

**Atelier B**

# **Prouveur Interactif**

**Manuel de référence**

**version 4.0**



ATELIER B  
Prouveur Interactif Manuel de référence  
version 4.0

Document établi par CLEARSY.

Ce document est la propriété de CLEARSY et ne doit pas être copié, reproduit, dupliqué totalement ou partiellement sans autorisation écrite.

Tous les noms des produits cités sont des marques déposées par leurs auteurs respectifs.

CLEARSY  
Maintenance ATELIER B  
Parc de la Duranne  
320 avenue Archimède  
Les Pléiades III - Bât.A  
13857 Aix-en-Provence Cedex 3  
France

Tél 33 (0)4 42 37 12 99  
Fax 33 (0)4 42 37 12 71  
email : [maintenance.atelierb@clearsy.com](mailto:maintenance.atelierb@clearsy.com)

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Notions de base</b>	<b>3</b>
2.1	Qu'est ce qu'une obligation de preuve? . . . . .	3
2.2	Qu'est ce qu'un prédicat bien typé? . . . . .	3
2.3	Qu'est ce qu'une expression bien définie? . . . . .	4
2.4	Qu'est ce qu'une règle? . . . . .	5
2.5	Qu'est ce qu'une théorie? . . . . .	7
2.6	Qu'est ce qu'une tactique? . . . . .	7
2.7	Qu'est ce qu'une preuve? . . . . .	8
2.8	Le prouveur . . . . .	9
2.9	Qu'est ce qu'un bouclage du prouveur? . . . . .	10
2.10	Qu'est ce qu'une commande? . . . . .	10
<b>3</b>	<b>Normalisation des obligations de preuve</b>	<b>13</b>
<b>4</b>	<b>Commandes interactives</b>	<b>15</b>
4.1	Abstract Expression . . . . .	15
4.2	Add hypothesis . . . . .	18
4.3	Arithmetic Proof . . . . .	20
4.4	Abstract predicate . . . . .	22
4.5	Apply rule . . . . .	23
4.6	Back . . . . .	30
4.7	Loop . . . . .	33
4.8	CurrentGoal . . . . .	39
4.9	CreateHyp . . . . .	40
4.10	Contradiction . . . . .	41
4.11	Special Contradiction . . . . .	43
4.12	Do cases . . . . .	44
4.13	Special do cases . . . . .	47

4.14 Deduction . . . . .	49
4.15 Display Term . . . . .	52
4.16 Use equality in hypothesis . . . . .	54
4.17 Force . . . . .	58
4.18 False hypothesis . . . . .	60
4.19 Forward . . . . .	62
4.20 Goto . . . . .	64
4.21 Goto with reset . . . . .	66
4.22 Global situation . . . . .	67
4.23 Graphical Trace . . . . .	70
4.24 Goto without save . . . . .	71
4.25 Help . . . . .	72
4.26 Logical Analysis . . . . .	73
4.27 Match goal . . . . .	75
4.28 Match Hypothesis . . . . .	76
4.29 Show litteral PO . . . . .	77
4.30 ModelChecking . . . . .	79
4.31 MiniProof . . . . .	83
4.32 Modus ponens on hypothesis . . . . .	84
4.33 Mono Lemma Prover . . . . .	86
4.34 Next . . . . .	89
4.35 Pmm compile . . . . .	90
4.36 Particularize hypothesis . . . . .	94
4.37 Predicate prover . . . . .	96
4.38 Prove . . . . .	100
4.39 PreviousPO . . . . .	106
4.40 Quit . . . . .	107
4.41 Reset PO . . . . .	108
4.42 Show reduced PO . . . . .	109
4.43 Repeat . . . . .	111
4.44 Suggest for exist . . . . .	113
4.45 Search hypothesis . . . . .	115
4.46 Submatch Goal . . . . .	118
4.47 Show Proof . . . . .	119
4.48 Submatch Hypothesis . . . . .	120
4.49 Save with question . . . . .	121
4.50 Search rule . . . . .	122
4.51 Simplify Set . . . . .	125

4.52 Step . . . . .	128
4.53 Save without question . . . . .	131
4.54 Try everywhere . . . . .	132
4.55 Proof by attempts . . . . .	138
4.56 User Simplification . . . . .	140
4.57 Well Definedness . . . . .	143
<b>5 Paramétrage du Prouveur</b>	<b>145</b>
5.1 Temps de Coupure Paramétrable . . . . .	145
5.2 Normalisation des formules $P \Rightarrow Q$ et $\neg P$ . . . . .	146
5.3 Paquetages de Règles Additionnelles . . . . .	147
5.4 Base de règles utilisateur . . . . .	148
5.5 Nombre Maximum d’Instanciation d’Hypothèses Quantifiées Universellement	149
5.6 Trace de règles utilisateur . . . . .	149
<b>6 Proof Manual Method : ajout de règles utilisateur</b>	<b>151</b>
<b>7 Patchprover : ajout direct de règles dans le prouveur</b>	<b>153</b>
<b>8 User Simplification : théories de simplification de l’utilisateur</b>	<b>155</b>
<b>9 User Pass : utilisation de passes configurables</b>	<b>157</b>
9.1 Présentation . . . . .	157
9.2 Filtres pour User_Pass . . . . .	158
<b>10 Système de trace</b>	<b>161</b>
10.1 Description . . . . .	161
10.2 Utilitaire . . . . .	162
<b>11 Liste des commandes disponibles</b>	<b>165</b>
<b>12 ANNEXE</b>	<b>169</b>



# Chapitre 1

## Introduction

Ce document constitue le manuel de référence du prouveur interactif (PRI) de l'Atelier B. Il contient l'ensemble des commandes utilisables, pour la version 4.0 de l'Atelier.

La vocation de ce manuel n'est pas de donner des techniques de preuve, mais d'indiquer à l'utilisateur la syntaxe et le domaine couvert par les commandes de preuve interactive.

Pour chaque fonction, seront indiqués la syntaxe à utiliser et le mode d'utilisation. Un exemple complètera la description de la fonction.

On trouvera aussi dans ce document :

- les notions de base, nécessaires pour le travail de preuve (voir chapitre 2 page 3)
- la normalisation des obligations de preuve (PO), effectuée par le prouveur et le générateur d'obligations de preuve (GOP) (voir chapitre 3 page 13)
- la présentation des possibilités d'enrichissement de la base de règles du prouveur (patchprover (voir chapitre 7 page 153), pmm (voir chapitre 6 page 151) )
- la liste des commandes disponibles, regroupées par thème et par ordre alphabétique (voir chapitre 11 page 165)



## Chapitre 2

# Notions de base

### 2.1 Qu'est ce qu'une obligation de preuve ?

Une obligation de preuve (PO) est constituée d'un but  $B$  et d'un ensemble d'hypothèses  $H$ . Démontrer une PO consiste à démontrer ce but  $B$ , en supposant que toutes les hypothèses de  $H$  sont vérifiées.

Les hypothèses  $H$  sont dites *hypothèses contextuelles*.

Si le but courant est de la forme  $P \Rightarrow Q$ , par application du principe de déduction,  $P$  va devenir une hypothèse et le nouveau but devient  $Q$ .

$P$  est alors appelé *hypothèse dérivée*.

Les *hypothèses courantes* sont constituées des hypothèses contextuelles et des hypothèses dérivées.

La *pile des hypothèses* contient l'ensemble des hypothèses contextuelles et des hypothèses dérivées, dans l'ordre dans lequel elles sont apparues.

### 2.2 Qu'est ce qu'un prédicat bien typé ?

Lorsque l'on saisit des expressions à l'attention du prouveur interactif (paramètres de commandes), il faut faire attention à ne pas introduire des erreurs de typage.

Par exemple :

- l'introduction du prédicat  $i = E$  alors que  $i$  et  $E$  sont typés comme suit :  $i \in NAT$  et  $E \in POW(NAT)$   
Le typage suivant est correct :  $i \in NAT$  et  $E \in NAT$ .
- l'introduction du prédicat  $E \leq F$  alors que  $E$  et  $F$  sont typés comme suit :  $E, F \in POW(NAT)$   
Le typage suivant est correct :  $E, F \in NAT$ .
- l'introduction du prédicat  $b = e_1 + e_2$  alors que  $b$ ,  $e_1$  et  $e_2$  sont typés comme suit :  $b \in BOOL$  et  $e_1, e_2$  appartenant à un ensemble énuméré  
Le typage suivant est correct :  $b, e_1, e_2 \in \mathbb{Z}$ .

D'une manière générale, une erreur de typage va introduire un but mal typé qui ne sera pas prouvable.

## 2.3 Qu'est ce qu'une expression bien définie ?

### 2.3.1 Présentation

Une expression mathématique est bien définie lorsqu'on peut lui donner un sens (voir Outil mdelta Manuel Utilisateur Version 1.0.). Dans le cas contraire on dira que l'expression est *dépourvue de sens*. On désigne par expression *potentiellement dépourvue de sens* toute expression nécessitant des conditions pour ne pas être dépourvue de sens.

Par exemple, soit l'expression :

$$y = \frac{x}{\frac{x+8}{c}} \quad (2.1)$$

Cette expression peut être vraie ou fausse, à la condition qu'elle soit bien définie. Si cette expression n'est pas bien définie, il est alors impossible de lui associer une valeur vrai ou faux. Cette mauvaise définition signifie qu'au moins un opérateur de l'expression a au moins un opérande qui n'appartient pas à son domaine de définition.

L'expression (3.1) est manifestement de nature arithmétique. On considère que l'expression est bien typée (opération réalisée par le vérificateur de types) et que y, x et c sont des entiers relatifs.

Les opérateurs apparaissant dans l'expression (3.1) sont :

- L'égalité  
Une égalité  $a=b$  est bien définie à condition que a et b soient bien définis
- l'addition  
Une addition  $a+b$  est bien définie à condition que a et b soient bien définis
- la division entière  
Une division entière  $a/b$  est bien définie si a et b sont bien définis et b est non nul

Il va donc falloir vérifier que :

- le dénominateur de  $x/(x+8/c)$  est non nul
- le dénominateur de  $8/c$  est non nul

La bonne définition de l'expression (3.1) passe par la démonstration des prédicats suivants :

$$(x+8)/c \neq 0 \quad (2.2)$$

$$c \neq 0 \quad (2.3)$$

Le contexte de l'expression doit contenir ces prédicats sous forme d'hypothèses ou doit permettre de les déduire.

Si tel n'est pas le cas, l'expression (3.1) est potentiellement mal définie.

On se reportera au tableau (1) pour la liste des expressions pouvant être mal définies.

### 2.3.2 Conditions de bonne définition

Les conditions de bonne définition sont recensées dans le tableau ci-dessous.

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

Tableau (1) : Expressions potentiellement dépourvues de sens

## 2.4 Qu'est ce qu'une règle ?

Une règle est une formule qui se présente sous la forme  $A \Rightarrow B$ .

A est appelé antécédent de la règle.

B est appelé conséquent de la règle.

A et B peuvent être des conjonctions de prédicats.

A peut être omis. Dans ce cas, la règle est dite atomique.

Une règle peut être :

— inductive (backward)

Si le but courant est B, alors pour prouver B, il suffit de prouver A.

A est sensé être plus simple que B ou, du moins, plus facilement prouvable que B.

— déductive (forward)

Si les hypothèses A apparaissent dans la pile des hypothèses, alors les hypothèses B sont générées et montées dans la pile, si elles n'existent pas déjà.

— de réécriture

Dans ce cas, B est de la forme  $C == D$ .

Si A est vérifié alors C est réécrit en D.

Ce type de règle ne s'applique que sur des sous-formules contenues dans le but courant, ou sur le but courant lui-même.

Par exemple, la règle `SimplifyIntMaxXY.3` :

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

peut s'appliquer sur le but :

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

pour le transformer en :

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

Les règles, contrairement aux hypothèses et au but, contiennent des jokers.

Un joker est une variable, qui peut prendre n'importe quelle valeur (littéral, expression, ...).

Si on lui affecte une valeur, on dit alors qu'il est instancié.

Un joker est représenté par une lettre de l'alphabet : on ne peut donc pas avoir plus de 52 jokers à l'intérieur d'une même règle (majuscules et minuscules).

## 2.5 Qu'est ce qu'une théorie ?

Une théorie consiste en un regroupement de règles, écrites en langage de théorie<sup>1</sup>.

Les règles ont pour nom  $t.n$ , avec

- $t$  : nom de la théorie,
- $n$  : index de la règle dans la théorie<sup>2</sup>.

Exemple :

```
THEORY th1 IS

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

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

END
```

Le prouveur tente toujours d'appliquer les règles d'index le plus élevé avant celles d'index plus faible (du "bas" de la théorie vers le "haut").

## 2.6 Qu'est ce qu'une tactique ?

Une tactique est une liste ordonnée de théories, qui détermine le parcours d'une base de règles<sup>3</sup> pour déterminer la ou les règles qui vont être appliquées.

Les théories sont classées en 2 catégories :

- Backward Le but à traiter est décomposé en sous-buts ou déchargé.
- Forward De nouvelles hypothèses sont générées.

---

1. voir le *Manuel de Référence du Logic Solver*

2. la règle positionnée en début de théorie correspond à l'index 1

3. Une base de règles est constituée d'un ensemble de théories.

Une tactique complète se présente comme une combinaison d’une tactique backward et d’une tactique forward :

```

<tactique>          ::= <tactique backward> , <tactique forward>
<tactique backward> ::= <tactique backward> ; <tactique backward>
                        <tactique backward> ~
                        nom de théorie

<tactique forward>  ::= <tactique forward> ; <tactique forward>
                        <tactique forward> ~
                        nom de théorie

```

La tactique forward est optionnelle.

