as such, $(v \in E)$ is added, by the prover, to this conjunction so that the proof can begin on this preliminary goal It is stressed that the user has to check that the predicate $v \in E$ is well-typed (see chapter 3.2 page 7) and well-defined (see chapter 3.3 page 8).

  The proof performed depends on the form of the expression v,E:

  – E is in the form $\{e_1\} \cup \ldots \cup \{e_n\}$
    the proof of G is then replaced by:
    $(v = e_1 \Rightarrow G) \wedge \ldots \wedge (v = e_n \Rightarrow G)$
  – E is in the form $(\{e_1\} \cup \ldots \cup \{e_n\}) \times \{f\}$
    the proof of G is then replaced by:
    $(v = (e_1 \mapsto f) \Rightarrow G) \wedge \ldots \wedge (v = (e_n \mapsto f) \Rightarrow G)$
  – E is in the form $(\{e_1\} \cup \ldots \cup \{e_n\}) \times F$
    where F is not in the form $\{f\}$
    The proof of G is then replaced by:
    $(v = (\{e_1\} \times F) \Rightarrow G) \wedge \ldots \wedge (v = (\{e_n\} \times F) \Rightarrow G)$

Example 1

Let us consider the following situation:

```
Hypothesis
    xx: 1..10 &
    yy: 1..10 &
    zz: 1..100 &
    xx: 1..5 => yy = 10 &
    xx: 6..10 => yy = 1 &
    yy*xx<=100 & xx*yy<=100 or (yy*yy<=100 & yy*yy<=100)
Goal
   xx+yy-1: 1..100
```

Due to the presence of the two hypotheses $xx : 1..5 => yy = 10$ and $xx : 6..10 => yy = 1$, the user decides to start a proof by case , for the predicate $xx : 1..5$.

```
PRI> dc(xx: 1..5)
```

The first goal processed is thus:

```
Goal
   xx: 1..5 => xx+yy-1: 1..100
```

The prover will therefore attempt to prove the current goal, first under the hypothesis $xx : 1..5$. To do this, the user applies the command **pr** (see chapter 5.34 page 99).

```
PRI> pr
```

The goal is proved by the automatic prover. The other case to be processed is thus $not(xx : 1..5)$.

```
Goal
    not(xx: 1..5) => xx+yy-1: 1..100
```

The user starts the automatic prover again.

```
PRI> pr
```

The PO is proved and its command line is:

```
    Force(0) &
      dd &
        dc(xx: 1..5) &
          pr &
          pr &
      Next
```

Example 2

We will now consider proof by case, for an enumerated set.

Given the following situation:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    ENS: FIN(NATURAL*{ENS.enum}) &
    not(ENS = {}) &
    xx: ENS &
    xx: {e1,e2,e3,e4}
Goal
    not(xx = e5)
```

The user starts a proof by cases for every value of `xx`, which belongs to enumerated set `ENS`.

```
PRI> dc(xx,ENS)
Do Cases on Enumerated: {5}\/{4}\/{3}\/{2}\/{1}
```

the interactive prover will thus launch 5 successive proofs:

```
Goal
    xx = e5 => not(xx = e5)

Goal
    xx = e4 => not(xx = e5)

Goal
    xx = e3 => not(xx = e5)

Goal
    xx = e2 => not(xx = e5)

Goal
    xx = e1 => not(xx = e5)
```

5.12 Special do cases

Proof by cases attempt on a disjunctive formula

Syntaxe

dcs(A or B)

with:

- (A or B) is a disjunctive predicate

Use

This command enables the proof by cases starting up.
If the goal to prove is B and the parameter supplied to dcs is X:

- If X is a disjunctive predicate of n disjunctions, the goal becomes a conjunction of n goals. Each of these goals is the implication of B by the conjunction of one of the disjunctive term of the X predicate and of the other terms negation. If the X predicate is not in hypothesis, the first goal to demonstrate is this predicate.

  For example if X is $A \vee not(B) \vee C$ and the goal is $Goal$, the new goals become:

  - $A \wedge B \wedge not(C) \Rightarrow Goal$
  - $not(A) \wedge not(B) \wedge not(C) \Rightarrow Goal$
  - $not(A) \wedge B \wedge C \Rightarrow Goal$

- if X is not a disjunctive predicate, the **dcs** command acts as the **dc** command.

Example

Let us consider the proof by cases, in the case of a disjunctive predicate.

Given the following situation:

```
Hypothesis
    xx: INTEGER &
    0<=xx &
    xx = 1 or xx = 10 or xx = 4 &
    btrue &
    0<=20 &
    20: INTEGER
Goal
    xx<=20
```

The user launches a proof by cases to fully use the hypothesis
`xx = 1 or xx = 10 or xx = 4`.

```
PRI> dcs(xx = 1 or xx = 10 or xx = 4)
Starting Do Cases
```

As the disjunctive predicate comes from the hypothesis, it is not necessary to check it. The prover starts three successives proofs:

```
Goal
    xx = 4 &
    not(xx = 10) &
    not(xx = 1)
    =>
    xx<=20

Goal
    xx = 10 &
    not(xx = 4) &
    not(xx = 1)
    =>
    xx<=20

Goal
    xx = 1 &
    not(xx = 4) &
    not(xx = 10)
    =>
    xx<=20
```

Each of these goals is easily discharged by the **pr** command (see chapter 5.34 page 99).

5.13 Deduction

Direct Deduction

Syntax

dd
dd(i)

with:

- i equals 0, 1, 2 ou 3

Use

This command enables to perform a direct deduction: if the current goal is in the form $P \Rightarrow Q$, the prover then attempts the Q proof under the P hypothesis.

The P hypothesis is raised in the hypotheses stack, without using the automatic prover (especially, the simplifications are not performed) and the current goal becomes Q.

It can be interesting, in certain cases, to use **dd** since the automatic prover may normalise, and not according to the user's wishes, a hypothesis added by the **ah** command (see chapter 5.2 page 22). **dd** is thus sometimes used after **ah** to raise a new hypothesis as it is.

The argument **dd(i)** enables fine parameterisation of the processing of the raising hypotheses. The i parameter represents the proof force that will be used temporarily when the hypotheses are raised.

For example, **dd(1)** corresponds to hypotheses raised in force 1.

Example

Given the following situation:

```
Hypothesis
    xx: 1..10 &
    yy: 1..10 &
    zz: 1..100 &
Goal
    xx+yy-1: 1..100
```

The user wishes to add the hypothesis $xx + yy : 2..20$.

```
PRI> ah(xx+yy: 2..20)
Starting Add Hypothesis
```

the hypothesis to be added must first be proved.

```
    Goal
        xx+yy: 2..20
```

the user starts the automatic proof:

```
PRI> pr
Starting Prover Call
```

the hypothesis is proved. the current goal thus becomes:

```
    Goal
        xx+yy: 2..20  => xx+yy-1: 1..100
```

The command **dd** then enables the hypothesis $xx+yy : 2..20$ to be raised in the hypotheses stack.

```
PRI> dd
Starting Deduction
```

The hypothesis has indeed been raised and the current goal is again $xx + yy - 1 : 1..100$.

```
    New Hypothesis since last command
        xx+yy: 2..20
    Goal
        xx+yy-1: 1..100
```

5.14 Display Term

Display formula terms

Syntax

dt
dt(f)

with:

- f is a list, that may possibly be just of one element, of terms with t.i form and separated by a comma.

Use

This command can be used after a logical analysis of formula **la** (see chapter 5.24 page 74) and enables to display the value of the various terms of the analysed formula.

Example

Given the following goal:

```
        "‘REVERSE_RGE preconditions in this component’" &
        rng: minrge..maxrge &
        jj: 0..maxidx &
        ii: 0..maxidx &
        "‘Local hypotheses’" &
        kk$0: INTEGER &
        0<=kk$0 &
        ll$0: INTEGER &
        0<=ll$0 &
        ii<=kk$0 &
        ii<=jj => kk$0<=ll$0+1 &
        ll$0<=jj &
        kk$0+ll$0 = ii+jj &
        arr_rge$2 = arr_rge$1<+{rng|->(arr_rge$1(rng)<+%xx.(xx: ii..jj
& (xx+1<=kk$0 or ll$0+1<=xx) | arr_rge$1(rng)(ii+jj-xx)))} &
kk$0+1<=ll$0 &
        "‘Check preconditions of called operation, or While loop
        construction, or Assert predicates’"
        =>
        ii<=kk$0+1
```

The formula logical analyser, performed by **la** command (see chapter 5.24 page 74), breaks down the goal formula in:

```
PRI > la(2)
Parsing formula
"‘REVERSE_RGE preconditions in this component’" &
t.1: t.2 &
t.3: t.4 &
t.5: t.4 &
"‘Local hypotheses’" &
t.6: t.7 &
t.8 <= t.6 &
t.9: t.7 &
t.8 <= t.9 &
t.5 <= t.6 &
(t.10=>t.11) &
t.9 <= t.3 &
t.12 = t.13 &
t.14 = t.15 &
t.16 <= t.9 &
"‘Check preconditions of called operation, or While loop construction,
or Assert predicates’"
=>
t.5 <= t.16
End of analysis
```

Terms t.1 to t.16 can then be displayed by the **dt** command:

```
PRI > dt
t.1 is put for rng
t.2 is put for minrge..maxrge
t.3 is put for jj
t.4 is put for 0..maxidx
t.5 is put for ii
t.6 is put for kk$0
t.7 is put for INTEGER
t.8 is put for 0
t.9 is put for ll$0
t.10 is put for ii<=jj
t.11 is put for kk$0<=ll$0+1
t.12 is put for kk$0+ll$0
t.13 is put for ii+jj
t.14 is put for arr_rge$2
t.15 is put for arr_rge$1<+{rng|->(arr_rge$1(rng)<+%xx.(xx: ii..jj &
(xx+1<=kk$0 or ll$0+1<=xx) | arr_rge$1(rng)(ii+jj-xx)))}
t.16 is put for kk$0+1
```

5.15 Use equality in hypothesis

Rewriting according to the equality in hypothesis

Syntax

eh(a, A, f)

with:

- the hypothesis a=A or the hypothesis A=a must be in the stack,
- we can also use the keyword _*h* either for a, either for A
- f can be
  - either the **Goal** keyword
  - either the **AllHyp** keyword
  - either the **Hyp(h)** keyword with h corresponding to the selected hypothesis

Use

This command enables us to replace a by A, either in the current goal , or in all the hypotheses, or in the h hypothesis.

Keyword _*h* enables us to use an equality of which we only know one member (right or left). In this case, the last satisfactory equality in hypothesis, is used.

If f = **Hyp(h)**, the goal becomes:
$H \Rightarrow G$
where H is obtained by replacing a by A in the hypothesis h, if it exists.

If f = **AllHyp**, the goal becomes:
$H \Rightarrow G$
where H is obtained by replacing a by A in all the hypotheses.

If f = **Goal**, the goal becomes:
G'
where G' is obtained by replacing a by A in all the hypotheses.

A proof often fails because an equality has not been used. The automatic provers have to take precautions with rewriting using equalities; indeed, this can generate loops (see chapter 3.9 page 14). However, the interactive prover can perform such rewritings, which are applied from time to time and under user control.

When a goal is rewritten, the interactive command may contradict a normalisation performed by the automatic prover; if we restart in automatic mode, this will immediately redo the inverse transform.

Nonetheless, the command can be useful if the user uses other interactive commands on the rewritten goal, before calling the automatic prover.

Example 1

Given the following situation:

```
    Hypothesis
        ENS = {e1,e2,e3,e4,e5} &
        ENS: FIN(NATURAL*{ENS.enum}) &
        not(ENS = {}) &
        tt: ENS &
        zz: ENS &
        not(zz = tt) &
        #kk.(kk: ENS & not(kk = zz) & not(kk = tt)) &
        zz: {e1,e2,e3,e4} => tt = e5 &
        zz = e5 => tt = e1 &
        zz = e5 or zz = e1 &
        uu = zz &
        !vv.(vv: ENS & (not(zz = vv) or not(tt = vv)) => zz = vv)
    Goal
        uu = e5 => zz = e1
```

It is possible to substitute uu by zz in the goal.

```
PRI> eh(uu,zz,Goal)
Starting use Equality in Hypothesis
```

the goal becomes:

```
    Goal
        zz = e5 => zz = e1
```

It is possible to perform the substitution for a hypothesis

```
PRI> eh(zz,uu,Hyp(zz = e5 or zz = e1))
Starting use Equality in Hypothesis
```

The goal becomes:

```
    Goal
        uu = e5 or uu = e1 =>  (zz = e5 => zz = e1)
```

It is possible to perform the substitution for all the hypotheses.

```
PRI> eh(zz,uu,AllHyp)
Starting use Equality in Hypothesis
```

All the new hypotheses appear as antecedent of the current goal:

```
Goal
    uu: ENS & not(uu = tt) &
    #kk.(kk: ENS & not(kk = uu) & not(kk = tt)) &
    (uu: {e1,e2,e3,e4} => tt = e5) &
    (uu = e5 => tt = e1) &
    (uu = e5 or uu = e1) &
    !vv.(vv: ENS & (not(uu = vv) or not(tt = vv)) => uu = vv)
    =>
    (uu = e5 or uu = e1 => ( zz = e5  => zz = e1))
```

Example 2

Given the following situation:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    ENS: FIN(NATURAL*{ENS.enum}) &
    not(ENS = {}) &
    zz: ENS &
    uu = tt or uu = zz &
    tt: {e1,e2,e3,e4} => zz = e5 &
    zz = e5 => tt: {e1,e2,e3,e4} &
    tt = e5 => zz = e1 &
    zz = e1 => tt = e5 &
    zz = e5
Goal
    e2 = e5 or e2 = zz
```

If the user wants to use an equality with $e5$ as right member, without taking care of the left member:

```
PRI> eh(_h,e5,Goal)
Starting use Equality in Hypothesis
```

using the equality $zz = e5$, the goal becomes:

```
Goal
    e2 = e5 or e2 = e5
```

If the user wishes to use the last equality whose left member is zz:

```
PRI> eh(zz,_h,Goal)
Starting use Equality in Hypothesis
```

the goal becomes:

```
Goal
    e2 = e5 or e2 = e5
```

The goal is indeed transformed, using the hypothesis $zz = e5$.

5.16 Force

Change of the prover current force

Syntax

ff(n)

with:

- n = Fast, 0, 1, 2 ou 3

Use

This command enables the user to change the current prover force and to replay the whole existing command line (see chapter 3.7 page 12) with a new force. If the force requested is equal to the current force, there is no change.

Force `Fast` gives maximum enhancement to processing speed. A proof in force 0 requires an average of 10 seconds per proof obligation. For certain complex PO, the proof can take several minutes. Higher forces use mechanisms consuming more CPU and memory resources. The processing time for a PO can become very long.

Given that around 90 % of PO proved are in force 0, it is advised to attempt the proof of a PO firstly in force 0.

for more details on proof forces, see "the choice of higher force" chapter in *Interactive prover User Manual* .

the **ba** command (see chapter 5.5 page 33), which backsteps the proof does not apply on the **ff** command.

Example

Given the following situation:

```
Hypothesis
    xx: 1..10 &
    yy: 1..10 &
    zz: 1..100 &
    xx+yy: 2..20
Goal
    xx+yy-1: 1..100
```

for which an interactive proof task exists in force(0):

```
Force(0) &
  ah(xx+yy: 2..20) &
    pr &
    dd &
      Next
```

the user changes the current prover force:

```
PRI> ff(1)
Switching to Force 1
```

The command line already executed will be replayed with the new ($ah(xx + yy \in 2..20)$ & pr & dd) force .

```
Starting Add Hypothesis
Starting Prover Call
dd not applicable: Goal is not p => q
```

Replay using different force is difficult since the proof path is a priori different. We particularly notice that the command **dd** does not apply.

The new command line is:

```
Force(0) &
  ah(xx+yy: 2..20) &
    pr &
    Next
```

5.17 False hypothesis

Attempt to prove using contradictory hypotheses

Syntax

fh(h)

with:

- h is a hypothesis

Use

this command enables to perform a demonstration, by proving that a hypothesis is contradictory to the others.

If the lemma to be demonstrated is
G under the hypotheses h_1 , ... , h_n
and the user suspects that one of the hypotheses h_i is contradictory to the others, we can demonstrate the lemma by demonstrating:
$\neg h_i$ under the hypotheses h_1, ... ,h_n

Example

Given the following situation:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    e2 = e5
Goal
    e5 = e1
```

It is clear that the hypothesis $e2 = e5$ is contradictory. By applying the command:

```
PRI> fh(e2=e5)
Starting False Hypothesis
```

the current goal becomes:

```
Goal
    not(e2 = e5)
```

by calling the automatic prover the goal can then be discharged.

```
PRI> pr
Starting Prover Call
```

5.18 Goto

Positioning on a PO

Syntax

go(f.n)

with:

- f is the name of an operation (or clause) of the current component
- n is the number of an existing PO for the operation (or clause) concerned

Use

This command enables the user to go to the beginning of the proof of the PO number n, for the f operation (or clause).
A confirmation to save will be requested from the user if:

- The previous command line and the new command line enable to prove the proof obligation,
- The previous command line and the new command line do not enable to prove the proof obligation.

Example

The user wishes to go to the PO *Initialisation*.1. A proof task has been performed and not saved.

```
PRI> go(Initialisation.1)
```

As the previous and new command line do not enable to prove the proof obligation, a confirmation to save is requested:

```
Last PO does not have a saved demo. Your new demo does not prove.
Do you want to save the new demo (will replace the old one)?
  Answer No to continue without saving (any other word to save):
```

The user does not wish to save his proof work:

```
  No
```

The command line is not saved and the current proof obligation becomes *Initialisation*.1.

```
No saving
Current PO : Initialisation.1
```

5.19 Goto with reset

Positioning on a PO in force 0

Syntax

gr(f.n)

with:

- f is the name of an operation (or clause) of the current component
- n is the number of an existing PO, for the operation concerned

Use

This command enables to move to PO f.n while it reduces the proof force to 0 and keeps the saved proof commands unchanged.

From force 1, hypotheses of a proof obligation are all simplified during loading. This command will be useful if a proof obligation is such that raising its hypotheses makes the prover loop for a particular force (1,2, or 3).

Example

Given the proof obligation whose command line is:

```
        Force(3) &
          Next
```

If the **gr** command is carried out

```
PRI> gr(Calculus.1)
```

the command line will be put back in force 0:

```
        Force(0) &
          Next
```

5.20 Global situation

Proof status for the current component PO

Syntax

gs
gs (k)
gs (o,e)
gs (o,e,f)

with

- k is either an operation (or clause) name of the current component or the keyword _all indicating all the component operations and clauses, or one of the following expressions: Unproved, Proved and Patt(g) with g a formula.
- o is a component clause name, or the keyword _all.
- e is a proof obligations status: Proved, Unproved or _all (if not specified).
- f is a formula -in parenthesis- or the keyword _all. f enables to filter the proof obligations according to the goal form that must match f. If f is the keyword _all, there is no filtering.

Use

This command enables to select and display the proof status (proved, unproved) and the goal (without hypotheses) of proof obligations.

The o argument enables to select the proof obligations corresponding to the specified clause (if it is given), e identifies the proof obligations status, and at last, f their goal form.

The k argument enables to select all the proof obligations of the current component that are in the specified proof status, if k equals Proved or Unproved. If k has the Patt(g) form, all the proof obligations of the current component whose goal matches g are selected and if k is a clause name of the current component, all the proof obligations corresponding to that clause are selected.

By default, gs, gs(o,e) respectively represents gs(_all,_all,_all) and gs(o,e,_all).

If k equals Proved or Unproved, gs(k) means gs(_all,k,_all), if k has the Patt(g) form, gs(k) points out gs(_all,_all,(g)) and in all other cases, gs(k) means gs(k,_all,_all).

Example 1

For the following component, we can remark the presence of the (*Initialisation* clause and of the *op0*) operation. The goal form of each proof obligation is given at the end of the line.

```
PRI> gs
State of all PO
    Initialisation
        PO1 Unproved xx = 3
        PO2 Unproved {0|->TRUE}: NAT +-> BOOL
        PO3 Unproved xx+1: INTEGER
        PO4 Proved 0<=xx+1
        PO5 Unproved xx+1<=2147483647
    op0
        PO1 Unproved zz+2: INTEGER
        PO2 Unproved 0<=zz+2
        PO3 Proved zz+2<=2147483647
        PO4 Unproved zz+2 = 3
End
```

Example 2

Now let us select the proof obligations of operation op0:

```
PRI> gs(op0)
State of All PO of operation op0
        PO1 Unproved    zz+2: INTEGER
        PO2 Unproved    0<=zz+2
        PO3 Proved      zz+2<=2147483647
        PO4 Unproved    zz+2 = 3
End
```

Example 3

We choose to display only the unproved proof obligations of operation op0:

```
PRI> gs(op0,Unproved)
Unproved PO of operation op0
        PO1 Unproved    zz+2: INTEGER
        PO2 Unproved    0<=zz+2
        PO4 Unproved    zz+2 = 3
End
```

Example 4

We are looking for the unproved proof obligations of the Initialisation clause whose goal matches $\{x\} : y \ \nrightarrow \ BOOL$:

```
PRI> gs(Initialisation,Unproved,({x}: y +-> BOOL))

Unproved PO of operation Initialisation
Matching with {x}: y +-> BOOL
        PO2 Unproved    {0|->TRUE}: NAT +-> BOOL
End
```

Example 5

We are looking for proved proof obligations among all the proof obligations of current component:

```
PRI> gs(Proved)

All Proved PO
    Initialisation
        PO4 Proved 0<=xx+1
    op0
        PO3 Proved zz+2<=2147483647
End
```

Example 6

We are looking for all the proof obligations of the current component whose goal matches the $x = y$ formula:

```
PRI> gs(Patt(x = y))

State of all PO
Matching with x = y
    Initialisation
        PO1 Unproved xx = 3
    op0
        PO4 Unproved zz+2 = 3
End
```

5.21 Graphical Trace

Selection of graphical trace mode

Syntax

gt(f)

with:

- f equals **on** or **off**

Use

This commande enables to switch on and off the graphical trace mode of the interactive prover. Selecting graphical trace mode is only possible when the current proof obligation demonstration has not been started yet. If it is not the case, a reset (**re** command) of the demonstration has to be done and command **gt(on)** to be typed again. After a moment, the graphical visualization tool DaVinci appears. The trace of the carried out commands will then be displayed in this window:

- typed commands appear in red
- intermediate goals in white
- proved goals in green

Example

The user started a proof and wants to switch into graphical trace mode.

```
PRI> gt(on)
Rewind your demo to start displaying the graphical trace
```

The user resets his demonstration, after saving it, using command **sw**. He can then switch to graphical trace mode:

```
PRI> gt(on)
Graphical Trace mode is on
```

When he ends the demonstration, he can disable the graphical trace mode:

```
PRI > gt(off)
Graphical Trace mode is off
```

5.22 Goto without save

Positioning on another proof obligation without saving the current proof

Syntax

gw(f.n)

with:

- f is the name of an operation (or clause)
- n is the number of a proof obligation of the operation (or clause) concerned

Use

This command allows the user to access the f.n proof obligation, discarding the work already made on the current proof obligation.

If f is not an operation or a clause or if n is not a correct number, the command will fail.

Example

The PO Initialisation test.1 has been proved. The user may not want to save the interactive proof that has been made.

The user goes to the PO test.Initialisation.2, without saving previous work.

```
PRI> gw(Initialisation.2)
Skipping without saving to Initialisation.2
Current PO : Initialisation.2
```

5.23 Help

On-line help about command syntax and function

Syntax

help
help(c)

with:
c name of an interactive command

Use

This command displays a description of command c.

If the argument is omitted, it displays the list of all available interactive commands.

Example

The user wants to know how the command **mini prover** `mp` works:

```
help(mp)
Help
mp - Mini Prover
Syntax
 mp
Description
 Call the automatic prover with the current force (see ff), so that no
proof by cases will be performed. This command is equivalent to pr(Red).
See Also
 ff, pr.
End help
```

5.24 Logical Analysis

Logical analysis of a formula

Syntax

la
la(n)
la(f)
la(f | n)

with:

- f is a formula (B expression or predicate)
- n is a positive integer

Use

This command enables to analyse a B formula according to various modes:

- **la**: current goal is analysed with an analysis depth of 1,
- **la(n)**: current goal is analysed with an analysis depth of n,
- **la(f)**: formula f is analysed with a depth of 1,
- **la(f | n)**: formula f is analysed with a depth of n.

When the formula to analyse has a depth greater than the analysis depth, expressions and predicates that cannot be detailed are replaced by terms of form t.i. This notation gives a global display of the formula. The term value is not automatically displayed so that display is not overloaded with complex terms. Command **dt**(see chapter 5.14 page 55) enables to display some or all of theses terms.

Example

Let us consider the following situation:

```
"‘SEARCH_MAX_EQL_RGE preconditions in this component’" &
rng: minrge..maxrge &
jj: 0..maxidx &
ii: 0..maxidx &
ii<=jj &
vv: VALUE &
```

```
      "‘Local hypotheses’" &
        nrr: INTEGER &
        0<=nrr &
        nrr<=2147483647 &
        nbb: BOOL &
        sol = (ii..jj<|arr_rge$1(rng))~[{vv}] &
        not(sol = {}) => nrr = max(sol) &
        nbb = bool(not(sol = {})) &
        not((ii..jj<|arr_rge$1(rng))~[{vv}] = {}) &
        "‘Check operation refinement - ref 4.4, 5.5’" &
        =>
        nrr = max((ii..jj<|arr_rge$1(rng))~[{vv}])
```

Applying the **la(3)** command breaks down the formula as follows:

```
Parsing formula
"‘SEARCH_MAX_EQL_RGE preconditions in this component’" &
rng: t.1..t.2 &
jj: t.3..t.4 &
ii: t.3..t.4 &
ii <= jj &
vv: VALUE &
"‘Local hypotheses’" &
nrr: INTEGER &
0 <= nrr &
nrr <= 2147483647 &
nbb: BOOL &
sol = t.5[t.6] &
(not(t.7)=>t.8 = t.9) &
nbb = bool(t.10) &
not(t.11 = t.12) &
"‘Check operation refinement - ref 4.4, 5.5’"
=>
nrr = max(t.11)
End of analysis
```

5.25 Show literal PO

Display of a proof obligation in its literal form

Syntax

lp(f.n)

with:

- f must be the name of an operation or clause of the current component
- n must be a valid proof obligation number for the f operation (or clause) that has been selected

Use

This command displays the proof obligation which has been selected, in its literal form, that is to say, as it was generated by the proof obligation generator.

In fact, the prover performs operations on the hypotheses and the goal (simplifications, ...). The displayed data therefore do not reflect exactly the proof obligation.

Example

When we move to proof obligation Initialisation.1, expressions within (hypotheses, goal) are normalised.

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    ENS: FIN(NATURAL*{ENS.enum}) &
    not(ENS = {})
Goal
   e1 = e3
   =>
   e1 = e2 or e1 = e3 or e1 = e5
```

The user would like to see proof obligation Initialisation.1 in its literal form:

```
PRI> lp(Initialisation.1)
```

Display of the proof obligation in its literal form shows us how this proof oblligations looked like before normalisation, as generated by the Proof Obligations Generator.

```
Show PO : Initialisation.1
  (1..5)*{ENS}: FIN(NATURAL*{ENS}) &
  not((1..5)*{ENS} = {}) &
  1|->ENS = 3|->ENS
  =>
  1|->ENS = 2|->ENS or 1|->ENS = 3|->ENS or 1|->ENS = 5|->ENS
End PO
```

5.26 ModelChecking

Verification of predicate validity using the comprehensive list of its free variable values

Syntax

mc
mc(l)

with:

- l is a list of no more than four items separated by | and all distinct (order in the list does not affect the command behaviour). These items are of four different functional categories and allow to specify the way the mc command works :
  - Proof management: _Auto or _Step. Keyword _Auto means that proof is handled automatically, that is to say, the automatic prover performs all demonstrations using the different values of variables. Keyword _Step means that the user will use the interactive prover to demonstrate the different cases generated by the prover. If no value is given, then proof is handled automatically by default.
  - User Tactic: Tac(None) or Tac(T). Tac(None) means there is no user-provided tactic. Tac(T), with T being a tactic, means *conversely* that the prover must try tactic T before possibly proceeding with an automatic demonstration. Note that the combination of _Step and Tac(T) (with $T \neq$ None) is not valid. Default is Tac(None).
  - Resolution Variable List: _Variables or L. Keyword _Variables tells that all the goal free variables will be used to perform the model checking. L is a list of variables names separated by commas. Thus L specifies the variable subset to be used to perform the model checking. By default, the _Variables setting is used.
  - Maximum Number of values: n. n is a positive integer representing the maximum number of values a variable can take. 20 by default.