Le parcours d’une tactique consiste donc à rechercher dans chaque théorie backward une règle qui peut s’appliquer. L’ordre de recherche correspond à l’ordre des théories listées. Si une règle peut s’appliquer, alors elle est appliquée et la recherche se poursuit avec la théorie suivante. Ce processus va se répéter jusqu’à épuisement des théories backward.

Si, au cours du parcours des théories backward, au moins une hypothèse a été générée, alors le processus décrit pour les théories backward va s’appliquer pour la liste des théories forward (s’il y en a), et ce à chaque montée d’hypothèse.

Une théorie ou un groupe de théories peut être “tildé”. Dans ce cas, il va y avoir tentative d’application d’une règle de cette théorie tant qu’il y a application de règles de cette théorie. Lorsqu’il n’y a plus de règles appliquées, la théorie suivante sera examinée.

Exemple :

$$((t1, t2) \sim, t3, t4 \sim)$$

Implicitement, la liste de théories la plus externe est “tildée”, c’est à dire que :

$$(t1; t2; t3; \dots; tn)$$

est équivalent à

$$(t1; t2; t3; \dots; tn) \sim$$

## 2.7 Qu’est ce qu’une preuve ?

Le prouveur de l’Atelier est composé d’un prouveur automatique et d’un prouveur interactif. Le prouveur automatique permet de tenter de démontrer automatiquement un ensemble d’obligations de preuve, en appliquant un ensemble donné de mécanismes généraux. Le prouveur est paramétrable selon sa force (force Rapide, force 0 à force 3). Plus la force est élevée, plus le temps de preuve s’accroît.

Le prouveur interactif permet à l’utilisateur d’aider le prouveur automatique dans son

travail de démonstration, en orientant la preuve (ajout d'hypothèses, preuve par contradiction, preuve par cas, ...).

Cette orientation se fait par l'intermédiaire de commandes interactives. Ces commandes sont appliquées pour une obligation de preuve donnée et sont sauvegardées pour cette PO. Elles constituent la ligne de commande.

Malgré tous nos efforts, le prouveur peut être parfois amené à boucler (voir chapitre 2.9 page 10), c'est à dire adopter un comportement itératif ininterrompu, fortement divergent. Les probabilités d'apparition d'un tel phénomène augmentent avec le niveau de force.

## 2.8 Le prouveur

Le prouveur automatique de l'Atelier est composé de :

- mécanismes  
Ces mécanismes doivent orienter la preuve, de manière à démontrer les buts des obligations de preuve. Ils permettent le déclenchement des règles de la base de règles.
- règles  
Ces règles sont celles de la base de règles du prouveur (voir chapitre 2.4 page 5). Elles peuvent aussi avoir été ajoutées par l'opérateur (règles manuelles), par l'intermédiaire de :
  - fichier **pmm** (Proof Manual Method) pour un composant donné (voir chapitre 6 page 151)
  - fichier **PatchProver** pour un projet complet (voir chapitre 7 page 153)  
Le fichier PatchProver doit se situer dans le répertoire bdp du projet concerné  
Attention !! L'utilisation de règles manuelles remet en cause l'intégrité de la preuve (des règles fausses peuvent permettre de prouver des obligations de preuve fausses). Il faudra donc, en cas d'ajout de règles, procéder à une démonstration rigoureuse de celles-ci et s'astreindre à une relecture par une tierce personne.
- Ressources  
On peut paramétrer le prouveur à l'aide d'un fichier de ressources en positionnant les ressources :
  - **Keep\_Non\_Simplified\_Hypothesis** qui, lorsqu'elle est à **TRUE**, permet de conserver en hypothèses, les prédicats simplifiés (par le prouveur) ainsi que leur version non simplifiée et lorsqu'elle vaut **FALSE**, indique au prouveur qu'il ne faut conserver que les versions simplifiées des hypothèses. Par défaut, elle vaut **TRUE**.
  - **Time\_Out** dont la valeur est un entier exprimant le temps de coupure des prouveurs satellites (prouveur de prédicats et prouveur mono-lemme) en *User\_Pass* et *Replay* i.e. le temps au bout duquel, on décide de stopper l'action des prouveurs satellites. Par défaut, il vaut 300 secondes.
  - **Use\_Rule\_Package** qui lorsqu'elle est positionnée à la valeur p1 permet d'utiliser des règles mathématiques supplémentaires.
  - **Max\_Number\_Of\_Universal\_Hypothesis\_Instantiation** qui est représenté par un quadruplet d'entiers naturels dont les valeurs correspondent respectivement

au nombre maximum d'instanciation d'hypothèses quantifiées universellement en force 0, 1, 2 et 3.

## 2.9 Qu'est ce qu'un bouclage du prouveur ?

Par exemple, on utilise la règle :

$$a*a == a*a*a/a$$

sur le but à prouver

$$cc(vv) = vv*vv$$

On obtiendra successivement les buts suivant :

$$\begin{aligned} cc(vv) &= vv*vv*vv/vv \\ cc(vv) &= (vv*vv*vv/vv)*vv/vv \\ &\dots \end{aligned}$$

Le noyau de preuve produit les messages suivants :

```
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.

...
```

Les messages *krt : sequence memory short* sont générés par le kernel qui alloue dynamiquement la mémoire qui lui fait défaut.

Cet exemple est simple. Des bouclages peuvent se produire pour des groupes de règles et peuvent être plus difficiles à détecter a priori.

## 2.10 Qu'est ce qu'une commande ?

Une commande peut-être :

— simple

Elle est composée d'une unique commande interactive. Les commandes simples sont listées à la fin du document (voir chapitre 11 page 165).

— composée

Une commande composée est une liste de commandes simples séparées par un `&`. Les différentes commandes de la liste vont être appliquées successivement. Lorsque l'on utilise la commande **rr** (voir chapitre 4.43 page 111) et que l'on vient de saisir une commande composée, cette commande composée va être intégralement rejouée. Exemple de commande composée :

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



## Chapitre 3

# Normalisation des obligations de preuve

Les hypothèses et les buts sont normalisés par le Générateur d'obligations de preuve et le prouveur. La normalisation permet de transformer une expression en une expression normale, qui sera par la suite utilisée, sous cette forme, dans l'ensemble des règles relatives à cette expression.

Cela permet de limiter le polymorphisme des règles de la base de règles du prouveur, donc leur nombre.

Les formes normales retenues sont :

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

Il convient, lors de l'écriture d'une règle, de vérifier que cette règle est bien normalisée. Dans le cas contraire, la règle va être normalisée lors de son chargement et peut ne plus pouvoir s'appliquer.

Par exemple, la règle suivante :

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

va être normalisée en

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

Or la garde **btest** n'accepte que des paramètres de la forme **a op b**, où **a** et **b** sont des entiers. Cette règle ne va donc jamais s'appliquer. Il aurait fallu écrire plutôt :

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

## Chapitre 4

# Commandes interactives

### 4.1 Abstract Expression

#### ABSTRACTION D'UNE EXPRESSION

##### Syntaxe

**ae**( $EX, ID$ )

avec :

- $EX$  est une expression mathématique dont au moins une occurrence apparaît dans le but courant,
- $ID$  est un identificateur (au sens du langage B) qui n'est libre ni dans le but ni dans les hypothèses.

##### Utilisation

Cette commande permet de générer le but  $\Delta_e(EX) \wedge (EX = ID \Rightarrow [EX := ID]G)$  où  $\Delta_e(EX)$  représente le lemme de bonne définition de l'expression  $EX$  (voir chapitre 2.3 page 4) et  $[EX := ID]G$  le but courant dans lequel ont été remplacées les diverses occurrences de l'expression  $EX$  (apparaissant dans le but) par l'identificateur  $ID$ .

Il est nécessaire de vérifier la bonne définition de l'expression  $EX$  car sortie de son contexte (le but courant), elle peut éventuellement être mal définie.

Supposons en effet que sous l'hypothèse  $xx + 1 \leq 1$  on ait le but  $xx + 1 \leq 1 \vee \max(1..xx) = xx$ . Ce but est bien défini puisque  $xx + 1 \leq 1$  est bien défini et  $\text{not}(xx + 1 \leq 1) \Rightarrow \Delta_p(\max(1..xx) = xx)$  i.e.  $\text{not}(xx + 1 \leq 1) \Rightarrow \Delta_e(\max(1..xx))$  soit encore  $\text{not}(xx + 1 \leq 1) \Rightarrow 1..x \cap \text{NAT} \in \text{FIN}(\text{NAT}) \wedge \text{not}(1..xx) = \emptyset$  (d'après la bonne définition de  $\vee$  et de  $\max$ ). Si on effectue **ae**( $\max(1..xx), \text{MAX}$ ) sans précaution concernant la bonne définition, on obtient le but  $\max(1..xx) = \text{MAX} \Rightarrow xx + 1 \leq 1 \vee \text{MAX} = xx$ . Puis une déduction (par **dd**) permet de faire monter en hypothèses la formule  $\max(1..xx) = \text{MAX}$  qui est maintenant dépourvue de sens puisqu'on a l'hypothèse  $xx + 1 \leq 1$  (et donc  $1..xx = \emptyset$ ).

Remarquons enfin que le prouveur tente une preuve automatique du lemme de bonne

définition de l'expression  $EX$  et si la preuve échoue, le lemme de bonne définition devra être démontrée interactivement.

### Exemple1

Soit le but à démontrer suivant :

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

L'opérateur désire remplacer l'expression  $p2 + 1$  par  $pp2$  :

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

On obtient le but suivant (le lemme de bonne définition de  $p2 + 1$ , réduit à **btrue**, est démontré automatiquement par le prouveur) :

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

On peut ensuite monter en hypothèse l'égalité  $pp2 = p2 + 1$  (en utilisant la commande **deduction dd**) et on vérifie cela à l'aide de la commande **Search Hypothesis** :

```
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
```

**Exemple2**

Soit le but à démontrer suivant :

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

L'utilisateur souhaite remplacer l'expression  $\max(aa..bb)$  par  $MAXI$  :

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

Le lemme de bonne définition associé à l'expression  $\max(aa..bb)$  est  $\text{not}(aa..bb = \emptyset) \wedge aa..bb \cap NATURAL : FIN(NATURAL)$ . Le prouveur automatique ne parvient pas à le démontrer, il génère donc le premier sous-but suivant :

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

Une fois ce but démontré (par exemple par  $\text{ah}(aa \leq bb) \ \& \ \text{pp}(\text{rp}.0)$ ), le prouveur génère le sous-but suivant :

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

La commande **pr** suffit alors à démontrer ce but. La bonne définition de l'expression  $\max(aa..bb)$  est donc démontrée. Le prouveur peut donc effectuer la substitution de  $\max(aa..bb)$  par  $MAXI$  dans le but de départ et générer le nouveau but :

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

## 4.2 Add hypothesis

### AJOUT D'UNE HYPOTHÈSE

#### Syntaxe

**ah**( $P$ )

avec :

- $P$  est un prédicat

#### Utilisation

Cette commande permet d'ajouter le prédicat  $P$  dans la pile des hypothèses.

Dans les hypothèses courantes, le prédicat  $P$  doit :

- être bien défini (voir chapitre 2.3 page 4),
- être bien typé (voir chapitre 2.2 page 3),
- pouvoir se déduire des hypothèses courantes.

Si la preuve courante était

prouver  $B$  sous les hypothèses  $h_1, \dots, h_n$

alors le prouveur va tenter de prouver successivement

prouver  $P$  sous les hypothèses  $h_1, \dots, h_n$

puis

prouver  $B$  sous les hypothèses  $h_1, \dots, h_n, P$

Cette preuve est effectuée avec la force de preuve courante.

Après avoir exécuté la commande **ah**(**P**), le but courant devient  $P$ . Si après la commande **pr** (voir chapitre 4.38 page 100), le but courant est toujours  $P$ , l'hypothèse  $P$  n'a pas pu être prouvée.

Lorsque l'hypothèse  $P$  a été prouvée, le but devient  $P \Rightarrow B$ . L'utilisateur a alors la possibilité soit de monter directement l'hypothèse  $P$  (commande **dd** (voir chapitre 4.14 page 49)), soit de lancer le prouveur automatique qui montera l'hypothèse  $P'$ , obtenue après traitements sur  $P$ .

Si  $P$  ne peut pas être prouvé avec la force courante, on peut essayer une force supérieure. Toute la ligne de commande sera alors réexécutée avec la nouvelle force.

Puisque cette commande n'est pas protégée contre le mauvais typage et la mauvaise définition, l'utilisateur doit donc vérifier que l'hypothèse ajoutée est bien typée et bien définie.

Ceci peut être vérifié *a posteriori* à l'aide l'outil **mdelta** (cf. Manuel Utilisateur Version 1.0.).

#### Exemple

Soit la situation suivante :

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

L'opérateur désire ajouter l'hypothèse  $xx + yy : 2..20$ . Il exécute la commande **ah** :

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

Le nouveau but devient :

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

Cette hypothèse est à prouver avant de pouvoir poursuivre. L'opérateur lance le coeur de preuve :

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

L'hypothèse  $xx + yy : 2..20$  a été prouvée : le but devient  $xx + yy : 2..20 \Rightarrow \text{but courant}$ .