Use

this command allows to perform demonstration by cases of the current goal: it divides the proof according to every value of the resolution variables of list L (the variable domains should be bounded, they are, if it is possible, inferred from the hypotheses).

If the initial goal is H $\Rightarrow$ G, then it is transformed by **mc**($x1$,$x2$,...,xn) into:

```
x1 = a1 & x2 = b1 & ... & xn = v1
=>
([x1, x2, ...,  xn := a1, b1, ..., v1]H
  =>
 [x1, x2, ...,  xn := a1, b1, ..., v1]G)
&
 ...
&
x1 = ai & x2 = bj & ... & xn = vk
=>
([x1, x2, ...,  xn  := ai, bj, ..., vk]H
  =>
 [x1, x2, ...,  xn  := ai, bj, ..., vk]G)
&
 ...
&
x1 = ap & x2 = bq & ... & xn = vr
=>
([x1, x2, ...,  xn := ap, bq, ..., vr]H
  =>
 [x1, x2, ...,  xn  := ap, bq, ..., vr]G)
```

With (ai,bj,...,vk) taking all possible values of the cartesian product of the domains of variation of the variables $x1$, $x2$, ..., xn which are respectively {$a1$, ..., ap}, {$b1$, ..., bq}, ..., {$v1$, ..., vr}.

Domains of variables have been inferred from the available hypotheses involving the resolution variables and have been represented by sets in extension. It should be noted that the domain inferer does not perform constraint resolution between variables and it is limited by the variables domain size: for instance, **ModelChecking** will fail if it is applied to a variable of the INT type whose domain cannot be further constrained by any other available hypothesis.

Hypotheses used by the domain inferer are typically predicates of membership to predefined enumerated sets like BOOL or to enumerated sets defined in the SET clause, predicates of membership to interval or set of integers given in extension, equalities or inequalities between variables and values, etc.

Two options are availables:

- The user can intent to prove interactively each case, by specifying the _Step parameter.
- The automatic prover, possibly assisted by a proof tactic (by Tac(T)), is applied on every case whithout the user interfering, by specifying _Auto. Note that if the prover fails in an attempt to prove one of the sub-goals, the command will fail and the user will get back to the current goal : he is not enabled to prove interactively the unproved subgoals.

Examples

Let us consider the proof obligation: bool(xx = 1) = yy, with xx: {0,2} and yy = FALSE among the hypotheses.

The command **mc**(_Step|xx,yy) makes the goal transform into:

```
Hypothesis
  ...
  xx: {0,2}
  yy = FALSE
Goal
  xx = 2 & yy = FALSE
   =>
  bool(2=1) = FALSE
```

The user proves it using **pr** and the prover generates the second case:

```
Hypothesis
  ...
  xx: {0,2}
  yy = FALSE
Goal
  xx = 0 & yy = FALSE
   =>
  bool(0=1) = FALSE
```

The command **mc**(_Auto) proves the two cases in a totally automatic way:

```
    Starting Model Checking in Automatic mode
    Case xx=2 & yy=FALSE proved
    Case xx=0 & yy=FALSE proved
    Proved by Model Checking
```

Finally, the command **mc**(xx|_Step|Tac(None)|3) would have generated the following goal in the first place:

```
Hypothesis
  ...
  xx: {0,2}
  yy = FALSE
Goal
  xx = 2
   =>
  bool(2=1) = yy
```

then, after using the command **pr**:

```
Hypothesis
  ...
  xx: {0,2}
  yy = FALSE
Goal
  xx = 0
   =>
  bool(0=1) = yy
```

To finish with, the user may wish to apply the command **ModelChecking** on a variable whose domain is too large. Consider the following goal:

```
Hypothesis
  ...
  xx: -15..25
Goal
      toto(xx) = MTP
```

Using **mc**(_Step | 22 | xx), where we specified 22 as the maximum variable values, produces the display of the following message:

```
Failure in Model Checking
```

The domain of the variable contains more than 22 elements indeed.

5.27 Modus ponens in hypothesis

Application of modus ponens

Syntax

mh(H)

with:

- H must be a hypothesis of the form $P \Rightarrow Q$

Use

This command allows the direct use of a hypothesis of the form $P \Rightarrow Q$. If $P \Rightarrow Q$ and P are hypotheses and G is the current goal, then the goal becomes $Q \Rightarrow G$. The command **mh** allows to use hypothesis $P \Rightarrow Q$ without involving the prover, that is to say without simplifying the generated hypothesis Q. In fact, the automatic prover applies the modus ponens systematically on each $P \Rightarrow Q$ and P hypothesis couple that are present in the hypothesis stack.

If $P \Rightarrow Q$ or P are not hypotheses, the command isn't applied.

Example

The following situation has been obtained directly:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    zz = e5 => tt = e1 &
    zz = e5
Goal
    not(e2 = e5)
```

The user wants to use the hypothesis $zz = e5 => tt = e1$ in order to generate the hypothesis $tt = e1$. He knows that hypothesis $zz = e5$ exists. The **mh** command is applied normally

```
PRI> mh(zz = e5 => tt = e1)
Starting Modus Ponens on Hypothesis
```

and the goal becomes:

```
Goal
    tt = e1 =>  not(e2 = e5)
```

The **pr** or **dd** command allow to raise this hypothesis in the stack.

5.28 Mono Lemma Prover

Call of mono lemma prover

Syntax

ml
ml(t)
ml(rp.n)
ml(rp.n|t)
ml(rp(f))
ml(rp(f)|t)
ml(ff(l))
ml(ff(l)|t)
ml(ff(l)|rp.n)
ml(ff(l)|rp.n|t)
ml(ff(l)|rp(f))
ml(ff(l)|rp(f)|t)

with:

- l : list of proof forces separated by semicolons. A force equals to one of the following values: Fast, 0, 1, 2 or 3. If ff(l) is not specified, the current force is used.
- t : mono lemma prover time-out (in seconds). If not provided, the mono lemma prover will stop after 60 seconds in interactive mode.
- rp.n indicates that the mono lemma prover is used on reduced hypotheses.
  - **rp** is a keyword.
  - n is an integer greater or equal to zero that gives the level of hypotheses taken in account. If n equals zero, only the goal (without the hypotheses) is processed by the mono lemma prover.
- rp(f) indicates that the mono lemma prover is applied on the hypotheses selected by formula f (whose form must be a or f+a, a being a keyword). Available keywords for a are:
  - **inv**: component invariant
  - **sees**: assertions and invariants of used and seen machines
  - **loc**: local hypotheses
  - **typ**: type predicates of concrete variables
  - **abs**: assertions and invariants of previous components
  - **used**: constraints of used machines
  - **inc**: properties of included, imported and extended machines
  - **prp**: properties and valuations of the component

Use

This command enables to use the mono lemma prover on the current goal. The mono lemma prover works just like the automatic prover except that it processes hypotheses differently.

This function can be used in the three following modes:

1. First mode applies the mono lemma prover to the goal and all the current hypotheses.
2. Second mode applies the mono lemma prover to the goal and the hypotheses of the reduced proof obligation. The selected hypotheses are the same as those returned by **rp**(see chapter 5.38 page 108).
3. Third mode applies the mono lemma prover to the goal and to the hypotheses that meet the criteria given by parameter f. For instance, **ml(rp(sees+loc+inv))** enables to proceed the goal proof under the assertions and invariants of seen and used machines, the local hypotheses and the component invariant.

In the three modes, the mono lemma prover is invoked with a time-out. If this time-out is not specified by the user, it is set to 60 seconds.

When proof is replayed in automatic mode, calls to the mono lemma prover are done with a time-out specified by the **Time_Out** resource given in the resource file of Atelier B (300 seconds by default).

To finish with, we can configure the proof force with the ff(l) argument. Proof will be attempted with successively each of the listed forces until proof succeeds or the list is exhausted.

Example

The mono lemma prover can be applied to the whole proof obligation (we suppose that the current force is 0):

```
PRI> ml
Starting Mono Lemma Prover Call
Proved by the Mono Lemma Prover
with force 0
```

or to the reduced proof obligation. This option is used when the proof obligation has many hypotheses:

```
PRI> ml(rp.1 | 5)
Starting Mono Lemma Prover Call
Proved by the Mono Lemma Prover
with force 0
```

Proof may be attempted with more selected hypotheses, but it may not succeed anymore.

```
PRI> ml(rp.5 | 10)
Starting Mono Lemma Prover Call
The Mono Lemma Prover failed to prove the current goal
```

The mono lemma prover can be used to prove a given proof obligation or a sub-goal. So it can be involved in a proof strategy by using it in the **te** (see chapter 5.48 page 128) command body.

Below it is used on reduced proof oblligations (1 iteration) with a time-out of 10 seconds.

```
PRI> te(ml(rp.1 | 10), Replace.Gen.All)
```

The prover may be used also with a list of forces to attempt. We go through the list of forces until one of them enables to achieve the proof. Here goal is discharged by force 1, we thus do not try force 3.

```
PRI> ml(ff(0;1;3) | rp.0 | 50)
Starting Mono Lemma Prover Call
Proved by the Mono Lemma Prover
with force 1
```

5.29 MiniProof

Mini Proof

Syntax

mp

Use

In **0** or **1** force, the **mp** command allows to use the prover without using proof by case. This command is similar to **pr(Red)**(see chapter 5.34 page 99).

Example

See command **pr(Red)**.

5.30 Next

Positioning on the next unproved proof obligation

Syntax

ne

Use

This command allows the user to go on to the next unproved proof obligation, if there is one. If there are no more unproved proof obligations, the **ne** command is ineffective.

This command can be used to go quickly to the first proof to be checked when the interactive proof of a component is opened.

Example

The component is made up of clause Initialisation and operation Calculus, two proof obligations which have been proved, and two unproved proof obligations. By using the **gs** (see chapter 5.20 page 68) command, the following situation can be obtained:

```
PRI> gs
State of all PO
    Initialisation
        PO1 Proved      not(e5 = e1)
        PO2 Proved      e1 = e5
    Calculus
        PO1 Unproved    not(e2 = e5)
        PO2 Unproved    e5 = e1
End
```

Let us suppose that the current proof obligation is *Calculus*.1. The user moves on to the next unproved proof obligation.

```
PRI> ne
Current PO : Calculus.2
```

By repeating this command, the user finds himself back to the proof obligation *Calculus*.1.

```
PRI> ne
Current PO : Calculus.1
```

5.31 Pmm compile

Loading user rules (Pmm)

Syntax

pc

Use

This command allows the loading and compilation of manual rules.

So as to be able to treat the most difficult proof cases, manual rules can be written in the .pmm file (proof methods manual) (see chapter 7 page 143), which is created by the user and has the name of the current component as a prefix.

This file should contain a group of valid theories, separated by &, written in the logic solver language. The use of these rules should remain marginal. In fact, these rules can be erroneous and cause the prover to prove false proof obligations.

When the interactive prover is launched, the pmm component file, if it exists, is automatically loaded into memory.

If the pmm file is modified by the user during the interactive proof, and if the latter wishes to use the latest pmm file rules, the **pc** command will allow the user to load the pmm file rules into memory.

While the pmm file is being loaded, the prover displays an acceptation or file error message.

Beware!
Whereas all the other functions of the interactive prover have been totally protected, this possibility of using manually written rules is not.

It is possible to key in a false rule, leading to false demonstrations. If no manual rule of this type has been used, the proof, whichever interactive commands have been used, is valid because the prover mechanisms themselves (automatic + interactive) are all valid.

However if manual rules have been added, then it will be necessary to check the validity of these rules. The use of a rules demonstrator can be recommended for this task; but it is clear that the interactive prover has been constructed so as to avoid the use of these manual rules.

Example

Given the following situation:

```
    Hypothesis
        xx: 1..10 &
        yy: 1..10 &
        zz: 1..109
    Goal
        (xx+1)*yy-1: 1..109
```

The user launches the prover

```
PRI> pr
Starting Prover Call
```

The goal $(xx+1)*yy-1 : 1..109$ is broken down into two sub-goals $1 \leq (xx+1)*yy-1$ and $(xx+1)*yy-1 \leq 109$. The first sub-goal is treated first.

The automatic prover stops because it can't solve the inequality $0 <= -2+yy+xx*yy$.

```
    New Hypothesis since last command
        2: 1..109 &
        2: 1..10 &
        0<=2 &
        2: NATURAL &
        2: INTEGER &
        0<=0 &
        0: NATURAL &
        0: INTEGER
    Goal
        0<= -2+yy+xx*yy
```

So the user decides to introduce a new rule, via the pmm file, and to use it in its proof.

So the *test.pmm* test file contains:

```
THEORY test IS

        binhyp(a: 1..10) &
        binhyp(b: 1..10)
        =>
        0<= -2+a+b*a

END
```

The rule is first loaded, then compiled.

```
PRI> pc
Loading theory test
```

In order to discharge the sub-goal $0 <= -2+yy+xx*yy$, the rule of the *test* theory is

applied.

```
PRI> ar(test.1,Once)
Starting Apply Rule
```

The first sub-goal is discharged and the automatic prover now tries to prove the second sub-goal:

```
    Hypothesis
        xx: 1..10 &
        yy: 1..10 &
        zz: 1..109
    Goal
        (xx+1)*yy-1<=109
```

The second sub-goal is now to be proved, but is not proved by the means of the **pr** command. So the user adds the rule allowing to discharge this goal. Eventually, the *pmm test.* file contains:

```
THEORY test IS

        binhyp(a: 1..10) &
        binhyp(b: 1..10)
        =>
        0<= -2+a+b*a;

        binhyp(a: 1..10) &
        binhyp(b: 1..10)
        =>
        (a+1)*b-1<=109

END
```

Since the pmm file has been modified, it has to be reloaded into memory. The previously loaded rules are replaced by the new ones.

```
PRI> pc
Loading theory test
```

So as to discharge the sub-goal $(a + 1) * b - 1 <= 109$, rule number two of the *test* theory is applied.