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

En utilisant la commande **pr** (voir chapitre 4.38 page 100) ou **dd** (voir chapitre 4.14 page 49), la preuve peut alors se poursuivre avec la nouvelle hypothèse.

## 4.3 Arithmetic Proof

### APPEL DU PROUVEUR ARITHMÉTIQUE

#### Syntaxe

**ap**  
**ap(n)**

avec :

- $n$  est une valeur numérique permettant de limiter le fonctionnement du mécanisme.  
Si cette valeur n'est pas précisée, la valeur de 400 est prise

#### Syntaxe

Le prouveur arithmétique est un mécanisme destiné à rechercher une contradiction dans un ensemble d'inéquations. Cette contradiction est recherchée en créant de nouvelles inéquations par combinaison linéaire. Le nombre d'inéquations est limité afin d'éviter un bouclage du mécanisme.

La commande **ap** permet d'appeler le mécanisme sur l'obligation de preuve courante. Le mécanisme va travailler sur les inéquations que comporte la pile des hypothèses. Si le but courant est une inéquation de la forme  $a \leq b$ , alors l'inéquation  $a > b$  est ajoutée à la liste des inéquations sur lesquelles va travailler le mécanisme.

#### Exemple 1

Soit l'obligation de preuve suivante :

```
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
```

Etant donné la forme du but et le nombre d'inéquations en hypothèses, l'utilisation de la commande **ap** est recommandé.

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

Le but courant est alors déchargé.

### Exemple 2

Cet exemple montre le comportement de la commande lorsque le mécanisme échoue dans son travail. Soit l'obligation de preuve suivante :

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

L'obligation de preuve étant fausse, la commande ne décharge pas le but.

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

La commande **ap** n'est pas enregistrée, le but courant n'est pas modifié.

## 4.4 Abstract predicate

### ABSTRACTION D'UN PRÉDICAT

#### Syntaxe

**aq**( $P$ )

avec :

—  $P$  est un prédicat

#### Utilisation

Cette commande permet de créer une nouvelle variable nommée  $PP$  ou  $PP\$i$  équivalente au prédicat  $P$  contenu dans le but courant et de remplacer le prédicat par cette nouvelle variable dans le but courant.

## 4.5 Apply rule

APPLICATION D'UNE RÈGLE OU UNE THÉORIE UTILISATEUR

### Syntaxe

**ar**(T)  
**ar**(r, M)

avec :

- T est un identifiant de théorie ou de tactique (voir chapitre 2.6 page 7),
- r est un identifiant de règle (Theorie.Numero)
- M est un mode qui dépend du type de règle :
  - Règles n'effectuant pas de réécriture :
    - M=**Once** : application de la règle une seule fois
    - M=**Multi** : application de la règle, tant qu'elle s'applique
    - M=**Fwd** : application de la règle en forward.
 La règle doit alors être de la forme
 
$$h_1 \wedge \dots \wedge h_n \Rightarrow g$$
 Pour tous les ensembles de n hypothèses valides qui coïncident avec  $h_1, \dots, h_n$ , l'hypothèse g correspondante est générée.
  - M=**Fwd(P)** : Idem cas précédent, mais on impose en plus que l'une des hypothèses  $h_1, \dots, h_n$  coïncide<sup>1</sup> avec P
- Règles de réécriture : M doit être de la forme A ou A.B, avec :
  - A = **AllHyp** :réécriture de toutes les hypothèses.
  - A = **Goal** : réécriture du but.
  - A = **Hyp(f)** :réécriture des hypothèses qui coïncident avec f.
  - B peut être **Part(g)**. Dans ce cas, la réécriture est restreinte aux sous formules des formules sélectionnées qui coïncident avec g.

### Utilisation

**ar** est une commande qui permet d'appliquer une règle ou une théorie sur différentes parties du lemme à prouver.

Pour des règles de réécriture, l'argument M permet d'agir de manière contrôlée sur n'importe quelle partie de l'obligation de preuve, but ou hypothèse.

---

1. On dit que H coïncide avec P si H et P ont même forme (c'est à dire qu'il est possible d'instancier les jokers de P pour obtenir exactement H).

Par exemple,  $xx = yy + 3 - \min(4..7)$  coïncide avec :

- a = b  
 car a peut être remplacé par xx et b par  $yy + 3 - \min(4..7)$
- c = d - e  
 car c peut être remplacé par xx, d par  $yy + 3$  et e par  $\min(4..7)$ .

Les règles utilisées sont celles de la base de règles du prouveur, des règles éventuelles contenues dans le fichier pmm (voir chapitre 6 page 151) associé au composant, ainsi que les règles contenues dans le fichier PatchProver (voir chapitre 7 page 153).

Pour accéder à la base de règles il faut cliquer sur le menu "Display/print" et ensuite sur le bouton "Display Rules Database" de la fenêtre **INTERACTIVE PROOF** du prouveur interactif.

Application en mode backward (M=Once ou M=Multi) :

- si la règle est de la forme  

$$a_1 \wedge \dots \wedge a_n \Rightarrow c$$
et que le but courant est de la forme  $c$ , alors il y a division de la preuve en  $n$  sous-buts  

$$a_1, \dots, a_n$$
- si la règle est de la forme  

$$a_1 \wedge \dots \wedge a_n \Rightarrow c == d$$
et que  $c$  est une sous-formule<sup>2</sup> du but courant, alors il y a division de la preuve en  $n$  sous-buts  

$$a_1, \dots, a_n$$
Si les  $n$  sous-buts sont prouvés, alors  $c$  se réécrit en  $d$ .

Application en mode forward (M=Fwd) :

- si la règle est de la forme  

$$a_1 \wedge \dots \wedge a_n \Rightarrow c$$
les antécédents  $a_1, \dots, a_n$  sont cherchés dans les hypothèses pour générer la nouvelle hypothèse  $c$ .  
Le procédé peut boucler facilement (par exemple : la règle boucle car elle se réapplique sur son conséquent), c'est pourquoi, l'option Fwd(P) permet d'imposer que l'une des hypothèses  $a_i$  trouvée coïncide avec  $P$ .

Application d'une tactique :

- si l'on n'utilise que des théories *Backward*  
La commande sera :  

$$\text{ar}(\text{tb1}; \text{tb2}; \dots; \text{tbn})$$
- si l'on utilise des théories *Backward* et des théories *Forward*  
La commande sera :  

$$\text{ar}((\text{tb1}; \text{tb2}; \dots; \text{tbn}; \text{DED}), (\text{tf1}; \text{tf2}; \dots; \text{tfp}))$$

---

2. Par exemple,  $xx + yy$  est une sous-formule de  $0 \leq \min(1..xx + yy)$

La théorie *DED* (théorie native du kernel réalisant la montée des hypothèses dans la pile) est obligatoire. Les théories *Forward* ne sont appelées qu'à l'occasion de chaque hypothèse montée. Les théories *Backward* génèrent des buts dérivés  $P \Rightarrow Q$  mais ne font pas monter les hypothèses  $P$ . Il faut donc leur associer une théorie permettant d'effectuer une déduction directe (*DED*). Lorsque la commande **ar** se terminera, les hypothèses seront associées au but courant, qui sera alors :

Hypothèses générées  $\Rightarrow$  But courant

Bien entendu, les théories peuvent être "tildées". Par exemple :

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

Implicitement

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

est équivalent à

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

Après l'application de la commande **ar**, si de nouvelles hypothèses sont générées, le but est de la forme  $H \Rightarrow B$  (l'opérateur peut voir les nouvelles hypothèses générées). Il faut exécuter la commande **pr** (voir chapitre 4.38 page 100) pour relancer la preuve et faire monter ces hypothèses.

Attention ! Si des règles utilisateur (pmm, PatchProver) sont utilisées, la validité de la preuve peut être remise en question. Il faut alors effectuer une démonstration mathématique pour chacune de ces règles.

### Exemple 1

Soit la situation suivante :

Hypothesis

xx: 1..10 &

yy: 1..10 &

zz: 1..100

Goal

xx+yy-1: 1..100

L'opérateur utilise le fichier *pmm* suivant :

```

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

```

La théorie *test* est lue et compilée, grâce à la commande **pc** (voir chapitre 4.35 page 90).

```

PRI> pc
Loading theory test

```

La règle *test.1* est alors appliquée en mode *forward* (génération d'hypothèses).

```

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

```

5 nouvelles hypothèses ont été générées. Le but devient :

```

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

```

Grâce à la commande **dd** (voir chapitre 4.14 page 49).

```

PRI> dd
Starting Deduction

```

les hypothèses sont ensuite montées dans la pile des hypothèses.

```
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
```

La règle *test.2* est alors utilisée en mode *backward*.

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

La règle s'applique (on vérifie que la ligne de commande contient *ar(test.2, Once)*) et les deux sous-buts  $1 \leq xx + yy - 1$  et  $xx + yy - 1 \leq 100$  vont être traités. Le premier sous-but est à prouver :

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

Le prouveur automatique est appelé une première fois :

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

Le premier sous-but est déchargé. Le second sous-but devient le but courant.

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

Par appel au prouveur automatique

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

le second sous-but est déchargé et l'obligation de preuve est prouvée, moyennant la justesse des règles contenues dans le fichier *pmm* !

La ligne de commande est finalement :

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

## Exemple 2

Soit la situation suivante :

```
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
```

associée au fichier *pmm* suivant :

```
THEORY test IS

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

END
&
THEORY testbis IS

  a = b &
  b: E
=>
  a: E

END
```

L'opérateur essaye d'appliquer la théorie backward *test* au but courant. Les hypothèses seront montées par la théorie *DED*. En cas de génération d'hypothèses, la théorie forward *testbis* sera alors essayée.

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

La règle *test.1* contient une garde permettant d'imprimer un message indiquant son déclenchement (*Application de test.1*). La règle *testbis.1* s'est déclenchée lors de la montée de l'hypothèse  $zz = e2$  et a permis de générer l'hypothèse  $zz \in \{e1, e2, e3, e4, e5\}$ . Toutes les hypothèses générées sont finalement rangées comme antécédent du but courant.

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

## 4.6 Back

RECU L D'UN PAS DE PREUVE

### Syntaxe

**ba**  
**ba(n)**

avec :

- n vaut **Node** pour reculer au noeud précédent dans l'arbre de preuve.
- n prend une valeur numérique lorsqu'on sait de combien de commandes on veut reculer.

### Utilisation

Cette commande provoque le recul d'un pas de preuve : l'effet de la dernière commande est annulé. Le but courant, les hypothèses et la ligne de commande reprennent leur état précédent. Il se peut que la commande **ba** ne produise pas de déplacement, si l'on est en début de ligne de commande (aucune commande exécutée).

Cette commande a un effet sur l'état de l'obligation de preuve : si celle-ci vient d'être prouvée et que l'on applique la commande **ba**, on va se replacer juste avant la dernière commande qui finit la démonstration et l'obligation de preuve redevient non prouvée.

La commande Back peut reculer du nombre de pas spécifié lors de l'appel de la commande.

Le paramètre **Node** fait reculer l'état de l'obligation de preuve, jusqu'au précédent niveau d'indentation dans la ligne de commande. Ce paramètre donne un Back très rapide car aucune commande n'est rejouée.

**ba** ne s'applique pas sur une commande **ff** (voir chapitre 4.17 page 58).

### Exemple 1

Soit l'obligation de preuve dont la ligne de commande est la suivante :

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

Si l'on applique la commande **ba**, on vérifie que la dernière commande (**pr**) a été supprimée :

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

En recommençant la même opération, on vérifie que la dernière commande (**pr**) a été supprimée :

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

On peut aussi reculer de plusieurs pas en une seule commande :

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

Dans ce cas, la ligne de commande devient :

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

**Exemple 2**

Si la ligne de commande est :

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

Après application de la commande **ba(Node)**, la ligne de commande devient :

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

## 4.7 Loop

APPLICATION DE COMMANDES INTERACTIVES LORS D'UNE DIVISION DE LA PREUVE

### Syntaxe

**bb(f)**

avec :

- L est une suite de commandes de la forme :  
 $c1 \ \& \ c2 \ \& \ \dots \ \& \ cn \ \& \ ll((d1 \ \& \ d2 \ \& \ \dots \ \& \ dm), \dots, (z1 \ \& \ z2 \ \& \ \dots \ \& \ zp))$

### Utilisation

Cette commande permet d'appliquer une suite de commandes sur toutes les branches d'une partie de l'arbre de preuve, lorsqu'il y a division de la preuve. Elle évite ainsi à l'utilisateur de saisir et d'exécuter plusieurs fois la même suite de commandes.

Si la série de commandes ne réussit pas à prouver une des branches, la boucle s'arrête.

La commande **bb** s'applique aussi lorsqu'il n'y a pas de division de preuve, mais l'intérêt de **bb** est, dans ce cas là, bien moindre.

La présence de **ll** au sein de la boucle permet d'appliquer des commandes interactives, dans le cas où une branche de preuve ne peut pas être prouvée automatiquement. Dans ce cas, les commandes  $(d1 \ \& \ d2 \ \& \ \dots \ \& \ dm)$  vont être essayées. Si la preuve de la branche aboutit, on change de branche et l'on réapplique  $c1 \ \& \ c2 \ \& \ \dots \ \& \ cn$ . Si la preuve n'aboutit pas, ce sont les commandes suivantes qui seront essayées. Ce processus est itéré jusqu'à ce que la preuve de la branche aboutisse ou que les commandes  $z1 \ \& \ z2 \ \& \ \dots \ \& \ zp$ .