```
PRI> ar(test.2,Once)
Starting Apply Rule
```

The proof obligation is therefore proved.

5.32 Particularize hypothesis

Instantiation of a universally quantified hypothesis

Syntax

ph$(v_1, \ldots, v_n, h)$

with:

- h is a universal hypothesis of the form
  $\forall(w_1, \ldots, w_n).(P(w_1, \ldots, w_n) \Rightarrow Q(w_1, \ldots, w_n))$

Use

This command allows the assignment of a value to variables which appear, in hypotheses, under the scope of a universal quantifier. The values $v_1, \ldots, v_n$ are affected to the variables $w_1, \ldots, w_n$. If the value of one or several variables is unknown, the keyword $\_h$ can be used to signify that the variable(s) will not be instantiated. For example:

```
ph(e1,ENS1,_h,(MAXINT-ff(3)), !(aa,bb,cc,dd).PP(aa,bb,cc,dd))
```

will generate the hypothesis corresponding to

```
!cc.PP(e1,ENS1,cc,(MAXINT-ff(3)))
```

G being the initial goal, the goal becomes:
$P(v_1, \ldots, v_n) \wedge Q(v_1, \ldots, v_n) \Rightarrow G$

- so the automatic prover will seek to demonstrate $P(v_1, \ldots, v_n)$
- if it succeeds, the proof will continue with $Q(v_1, \ldots, v_n)$ in hypothesis.

The predicates $P(w_1, \ldots, w_n)$ contain the typing of $v_1, \ldots, v_n$.
The user must be aware, though, that the particularization of universally quantified hypotheses is not protected against ill-typing (see chapter 3.2 page 7) nor ill-definedness (see chapter 3.3 page 8). A bounded variable may be instantiated by an ill-typed or ill-defined value. Thus the user must verify the well-typing and well-definedness before using this command.
This can be checked *with hindsight* thanks to the **mdelta** tool (cf. User Manual Version 1.0.).

Example

Given the following:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    tt: ENS &
    uu: ENS &
    zz: ENS &
    !vv.(vv: ENS & (not(uu = vv) or not(tt = vv)) => zz = vv)
Goal
    not(tt = uu)
```

The user wishes to use the hypothesis $\forall vv.(vv \in ENS \ \wedge \ (\neg(uu = vv) \ \vee \ \neg(tt = vv)) \Rightarrow zz = vv)$, by instantiating vv with the value e_1.

The proof of $vv \in ENS \wedge (\neg(uu = vv) \vee \neg(tt = vv)) \Rightarrow zz = vv$ will, after instanciation, split itself into two parts:

- $vv \in ENS$
- then $\neg(uu = vv) \ \vee \ \neg(tt = vv)$

If these two sub-goals are proved then the sub-goal becomes $zz = e1 \ \Rightarrow \ \neg(tt = uu)$.

```
PRI> ph(e1,!vv.(vv: ENS & (not(uu = vv) or not(tt = vv)) => zz = vv))
Starting Particularize Hypothesis
```

The predicate of instanciated typing must be proved first:

```
    Goal
        e1: ENS
```

The **pr** command allows the discharging of this sub-goal.

```
PRI> pr
```

The following sub-goal is therefore:

```
    Goal
        not(uu = e1) or not(tt = e1)
```

Then the user launches the proof kernel:

```
PRI> pr
Starting Prover Call
```

The goal is proved. The next goal is generated in this way:

```
    Goal
        zz = e1 => not(tt = uu))
```

The predicate $zz = e_1$ can be placed under hypothesis by one of the two **pr** or **dd** (see chapter 5.13 page 53) commands.

5.33 Predicate prover

Calling the predicate prover

Syntax

pp
pp(t)
pp(rp.n)
pp(rp.n|t)
pp(rp(f))

with:

- t : predicate prover time-out, in seconds. If this value is not indicated, the predicate prover stops after 60 seconds.
- rp.n indicates that the predicate prover is used on reduced hypotheses.
  - **rp** is a keyword.
  - n is a positive integer which indicates the level of hypotheses considered. If n equals zero, only the goal (without the hypotheses) is processed by the predicate prover.
- rp(f) indicates that the predicate prover is applied on the hypotheses selected by formula f (of form a or f+a, a being a keyword). Available keywords for a are:
  - **inv**: component invariant
  - **sees**: assertions and invariants of used and seen machines
  - **loc**: local hypotheses
  - **typ**: type predicates of concrete variables
  - **abs**: assertions and invariants of previous components
  - **used**: constraints of used machines
  - **inc**: properties of included, imported and extended machines
  - **prp**: properties and valuations of the component

Use

This command allows the predicate prover to be used on the current goal.
This has three modes of functioning:

1. The first mode calls the predicate prover to the goal and to all the current hypotheses. This mode is not suitable for a large number of hypotheses.
2. The second mode calls the predicate prover to the goal and the hypotheses of reduced proof obligation. The hypotheses selected are the same as for the **rp**(see chapter 5.38 page 108) function.

3. Third mode applies the predicate prover to the goal and to the hypotheses that meet the criteria given by parameter f. For instance, **ml(rp(sees+loc+inv))** enables to proceed the goal proof under the assertions and invariants of seen and used machines, the local hypotheses and the component invariant.

In the three modes, the predicate prover is launched with a time-out. If the user does not define it, the time-out is of 60 seconds.

When the proof is replayed automatically, the calls to the predicate prover are made with a time-out specified by the **Time_Out** resource given in the resource file of Atelier B (300 seconds by default). This margin allows a successful recall of the predicate prover, on a less powerful machine.

Example

The predicate prover can be used on the complete proof obligation:

```
PRI> pp
Starting Prover Predicate Call
Proved by the Predicate Prover
```

or on the reduced proof obligation. This option is used when the proof obligation has many hypotheses:

```
PRI> pp(rp.1 | 5)
Starting Prover Predicate Call
Proved by the Predicate Prover
```

The proof can be tried with more selected hypotheses, but success is no longer guaranteed.

```
PRI> pp(rp.5 | 10)
Starting Prover Predicate Call
The Predicate Prover don't prove the current goal
```

The predicate prover can be used to prove a specific proof obligation, or to prove a subgoal. It can thus be part of a proof strategy, being used in the command body **te** (see chapter 5.48 page 128).

Here it is used on reduced proof obligations (1 iteration) with a time-out of 10 seconds.

```
PRI> te(pp(rp.1 | 10), Replace.Gen.All)
Begin TryEveryWhere
```

The work done by the predicate prover is then displayed:

```
+--+
Summary
Initialisation.1 transformed   Unproved --> Proved,   pp(rp.1)
Initialisation.4 transformed   Unproved --> Proved,   pp(rp.1)
End TryEveryWhere
```

Two proof obligations (*Initialisation*.1 and *Initialisation*.4) have been discharged.

5.34 Prove

Calling the automatic prover

Syntax

pr
pr(r.b.h,f,s)
pr(Tac(t),r.b.h,f,s)
pr(Tac(t))
pr(Red)
pr(Red,r.b.h,f,s)

with:

- r = **None** (display the name of the rules applied) or **Ru** (display the name and body of the rules applied)
- b = **Goal** (display of the goals) or **Stop** (display and stop on each goal)
- h = **None** (No displaying about hypotheses), **Gen** (display generated hypotheses) or **Full** (display generated hypotheses and hypotheses that are raised into the stack)
- f = **File** (generation of a trace file, visible with the command **Show Proof Tree**) or **NoFile** (no generation, default value)
- s = **Simpl** (Display the simplifications realized on every expression) or **NoSimpl** (The simplifications are not displayed - default value)
- t is a tactic (see chapter 3.6 page 11)

Use

This command allows the use of the automatic prover, to prove the current proof obligation.

The **pr** command is also useful to start off again a proof which almost succeeded, but which stopped because the maximum number of tries has been reached. Effectively, the automatic prover has a certain number of counters which limit the number of applications of certain mechanisms in one call, so as to avoid loops. Launching **pr** several times in a row can therefore have an effect.

If a **pr** call has really not been effective, this is signalled by the message

```
Prover call did nothing
```

In this case, it isn't worthwhile re-launching it.

The commands **pr(r.b.h,f,s)** and **pr(Tac(t),r.b.h,f,s)** allow the launching of the prover in trace mode (see chapter 11 page 151). The parameters f, s and Tac(t) are optional, but if one wishes to give s, then one needs to give f (the order of the parameters must be respected).

The following information are available:

- simplification of the goal, by application of a rule or a mechanism
- unloading a goal
- launching a case proof
- launching and ending proofs by attempt (internal sub-proofs)
- application of simplification rules

The presence of the Tac(t) parameter allows the use of user rules(pmm (see chapter 7 page 143), patchprover (see chapter 8 page 145)) within the automatic prover. The user's backward tactics can be applied once the local hypotheses have been raised in the stack, and before calling on the rules base. The forward tactics behave like the rules of a single theory. These rules are used with the prover's forward rules. The user cannot use complex tactics with Forward rules.

For example, the following forward tactic is not valid:

```
Fwd1~;(Fwd2;Fwd3)
```

If we use command **pr(Tac(backward,forward))**, the interactive prover will attempt to apply the rules of the backward tactic. If these rules generate hypotheses, the **DED** predefined theory must appear in the **backward** tactic. In that case, the rules of tactic **forward** will process the raising hypotheses.

In force 0 or 1, the **pr(Red)** command allows the use of the prover without starting proof by case. This use of the automatic prover is limited to:

- traversal of the rules base,
- processing of existential goals,
- over-typing (generating further typing hypotheses).

As far as forces Fast, 2 and 3 are concerned, command **pr(Red)** behaves the same as **pr** and can probably attempt proofs by cases.

Example 1

Given the following situation:

```
    New Hypothesis since last command
        e1: ENS &
        1: 1..5 &
        1: 1..100 &
        1: 1..10 &
        0<=1 &
        1: NATURAL &
        1: INTEGER &
        0<=zz &
        0<=yy &
        0<=xx &
        not(uu = e5) &
        not(1: NATURAL) => -1: NATURAL
    Goal
        not(uu = e1)
```

A first call to the automatic prover has not allowed the current goal to be discharged. The **pr** command is tried a second time, to see if the automatic prover has not failed in the proof, because of the limited number of applications of rules (internal prover counters, limiting the risk of loops (see chapter 3.9 page 14)).

```
PRI> pr
Starting Prover Call
```

The message *Prover call did nothing* indicates that the prover has not succeeded in a definite way, in proving the current goal, and that it has not produced another hypothesis.

```
Prover call did nothing

    Goal
        not(uu = e1)
```

Example 2

Let us now observe the functioning of the Prover in Trace mode.

Let us consider the following situation:

```
    Hypothesis
        xx: 1..10 &
        yy: 1..10 &
        zz: 1..100
    Goal
        xx+yy-1: 1..100
```

The prover is started in trace mode; the rule bodies along with information relative to the hypotheses are not displayed, all the goals are listed.

```
PRI> pr(None.Goal.None)

  Starting Trace in mode None.Goal.None , NoFile

Starting Prover Call

  After deduction, goal is now
        xx+yy-1: 1..100
```

The initial goal is decomposed into two sub-goals.

```
  By applying atomic rule InSetXY.13,
  the goal xx+yy-1: 1..100  is now
       1<=xx+yy-1
   and xx+yy-1<=100

  Goal
       1<=xx+yy-1
  is simplified in
       0<= -2+xx+yy

  Because 0  is a lower bound of -2+xx+yy - 0
  Goal 0<= -2+xx+yy is discharged.
```

The first sub-goal has been simplified then discharged. The second sub-goal can then be processed

As $(xx, yy) \in (1..10) \times (1..10)$, $101 - xx - yy$ is bounded by 81:

```
Goal
     xx+yy-1<=100
is simplified in
     0<=101-xx-yy

Because 81  is a lower bound of 101-xx-yy - 0
Goal 0<=101-xx-yy is discharged.

End of trace
```

If the **pr(Ru.Goal.None)** command had been applied, the part of the trace concerning the $InSetXY.13$ rule, that is to say:

```
By applying atomic rule InSetXY.13,
```

would have been:

```
By applying atomic rule InSetXY.13,
     n<=a &
     a<=p
     =>
     a: n..p
```

Example 3

The following example shows the use of the **Tac** parameter, to use backward and forward tactics.

Let us consider the following proof obligation:

```
        integers <: INTEGER &
        xx: INTEGER &
        xx-1: integers &
        =>
        xx: integers
```

The associated file **PMM** contains the **backward** and **forward** theories:

```
THEORY backward IS
        xx-1: integers => xx: integers => p
        =>
        p
END
&
THEORY forward IS
        xx-1: integers
        =>
        xx: integers
END
```

The **pr(Tac((backward;DED),forward))** command enables to discharge the current goal.

5.35 PreviousPO

Syntax

pv

Use

This command enables to go to the first previous unproved proof obligations, if there is one. If there is no more unproved proof obligation left, command pv is ineffective.

Example

The component have 2 operations, 1 proved proof obligation and 3 unproved proof obligations. By launching the **gs** command, we get the following situation:

```
PRI> gs
State of all PO
    Initialisation
        PO1 Proved      not(e5 = e1)
        PO2 Unproved     e1 = e5
    Calculus
        PO1 Unproved    not(e2 = e5)
        PO2 Unproved    e5 = e1
End
```

Let suppose that the current proof operation is *Calculus*.1. The user goes to the first previous unproved obligation.

```
PRI> pv
Current PO : Initialisation.2
```

By repeating the command, we go to proof obligation *Calculus*.2.

```
PRI> pv
Current PO : Calculus.2
```

5.36 Quit

Syntax

qu

Use

This command allows to leave the interactive prover.

If the current proof state has changed since its loading or if the command line has been modified, the prover will ask the user if he wants to save the new command line before quitting.

Example

A proof task has been performed on a current proof obligation. The user wishes to quit the current interactive proof session. The prover asks the user if he wishes to save the proof work of the last proof obligation used.

```
PRI> qu
Last PO does not have a saved demo. Your new demo does not Prove.
Do you want to save the new demo (will replace the old one)?
  Answer No to continue without saving (any other word to save):
```

5.37 Reset PO

Resetting all the commands of the command line

Syntax

re

Use

This command allows the resetting of all the commands on the command line, for the current proof obligation.

Example

Given the proof obligation which has the following command line:

```
Force(0) &
  ah(uu: ENS => (uu = e5 => tt = e1)) &
    pr &
    dd &
      dd &
        Next
```

The command **re**

```
PRI> re
Resetting PO
```

allows the re-initialisation of the command line. The command line is taken back to its starting point.

```
Force(0) &
  Next
```

5.38 Show reduced PO

Displaying the proof obligation with reduced hypotheses

Syntax

rp or **rp(n)**

with:

- n is a positive integer

Use

This command allows the display of the current proof obligation in the form of reduced hypotheses.

If n = 1, only the hypotheses which have a common symbol with the goal are selected.

If n = 2, the process is reiterated, by selecting the hypotheses which have a common symbol with the goal, or with the previously selected hypotheses.

Command **rp** is equivalent to **rp(1)**.

rp allows the user to find quickly the hypotheses likely to help demonstrating the goal. In particular, this applies to the proof obligations coming from machines doing a lot of SEES or INCLUDES, which may have a lot of hypotheses characterising variables which do not appear in the goal.

Example

Given the following situation:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    ENS: FIN(NATURAL*{ENS.enum}) &
    xx: 1..10 &
    yy: 1..10 &
    zz: 1..100 &
    tt: ENS &
    uu: ENS &
    not(uu = tt) &
    uu: {e1,e2,e3,e4} => tt = e5 &
    uu = e5 => tt = e1
Goal
    not(uu = e1)
```

With one iteration,

```
PRI> rp
Reducing hypothesis of lemma, 1 inclusion iteration(s)...
```

The proof obligation in its reduced form is therefore:

```
    Goal
        not(uu = e1)
    Hypothesis (1 pass(es) of inclusion by common symbols from goal)
        uu: ENS &
        not(uu = tt) &
        uu: {e1,e2,e3,e4} => tt = e5 &
        uu = e5 => tt = e1
End of reduced PO
```

With 2 iterations

```
PRI> rp(2)
Reducing hypothesis of lemma, 2 inclusion iteration(s)...
```

we get:

```
    Goal
        not(uu = e1)
    Hypothesis (2 pass(es) of inclusion by common symbols from goal)
        ENS: FIN(NATURAL*{ENS.enum}) &
        tt: ENS &
        uu: ENS &
        not(uu = tt) &
        uu: {e1,e2,e3,e4} => tt = e5 &
        uu = e5 => tt = e1
End of reduced PO
```

5.39 Repeat

Repetition of the last command

Syntax

rr

Use

This command allows to repeat the last command keyed in by the user.

Example

Given the **dd** command keyed in by the user:

```
PRI> dd
Starting Deduction
```

The **rr** command

```
PRI> rr
```

allows to repeat the last command keyed in.

```
Repeat: dd
Starting Deduction
```

If the keyed in command is a simultaneous command (see chapter 3.10 page 14),the **rr** command allows to replay these commands one after the other.

The user performs the command:

```
PRI> dd & dd & pr
Starting Deduction
Starting Deduction
Starting Prover Call
```

The **rr** command

```
PRI> rr
```

allows to repeat these 3 commands

```
Repeat: dd & dd & pr
dd not applicable: Goal is not p => q
dd not applicable: Goal is not p => q
Starting Prover Call
```

5.40 Suggest for exist

Instantiation of the existentially qualified goal

Syntax

se$(v_1, \ldots, v_n)$

with:

- $v_1, \ldots, v_n$ are valid expressions or the keyword $\_h$.

Use

This command allows to chose the instantiation of variables, under the scope of an existential quantifier which appears in the current goal. If the goal is as follows:

$\exists(w_1, \ldots, w_n).P(w_1, \ldots, w_n)$

then the goal becomes:

$P(v_1, \ldots, v_n)$

This command is not protected against ill-typing (see chapter 3.2 page 7) nor ill-definedness (see chapter 3.3 page 8) of the values with which we instantiate variables. Thus one must be careful to not introduce ill-typed or ill-defined expressions.

This can be checked *with hindsight* thanks to the **mdelta** tool (cf. User Manual Version 1.0.).

If the value of one or several variables is unknown or must remain undetermined, it is possible to use the keyword $\_h$, so as not to instantiate the chosen variables. For example, if the goal is:

```
#(aa,bb,cc,dd).P(aa,bb,cc,dd)
```

then

```
se(e1,ENS1,_h,(MAXINT-ff(3)))
```

will transform the goal into

```
#cc.P(e1,ENS1,cc,(MAXINT-ff(3)))
```

Example

Given the following situation:

```
Hypothesis
    ENS = {e1,e2,e3,e4,e5} &
    uu: ENS &
    zz: ENS &
Goal
    #kk.(kk: ENS & not(kk = uu) & not(kk = zz))
```

The user may replace the *kk* variable by a judiciously chosen value.

```
PRI> se(e1)
Starting Suggest for Exist
```

In this case, the current goal is replaced by the goal containing the instantiated *kk* variable.

```
    Goal
        e1: ENS & not(e1 = uu) & not(e1 = zz)
```

5.41 Search hypothesis

Search hypothesis

Syntax

sh(P,A)
sh(P)

with:

- P is a group of formulae separated by the operators **_and**, **_or**, **_not**
- A is a group of formulae separated by the operators **_and**, **_or**, **_not**.

Use

This command allows to search among the hypotheses, for the ones which satisfy certain criteria.

Argument P represents what the parts of a hypothesis must verify so that it is selected. Argument P must be a group of sub-formulae separated by the 3 special operators _and, _or, _not.

For example: if P is ($var1$ **_and** $var2$) **_or** $var3$, the hypotheses containing either both $var1$ and $var2$, or $var3$ are selected

Formulae with wildcards are accepted.

A represents what the entire hypothesis is to verify, in the same language as before.

For example, $(a = b)$ **_or** $(a \Rightarrow b)$ selects the hypotheses which are either equalities or implications.

Beware!! Inside a **_not**, the use of **_and** and **_or** is not recognised. In the same way, A must not be the equivalent of a term such as $((a = b)$ **_and** $(a \Rightarrow b))$, otherwise no hypothesis would be selected.

The argument A may be omitted.

If one of the elements of P is a variable, then the hypotheses that are found must contain the variable in question.

For example sh(var) selects the hypotheses containing var, but not $var\$i$ or my_var (here var is considered as a formula, and not as a string of letters).

A classical use consists in searching all the hypotheses which refer to a given variable.

Example

Given the following situation:

```
    Hypothesis
        ENS = {e1,e2,e3,e4,e5} &
        ENS: FIN(NATURAL*{ENS.enum}) &
        not(ENS = {}) &
        xx: 1..10 &
        yy: 1..10 &
        zz: 1..100 &
        tt: ENS &
        uu: ENS &
        not(uu = tt) &
        uu: {e1,e2,e3,e4} => tt = e5 &
        uu = e5 => tt = e1 &
        1<=xx &
        xx<=10 &
        1<=yy &
        yy<=10 &
        1<=zz &
        zz<=100
    Goal
        not(uu = e1)
```

We begin by searching all the hypotheses which contain the *uu* variable.

```
PRI> sh(uu)
```

The result obtained is:

```
Searching all Hypothesis that:
  contain uu
  match with a
Starting search...
Found hypothesis List is
        uu = e5 => tt = e1 &
        uu: {e1,e2,e3,e4} => tt = e5 &
        not(uu = tt) &
        uu: ENS
End of found hypothesis
```

The message **match with a** is displayed because, when parameter *A* is omitted, the hypotheses must coincide with the wildcard **a** pattern.

We then search the hypotheses which refer simultaneously to *uu* and *tt*.

```
PRI> sh(uu _and tt)
```

The result obtained is the following:

```
Searching all Hypothesis that:
  contain uu _and tt
  match with a
Starting search...
Found hypothesis List is
        uu = e5 => tt = e1 &
        uu: {e1,e2,e3,e4} => tt = e5 &
        not(uu = tt)
End of found hypothesis
```

We now search all the hypotheses which refer to the variable uu or which contain the expression $not(a)$ (a is a wildcard).

```
PRI> sh(uu _or not(a))
```

The selected hypotheses are:

```
Searching all Hypothesis that:
  contain uu _or not(a)
  match with a
Starting search...
Found hypothesis List is
        uu = e5 => tt = e1 &
        uu: {e1,e2,e3,e4} => tt = e5 &
        not(uu = tt) &
        uu: ENS
End of found hypothesis
```

Finally, we search the hypotheses which refer to the variable uu and whose form is $a : b$ or $a = b$.

```
PRI> sh(uu, (a:b _or a=b))
```

The selected hypotheses are:

```
Searching all Hypothesis that:
  contain uu
  match with a: b
Starting search...
Found hypothesis List is
        uu: ENS
End of found hypothesis
```

5.42 Show Proof

Display of saved proof commands

Syntax

sp(o.i)

with:

- o.i a proof obligation of the component currently processed.

Use

This command enables to display the force level and the saved proof commands of the o.i obligation proof.

Example

The current proof obligation is *Initialisation*.5. The user is attempting to prove it and wants to use the proof of the *Calculus*.7 proof obligation that was previously proved.

```
PRI> sp(Calculus.7)
```

The following message is then displayed:

```
Saved Proof Commands of Calculus.7: Force(0) & ar(PatchProverH0)
& ah(foo >= SRAM) & pp(rp.0)
```

5.43 Save with question

SAVE WITH QUESTION

Syntax

sq

Use

This command allows to save a proof task (command line) made upon current proof obligations, if necessary.

If there is a risk of loss or regression, the user is asked to confirm. The user who has been working for a long time on the same proof obligation should use the **sq** command so as not to lose his work in case of a power cut.

Example

The current proof obligation is *Initialisation*.1. The user has completed a proof task and wishes to save it, if it is necessary.

```
PRI> sq
```

The previous line and the new command line do not allow to prove the current obligation. A confirmation of saving is asked:

```
Last PO does not have a saved demo. Your new demo does not Prove.
Do you want to save the new demo (will replace the old one)?
  Answer No to continue without saving (any other word to save):
```

The user wishes to save his work.

```
  Yes
```

The proof obligation is saved:

```
PO Initialisation.1 saved
Current PO : Initialisation.1
```

5.44 Search rule

SEARCH RULE

Syntax

sr(T,CO,AN)
sr(T,CO)
sr

with:

- T is a list of theories separated by dots (for example: $t_1.t_2.t_3$)
  t_i is either the name of a theory, taken from the theories making up the rule base, or a keyword covering several names of theories from the rule base.

  The keywords are the following:

  - **All** : all the theories of the rule base
  - **Rewr** : the theories containing rewrite rules
  - **Back** : the theories containing deduction rules to be used in backward tactics
  - **Fwd** : the theories containing deduction rules to be used in forward tactics

- CO allows to define the selection criteria of the consequents of rules. It can have one of the following forms:

  - **Goal**: The selected rules must be appliable to the current goal within the current hypothesis context
  - **Goal2**: Consequent of selected rules matches the current goal (this constraint is weaker than that of the **Goal** keyword)
  - **abs**(C),D: The selected rules must have a consequent of the same form as C and contain at least one sub-formula of the same form as D.

- AN allows to define the selection criteria for the antecedents of the rules.

  - **abs**(A),B : The selected rules must have an antecedent of the same form as A and contain at least one sub-formula of the same form as B.

Use

This command allows to search one or several rules in the group of rules used by the prover.

CO and **AN** are lists of formulae and translate the criteria that the selected rules have to verify.

CO identifies the constraints related to the consequent of the rules and **AN** those related to the antecedent.

The search criteria which may concern the antecedent and/or the consequent of a rule, are of two types:

- **The absolute type**, identified by the prefix **abs**, operates a selection according to the global form of the consequent or antecedent of the rule.
- **The relative type** operates according to the form of one or more sub-formulae of the consequent or antecedent

If the selection only concerns the consequent of the rule, AN can be omitted.

If the user is only searching for the rules applying to the current proof obligation, he just keys in **sr**, and the choice will be made among the rules selected by the keywords Back and Rewr (**sr** is equivalent to **sr(Back.Rewr, Goal)**).

The criteria C, D, A and B can be omitted. If the criteria A or B are present, the criteria C et D must also be present (the order of parameters is important). In this case, the user will use **abs(No),No** for C, D so as to show that no criteria applies to the consequent. The command will be **sr(T, abs(No), No, abs(A), B)**.

Generally, the formulae keyed in are made of expressions separated by special operators **_and**, **_or** and **_not**. **The parameter of _not should not be put between brackets**.

So as to be selected, a rule must contain the expressions which are concerned according to the indications given by these operators. For example, the following formula allows to select the rules whose consequent is of the form a=b (absolute constraint), which contains overloadings, unions but not sets in comprehension:

$\mathbf{abs}(a = b), ((a < +b)$ **_and** $(a \cup b)$ **_and** $(\mathbf{\_not}(\{x|Q\})))$

Keywords **_and** and **_or** are forbidden inside a **_not** where their use would be superfluous. In fact, **_not**(X **_and** Y) is equivalent to **_not**(X) **_or** **_not**(Y).

Example

By default, **sr** is equivalent to **sr(Rewr.Back,Goal)**.

```
PRI> sr
Searching rules matching with goal in : Rewr.Back
```

Let us search in the theories SimplifyX and DifferenceX the rules whose consequent matches with the current goal.

```
PRI> sr(SimplifyX.DifferenceXY, Goal)
Searching rules matching with goal in : SimplifyX.DifferenceXY
```

Finally, it is possible to select the rules according to the general form and the expressions appearing in the consequents and antecedents.

We are searching for all the rules whose consequent is of the form $a = b$ and which contain at least one of the formula of the form $\neg(c)$:

```
PRI> sr(All, abs(a=b), not(c))
Searching in All rules with filter
    consequent should contain not(c)
    consequent should match with a = b
```

We are looking for rules whose consequent is $a = b$ and which contain at least one formula of the form $\neg(c)$, and whose antecedent is $a \geq b$:

```
PRI> sr(All, abs(a =b), not(c), abs(a>=b))
Searching in All rules with filter
    consequent should contain not(c)
    consequent should match with a = b
    antecedent should match with a >= b
```

We are looking for the rules which have an antecedent of the form $a \geq b$:

```
PRI> sr(All, abs(No), No, abs(a>=b))
Searching in All rules with filter
    antecedent should match with a >= b
```

We are looking for rules which have the form $a \geq b$ as a consequent:

```
PRI> sr(All, No, abs(a>=b))

Searching in All rules with filter
    consequent should match with a >= b
```

We are looking for rules that have one of their antecedents including at least one formula $\neg(a - b)$:

```
PRI> sr(Goal.Rewr, abs(No), No, abs(No), not(a-b))
Searching in Goal.Rewr rules with filter
    antecedent should contain not(a-b)
```

5.45 Simplify Set

Simplification of set expressions in the current goal

Syntax

ss

Use

This command launches some simplifications on the set expressions appearing in the goal. Involved mechanisms are much more powerful than the rules from the rule base. This command must therefore make the simplification of the goal better.

Set simplification mainly uses three tools.

- The first tool makes the simplifications on expressions composed of literal values.
- The second tool works on any term.
- The third tool tries to use information from the hypotheses stack.

The first tool works on the following set and functional operators:

- union ($A \cup B$)
- intersection ($A \cap B$)
- set difference ($A - B$)
- generalised union ($\mathsf{union}(E)$)
- generalised intersection ($\mathsf{inter}(E)$)
- inverse of a relation (r^{-1})
- domain ($\mathsf{dom}(r)$)
- range ($\mathsf{ran}(r)$)
- identity relation ($\mathsf{id}(r)$)
- domain restriction ($u \lhd r$)
- range restriction ($u \rhd r$)
- domain subtraction ($r \rhd v$)
- range subtraction ($r \rhd v$)
- image of a set ($r[w]$)
- application of a function ($f(x)$)

- overriding ($r \Leftarrow q$)
- direct product ($p \otimes q$)
- composition ($p; q$)
- parallel product ($p||q$)
- first and second projection ($\mathsf{prj}_1(E, F), \mathsf{prj}_2(E, F)$)
- cartesian product($A \times B$)
- cardinal ($\mathsf{card}(E)$)
- transformation of an interval ($a..b$) into an enumeration

Moreover, it can manage the BOOL set, and some set operations such as set membership. Because of the algorithm complexity, only the set operators are recognised by the second tool:

- union ($A \cup B$)
- intersection ($A \cap B$)
- set difference ($A - B$)
- generalised union ($\mathsf{union}(E)$)
- generalised intersection($\mathsf{inter}(E)$)

The third tool recognises the following operators:

- union ($A \cup B$)
- intersection ($A \cap B$)
- set difference ($A - B$)
- inverse of a relation (r^{-1})
- domain ($\mathsf{dom}(r)$)
- range ($\mathsf{ran}(r)$)
- identity relation ($\mathsf{id}(r)$)
- domain restriction ($u \lhd r$)
- range restriction ($r \rhd v$)
- image of a set ($r[w]$)
- overriding ($r \Leftarrow q$)
- composition ($p; q$)

It is worthy noting an important limitation of this command : The maximum number of elements of an enumerated set that can be handled by **ss** is set to ten because of the processing complexity.

Example

Given a proof obligation transformed by command **mp** into:

```
    Hypothesis
        ff: INTEGER +-> INTEGER &
        xx: INTEGER &
        yy: INTEGER &
        ff: INTEGER <-> INTEGER &
        dom(ff) <: INTEGER &
        ran(ff) <: INTEGER
    Goal
        ({2|->3,3|->4}/\{2|->xx})-{yy|->3} = ff
```

ss is applied to simplify the goal.