### Exemple 1

Soit l'obligation de preuve suivante :

```
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
```

On peut exécuter une preuve par cas :

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

Pour éviter de saisir 5 fois la commande **pr**, on peut exécuter :

```
bb(pr)
```

On obtient alors les messages :

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

et l'obligation de preuve est démontrée.

La fenêtre Commands (Executed / Next) contient alors :

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

Chaque  $fc(1)$ <sup>3</sup> indique le début d'exécution de la commande **pr** (paramètre de la commande de boucle). Le numéro (ici 1) indique le nombre de boucles imbriquées.

On retrouve bien les 5 cas de la preuve par cas ( $fc(1)$ ).

**Exemple 2**

Soit l'obligation de preuve suivante :

```

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

```

On peut exécuter une preuve par cas :

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

Si l'on exécute la commande :

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

on obtient :

```

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

```

et l'obligation de preuve est démontrée.

La fenêtre Commands (Executed / Next) contient alors :

```
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
```

Chaque `fc()`<sup>4</sup> indique le début d'exécution d'une suite de commandes (paramètre de la commande de boucle). Le numéro (1, 2 ou 3) indique le nombre de boucles imbriquées.

On retrouve bien les 5 cas de la preuve par cas (`fc(3)`).

Après sauvegarde, la ligne de commande sauvegardée est :

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

La commande **ll()** a été ajoutée à la sauvegarde et contient les éventuelles commandes interactives ajoutées par l'utilisateur, au cours de la preuve et à l'intérieur de boucles. Dans ce cas, aucune commande n'a été ajoutée et on obtient donc **ll(?)**.

Cette ligne de commande peut d'ailleurs être saisie directement par l'utilisateur : elle conduira au même résultat. Il faut par contre que la commande **ll()** soit à la fin de la ligne de commandes contenue dans **bb()**.

Par exemple, la ligne de commandes suivant est incorrecte :

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

et donne à l'exécution :

```
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
```

Par contre, la ligne de commandes suivante est correcte :

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

et donne à l'exécution :

```
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
```

On s'aperçoit que la commande **dd** contenue dans **ll** n'a pas été exécutée.

## 4.8 CurrentGoal

AFFICHAGE DU BUT COURANT

### Syntaxe

**cg**

### Utilisation

Cette commande permet d'afficher le but de preuve courant. Elle correspond ainsi à la fenêtre “Current Goal” de l'interface Motif.

### Exemple

Soit le but courant suivant à démontrer :

```
toto = plus(nn)
```

L'utilisateur recherche alors les hypothèses se rapportant à toto, en utilisant **sh(toto)** :

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

Pour visualiser le but courant, l'utilisateur a deux choix : manipuler l'interface ou utiliser la commande **CurrentGoal** qui provoque l'affichage suivant :

```
Goal  
  toto = plus(nn)
```

## 4.9 CreateHyp

### CRÉATION D'HYPOTHÈSES

#### Syntaxe

**ch**

#### Utilisation

Ce mécanisme provoque la création d'hypothèses en rapport avec la forme du but. Après son application, le but se présente sous la forme d'une implication, les hypothèses générées étant en antécédent de l'implication.

#### Exemple

Soit une obligation de preuve dont le but courant est :

```
toto = plus(nn)
```

L'application de la commande **ch**

```
PRI> ch
```

donne le but suivant :

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

L'antécédent de cette implication contient les hypothèses générées par la commande.

L'utilisateur peut ensuite effectuer le traitement qu'il désire sur ces hypothèses.

## 4.10 Contradiction

### TENTATIVE DE PREUVE PAR CONTRADICTION

#### Syntaxe

`ct`

#### Utilisation

Cette commande permet de tenter une preuve par contradiction.

Si le but courant est  $B$ , alors il est transformé en :

$$\neg B \Rightarrow bfalse$$

Il faut alors que les hypothèses, complétées par  $\neg B$ , permettent de générer  $bfalse$ .

Dans ce cas, on obtient :

$$bfalse \Rightarrow bfalse$$

ce qui est vrai.

La preuve par contradiction peut s'utiliser notamment :

- si le but est de la forme  $\neg P$
- s'il y a plusieurs hypothèses contradictoires

#### Exemple

Soit l'obligation de preuve suivante :

```
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)
```

On tente une preuve par contradiction, vu la forme du but.

```
PRI> ct
Starting Contradiction
```

$\neg\neg e2 = e5$  est simplifié en  $e2 = e5$  puis passe en hypothèse. Le but devient *bfalse*.

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

Un appel au prouveur automatique est effectué ensuite.

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

La ligne de commande devient alors :

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

## 4.11 Special Contradiction

TENTATIVE DE PREUVE PAR CONTRADICTION SANS MONTÉE AUTOMATIQUE D'HYPOTHÈSES

### Syntaxe

`cts`

### Utilisation

Cette commande permet de tenter une preuve par contradiction, sans montée automatique d'hypothèses.

Si le but courant est  $B$ , alors il est transformé en :

$$\neg B \Rightarrow \text{bfalse}$$

Il faut alors que les hypothèses, complétées par  $\neg B$ , permettent de générer  $\text{bfalse}$ .

Cette commande est identique à `ct`, sauf que les hypothèses générées ne sont pas automatiquement montées.

### Exemple

Soit l'obligation de preuve suivante :

```
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)
```

On tente une preuve par contradiction, vu la forme du but.

```
PRI> ct
Starting Contradiction
```

$\neg\neg e2 = e5$  est simplifié en  $e2 = e5$ . Le but devient :

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

## 4.12 Do cases

TENTATIVE DE PREUVE PAR CAS

### Syntaxe

**dc(f)**  
**dc(v,E)**

avec :

- f est un prédicat correctement typé par rapport au contexte des hypothèses actuelles,
- v est une variable et E est le nom d'un ensemble en extension (maximum de 50 éléments)

### Utilisation

Cette commande permet le déclenchement d'une preuve par cas.

Si le but à prouver est B :

- si X est un prédicat,  
la preuve se décompose en 2 sous-buts :  

$$X \Rightarrow B$$
et  

$$\neg X \Rightarrow B$$
- si X est de la forme v,E  
le but devient une conjonction de n buts correspondants aux différentes valeurs possibles de v dans E. Si l'hypothèse  $v \in E$  n'existe pas telle quelle,  $(v \in E)$  est ajouté, par le prouveur, à cette conjonction afin que la preuve commence sur ce but préliminaire  
Remarquons que l'utilisateur doit vérifier que le prédicat  $v \in E$  est bien typé (voir chapitre 2.2 page 3) et bien défini (voir chapitre 2.3 page 4).  
La preuve à venir dépend de la forme de l'expression v,E :
  - E est de la forme  $\{e_1\} \cup \dots \cup \{e_n\}$   
La preuve de B est alors remplacée par :  

$$(v = e_1 \Rightarrow B) \wedge \dots \wedge (v = e_n \Rightarrow B)$$
  - E est de la forme  $(\{e_1\} \cup \dots \cup \{e_n\}) \times \{f\}$   
La preuve de B est alors remplacée par :  

$$(v = (e_1 \mapsto f) \Rightarrow B) \wedge \dots \wedge (v = (e_n \mapsto f) \Rightarrow B)$$
  - E est de la forme  $(\{e_1\} \cup \dots \cup \{e_n\}) \times F$   
où F n'est pas de la forme  $\{f\}$   
La preuve de B est alors remplacée par :  

$$(v = (\{e_1\} \times F) \Rightarrow B) \wedge \dots \wedge (v = (\{e_n\} \times F) \Rightarrow B)$$

## Exemple 1

Soit la situation suivante :

```
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
```

Du fait de la présence des deux hypothèses  $xx : 1..5 \Rightarrow yy = 10$  et  $xx : 6..10 \Rightarrow yy = 1$ , l'opérateur décide de lancer une preuve par cas, pour le prédicat  $xx : 1..5$ .

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

Le premier but traité est donc :

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

Le prouveur va donc tenter de prouver le but courant, d'abord sous l'hypothèse  $xx : 1..5$ . Pour cela, l'opérateur applique la commande **pr** (voir chapitre 4.38 page 100).

```
PRI> pr
```

Le but est prouvé par le prouveur automatique. L'autre cas à traiter est donc  $not(xx : 1..5)$ .

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

L'opérateur lance de nouveau le prouveur automatique.

```
PRI> pr
```

La PO est prouvée et sa ligne de commande est :

```

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

```

## Exemple 2

Considérons maintenant la preuve par cas, dans le cas d'un ensemble énuméré.  
Soit la situation suivante :

```

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)

```

L'opérateur lance une preuve par cas, relativement à  $xx$ , qui appartient à l'ensemble énuméré  $ENS$ .

```

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

```

Le prouveur interactif va donc déclencher 5 preuves successives :

```

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)

```

## 4.13 Special do cases

TENTATIVE DE PREUVE PAR CAS SUR FORMULE DISJONCTIVE

### Syntaxe

**dc**s(**A or B**)

avec :

- (**A or B**) est un prédicat disjonctif

### Utilisation

Cette commande permet le déclenchement d'une preuve par cas.

Si le but à prouver est *B* :

- si *X* est un prédicat disjonctif de *n* disjonctions, le but devient une conjonction de *n* buts. Chacun de ces buts est la conjonction d'un des termes disjonctifs du prédicat *X* et de la négation des autres termes. Si le prédicat *X* n'est pas en hypothèse, le premier but à démontrer est ce prédicat.

Par exemple si *X* est  $A \vee \text{not}(B) \vee C$  et que le but est *But*, les nouveaux but deviennent :

- $A \wedge B \wedge \text{not}(C) \Rightarrow \text{But}$
- $\text{not}(A) \wedge \text{not}(B) \wedge \text{not}(C) \Rightarrow \text{But}$
- $\text{not}(A) \wedge B \wedge C \Rightarrow \text{But}$
- si *X* n'est pas un prédicat disjonctif, la commande **dc**s se comporte comme la commande **dc**.

### Exemple

Considérons la preuve par cas, dans le cas d'un prédicat disjonctif.

Soit la situation suivante :

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

L'utilisateur lance une preuve par cas pour pouvoir pleinement utiliser l'hypothèse  $xx = 1 \text{ or } xx = 10 \text{ or } xx = 4$ .

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

Puisque le prédicat disjonctif provient des hypothèses, aucune vérification à son sujet n'est nécessaire. Le prouveur va déclencher 3 preuves successives :

```
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
```

Chacun de ces but est facilement déchargé par le commande **pr** (voir chapitre 4.38 page 100).

## 4.14 Deduction

### DÉDUCTION DIRECTE

#### Syntaxe

**dd**  
**dd(i)**

avec :

— *i* vaut 0, 1, 2 ou 3

#### Alias

**d0** est équivalent à **dd(0)**  
**d1** est équivalent à **dd(1)**  
**d2** est équivalent à **dd(2)**

#### Utilisation

Cette commande permet de réaliser une déduction directe : si le but courant est de la forme  $P \Rightarrow Q$ , alors le prouveur tente la preuve de  $Q$  sous l'hypothèse  $P$ .

L'hypothèse  $P$  est montée dans la pile des hypothèses, sans passer par le prouveur automatique (en particulier, les simplifications ne sont pas effectuées) et le but courant devient  $Q$ .

Il peut être intéressant, dans certains cas, d'utiliser **dd** car le prouveur automatique peut normaliser, d'une manière non souhaitée par l'opérateur, une hypothèse ajoutée par la commande **ah** (voir chapitre 4.2 page 18).

**dd** est donc parfois utilisé après **ah** pour faire monter la nouvelle hypothèse telle quelle.

L'argument de **dd(i)** permet de paramétrer finement les traitements qui sont réalisés sur les hypothèses montantes. Le paramètre *i* représente la force de preuve utilisée temporairement pendant la montée des hypothèses.

Par exemple, **dd(1)** correspond à la montée des hypothèses en force 1.

#### Exemple

Soit la situation suivante :

<p><b>Hypothesis</b>  xx: 1..10 &amp;  yy: 1..10 &amp;  zz: 1..100 &amp;</p> <p><b>Goal</b>  xx+yy-1: 1..100</p>
----------------------------------------------------------------------------------------------------------------------------------

L'opérateur désire ajouter l'hypothèse  $xx + yy : 2..20$ .

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

L'hypothèse à ajouter doit d'abord être prouvée.

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

L'opérateur lance le coeur de preuve :

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

L'hypothèse est prouvée. Le but courant devient donc :

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

La commande **dd** permet alors de faire monter l'hypothèse  $xx + yy : 2..20$  dans la pile des hypothèses.

```
PRI> dd
Starting Deduction
```

L'hypothèse a effectivement été montée et le but courant est à nouveau  $xx + yy - 1 : 1..100$ .

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

## 4.15 Display Term

### AFFICHAGE DES TERMES D'UNE FORMULE

#### Syntaxe

**dt**  
**dt(f)**

avec :

- f est une liste, éventuellement réduite à un seul élément, de termes de la forme t.i et séparés par une virgule

#### Utilisation

Cette commande s'utilise à la suite d'une analyse logique de formule **la** (voir chapitre 4.26 page 73) et permet d'afficher la valeurs des différents termes de la formule analysée.

#### Exemple

Soit le but suivant :

```
"'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
```

L'analyse logique de formule, exécutée par la commande **la**(voir chapitre 4.26 page 73), décompose la formule du but en :

```
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
```

Les termes t.1 à t.16 peuvent alors être affichés, par la commande **dt** :

```
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
```

## 4.16 Use equality in hypothesis

### RÉÉCRITURE D'APRÈS UNE ÉGALITÉ EN HYPOTHÈSE

#### Syntaxe

**eh(a, A, f)**

avec :

- L'hypothèse  $a=A$  ou l'hypothèse  $A=a$  doit être dans la pile,
- on peut aussi utiliser le mot-clé  $\_h$  soit pour  $a$ , soit pour  $A$
- $f$  peut être
  - soit le mot-clé **Goal**
  - soit le mot-clé **AllHyp**
  - soit le mot-clé **Hyp(h)** avec  $h$  correspondant à l'hypothèse choisie

#### Utilisation

Cette commande permet de remplacer  $a$  par  $A$ , soit dans le but courant, soit dans toutes les hypothèses, soit enfin dans l'hypothèse  $h$ .

Le mot-clé  $\_h$  permet d'utiliser une égalité dont on ne connaît qu'un membre (droit ou gauche). Dans ce cas, la dernière égalité en hypothèse, satisfaisante, est utilisée.

Si  $f = \mathbf{Hyp}(h)$ , le but devient :

$$H \Rightarrow B$$

où  $H$  est obtenu en remplaçant  $a$  par  $A$  dans l'hypothèse  $h$ , si elle existe.

Si  $f = \mathbf{AllHyp}$ , le but devient :

$$H \Rightarrow B$$

où  $H$  est obtenu en remplaçant  $a$  par  $A$  dans toutes les hypothèses.

Si  $f = \mathbf{Goal}$ , le but devient :

$$B'$$

où  $B'$  est obtenu en remplaçant  $a$  par  $A$  dans le but courant.

Très souvent, une preuve échoue parce qu'une égalité n'a pas été employée. Les démonstrateurs automatiques sont obligés de prendre des précautions avec les réécritures utilisant des égalités ; en effet, celles-ci peuvent générer des bouclages (voir chapitre 2.9 page 10).

Le prouveur interactif peut, quant à lui, se permettre de telles réécritures, qui sont appliquées ponctuellement et sous le contrôle de l'opérateur.

Dans le cas d'une réécriture du but, il se peut que la commande interactive aille à l'encontre d'une normalisation faite par le prouveur automatique ; si on relance en automatique, celui-ci refera immédiatement la transformée inverse.

Néanmoins, la commande peut être utile si l'opérateur utilise d'autres commandes interactives sur le but réécrit, avant de rappeler le démonstrateur automatique.

**Exemple 1**

Soit la situation suivante :

```
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
```

Il est possible de substituer *uu* par *zz* dans le but.

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

Le but devient :

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

Il est possible d'effectuer la substitution pour une hypothèse

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

Le but devient :

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

Il est possible d'effectuer la substitution pour toutes les hypothèses.

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

Toutes les nouvelles hypothèses apparaissent comme antécédent du but courant :

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))
```

## Exemple 2

Soit la situation suivante :

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
```

Si l'opérateur désire utiliser une égalité avec  $e5$  en membre de droite, sans s'occuper du membre de gauche :

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

en utilisant l'égalité  $zz = e5$ , le but devient :

Goal

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

Si l'opérateur désire utiliser la dernière égalité en hypothèse dont le membre gauche est  $zz$  :

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

le but devient :

Goal  
e2 = e5 or e2 = e5

Le but est effectivement transformé, en utilisant l'hypothèse  $zz = e5$ .

## 4.17 Force

### CHANGEMENT DE LA FORCE COURANTE DU PROUVEUR

#### Syntaxe

**ff(n)**

avec :

— n = Rapide, 0, 1, 2 ou 3

#### Utilisation

Cette commande permet de changer la force courante du prouveur et de rejouer toute la ligne de commande existante (voir chapitre 2.7 page 8) avec la nouvelle force.

Si la force demandée est égale à la force courante, alors rien ne se passe.

La force Rapide privilégie au maximum la rapidité des traitements.

Une preuve en force 0 demande en moyenne jusqu'à 10 secondes par obligation de preuve. Pour certaines PO complexes, la preuve peut demander plusieurs minutes. Les forces supérieures mettent en oeuvre des mécanismes plus gourmands en ressources mémoire et CPU. Le temps de traitement par PO peut devenir très important.

Étant donné qu'environ 90 % des PO prouvées le sont en force 0, il est conseillé de tenter la preuve d'une PO d'abord en force 0.

Pour plus de précisions sur les forces de preuve, on se reportera au chapitre "Le choix d'une force supérieure" du *Manuel utilisateur du prouveur interactif*.

La commande **ba** (voir chapitre 4.6 page 30), qui effectue le recul de la preuve, ne s'applique pas sur une commande **ff**.

#### Exemple

Soit la situation suivante :

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

pour laquelle un travail de preuve interactive existe, en force(0) :

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

L'opérateur change la force courante du prouveur :

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

La ligne de commande déjà exécutée va être rejouée dans la nouvelle force ( $ah(xx + yy \in 2..20) \ \& \ pr \ \& \ dd$ ).

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

Le jeu en force différente est difficile puisque le cheminement de la preuve est a priori différent. On remarque en particulier que la commande **dd** ne s'applique pas.

La nouvelle ligne de commande est :

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

## 4.18 False hypothesis

TENTATIVE DE PREUVE PAR HYPOTHÈSES CONTRADICTOIRES

### Syntaxe

**fh(h)**

avec :

— h est une hypothèse

### Utilisation

Cette commande permet de réaliser une démonstration, en prouvant qu'une hypothèse est contradictoire avec les autres.

Si le lemme à démontrer est

B sous les hypothèses  $h_1, \dots, h_n$

et que l'opérateur se doute que l'une des hypothèses  $h_i$  est contradictoire avec les autres, alors on peut démontrer le lemme en démontrant :

$\neg h_i$  sous les hypothèses  $h_1, \dots, h_n$

### Exemple

Soit la situation suivante :

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

Manifestement, l'hypothèse  $e2 = e5$  est contradictoire. En appliquant la commande :

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

le but courant devient :

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

Un appel au prouveur automatique permet alors de décharger le but.

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

## 4.19 Forward

PASSAGE À LA COMMANDE SAUVEGARDÉE SUIVANTE

### Syntaxe

**fw**

### Utilisation

Cette commande permet d'ignorer la commande suivante de la ligne de commande sauvegardée.

### Exemple

Soit l'obligation de preuve dont la ligne de commande sauvegardée est :

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

Exécutons la première commande sauvegardée :

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

La première commande sauvegardée s'applique :

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

Il est possible de ne pas exécuter la commande sauvegardée suivante (**dd**) en faisant appel à la commande **fw** :

```
PRI> fw
```

La prochaine commande sauvegardée sera donc pr :

```
PRI> st
Next step pr
Starting Prover Call
```

La ligne de commande obtenue est donc :

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

## 4.20 Goto

POSITIONNEMENT SUR UNE PO

### Syntaxe

**go(f.n)**

avec :

- f est le nom d'une opération (ou d'une clause) du composant courant
- n est le numéro d'une PO existante, pour l'opération (ou la clause) concernée

### Utilisation

Cette commande permet de se placer au début de la preuve de la PO numéro n, pour l'opération (ou clause) f.

Une confirmation de sauvegarde sera demandée à l'opérateur si :

- La ligne de commande précédente et la nouvelle ligne de commande permettent de prouver l'obligation de preuve,
- La ligne de commande précédente et la nouvelle ligne de commande ne permettent pas de prouver l'obligation de preuve.

### Exemple

L'opérateur désire se déplacer sur la PO *Initialisation.1*. Un travail de preuve a été effectué et n'a pas été sauvegardé.

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

Comme l'ancienne et la nouvelle ligne de commande ne permettent pas de prouver l'obligation de preuve, une confirmation de sauvegarde est demandée :

```
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):
```

L'opérateur ne désire pas sauvegarder son travail de preuve :

No

La ligne de commande n'est pas sauvegardée et l'obligation de preuve courante devient *Initialisation.1*.

No saving  
Current PO : Initialisation.1

## 4.21 Goto with reset

POSITIONNEMENT SUR UNE PO EN FORCE 0

### Syntaxe

**gr(f.n)**

avec :

- f est le nom d'une opération (ou d'une clause) du composant courant
- n est le numéro d'une PO existante, pour l'opération (ou la clause) concernée

### Utilisation

Cette commande permet de se positionner sur la PO f.n en remettant la force de preuve à 0 et en conservant inchangées les commandes de preuve sauvegardées.

A partir de la force 1, les hypothèses d'une obligation de preuve sont toutes simplifiées lors de son chargement. Cette commande sera utile si une obligation de preuve est telle que la montée de ses hypothèses fait boucler le prouveur pour une force donnée (1, 2 ou 3).

### Exemple

Soit l'obligation de preuve, dont la ligne de commande est :

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

Si l'on exécute la commande **gr**

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

la ligne de commande sera remise en force 0 :

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

## 4.22 Global situation

ÉTAT DE LA PREUVE POUR LES PO DU COMPOSANT COURANT

### Syntaxe

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

avec :

- `k` est soit le nom d'une opération (ou d'une clause) du composant courant ou le mot-clef `_all` désignant toutes les opérations et les clauses du composant, soit l'une des expressions suivantes : `Unproved`, `Proved` et `Patt(g)` avec `g` une formule.
- `o` est le nom d'une clause du composant ou le mot-clef `_all`.
- `e` est l'état des obligations de preuve : `Proved`, `Unproved` ou `_all` (quand on ne le spécifie pas).
- `f` est une formule -entre parenthèses- ou le mot-clef `_all`. `f` permet de filtrer les obligations de preuve en fonction de la forme du but qui doit coïncider avec `f`. Si `f` est le mot-clef `_all`, il n'y a aucun filtrage.

### Utilisation

Cette commande permet de sélectionner et d'afficher l'état de preuve (prouvée, non prouvée) et le but (sans les hypothèses) des obligations de preuve.

L'argument `o` permet de sélectionner les obligations de preuves correspondantes à la clause spécifiée (si elle existe), `e` identifie l'état des obligations de preuve et enfin, `f` la forme de leur but.

L'argument `k` permet de sélectionner toutes les obligations de preuve du composant courant qui sont dans l'état de preuve spécifié, si `k` vaut `Proved` ou `Unproved`, toutes les obligations de preuve du composant courant dont le but coïncide avec `g`, si `k` est de la forme `Patt(g)` et toutes les obligations de preuve correspondantes à la clause `k`, si `k` est un nom de clause du composant courant.

Par défaut, `gs`, `gs(o,e)` représentent respectivement `gs(_all,_all,_all)` et `gs(o,e,_all)`.

Si `k` vaut `Proved` ou `Unproved`, `gs(k)` signifie `gs(_all,k,_all)`, si `k` est de la forme `Patt(g)`, `gs(k)` désigne `gs(_all,_all,(g))` et dans tout les autres cas, `gs(k)` signifie `gs(k,_all,_all)`.

### Exemple1

Pour le composant ci-dessous, on note la présence de la clause *Initialisation* et de l'opération *op0*. La forme du but de chaque obligation de preuve est donnée en fin de ligne.

```

PRI> gs
State of all P0
  Initialisation
    P01 Unproved xx = 3
    P02 Unproved {0|->TRUE}: NAT +-> BOOL
    P03 Unproved xx+1: INTEGER
    P04 Proved 0<=xx+1
    P05 Unproved xx+1<=2147483647
  op0
    P01 Unproved zz+2: INTEGER
    P02 Unproved 0<=zz+2
    P03 Proved zz+2<=2147483647
    P04 Unproved zz+2 = 3
End

```

### Exemple2

On ne sélectionne maintenant que les obligations de preuve de l'opération op0 :

```

PRI> gs(op0)
State of All P0 of operation op0
  P01 Unproved    zz+2: INTEGER
  P02 Unproved    0<=zz+2
  P03 Proved      zz+2<=2147483647
  P04 Unproved    zz+2 = 3
End

```

### Exemple3

On choisit d'afficher seulement les obligations de preuve non prouvées de l'opération op0 :

```

PRI> gs(op0,Unproved)
Unproved P0 of operation op0
  P01 Unproved    zz+2: INTEGER
  P02 Unproved    0<=zz+2
  P04 Unproved    zz+2 = 3
End

```

**Exemple4**

On recherche les obligations de preuve non prouvées de la clause Initialisation dont le but coïncide avec  $\{x\} : y \rightarrow \text{BOOL}$  :

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

Unproved P0 of operation Initialisation
Matching with {x}: y +-> BOOL
      P02 Unproved      {0|->TRUE}: NAT +-> BOOL
End
```

**Exemple5**

On recherche les obligations de preuve prouvées parmi toutes les obligations de preuve du composant courant :

```
PRI> gs(Proved)

All Proved P0
      Initialisation
      P04 Proved 0<=xx+1
      op0
      P03 Proved zz+2<=2147483647
End
```

**Exemple6**

On recherche parmi toutes les obligations de preuve du composant courant, celles dont le but coïncide avec la formule  $x = y$  :

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

State of all P0
Matching with x = y
      Initialisation
      P01 Unproved xx = 3
      op0
      P04 Unproved zz+2 = 3
End
```

## 4.23 Graphical Trace

### SÉLECTION DU MODE TRACE GRAPHIQUE

#### Syntaxe

**gt(f)**

avec :

— f vaut **on** ou **off**

#### Utilisation

Cette commande permet de sélectionner (**gt(on)**) ou de désélectionner (**gt(off)**) le mode trace graphique du prouveur interactif. La sélection du mode trace graphique ne peut se faire que lorsque la démonstration de l'obligation de preuve courante n'a pas encore été jouée. Si cela n'est pas le cas, il faut faire un reset (commande **re**) de la démonstration jouée et saisir à nouveau la commande **gt(on)**. Après quelques instants, l'outil de visualisation graphique DaVinci apparaît. La trace d'exécution des commandes apparaîtra alors dans cette fenêtre :

- les commandes saisies apparaissent en rouge
- les buts intermédiaires en blanc
- les buts prouvés en vert

#### Exemple

L'opérateur a commencé un preuve et désire passer en mode trace graphique.