```
PRI> ss
Begin SimplifySet
```

and the new goal is:

```
    Goal
        ({2|->3}/\{2|->xx})-{yy|->3} = ff
```

This goal is indeed simpler. Without any hypothesis involving `xx` and `yy`, the simplification process cannot go further.

5.46 Step

Execution of the next saved command

Syntax

st
st(n)

with:
n is worth

- a numerical value, indicating the number of steps to take
- **End** to replay the whole saved proof

Use

This command allows to execute the next command of the saved command line. It allows to replay step by step interactive commands of a previous proof session.

The saved command line is made up of interactive commands which have been entered during a previous interactive proof (otherwise the command line only contains the command **pr** (see chapter 5.34 page 99)). When one goes to a proof obligation, no commands are processed beforehand. So, the user can replay the previous proof work which has been saved, thanks to the **st** command, and /or use other interactive commands.

The **n** parameter allows to apply several saved commands at the same time. Its numerical value allows to apply a specific number of commands. If the number is greater than the number of saved commands, the Step command sends back an error message.

End allows the replay of all the saved interactive commands, from the current position of the saved command.

Example

Given the proof obligation whose saved line is:

```
    Command line :
        Force(0) &
          Next
    Saved line pos 1
        Force(0) &
        ar(test.1,Fwd) &
        dd &
        dd &
        ar(test.2,Once) &
        pr &
        pr
```

There are 6 interactive saved commands ($ar(test.1, Fwd) \wedge dd \wedge dd \wedge ar(test.2, Once) \wedge pr \wedge pr$). We are going to replay them, one after the other.

The indicator *Saved line pos* 1 shows that the next **st** command will be command #1, that is to say $ar(test.1, Fwd)$ because $Force(0)$ is only a force indicator.

```
PRI> st
Next step: ar(test.1,Fwd)
```

The first saved command will be applied:

```
Starting Apply Rule
    Command line :
        Force(0) &
          ar(test.1,Fwd) &
            Next
    Saved line pos 2
```

The first command $ar(test.1, Fwd)$ has been replayed. The indicator *Saved line pos* shows that the next command to be made by the **st**will be command #2.

```
PRI> st
Next step: dd
```

The second saved command is applied:

```
Starting Deduction
    Command line :
        Force(0) &
          ar(test.1,Fwd) &
            dd &
              Next
    Saved line pos 3
```

It is possible to replay all the commands, up to the last one.

```
PRI> st(End)
```

The last four saved commands are then executed:

```
Starting Deduction
Starting Apply Rule
Starting Prover Call
Starting Prover Call
```

The command line obtained is therefore:

```
   Command line :
        Force(0) &
          ar(test.1,Fwd) &
            dd &
              dd &
                ar(test.2,Once) &
                  pr &
                  pr &
          Next
    Saved line pos 7
```

There are no more commands to be replayed because the position indicator is at number 7, that is to say, it is pointing towards the end of the saved commands.

```
PRI> st
Nothing to step
```

5.47 Save without question

Forced saving of the current command syntax

Syntax

sw

Use

This command allows to save without question the proof work done on the current proof obligation.

Example

Given the following proof obligation Calculus.2 whose command line is

```
Force(0) &
  pr &
    Next
```

The current proof obligation is then saved (status, command line), thanks to the command **sw**.

```
PRI> sw
PO Calculus.2 saved
```

5.48 Try everywhere

Application of a series of commands to a group of proof obligations

Syntax

te(f, m.n.p)
te(f,m.n')
te(f,n")
te(f)

with:

- f represents the command line to test for.
  - Either f is a series of commands separated by & and in brackets
  - Or f is the name of a proof obligation. In this case, the **saved** command line of the t.n. proof obligation is used.
- m can be:
  - **Append** : retry the proofs, placing the *f* commands after the saved commands
  - **Prepend** : retry the proofs, placing the *f* commands before the saved commands
  - **Replace** : retry the proofs, using the *f* commands instead of the saved commands
- n is worth:
  - **Loc** : only the proof obligations of the operation (or clause) which is being processed are concerned
  - **Gen** : process all the component proof obligations
  - **List(L)** : process all proof obligations of the L list (list of proof obligations identifier, that is to say Operation_Name . Obligation_Number, separated by &)
  - **Patt(P)** : process all proof obligations whose goal matches the P formula
  - **Op(O)** : process proof obligations of the operation (or clause) named O
  - **O[A..B]** : process proof obligations of operation (or clause) named O bounded by O.A and O.B with A **positive** and B **greater or equal to** A

- n' can be:
  - **List(L)**
  - **Patt(P)**
  - **Op(O)**
  - **O[A..B]**
- n" can be:
  - **List(L)**
  - **Patt(P)**
  - **Op(O)**
  - **O[A..B]**
  - **O.I** : process proof obligation named O.I where O is an operation or clause name and I a proof obligation number
- p can be:
  - **All** : process all proof obligations (even the proved ones)
  - **Unproved** : only process the unproved proof obligations

Use

This command allows the user to try to apply a demonstration to all the proof obligations of a component, while memorising only the demonstration which succeeded. The user normally prepares this series of commands when he is demonstrating one of the proof obligations. Different modes are proposed, so that the latest demonstration can fit well into the already existing demonstrations of every proof obligations.

Messages sent to the user indicate which proof obligations have changed. The command line of proof obligations which have been proved is also modified. Only the efficient commands (those which have had an effect on the state of the proof) will be saved.

A series of commands can only include one force command. This force command can be placed anywhere, since the whole command line is carried out under constant force.

te(f) is equivalent to **te(f, Replace.Loc.Unproved)**.

te(f,n") is equivalent to **te(f,Replace.n".Unproved)** with n" distinct from List(L) and O.I.

te(f,List(L)) is equivalent to **te(f, m.List(L).All)**. Moreover the Unproved option is not available with List(L).

te(f,O.I) is equivalent to **te(f,O[I..I])**.

te(f,m.n') is equivalent to **te(f,m.n'.Unproved)** with n' distinct from List(L).

te(f,m.List(L)) is equivalent to **te(f,m.List(L).All)**.

Example 1

Given the component containing clause Initialisation, operation Calculus and seven proof obligations, all unproved. The proof status is displayed with the **gs** command (see chapter 5.20 page 68):

```
PRI> gs
```

The user obtains:

```
State of all PO
    Initialisation
        PO1 Unproved    1: 1..10
        PO2 Unproved    1: 1..100
        PO3 Unproved    not(e5 = e1)
        PO4 Unproved    e1 = e5
    Calculus
        PO1 Unproved    xx+yy-1: 1..100
        PO2 Unproved    not(uu = e1)
        PO3 Unproved    e1 = e5
End
```

The user tries to apply the command line *dd* & *pr* for all the component proof obligations, by replacing the existing command line by *dd* & *pr*, if the proof succeeds. The existing command line is ignored (argument *Replace*).

```
PRI> te((dd & pr), Replace.Gen.All)
```

the proof obligations *Initialisation*.1, *Initialisation*.2, *Initialisation*.3, *Initialisation*.4 and *Calculus*.1 are proved and saved.

```
Begin TryEveryWhere
+++---
```

Then, the result of the application of the command line is displayed.

```
Summary
Calculus.1 :   Unproved --> Proved,   dd & pr
Initialisation.4 :   Unproved --> Proved,   dd & pr
Initialisation.3 :   Unproved --> Proved,   dd & pr
Initialisation.2 :   Unproved --> Proved,   dd & pr
Initialisation.1 :   Unproved --> Proved,   dd & pr
End TryEveryWhere
```

Using command **gs** (see chapter 5.20 page 68), we can verify that 5 proof obligations have really been proved.

```
PRI> gs
State of all PO
    Initialisation
        PO1 Proved    1: 1..10
        PO2 Proved    1: 1..100
        PO3 Proved    not(e5 = e1)
        PO4 Proved    e1 = e5
    Calculus
        PO1 Proved    xx+yy-1: 1..100
        PO2 Unproved  not(uu = e1)
        PO3 Unproved  e1 = e5
End
```

Example 2

Given the component containing clause Initialisation, operation Analysis and six proof obligations, all unproved. The proof status is displayed with command **gs** (see chapter 5.20 page 68):

```
PRI> gs
```

We obtain:

```
State of all PO
    Initialisation
        PO1 Unproved    ff: INTEGER +-> BOOL
        PO2 Unproved    ff(1) = TRUE
        PO3 Unproved    xx: dom(ff)
    Analysis
        PO1 Unproved    a1: 1..10
        PO2 Unproved    a1 = a2
        PO3 Unproved    a2 <= 11
End
```

The user tries to apply the *dd & pr* command line to the proof obligations number 1 of Initialisation and number 2 and 3 of Analysis. He uses default options, that is to say Replace and All:

```
PRI> te((dd & pr),List(Initialisation.1 & Analysis.2 & Analysis.3))

Begin TryEveryWhere
+++
Summary
Initialisation.1 transformed   Unproved --> Proved,   dd & pr
Analysis.2 transformed         Unproved --> Proved,   dd & pr
Analysis.3 transformed         Unproved --> Proved,   dd & pr
End TryEveryWhere
```

Then the user uses the saved proof of proof obligation Analysis.2 to try it on Analysis.1:

```
PRI> te(Analysis.2,Analysis.1)

Begin TryEveryWhere
+
Summary
Analysis.1 transformed   Unproved --> Proved,   dd & pr
End TryEveryWhere
```

Afterwards, the user decides to try again the *dd* command followed by command *pr* on the unproved proof obligations of clause Initialisation:

```
PRI> te((dd & pr),Replace.Op(Initialisation).Unproved)

Begin TryEveryWhere
++
Summary
Initialisation.2 transformed   Unproved --> Proved,   dd & pr
Initialisation.3 transformed   Unproved --> Proved,   dd & pr
End TryEveryWhere
```

We notice therefore that in our case, starting with proof obligations that were all unproved, it would have been better, for instance, to apply commands *dd* and *pr* to all the proof obligations of the Initialisation clause:

```
PRI> te((dd & pr),Replace.Initialisation[1..3].All)

Begin TryEveryWhere
+++
Summary
Initialisation.1 transformed   Unproved --> Proved,   dd & pr
Initialisation.2 transformed   Unproved --> Proved,   dd & pr
Initialisation.3 transformed   Unproved --> Proved,   dd & pr
End TryEveryWhere
```

To finish with, we could have used the previous proof commands only on the proof obligations whose goal matches the x : y formula, that is to say :

```
PRI> te((dd & pr),Replace.Patt(x : y).All)

Begin TryEveryWhere
+++
Summary
Initialisation.1 transformed   Unproved --> Proved,   dd & pr
Initialisation.3 transformed   Unproved --> Proved,   dd & pr
Analysis.1 transformed         Unproved --> Proved,   dd & pr
End TryEveryWhere
```

5.49 Proof by attempts

PROOF BY ATTEMPTS

Syntax

tp(m)
tp(m,n)

with:

- m is worth
  - Goal for a proof by attempts based on the form of the goal
  - Hyp for a proof by attempts based on hypotheses
- n is a numerical value indicating the maximum number of attempts to be made.

Use

This command can be used in two ways. The first method is based on the goal form, and attempts to generate further hypotheses based on rules which could be applied. This method uses automatically generated rules called alpha rules. The second method is based on hypotheses. The used rules belong to the classical rules of proof by attempts of the prover.

In both cases, a numerical value can indicate the maximum number of sub-proofs to be tried. The default value of this parameter is 20.

Example

If the current goal is:

```
Goal
    aa <: xx\/yy
```

The application of the tp(Goal,20) command gives the following result:

```
Goal
    aa <: xx &
    aa <: yy\/xx &
    xx <: xx\/yy &
    xx <: xx\/yy &
    aa <: xx\/yy/\aa &
    aa <: aa/\(xx\/yy) &
    POW(xx) <: POW(xx\/yy)
    =>
    aa <: xx\/yy
```

The hypotheses generated are hypotheses which will be used to prove the goal.

5.50 User Simplification

Use of user-provided rewrite rules

Syntax

us(T)
us(T | M)

with:

- T rewrite tactic:
  - t name of a user-provided theory (from the PatchProver or the Pmm file) only containing rewrite rules. These rules may be guarded but must not have any other antecedent.
  - t.n, name of a rewrite rule of theory t provided by the user,
  - t;U, where t is the name of a user-provided theory and U a rewrite tactic,
  - t.n;U, where t.n is the name of a user-provided rewrite rule and U a rewrite tactic.
- M application mode for rewriting:
  - either keyword **_Goal** (default mode when no other M mode is specified)
  - or keyword **_AllHyp**
  - or keyword **_Hyp(h)** with h corresponding to the chosen hypothesis

Use

This command enables to use user-provided rewrite rules (from PatchProver and/or a Pmm file), either on the h hypothesis, or on all hypotheses, or finally on the current goal while minimizing memory consumption.

When a compound tactic (that is to say a list of tactics separated by semicolons) is used, rules are applied successively by going through the tactic list from left to right.

If M = **Hyp(h)**, the goal becomes $H \Rightarrow G$ where H is obtained by applying the rewritings on hypotheses h, if they exist.

If M = **AllHyp**, the goal becomes $H \Rightarrow G$ where H is obtained by applying rewriting on all the hypotheses.

If M = **Goal**, the goal becomes G' where G' is obtained by rewriting the current goal G with the given rules.

The prover attempt to apply a given rewrite rule as long as it is likely to be applied.

Example

Given the following user-provided theories contained either in PatchProver or in a Pmm file:

```
THEORY My_Simplifications IS

        x: f[{a}] == {x |-> a} <: f;

        (x + y)*z == (x*z + y*z)
END

&

THEORY Enum_Simp IS

        binhyp(A : INTEGER) &
        binhyp(B : INTEGER)
=>
        (x: {A}\/{B} == (x = A) or (x = B))

END
```

Let us consider then the following proof obligation:

```
Hypotheses
    ...
    aa : INTEGER &
    bb : INTEGER &
    6 <= (xx+2)*3 &
    yy: {aa,bb}
    ...
Goal
    xx: ENS => not((xx+yy)*2 : gg[{5}])
```

We can rewrite the goal by using the rewritings from My_Simplifications:

```
PRV> us(My_Simplifications|_Goal)
```

We get the new goal:

```
Goal
    xx: ENS => not({(xx*2 + yy*2) |-> 5} <: gg)
```

We may want also, for instance, to apply only the first rewrite rule of My_Simplifications. We must thus provide the name of the rule we want to use. In our case, the name of the first rule of the My_Simplifications theory is *My_Simplifications.1*.