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

L'opérateur réinitialise sa démonstration jouée, après avoir sauvegardée celle-ci, par la commande **sw**. Il peut alors passer en mode trace graphique :

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

A la fin de son travail de démonstration, il peut désactiver le mode trace graphique :

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

## 4.24 Goto without save

POSITIONNEMENT SUR UNE AUTRE OBLIGATION DE PREUVE SANS SAUVEGARDE DE LA PREUVE COURANTE

### Syntaxe

**gw(f.n)**

avec :

- f est le nom d'une opération (ou d'une clause)
- n est le numéro d'une obligation de preuve, pour l'opération (ou la clause) concernée

### Utilisation

Cette commande permet d'aller à l'obligation de preuve f.n, en abandonnant sans sauvegarde le travail de preuve sur l'obligation de preuve courante.

Si f n'est pas une opération ou une clause ou si n n'est pas un bon numéro, la commande ne s'exécute pas.

### Exemple

La PO test.Initialisation.1 a été prouvée. L'opérateur peut désirer ne pas sauvegarder le travail de preuve interactive réalisé.

L'opérateur se déplace sur la PO test.Initialisation.2, sans sauvegarde.

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

## 4.25 Help

AIDE EN LIGNE SUR LA SYNTAXE DES COMMANDES ET LEUR FONCTIONNEMENT

### Syntaxe

**help**

**help(c)**

avec :

c est le nom d'une commande interactive

### Utilisation

Cette commande permet d'afficher la description du fonctionnement de la commande c. Sans argument, elle affiche la liste des commandes interactives disponibles.

### Exemple

L'utilisateur souhaite connaître le fonctionnement de la commande **mini prover mp** :

```
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
```

## 4.26 Logical Analysis

### ANALYSE LOGIQUE D'UNE FORMULE

#### Syntaxe

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

avec :

- f est une formule (expression ou prédicat B)
- n est un entier strictement positif

#### Utilisation

Cette commande permet d'analyser une formule B selon différents modes :

- **la** : le but courant est analysé avec une profondeur d'analyse de 1,
- **la(n)** : le but courant est analysé avec une profondeur d'analyse de n,
- **la(f)** : la formule f est analysée avec une profondeur de 1,
- **la(f | n)** : la formule f est analysée avec une profondeur de n.

Lorsque la formule à analyser a une profondeur supérieure à la profondeur d'analyse, alors les expressions et prédicats ne pouvant pas être détaillés sont remplacés par des termes notés t.i. Cette notation permet d'obtenir un affichage plus synthétique. La valeur de ces termes n'est pas affichée automatiquement afin d'alléger l'affichage. La commande **dt** (voir chapitre 4.15 page 52) permet d'afficher la valeur de tout ou partie de ces termes.

#### Exemple

Soit la situation suivante :

```
"'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}])

```

L'application de la commande **la(3)** permet d'obtenir la décomposition du but suivante :

```

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

```

## 4.27 Match goal

MATCH SUR UN BUT

### Syntaxe

**Mgoal(p)**

avec :

- **p** est un pattern destiné à matcher le but

### Utilisation

Cette commande est remplacée par un but lorsque celui-ci matche avec le pattern passé en paramètre. Si le but ne correspond pas au pattern, la commande échoue.

Cette commande est utilisée en paramètre d'autres commandes afin de généraliser une preuve.

### Exemple

Soit le but suivant.

<pre>Goal   var1 : fonction[ens1]</pre>
-----------------------------------------

La commande **Mgoal(a : b)** correspondra au but courant, avec **a = var1** et **b = fonction[ens1]**. Ainsi, pour rechercher l'ensemble des hypothèses impliquant le but, il est possible d'utiliser la commande **sh(h => Mgoal(a : b))**.

## 4.28 Match Hypothesis

MATCH D'UNE HYPOTHÈSE

### Syntaxe

**Mhyp**(p)

avec :

- p est un pattern destiné à matcher une hypothèse

### Utilisation

Cette commande est expansée en une hypothèse matchant le pattern passé en paramètre. Si aucune hypothèse correspondante n'est trouvée, la commande échoue. Si plusieurs hypothèses correspondent, **Mhyp** est remplacée par l'hypothèse la plus récente.

Cette commande est utilisée en paramètre d'autres commandes afin de généraliser une preuve.

### Exemple

Soit l'hypothèse suivante :

```
Hypothesis
  var1 : fonction[ens1]
```

La commande **Mhyp**(x : e[y]) peut être utilisée pour référencer cette l'hypothèse.

Ainsi, la commande **ah**(**Mhyp**(x : e[y])) ajoutera l'hypothèse **var1 : fonction[ens1]** au but courant.

## 4.29 Show litteral PO

AFFICHAGE D'UNE OBLIGATION DE PREUVE SOUS SON FORMAT LITTÉRAL

### Syntaxe

**lp(f.n)**

avec :

- f doit être le nom d'une opération (ou d'une clause) du composant courant
- n doit être un numéro d'obligation de preuve valide pour l'opération (ou la clause) f sélectionnée

### Utilisation

Cette commande affiche l'obligation de preuve sélectionnée, sous son format littéral, c'est à dire telle qu'elle a été générée par le générateur d'obligations de preuve.

En effet, le prouveur effectue des traitements sur les hypothèses et le but (simplifications, ...). Les données affichées ne sont donc plus l'exact reflet de l'obligation de preuve.

### Exemple

Lorsque l'on se déplace sur l'obligation de preuve *Initialisation.1*, les expressions qu'elle contient (hypothèses, but) sont normalisées.

```
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
```

L'opérateur désire voir l'obligation de preuve *Initialisation.1* sous sa forme littérale :

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

L'affichage de l'obligation de preuve sous sa forme littérale nous montre alors l'écriture de cette obligation de preuve sans normalisation, telle qu'elle est générée par le Générateur d'obligations de preuve.

```
Show P0 : 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 P0
```

## 4.30 ModelChecking

VÉRIFICATION DE LA VALIDITÉ D'UN PRÉDICAT À L'AIDE DE LA LISTE EXHAUSTIVE DES VALEURS DE SES VARIABLES LIBRES

### Syntaxe

**mc**

**mc(l)**

avec *l* une liste d'au plus quatre éléments séparée par des | et tous distincts (leur ordre dans la liste n'ayant pas d'incidence sur le comportement de la commande). Ces éléments appartiennent à des catégories fonctionnelles différentes et permettent de spécifier le mode d'utilisation de la commande **mc** :

- Gestion de la Preuve : **\_Auto** ou **\_Step**. Le mot-clef **\_Auto** signifie que la preuve est gérée de manière automatique, au sens où le prouveur automatique effectue toutes les démonstrations en fonction des différentes valeurs des variables. Le mot-clef **\_Step** signifie quant à lui que l'utilisateur va démontrer en preuve interactive les différents cas générés par le prouveur. Si l'on ne spécifie pas de valeur pour ce paramètre, la gestion sera automatique par défaut.
- Tactique Utilisateur : **Tac(None)** ou **Tac(T)**. **Tac(None)** signifie qu'il n'y a pas de tactique définie par l'utilisateur. **Tac(T)**, où *T* est une tactique, signifie *a contrario* que le prouveur doit essayer la tactique *T* avant de continuer éventuellement la démonstration. Il est à noter que la combinaison **\_Step** et **Tac(T)** (avec  $T \neq \text{None}$ ) n'est pas valide. Le mode par défaut est **Tac(None)**.
- Liste des Variables de Résolution : **\_Variables** ou **L**. Le mot-clef **\_Variables** indique que l'on va utiliser toutes les variables libres du but pour effectuer le model checking. **L** est une liste de noms de variables séparées par des virgules. **L** spécifie ainsi le sous-ensemble des variables à partir desquelles on va effectuer le model checking. Par défaut, la liste des variables de résolution est spécifiée par **\_Variables**.
- Nombre Maximum de Valeurs : **n**. *n* est un entier positif spécifiant le nombre maximum de valeurs que peut prendre une variable. 20 par défaut.

### Utilisation

Cette commande permet d'effectuer des démonstrations par cas du but courant : elle divise la preuve pour chaque valeur des variables de résolution de la liste **L** (les domaines des variables, devant être bornés, étant, si possibles, inférés à partir des hypothèses).

Si la preuve initiale est :  $H \Rightarrow G$

La preuve transformée par **mc**(*x*<sub>1</sub>,*x*<sub>2</sub>,...,*x*<sub>*n*</sub>) est :

```

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)

```

Avec  $(ai, bj, \dots, vk)$  parcourant le produit cartésien des domaines de variation des variables  $x1, x2, \dots, xn$  qui sont respectivement  $\{a1, \dots, ap\}, \{b1, \dots, bq\}, \dots, \{v1, \dots, vr\}$ .

Les domaines des variables ont été inférés à partir des hypothèses disponibles sur les variables de résolution et représentés par des ensembles en extension. Il est à noter ici que l'inférieur de domaine n'effectue pas de résolution de contraintes entre variables et est limité par la taille du domaine des variables : par exemple, **ModelChecking** échouera si on l'utilise sur une variable du type INT et sans aucune autre hypothèse susceptible de contraindre davantage le domaine de la variable.

Les hypothèses utilisées par l'inférieur de domaine sont typiquement les prédicats d'appartenance à un ensemble énuméré prédéfini tel que BOOL ou un ensemble énuméré défini dans la clause SETS, d'appartenance à un intervalle, d'appartenance à un ensemble d'entiers en extension, d'égalité et d'inégalité entre variables et valeurs, *etc.*

Deux options sont possibles :

- L'utilisateur peut entreprendre la démonstration de chacun des cas de manière interactive, en spécifiant le paramètre `_Step`.
- Le prouveur automatique, éventuellement complété d'une tactique de preuve (par `Tac(T)`), est utilisé pour chaque cas sans intervention de l'utilisateur, en spécifiant `_Auto`. On peut noter que si le prouveur échoue sur une tentative de preuve d'un des sous-buts, la commande échoue et on revient au but précédent : on ne rend pas la main à l'utilisateur sur les sous-preuves en échec.

## Exemples

Soit l'obligation de preuve :  $\text{bool}(xx = 1) = yy$ , avec  $xx : \{0,2\}$  et  $yy = \text{FALSE}$  parmi les hypothèses.

La commande `mc(_Step|xx,yy)` provoque la transformation du but courant en :

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

L'utilisateur le démontre en utilisant `pr` et le prouveur génère le second cas :

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

La commande `mc(_Auto)` effectue les démonstrations des deux cas de manière totalement automatique :

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

Enfin, la commande `mc(xx|_Step|Tac(None)|3)` aurait généré en premier lieu le but suivant :

```

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

```

Et ensuite, après utilisation de la commande **pr** :

```

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

```

Il se peut enfin que l'utilisateur souhaite utiliser la commande **ModelChecking** sur une variable dont le domaine est trop grand. Soit donc le but suivant :

```

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

```

L'utilisation de **mc**(\_Step | 22 | xx), où l'on a spécifié un nombre maximum de valeurs de variables à 22, provoque l'affichage du message suivant :

```

Failure in Model Checking

```

En effet le domaine de définition de la variable contient plus de 22 éléments.

## 4.31 MiniProof

PREUVE RÉDUITE

### Syntaxe

**mp**

### Utilisation

En force **0** ou **1**, la commande **mp** permet d'utiliser le prouveur sans tenter de preuve par cas. Cette commande est identique à **pr(Red)** (voir chapitre 4.38 page 100).

### Exemple

Voir la commande **pr(Red)**.

## 4.32 Modus ponens on hypothesis

### APPLICATION DU MODUS PONENS

#### Syntaxe

**mh(H)**

avec :

- H doit être une hypothèse de la forme  $P \Rightarrow Q$

#### Utilisation

Cette commande permet l'utilisation directe d'une hypothèse de la forme  $P \Rightarrow Q$ .

Si  $P \Rightarrow Q$  et  $P$  sont des hypothèses et que le but courant est  $B$ , alors le but devient  $Q \Rightarrow B$ .

La commande **mh** permet d'utiliser une hypothèse en  $P \Rightarrow Q$  sans passer par le prouveur, c'est à dire sans simplifier l'hypothèse  $Q$  générée.

En effet, le prouveur automatique applique systématiquement le modus ponens sur tout couple d'hypothèses  $P \Rightarrow Q$  et  $P$  présent dans la pile des hypothèses.

Si  $P \Rightarrow Q$  ou  $P$  ne sont pas des hypothèses, la commande ne s'applique pas.

#### Exemple

Soit la situation suivante que l'on a obtenu directement :

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

L'opérateur désire utiliser l'hypothèse  $zz = e5 \Rightarrow tt = e1$  pour générer l'hypothèse  $tt = e1$ . Il sait que l'hypothèse  $zz = e5$  existe.