```
PRI> us(My_Simplifications.1)
```

The goal becomes:

```
Goal
    xx: ENS => not({(xx+yy)*2 |-> 5} <: gg)
```

We have only applied the first rule.

It is also possible to apply the simplifications on all the hypotheses.

```
PRI> us(My_Simplifications;Enum_Simp|_AllHyp)
```

All the new hypotheses appear as antecedents of the current goal:

```
    Goal
        6<=(xx*3 + xx*3) &
        yy = aa or yy = bb
        =>
        xx: ENS => not((xx+yy)*2 : gg[{5}])
```

To finish with, we may decide to simplify only one hypothese:

```
PRV> us(My_Simplifications|_Hyp(6<=(xx+2)*3))
```

We then get:

```
    Goal
        6<=(xx*3 + xx*3)
        =>
        xx: ENS => not((xx+yy)*2 : gg[{5}])
```

5.51 Validation of a rule

PROOF OF A USER-PROVIDED RULE

Syntax

vr(t,r | i)

with:

- t indicates the rule type and can take the 2 values:
  - **Back** for a Backward rule. This is the default value.
  - **Fwd** for a Forward rule.
- r is actually the rule.
- i is the predicate prover time-out expressed in seconds. If omitted, the time-out is 60s in interactive proof.

Use

This command enables to prove a user-provided rule that the user wants to use for his proof work. The predicate prover attempts to prove the rule.

The predicate prover is always used with a time-out. In fact, although it is built to not loop, the predicate prover can indeed take a very long time to prove (or not) a goal. This time-out is set by default to 60s but can be changed by the user.

The rules are translated in predicate language before their proof by the predicate prover. It is done in the same way as in the OPR (see OPR Reference Manual version 1.0.)

For instance, rules of form $binhyp(P) \ \wedge \ G \Rightarrow D$ are translated into $P \ \wedge \ G \Rightarrow D$.

Likewise rules of the form $binhyp(P) \wedge Q \Rightarrow (G == D)$ are translated into $P \ \wedge \ Q \Rightarrow (G == D)$.

Example

The user tries to prove the $A <: B \ \& \ B <: C => A <: C$ rule:

```
PRI> vr(Back, (binhyp(A<:B) & binhyp(B<:C) => A<:C))
```

The rule is proved by the predicate prover.

```
The rule was proved
```

6 Customizing the prover

The prover uses a resource management system built in Atelier B. This system enables to modify the prover behaviour through options that are taken into account when a B project is opened (cf. paragraph 2.6. "Atelier B customization" of Atelier B User Manual).

6.1 User-definable time-out

Resource : `ATB*PR*Time_Out`.
Value : positive integer.
Meaning : time-out in seconds.
Default value : 300 (seconds).
This option allows to modify the time-out value of the satellite provers PP (Predicate Prover) and ML (Mono Lemma Prover) in user-definable mode "`User_Pass`" (see chapter 10 page 149) or replay mode "`Replay`".

This option enables to test, when in User Pass, proof tactics that use massively the predicate prover. This opportunity of customization therefore enables to launch proofs with small time-outs (maximum time of computation allowed before the proof process stops) so one can quickly test such a tactic efficiency.

Alternatively one can also increase PP and ML time-outs when one is replaying (`prove replay`) a project on a slower machine: thus one can be confident that if some proofs succeed on fast machines, they will also succeed on slower ones.

Example :

If we have the following `User_Pass` theory at our disposal:

```
THEORY User_Pass IS
    ff(0) & dd(0) & pp(rp.0)
END
```

and if we want to know whether this series of commands is efficient, we just have to set resource `Time_Out` to a small value, 10 seconds for instance, and start the automatic proof in `User_Pass` mode. Generally if a command run-time (for instance, the one of `pp`) lasts more than 10 seconds, it suggests that this command will never succeed. So if the series above succeeds in less than 10 seconds, it's OK and the series is efficient, on the contrary, if one of its commands is still running after 10 seconds, the proof is stopped and the user may consider that the series is inefficient.

6.2 Normalisation of formulae $P \Rightarrow Q$ and $\neg P$

Resource : `ATB*PR*Keep_Non_Simplified_Hypothesis`.
Value : `TRUE` or `FALSE`.
Meaning : Non simplified predicates are kept only if this resource equals to `TRUE`.
Default value : `TRUE`.
When the simplication mechanism is applied to predicates of form $P \Rightarrow Q$ and $\neg P$, it transforms these formulae into $P \wedge P' \Rightarrow Q$ and $\neg(P \wedge P')$ (where P' is the simplified form of P) if the resource is set to `TRUE`.

On the contrary, the `FALSE` value enables to keep only simplified predicates, *that is to say* to transform the above formulae into $P' \Rightarrow Q$ and $\neg P'$.

In some cases, the presence of both the non-simplified predicate and its simplified form prevents the prover from applying some of its mechanisms. The prover cannot, for instance, perform a *Modus Ponens* on the P, P' and $(P \wedge P') \Rightarrow Q$ hypotheses though it performs it on the P' and $P' \Rightarrow Q$ hypotheses.

Example :

Let us consider the following proof obligation:

```
Goal
        x1<=3-y1 => x1<=2+ii &
        x1+y1<=3 &
        =>
        5+z1 = y1
```

The resource is set to `TRUE`. If we perform a `dd(0)` (to raise in the hypothesis stack the two local hypotheses) followed by a `sh(x1)` (to display hypotheses containing the $x1$ term) we get:

```
Found hypothesis List is
    x1+y1<=3 &
    0<=3-x1-y1 &
    0<=3-x1-y1 & x1<=3-y1 => x1<=2+ii
End of found hypothesis
```

Let us notice that the $0 <= 3 - x1 - y1$ hypothesis (respectively $0 <= 3 - x1 - y1 \wedge x1 <= 3 - y1 \Rightarrow x1 <= 2 + ii$) is the simplified version of formula $x1 + y1 <= 3$ (respectively $x1 <= 3 - y1 => x1 <= 2 + ii$), as expected the non-simplified versions have been kept.

Here, we can not do a *modus ponens* on hypothesis $0 <= 3 - x1 - y1 \wedge x1 <= 3 - y1 \Rightarrow x1 <= 2 + ii$ as we do not have the non-simplified hypothesis $x1 <= 3 - y1$ at hand:

```
Invalid argument / Inexistent a=>b Hypothesis in
mh(0<=3-x1-y1 & x1<=3-y1 => x1<=2+ii)
```

Let us set the resource to `FALSE`. By performing the same interactive commands on the same proof obligation, we get:

```
Found hypothesis List is
    x1<=2+ii &
    0<=3-x1-y1 &
    0<=3-x1-y1 => x1<=2+ii
End of found hypothesis
```

This time, only the simplified versions of the formulae have been kept. Note that we have now at our disposal the hypothesis $x1 <= 2 + ii$ that was added by a *modus ponens* automatically applied to $0 <= 3 - x1 - y1 \Rightarrow x1 <= 2 + ii$. The prover functionning was not impeded by non-simplified hypotheses.

6.3 Additional rule packages

Resource : `ATB*PR*Use_Rule_Package`.
Value : list of packages (or theories) identifiers separated by commas.
Meaning : list of additional rule packages (*simplification*, *backward* and *forward*) to be added to the prover native rule base.
Default value : symbol "?".
This new functionnality enables the use of additional rule packages. These packages of validated rules are made of three different rule categories: *simplification*, *backward* and *forward* (see chapter 3.4 page 9). They are used by the automatic prover like rules of the native rule base.

As for now, only the `p1` package has been added. To enable the use of the simplification (respectively, backward and/or forward) rules, it is enough to set this resource to `s1` (respectively `b1` and/or `f1`). If we want to use all the `p1` rules, we just have to specify the `p1` value in the resource file.

Setting the resource to "?" means that no additional rule package is used.

In the future, we aim to add several rule packages to the automatic prover.

Rules of the `p1` package enable to process B language operators that were not fully handled by the native rule base of the prover: modulo mod, minimum min, maximum max, integer division /, generalised sum $\sum$ and product $\prod$.

Example :

Given the $xx \ mod \ tt <= nn$ goal under the hypotheses:

```
tt<=nn &
1<=tt &
tt:INT1 &
xx<=nn &
xx:INT1 &
1<=nn &
nn:INT1
```

When the resource *is not* set to `p1`, the `pr` command fails. On the contrary, this command succeeds when the resource equals `p1`.

6.4 Maximum Number of Instantiations of Universally Quantified Hypotheses

Resource : `ATB*PR*Max_Number_Of_Universal_Hypothesis_Instantiation`.
Value : quadruplet of positive integers separated by commas.
Meaning : Maximum number of applications of the **GenAny** mechanism for each of the proof forces.
Default value : (100, 200, 1000, 10000).
This resource enables to limit, for each proof force used, the number of applications of the forward rules **GenAny** to universally quantified hypotheses. Therefore, the first value of the quadruplet stands for the maximum number of applications of the rules of GenAny to one universally quantified hypothesis for force 0, the second one for force 1, the third one for force 2 and the last one for force 3.

The **GenAny** mechanism of the prover enables to particularize hypotheses of the form $\forall(X).(P(X) \Rightarrow Q)$ (where $P(X)$ is a predicate verified by X) according to hypotheses $P(X_i)$, that is to say, to generate hypotheses $[X := X_i]Q$ for all X_i verifying P.

This option thus enables the user to limit the number of applications of the GenAny rules to universally quantified hypotheses, for each proof force separately.

When a B component contains several universally quantified hypotheses, it is generally interesting to limit the number of applications of the GenAny rules to each of the hypotheses: it allows to avoid the generation of too many hypotheses since most of them will not be useful for the proof process and so to avoid a combinatorial explosion.

7 Proof Manual Method: adding user rules

The automatic prover contains general proof mechanisms, which stem from a basis of rules which are not universal.

These mechanisms have been designed in order to solve a large range of proof obligations, principally simple proof obligations. Therefore, the prover has a limited power of resolution.

So as to be able to handle the more difficult cases, it is possible either to orientate the proof by interactive commands (**ah**,**dd**, **ph**, **se**, ...), or to use manual rules. These rules can correspond to omissions in the rule base. They also allow the user to discharge a proof obligation, during the first step of an interactive proof, by incorporating the proof obligation (in its "wildcardised" form) in the component pmm file.

This file is located in the "project database" directory (*bdp*) and its extension is *pmm*. It is written in the logic solver language [1].

The rules contained in the *.pmm* file are loaded by the **pc** (see chapter 5.31 page 89) command and applied by the **ar** command (see chapter 5.4 page 26).

When accessing the .pmm file, the prover displays a message of acceptation of files or an error message.

The rules must be equipped with the trace system (see chapter 11 page 151) in order to be used with the trace system or to appear in the proof tree. If *pmm* rules are applied and aren't traced, the behavior of the proof tree generation module is not guaranteed anymore.
Caution!

While the other functions of the interactive prover are fully protected, this possibility of applying manually written rules is not.

It is possible to enter a false rule, which provokes false demonstration. If no manual rule of this type has been used, then the validity of the demonstrator (automatic+interactive) is sufficient to ensure the validity of the proof, whichever interactive commands have been used.

However, if manual rules have been added, then the user will have to be sure of these rules. The use of a rule demonstrator can be recommended for this task, but it is clear that the interactive prover was designed in order to avoid the use of these manual rules.

[1] see the *Logic Solver Reference Manual*

8 Patchprover: adding rules directly to the prover

This system can be used in the following way:

- Create a file called PatchProver in the bdp of the project
- Program in this file with the Logic solver language, while keeping in mind that:
  - The rules of the $PatchProverB_i$ theory, where i is the force, will be applied by the force i BEFORE the prover rules and mechanisms.
  - The rules of the $PatchProverA_i$ theory, where i is the force, will be applied by the force i AFTER the prover rules and mechanisms (just before failure).
  - The rules of the $PatchProverH_i$ theory will be applied on the conjunctive form of each group of hypotheses that is loaded in force i.
  - The $Fast$ force is not equipped with the $PatchProver$ mechanism.

  In $PatchProverB_i$, B stands for "Before" and in $PatchProverA_i$, A stands for "After".
- These theories are initially empty in the prover:
  - PatchProverH0 PatchProverH1 PatchProverH2 PatchProverH3
  - PatchProverB0 PatchProverB1 PatchProverB2 PatchProverB3
  - PatchProverA0 PatchProverA1 PatchProverA2 PatchProverA3
- NO NORMALISATION IS MADE IN THIS FILE. So the user should not use the formulae of the left row of the normalisation table. Particularly, as far as the inner notation used by the prover is concerned, {e} must always be a singleton. Otherwise, the subsequent behaviour is not guaranteed.
- All these theories are "backward". The $PatchProverH_i$ theories must contain rewrite rules. They are applied to each group of hypotheses by