La commande **mh** s'applique correctement

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

et le but devient :

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

La commande **pr** ou **dd** permet alors de faire monter cette hypothèse dans la pile.

## 4.33 Mono Lemma Prover

APPEL DU PROUVEUR MONO-LEMME

### Syntaxe

```
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)
```

avec :

- *l* : liste de forces de preuve séparées par des points-virgules. Une force est l'une des valeurs suivantes : Rapide, 0, 1, 2 ou 3. Si le paramètre *ff(l)* n'est pas spécifié, on utilisera la force courante.
- *t* : temps de coupure du prouveur mono-lemme en seconde. Si cette valeur n'est pas indiquée, le prouveur mono-lemme s'arrête au bout de 60 secondes en interactif.
- *rp.n* indique que l'on utilise le prouveur mono-lemme sur les hypothèses réduites.
  - **rp** est un mot clef.
  - *n* est un entier positif ou nul qui indique le niveau des hypothèses prises en compte. Si *n* est nul, le but seul, sans les hypothèses, est transmis au prouveur mono-lemme.
- *rp(f)* indique que l'on utilise le prouveur mono-lemme sur les hypothèses sélectionnées par la formule *f* (de la forme *a* ou *f+a*, *a* désignant un mot clé). Les mots clés utilisables pour *a* sont :
  - **inv** : invariant du composant
  - **sees** : assertions et invariants des machines vues et utilisées
  - **loc** : hypothèses locales
  - **typ** : prédicats de typage des variables concrètes
  - **abs** : assertions et invariants des composants précédents
  - **used** : contraintes des machines utilisées
  - **inc** : propriétés des machines incluses, importées et étendues
  - **prp** : propriétés et valuations du composant

### Utilisation

Cette commande permet d'utiliser le prouveur mono-lemme sur le but courant, le prouveur mono-lemme étant identique au prouveur automatique mais avec un traitement des hypothèses sensiblement différent.

Cette fonction a trois modes de fonctionnement :

1. Le premiers mode appelle le prouveur mono-lemme sur le but et toutes les hypothèses courantes.
2. Le deuxième mode appelle le prouveur mono-lemme sur le but et les hypothèses de l'obligation de preuve réduite. Les hypothèses sélectionnées sont les mêmes que pour la fonction **rp**(voir chapitre 4.42 page 109).
3. Le troisième mode appelle le prouveur mono-lemme sur le but et les hypothèses sélectionnées, de manière additive. Par exemple, **ml(rp(sees+loc+inv))** permet de lancer la preuve du but sous les les assertions et invariants des machines vues et utilisées, les hypothèses locales et les invariants du composant.

Dans les trois modes, le prouveur mono-lemme est lancé avec un temps de coupure. Sans précision de la part de l'utilisateur, ce temps de coupure est de 60 secondes en interactif. Lorsque la preuve est rejouée en automatique, les appels au prouveur mono-lemme se font avec un temps de coupure spécifié par la ressource **Time\_Out** du fichier de ressource de l'Atelier B (300 secondes par défaut).

On peut enfin paramétrer la force de preuve que l'on veut utiliser (avec l'argument **ff(l)**). Si **l** est une liste de forces, la preuve sera tentée successivement avec chacune des forces de la liste jusqu'à ce que la preuve réussisse ou bien qu'on ait épuisé la liste.

### Exemple

Le prouveur mono-lemme peut s'utiliser sur l'obligation de preuve complète (on notera qu'ici la force courante est 0) :

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

ou sur l'obligation de preuve réduite. Cette option s'utilise lorsque l'obligation de preuve a de nombreuses hypothèses :

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

La preuve peut être tentée avec plus d'hypothèses sélectionnées, mais le succès n'est plus garanti.

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

Le prouveur mono-lemme peut s'utiliser pour prouver une obligation de preuve donnée ou pour prouver un sous-but. Il peut à ce titre faire partie d'une stratégie de preuve, en étant

utilisé dans le corps de la commande **te** (voir chapitre 4.54 page 132).

Il est ici utilisé sur les obligations de preuve réduites (1 itération) avec un temps de coupure de 10 secondes.

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

Le prouveur peut aussi s'utiliser avec une liste de forces à tenter. On parcourt la liste de forces jusqu'à ce qu'une d'elles permette d'effectuer la preuve. Ici la force 1 permet de décharger le but, on n'essaie donc pas la 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
```

## 4.34 Next

POSITIONNEMENT SUR LA PROCHAINE OBLIGATION DE PREUVE NON PROUVÉE

### Syntaxe

**ne**

### Utilisation

Cette commande permet de passer à l'obligation de preuve suivante non prouvée, s'il y en a une. S'il n'y a plus d'autre obligation de preuve non prouvée, la commande **ne** est sans effet.

Cette commande peut être utilisée pour se placer rapidement sur la première preuve à faire quand on ouvre la preuve interactive d'un composant.

### Exemple

Le composant comprend la clause Initialisation et l'opération Calcul, deux obligations de preuve prouvées et deux obligations de preuve non-prouvées. En exécutant la commande **gs** (voir chapitre 4.22 page 67), on obtient la situation suivante :

```
PRI> gs
State of all P0
  Initialisation
    P01 Proved      not(e5 = e1)
    P02 Proved      e1 = e5
  Calcul
    P01 Unproved    not(e2 = e5)
    P02 Unproved    e5 = e1
End
```

Supposons que l'obligation de preuve courante est *Calcul.1*. L'opérateur passe à la prochaine obligation de preuve non prouvée.

```
PRI> ne
Current P0 : Calcul.2
```

En répétant la commande, on revient à l'obligation de preuve *Calcul.1*.

```
PRI> ne
Current P0 : Calcul.1
```

## 4.35 Pmm compile

CHARGEMENT DE RÈGLES UTILISATEUR (PMM)

### Syntaxe

**pc**

### Utilisation

Cette commande permet le chargement et la compilation des règles manuelles.

Pour permettre de traiter les cas de preuve les plus difficiles, il est possible d'écrire des règles manuelles dans le fichier .pmm (proof manual methods) (voir chapitre 6 page 151), créé par l'opérateur et ayant comme préfixe le nom du composant traité.

Ce fichier devra contenir un ensemble de théories valides séparées par des &, écrites en langage de théorie.

L'utilisation de ces règles devrait rester marginale. En effet, ces règles peuvent être fausses et conduire le prouveur à prouver des obligations de preuve fausses.

Lors du lancement du prouveur interactif, le fichier pmm du composant, s'il existe est chargé automatiquement en mémoire.

Dans le cas où le fichier pmm a été modifié par l'opérateur au cours de la preuve interactive et que celui-ci désire utiliser les règles du fichier pmm dans sa dernière version, la commande **pc**, spécifique, permet de charger en mémoire les règles du fichier pmm correspondant au composant.

Lors du chargement du fichier pmm, le prouveur affiche un message d'acceptation du fichier ou un message d'erreur.

### Attention !

Alors que toutes les autres fonctionnalités du prouveur interactif sont totalement protégées, cette possibilité d'application de règles écrites manuellement ne l'est pas.

Il est possible d'entrer une règle fausse, provoquant ainsi des démonstrations fausses. Si aucune règle manuelle de ce type n'a été utilisée, alors la validité du prouveur (automatique + interactif) suffit à assurer la validité de la preuve, quelles que soient les commandes interactives appelées.

Si par contre, des règles manuelles ont été ajoutées, alors il faudra s'assurer de la validité de ces règles. L'emploi d'un démonstrateur de règles pourra être préconisé pour cette tâche ; mais il est clair que l'esprit dans lequel est fait le prouveur interactif est d'éviter l'emploi de ces règles manuelles.

**Exemple**

Soit la situation suivante :

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

L'opérateur lance le coeur de preuve :

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

Le but  $(xx + 1) * yy - 1 : 1..109$  est décomposé en 2 sous-buts  $1 \leq (xx + 1) * yy - 1$  et  $(xx + 1) * yy - 1 \leq 109$ . Le premier sous-but est d'abord traité.

Le prouveur automatique s'arrête car il ne sait pas résoudre l'inégalité  $0 \leq -2 + yy + xx * yy$ .

```
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
```

L'opérateur décide donc d'introduire une nouvelle règle par l'intermédiaire du fichier pmm et de l'utiliser dans le cas de la preuve.

Le fichier *test.pmm* contient alors :

```
THEORY test IS

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

END
```

La règle est d'abord chargée puis compilée.

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

Pour décharger le sous-but,  $0 \leq -2 + yy + xx * yy$ , on applique la règle numéro 1 de la théorie *test*.

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

Le premier sous-but est déchargé et le prouveur automatique essaye maintenant de prouver le second sous-but :

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

Le deuxième sous-but est alors à prouver, mais n'est pas prouvé par l'intermédiaire de la commande **pr**.

L'opérateur ajoute alors la règle permettant de décharger ce but.

Le fichier *test.pmm* contient finalement :

```
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
```

Le fichier *pmm* ayant été modifié, il faut le recharger en mémoire. Les règles anciennement chargées sont écrasées par les nouvelles.

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

Pour décharger le sous-but,  $(a + 1) * b - 1 \leq 109$ , on applique la règle numéro 2 de la théorie *test*.

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

L'obligation de preuve est alors prouvée.

## 4.36 Particularize hypothesis

INSTANCIATION D'UNE HYPOTHÈSE UNIVERSELLEMENT QUANTIFIÉE

### Syntaxe

**ph**( $v_1, \dots, v_n, h$ )

avec :

- $h$  est une hypothèse universelle de la forme  $\forall(w_1, \dots, w_n). (P(w_1, \dots, w_n) \Rightarrow Q(w_1, \dots, w_n))$

### Utilisation

Cette commande permet d'assigner une valeur aux variables qui apparaissent, en hypothèse, sous la portée d'un même quantificateur universel. On affecte les valeurs  $v_1, \dots, v_n$  aux variables  $w_1, \dots, w_n$ . Si la valeur d'une ou plusieurs variables n'est pas connue, on peut utiliser le mot-clé `_h` pour signifier que la ou les variables ne seront pas instanciées. Par exemple :

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

permettra de générer l'hypothèse correspondant à

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

B étant le but initial, le but devient :

$P(v_1, \dots, v_n) \wedge (Q(v_1, \dots, v_n) \Rightarrow B)$

- le prouveur automatique va alors chercher à démontrer  $P(v_1, \dots, v_n)$
- si c'est un succès, la preuve continue avec  $Q(v_1, \dots, v_n)$  en hypothèse.

Les prédicats  $P(v_1, \dots, v_n)$  contiennent les typages de  $v_1, \dots, v_n$ .

Notons cependant que la particularisation des hypothèses quantifiées universellement n'est pas protégée contre le mauvais typage (voir chapitre 2.2 page 3) ainsi que la mauvaise définition (voir chapitre 2.3 page 4). On peut instancier une variable liée par une valeur mal typée ou mal définie. L'utilisateur doit donc vérifier la correction du typage et la bonne définition de ces valeurs avant d'utiliser cette commande.

Ceci peut être vérifié *a posteriori* à l'aide de l'outil **mdelta** (cf. Manuel Utilisateur Version 1.0.).

### Exemple

Soit la situation suivante :

```

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)

```

L'opérateur désire utiliser l'hypothèse  $\forall vv.(vv \in ENS \wedge (not(uu = vv) \vee not(tt = vv)) \Rightarrow zz = vv)$ , en instanciant  $vv$  avec la valeur  $e_1$ .

La preuve de  $vv \in ENS \wedge (not(uu = vv) \vee not(tt = vv)) \Rightarrow zz = vv$  va se décomposer en deux parties (après instanciation) :

- $e_1 \in ENS$
- puis  $not(uu = e_1) \vee not(tt = e_1)$

Si ces deux sous-buts sont prouvés alors le but deviendra  $zz = e_1 \Rightarrow not(tt = uu)$ .

```

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

```

Le prédicat de typage instancié doit d'abord être prouvé :

```

Goal
  e1: ENS

```

La commande **pr** permet de décharger ce premier sous-but.