  ```
  bguard((PatchProveri~;RES): bresult(H), Q)
  ```

  Other theories can be created, `bcall` and `bguard`, etc. can be used, In order to display messages, use

```
bcall(WRITE: bwritef(...))
```

- BEWARE : THE USE OF PATCHPROVER IS RESERVED TO THOSE WHO KNOW THE LOGIC SOLVER LANGUAGE. THIS IS NOT A SECURE METHOD. PatchProver is only read when the automatic or interactive prover starts. Once the file has been modified, it is better to unprove all the proof obligations and replay the proof commands.
- If PatchProver contains syntax errors, it isn't taken into account, and has no influence on the proof.

The rules and calls to the mechanisms must be equipped with the trace system (see chapter 11 page 151) , if the trace system is to be used during demonstrations, and the proof tree is to be obtained. If the PatchProver rules are applied, and aren't traced, the behaviour of the proof generation tree is no longer guaranteed.

9 User Simplification: user-provided simplification theories

This function allows to use user-provided rewrite theories in an efficient way. This theories are written in the PatchProver and/or in the associated Pmm file, they contain only rewrite rules and can be used with the following guard of the Logic Solver language:

bguard(UserSimpX: UserSimpG(T | B), R)

where **T** is the proof **tactic**, **B** a formula we want to simplify and **R** a **wildcard** (syntaxically: a single letter) that receives the result of the tactic **T** rule application to formula **B**.

The order of rule applications is given by the proof tactics.

Syntax is as follows:

Tactic ::= T | T**.**n | T **; Tactic** | T**.**n **; Tactic**

where T is a rewrite theory name and n a positive integer (therefore T**.**n is a name of a rule from theory T). If the tactic is just a theory name, all the rewrite rules within it will be tried. If the tactic is a rule name, only the rewrite rule of the same name will be used. Finally, if the tactic is of form **U ; V** where **U** and **V** are tactics, Tactic **U** will be executed at first and then **V**.

EXAMPLE

Let us consider, for instance, the following rewrite rules that are contained in the Patch-Prover file:

```
THEORY Maplet IS

        x: f[{a}] == {x |-> a} <: f

END

&

THEORY Enum_Simp IS

        binhyp(A : INTEGER) &
        binhyp(B : INTEGER)
=>
        (x: {A} \/ {B} == (x = A) or (x = B))

END
```

These rules can be used in other user-provided rules in an efficient way with the predefined theory *UserSimpX*:

```
THEORY Assumed_Proof IS

bguard(UserSimpX: UserSimpG(Maplet|x:f[{a}]),R) &
bsubfrm({x|->a},btrue,R,r) &
bnum(a) &
binhyp(not(Eval({x|->a}) = {y,z}))
=>
not(x:{y,z} &  x:f[{a}])

END
```

10 User Pass: Using configurable passes

10.1 Presentation

It is possible to define proof tactics and to use them in automatic proof. These proof tactics are made up of interactive command lines. Each command line will be tested on all the proof obligations that remain to be proved. Call to the interactive prover is performed in the "Automatic User Pass" mode.

The proof tactics are defined:

- either in the PatchProver file (see chapter 8 page 145)
- or in the pmm file(see chapter 7 page 143) associated with each component to be proved.

They must be contained in the **User_Pass** theory. If the **User_Pass** theory is defined both in the PatchProver and in the pmm file, only the theory contained in the PatchProver will be considered, and the following message will be displayed:

```
Theory User_Pass Not Loaded because name clashes with native Theories
```

An example of **User_Pass** theory is given below:

```
THEORY User_Pass IS

   ff(0) & dd(0) & pr(Red);
   ff(0) & dd(1) & pr(Red) & ch & dd(1) & pr(Red);
   ff(0) & dd(1) & tp(Goal,10)

END
```

The first command line used will be:

```
ff(0) & dd(0) & pr(Red)
```

After application of this first command line, the following command line will be applied to the unproved proof obligations that remain:

```
ff(0) & dd(1) & pr(Red) & ch & dd(1) & pr(Red)
```

Finally, for the remaining proof obligations, the last command line will be used:

```
ff(0) & dd(1) & tp(Goal,10)
```

10.2 User_Pass filters

It is possible to filter proof tactics by doing what follows:

- Use of the `Operation` keyword: If the user wish to apply some of the proof commands only to unproved proof obligations of an operation (or clause) o, he needs to add the keyword `Operation(o)` in the User_Pass theory command line,
- Use of the `Pattern` keyword: If the user wish to apply some of the proof commands only to unproved proof obligations whose goal (without local hypotheses) matches formula f, he just needs to add the keyword `Pattern(f)` in the User_Pass theory command line.

The filter position in the command list does not matter.

It is also possible to combine the two filters above to apply commands only to proof obligations of a given operation (or clause) whose goal matches some formula.

Let us consider, for instance, the following User_Pass theory:

```
THEORY User_Pass IS

        Operation(op0) & ff(0) & dd(0) & pr(Red);
        Pattern(x=y) & ff(0) & dd(1) & pr(Red) & dd(1) & pr(Red);
        Operation(op1) & Pattern(x:X) & ff(0) & dd(1) & tp(Goal,10)

END
```

The first command line to be used on the unproved proof obligations of operation op0 will be:

```
ff(0) & dd(0) & pr(Red)
```

To the proof obligations that remain unproved after applying this first command line and whose goal matches pattern $x = y$, the following command line will be applied:

```
ff(0) & dd(1) & pr(Red) & dd(1) & pr(Red)
```

Finally, the last command line will be used on the remaining unproved proof obligations of operation op1 and whose goal matches pattern $x \in X$:

```
ff(0) & dd(1) & tp(Goal,10)
```

The advantage of using filters is that it prevents from uselessly attempting to apply commands to proof obligations, when it is known to be unsuccessful (particularly because of the goal form).

11 Trace system

11.1 Description

The trace system allows the user to follow the application of the prover rule base, the goal simplifications done when they are performed, the proofs by case and by attempt, the generation, simplification and rising of derived hypotheses.

The equipment of rules is done the following way:

- Backward atomic rule

  The original rule has no antecedent and has the form:

  ```
  Q
  ```

  The equivalent rule equipped with the trace system is:

  ```
  bcall1(AtomicRule(first_rule)) => Q
  ```

  first_rule is nickname of the rule, it is independent of the theory where the rule is specified.

- Backward non-atomic rule

  The original rule is of the form:

  ```
  P => Q
  ```

  The equivalent rule equipped with the trace system is:

  ```
  bcall1(BackwardRule(second_rule)) & P => Q
  ```

- Forward Rule

  The original rule is of the form:

  ```
  P => Q
  ```

  The equivalent rule equipped with the trace system is:

```
P => Q & bcall1(ForwardRule(Third_rule))
```

For the (PatchProver) mechanism equipment, the entry in and exit from the mechanism will be traced. If the MECH mechanism calling code is the following:

```
Some_processing
=>
MECH(I, O);
```

with I corresponding to the input data of the mechanism, and O corresponding to the output data, the equivalent mechanism equipped with the trace system will be:

```
Some_processing &
bcall1(SimplifyNewH(I,O))
=>
MECH(I, O);
```

if it is a mechanism transforming or generating hypotheses and

```
Some_processing &
bcall1(SimplifyNewG(I,O))
=>
MECH(I, O);
```

if it is a mechanism transforming the goal.

11.2 Help tool

The source code of a program allowing to equip automatically rules of a file with the trace mechanism is provided in appendix.

To use this program, do the following:

1. The program must be compiled (file *equipe.src*)

   ```
   krt -c equipe.src equipe.kin
   ```

   before this, copy the symbol table from `AB/press/lib/Bsym/B_ST` to the directory containing `equipe.src`.

2. A *equipe.ex* file must be created, containing the **Equipe** formula, parameterized by the name of the file to be equipped, and, possibly, the list of Forward theories (By default, it is considered that all theories contain Backward rules).

   For example, if the *equipe.ex* file contains:

   ```
   Equipe(‘‘test.src’’)
   ```

   the rules contained in the file “test.src” will be equipped with the backward mode trace system.

   If the *equipe.ex* file contains:

```
Equipe(‘‘test.src’’ | (foo , foobar, gnu))
```

The rules contained in the “test.src” file will be equipped with the trace system in backward mode, except those of the foo, foobar and gnu theories, which will be equipped with the system in forward mode.

3. The program is to be executed

```
krt -b equipe.kin equipe.ex
```

Example of Use

If the *equipe.ex* file contains:

```
Equipe(‘‘test.src’’ | ForwardDirection)
```

and the *test.src* file contains:

```
THEORY stuff IS

 foo
 =>
 gnat;

 foobar
 =>
 machin;

 foo;

 foobar

END

&

THEORY ForwardDirection IS

 foobar
 =>
 foo;

 gnu
 =>
 gnat

END
```

the launching of the *equipe* program

```
krt -b equipe.kin equipe.ex
```

will result in:

```
THEORY stuff IS
bcall1(BackwardRule(stuff.1)) &
foo
=>
gnat

;
bcall1(BackwardRule(stuff.2)) &
foobar
=>
machin

;
bcall1(AtomicRule(stuff.3))
=>
foo

;
bcall1(AtomicRule(stuff.4))
=>
foobar

END

&

THEORY ForwardDirection IS
foobar
=>
foo &
 bcall1(ForwardRule(ForwardDirection.1))

;
gnu
=>
gnat &
 bcall1(ForwardRule(ForwardDirection.2))

END
```

12 List of available commands

List of commands by theme

category	sub-category	command	meaning	page
Proof construction	Change proof level	ff	Change force	61
	Prover call	pr	Prove	99
		ml	Mono Lemma Prover	84
		mp	MiniProof	87
		pp	Predicate Prover	96
		ap	Arithmetic Prover	24
		ss	Simplify Set	121
		mc	ModelChecking	78
		tp	Proof by attempts	133
	Rule Application	ar	Apply rule	26
		us	User Simplification	134
	Rewrite	ae	Abstract Expression	19
		eh	Use equality in hypothesis	57
	Special case of inference rule	ct	Contradiction	44
		cts	Special contradiction	46
		fh	False hypothesis	63
		dc	Do cases	48
		dcs	Special do cases	51
		se	Suggest for exist	111
	Operations on hypotheses	dd	Deduction	53
		ch	Create Hypothesis	43
		ph	Particularize hypothesis	93
		mh	Modus ponens hypothesis	82
		ah	Add hypothesis	22

category	sub-category	command	meaning	page
Search and display of information	Search in base	sr	Search rule	118
	Search in component	gs	Global situation	68
		sp	Show Proof	116
	Search in PO	sh	Search hypothesis	113
		la	Logical Analysis	74
		dt	Display Term	55
		gt	Graphical Trace	71
		lp	Show literal PO	76
		rp	Show reduced PO	108
		cg	Current Goal	42
	Search in commands	help	Help	73
Browsing PO		ba	Back	33
		re	Reset	107
		ne	Next	88
		pv	PreviousPO	105
		go	Goto	65
		gr	Goto with reset	67
		gw	Goto without save	72
Command repetition		rr	Repeat	110
		bb	Loop	36
		te	Try everywhere	128
		st	Step	124
Save the PO		sw	Save without question	127
		sq	Save with question	117
User-provided theories		pc	Pmm compile	89
		vr	Validation of rule	137
Quit		qu	Quit	106

List of commands in alphabetical order

Command	Meaning	page
sq	Save with question	117
sr	Search Rule	118
ss	Simplify Set	121
st	Step	124
sw	Save without question	127
te	Try everywhere	128
tp	Proof by attempts	133
us	User Simplification	134
vr	Validation of rule	137

13 Appendix

Program of equipment with the trace system

Here is a program, written in the logic solver language, which allows the automatic equipment of a file with the trace system.

```
‘B_ST

THEORY Main IS

        bget(F, R)
&       EquipeTheories(R | __PasDeTheorie__)
        =>
        Equipe(F);

        bget(F, R)
&       EquipeTheories(R | L)
        =>
        Equipe(F | L);

        bcall(WRITE: bwritef("\nTHEORY % END\n", T))
        =>
        EquipeTheories((THEORY T END) | L);

        bcall(WRITE: bwritef("\nTHEORY % IS\n", T))
&       bcall(MODR: bmodr(IndexRegle.1,0))
&       EquipeReglesBackward(C | T)
&       bcall(WRITE: bwritef("\nEND\n"))
        =>
        EquipeTheories((THEORY T IS C END) | L);

        bsearch(T, (L ,  btrue), R)
&       bcall(MODR: bmodr(IndexRegle.1,0))
&       bcall(WRITE: bwritef("\nTHEORY % IS\n", T))
&       EquipeReglesForward(C | T)
&       bcall(WRITE: bwritef("\nEND\n"))
        =>
        EquipeTheories((THEORY T IS C END) | L);


        EquipeTheories((A) | L)
&       bcall(WRITE: bwritef("\n&\n"))
&       EquipeTheories((B) | L)
        =>
        EquipeTheories((A & B) | L);
```

```
        brule(IndexRegle.1, N)
&       bguard((ARI;MODR): bmodr(IndexRegle.1, (N+1)))
&       brule(IndexRegle.1, M)
&       bcall(WRITE: bwritef("%\n=>\n% &\n bcall1(ForwardRule(%.%))\n", A, B, T, M))
        =>
        EquipeReglesForward((A=>B) | T);

        EquipeReglesForward(A | T)
&       bcall(WRITE: bwritef("\n;\n"))
&       EquipeReglesForward(B | T)
        =>
        EquipeReglesForward((A;B) | T);

        brule(IndexRegle.1, N)
&       bguard((ARI;MODR): bmodr(IndexRegle.1, (N+1)))
&       brule(IndexRegle.1, M)
&       bcall(WRITE: bwritef("bcall1(AtomicRule(%.%)) \n=>\n%\n", T, M, A))
        =>
        EquipeReglesBackward(A | T);

        brule(IndexRegle.1, N)
&       bguard((ARI;MODR): bmodr(IndexRegle.1, (N+1)))
&       brule(IndexRegle.1, M)
&       bcall(WRITE: bwritef("bcall1(BackwardRule(%.%)) &\n%\n=>\n%\n", T, M, A, B))
        =>
        EquipeReglesBackward((A=>B) | T);

        EquipeReglesBackward(A | T)
&       bcall(WRITE: bwritef("\n;\n"))
&       EquipeReglesBackward(B | T)
        =>
        EquipeReglesBackward((A;B) | T)

END

&

THEORY IndexRegle IS
        0
END
```