```

PRI> pr

```

Le but suivant est donc :

```

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

```

L'opérateur lance alors le coeur de preuve :

```

PRI> pr
Starting Prover Call

```

Le but est prouvé. Le but suivant est alors généré :

```

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

```

Le prédicat  $zz = e_1$  peut alors être mis en hypothèse par l'une des deux commandes **pr** ou **dd** (voir chapitre 4.14 page 49).

## 4.37 Predicate prover

### APPEL DU PROUVEUR DE PRÉDICATS

#### Syntaxe

```

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

```

avec :

- **t** : temps de coupure du prouveur de prédicat en seconde. Si cette valeur n'est pas indiquée, le prouveur de prédicat utilise la ressource **ATB\*PR\*Time\_Out** (voir chapitre 5.1 page 145), si celle-ci existe. Sinon le prouveur de prédicat s'arrête au bout de 60 secondes en interactif. Ce temps de coupure est un temps "CPU", donc indépendant de la charge de la machine.
- **rp.n** indique que l'on utilise le prouveur de prédicat sur les hypothèses réduites.
  - **rp** est un mot clef.
  - **n** est un entier positif ou nul qui indique le niveau des hypothèses prises en compte. Si **n** est nul, le but seul, sans les hypothèses, est transmis au prouveur de prédicats
- **rt.n** indique que l'on utilise le prouveur de prédicats sur les hypothèses réduites (de la même manière que **rp.n**), ainsi que sur les hypothèses de typage des identificateurs apparaissant dans les hypothèses prises en compte.
- **rp(f)** indique que l'on utilise le prouveur de prédicat sur les hypothèses sélectionnées par la formule **f** (de la forme **a** ou **f+a**, où **a** désigne un mot clé). Les mots clés utilisables pour **a** sont :
  - **inv** : invariant du composant
  - **sees** : assertions et invariants des machines vues et utilisées
  - **loc** : hypothèses locales
  - **typ** : prédicats de typage des variables concrètes
  - **abs** : assertions et invariants des composants précédents
  - **used** : contraintes des machines utilisées
  - **inc** : propriétés des machines incluses, importées et étendues
  - **prp** : propriétés et valuations du composant

#### Alias

```

p0 est équivalent à pp(rp.0)
p0(t) est équivalent à pp(rp.1|t)
p1 est équivalent à pp(rp.1)
p1(t) est équivalent à pp(rp.1|t)

```

```

t0 est équivalent à pp(rt.0)

```

`t0(t)` est équivalent à `pp(rt.1|t)`  
`t1` est équivalent à `pp(rt.1)`  
`t1(t)` est équivalent à `pp(rt.1|t)`

## Utilisation

Cette commande permet d'utiliser le prouveur de prédicats sur le but courant.  
 Cette fonction a trois modes de fonctionnement :

1. Le premiers mode appelle le prouveur de prédicats sur le but et toutes les hypothèses courantes. Ce mode n'est pas adapté au traitement d'un grand nombre d'hypothèses.
2. Le deuxième mode appelle le prouveur de prédicats sur le but et les hypothèses de l'obligation de preuve réduite. Les hypothèses sélectionnées sont les mêmes que pour la fonction `rp` (voir chapitre 4.42 page 109).
3. Le troisième mode appelle le prouveur de prédicats sur le but et les hypothèses sélectionnées, de manière additive. Par exemple, `pp(rp(sees+loc+inv))` permet de lancer la preuve du but sous les assertions et invariants des machines vues et utilisées, les hypothèses locales et les invariants du composant.

Dans les trois modes, le prouveur de prédicats est lancé avec un temps de coupure. Sans précision de la part de l'utilisateur, ce temps de coupure est de 60 secondes en interactif.

Lorsque la preuve est rejouée en automatique, les appels au prouveur de prédicats se font avec un temps de coupure spécifié par la ressource **Time\_Out** du fichier de ressource de l'Atelier B (300 secondes par défaut). Cette marge permet de rejouer un appel au prouveur de prédicat, couronné de succès, sur une machine moins puissante.

## Exemple

Le prouveur de prédicats peut s'utiliser sur l'obligation de preuve complète :

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

ou sur l'obligation de preuve réduite. Cette option s'utilise lorsque l'obligation de preuve a de nombreuses hypothèses :

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

La preuve peut être tentée avec plus d'hypothèses sélectionnées, mais le succès n'est plus garanti.

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

Le prouveur de prédicats peut s'utiliser pour prouver une obligation de preuve donnée ou pour prouver un sous-but. Il peut à ce titre faire partie d'une stratégie de preuve, en étant utilisé dans le corps de la commande **te** (voir chapitre 4.54 page 132).

Il est ici utilisé sur les obligations de preuve réduites (1 itération) avec un temps de coupure de 10 secondes.

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

Le travail effectué par le prouveur de prédicat est alors affiché :

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

Deux obligations de preuve *Initialisation.1* et *Initialisation.4* ont été déchargées.

## 4.38 Prove

APPEL DU PROUVEUR AUTOMATIQUE

### Syntaxe

```
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)
```

avec :

- r = **None** (affichage du nom des règles appliquées) ou **Ru** (affichage du nom et du corps de la règle)
- b = **Goal** (affichage des buts) ou **Stop** (affichage et arrêt sur chaque but)
- h = **None** (aucun affichage à propos des hypothèses), **Gen** (hypothèses générées) ou **Full** (hypothèses générées et réellement montées dans la pile)
- f = **File** (génération d'un fichier trace visualisable par la fonction **Show Proof Tree**) ou **NoFile** (pas de génération - option par défaut)
- s = **Simpl** (Affichage des simplifications réalisées sur toutes les expressions) ou **NoSimpl** (Les simplifications ne sont pas affichées - option par défaut)
- t est une tactique (voir chapitre 2.6 page 7)

### Utilisation

Cette commande permet de faire appel au démonstrateur automatique pour prouver l'obligation de preuve courante.

La commande **pr** est aussi utile pour relancer une preuve qui a presque réussi, mais qui s'est arrêtée parce qu'un nombre d'essais maximum était atteint. En effet, le prouveur automatique possède un certain nombre de compteurs qui limitent le nombre d'applications de certains mécanismes dans un même appel, pour éviter les bouclages.

Lancer **pr** plusieurs fois de suite peut donc avoir une action.

Si un appel à **pr** n'a réellement rien fait, ceci est signalé par le message

```
Prover call has not done anything
```

Dans ce cas, il est inutile de le relancer.

Le prouveur retourne un message indiquant si la preuve a échoué ou réussi.

Les commandes **pr(r.b.h,f,s)** et **pr(Tac(t),r.b.h,f,s)** permettent de lancer le prouveur en mode trace (voir chapitre 10 page 161). Les paramètres *f*, *s* et *Tac(t)* sont facultatifs, mais si l'on désire expliciter *s*, il faut alors expliciter aussi *f* (l'ordre des paramètres doit être conservé).

Les informations suivantes sont disponibles :

- simplification du but, par application d'une règle ou d'un mécanisme
- déchargement du but
- lancement d'une preuve par cas
- lancement et fin d'une preuve par tentative (preuve annexe)
- application des règles de simplification

La présence du paramètre *Tac(t)* permet d'utiliser les règles utilisateurs (pmm (voir chapitre 6 page 151), PatchProver (voir chapitre 7 page 153)) au sein même du prouveur automatique. Les tactiques backward de l'utilisateur s'appliquent après la montée des hypothèses locales dans la pile des hypothèses et avant l'appel à la base de règles.

Les tactiques forward se comportent comme les règles d'une seule théorie. Ces règles sont employées avec les règles forward du prouveur. L'utilisateur ne peut pas utiliser des tactiques complexes avec les règles Forward. Par exemple, la tactique forward suivante n'est pas valable :

**Fwd1~ ; (Fwd2;Fwd3)**

Si on utilise la commande **pr(Tac(arriere,avant))**, le prouveur interactif va tenter d'appliquer les règles de la théorie *arriere*. Si celles-ci génèrent des hypothèses, il faut que la théorie prédéfinie **DED** apparaisse dans la tactique *arriere*. Dans ce cas, les règles de la tactique *avant* vont traiter les hypothèses montantes.

En force 0 ou 1, la commande **pr(Red)** permet d'utiliser le prouveur sans tenter de la preuve par cas. Cet appel au prouveur automatique se limite :

- au parcours de la base de règles,
- au traitement des buts existentiels
- au surtypage (génération d'hypothèses de typage supplémentaires).

En ce qui concerne les forces Rapide, 2 et 3, la commande **pr(Red)** a le même comportement que **pr** et peut donc éventuellement déclencher des preuves par cas.

### Exemple 1

Soit la situation suivante :

```
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)
```

Un premier appel au prouveur automatique n'a pas permis de décharger le but courant. On essaye une deuxième application de la commande **pr** pour voir si le prouveur automatique n'a pas échoué dans la preuve, à cause de la limitation du nombre d'applications de règles (compteurs internes du prouveurs limitant le risque de bouclage (voir chapitre 2.9 page 10)).

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

Le message *Prover call did nothing* nous indique que le prouveur n'a pas réussi, de manière définitive, à prouver le but courant et qu'il n'a pas produit d'hypothèse supplémentaire.

```
Prover call did nothing

Goal
  not(uu = e1)
```

**Exemple 2**

Voyons maintenant le fonctionnement du prouveur en mode Trace.

Soit la situation suivante :

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

Le prouveur est lancé en mode Trace : le corps des règles ainsi que les informations relatives aux hypothèses ne sont pas affichées, tous les buts sont listés.

```
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
```

Le but initial est décomposé en 2 sous-buts.

```
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.
```

Le premier sous-but a été simplifié puis déchargé. Le deuxième sous-but est alors traité.

Comme  $(xx, yy) \in (1..10) \times (1..10)$ ,  $101 - xx - yy$  est minoré par 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
```

Si l'on avait appliqué la commande **pr(Ru.Goal.None)**, la partie de la trace concernant la règle *InSetXY.13*, c'est à dire :

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

aurait été :

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

### Exemple 3

L'exemple suivant présente l'utilisation du paramètre **Tac**, pour l'utilisation de tactiques backward et forward.

Soit l'obligation de preuve suivante :

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

Le fichier **PMM** associé contient les théories **arriere** et **avant** :

```
THEORY arriere IS  
  xx-1: entiers => xx: entiers => p  
  =>  
  p  
END  
&  
THEORY avant IS  
  xx-1: entiers  
  =>  
  xx: entiers  
END
```

La commande **pr(Tac((arriere;DED),avant))** permet de décharger le but courant.

## 4.39 PreviousPO

### Syntaxe

**pv**

### Utilisation

Cette commande permet de passer à la première obligation de preuve précédente non prouvée, s'il y en a une. S'il n'y a plus d'autres obligations de preuve non prouvées, la commande **pv** est sans effet.

### Exemple

Le composant comprend 2 opérations, 1 obligation de preuve prouvée et 3 obligations de preuve non-prouvées. En exécutant la commande **gs**, on obtient la situation suivante :

```
PRI> gs
State of all PO
  Initialisation
    P01 Proved      not(e5 = e1)
    P02 Unproved    e1 = e5
  Calcul
    P01 Unproved    not(e2 = e5)
    P02 Unproved    e5 = e1
End
```

Supposons que l'obligation de preuve courante soit *Calcul.1*. L'opérateur passe à la première obligation de preuve précédente non prouvée.

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

En répétant la commande, on passe à l'obligation de preuve *Calcul.2*.

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

## 4.40 Quit

### Syntaxe

**qu**

### Utilisation

Cette commande permet de quitter le prouveur interactif.

Si l'état de la preuve en cours a changé par rapport à son état lors de son chargement ou si la ligne de commande a été modifiée, le prouveur demande à l'utilisateur de sauvegarder ou non la nouvelle ligne de commande avant de quitter le prouveur interactif.

### Exemple

Un travail de preuve a été effectué sur l'obligation de preuve courante. L'opérateur désire quitter la session de preuve interactive. Le prouveur demande à l'opérateur s'il désire sauvegarder le travail de preuve de la dernière obligation de preuve utilisée.

```
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):
```

## 4.41 Reset PO

ANNULATION DE TOUTES LES COMMANDES DE LA LIGNE DE COMMANDE

### Syntaxe

**re**

### Utilisation

Cette commande permet d'annuler toutes les commandes de la ligne de commande (hormis la force courante), pour l'obligation de preuve courante.

### Exemple

Soit l'obligation de preuve qui a pour ligne de commande :

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

La commande **re**

```
PRI> re
Resetting PO
```

permet d'initialiser la ligne de commande. L'obligation de preuve retrouve son état initial.

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

## 4.42 Show reduced PO

AFFICHAGE DE L'OBLIGATION DE PREUVE AVEC HYPOTHÈSES RÉDUITES

### Syntaxe

**rp** ou **rp(n)**

avec :

— n est un entier strictement positif

### Utilisation

Cette commande permet d'afficher l'obligation de preuve courante en format hypothèses réduites.

Si  $n = 1$ , on ne sélectionne que les hypothèses qui ont un symbole commun avec le but.

Si  $n = 2$ , on réitère le processus en sélectionnant les hypothèses qui ont un symbole commun avec le but ou les hypothèses précédemment sélectionnées.

**rp** est équivalent à **rp(1)**.

**rp** permet de trouver rapidement les hypothèses susceptibles de participer à la démonstration du but. Cela s'applique, en particulier, pour les obligations de preuve issues de machines faisant beaucoup de SEES ou INCLUDES, qui peuvent avoir beaucoup d'hypothèses caractérisant des variables qui n'interviennent pas dans le but.

### Exemple

Soit la situation suivante :

```
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)
```

Avec une itération,

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

L'obligation de preuve sous format réduite est donc :

```
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
```

Avec 2 itérations

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

on obtient :

```
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
```

## 4.43 Repeat

RÉPÉTITION DE LA DERNIÈRE COMMANDE

### Syntaxe

**rr**

### Utilisation

Cette commande permet de répéter la dernière commande saisie par l'opérateur.

### Exemple

Soit la commande **dd** saisie par l'opérateur :

```
PRI> dd
Starting Deduction
```

La commande **rr**

```
PRI> rr
```

permet de répéter la dernière commande saisie.

```
Repeat: dd
Starting Deduction
```

Dans le cas où la commande saisie est une commande simultanée (voir chapitre 2.10 page 10), la commande **rr** permet de rejouer toutes ces commandes successivement.

L'opérateur exécute la commande :

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

La commande **rr**

```
PRI> rr
```

permet de répéter ces 3 commandes

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

## 4.44 Suggest for exist

INSTANCIATION DU BUT EXISTENTIELLEMENT QUANTIFIÉ

### Syntaxe

**se**( $v_1, \dots, v_n$ )

avec :

- $v_1, \dots, v_n$  sont des expressions valides ou bien le mot-clé `_h`.

### Utilisation

Cette commande permet de choisir l’instanciation de variables, sous la portée d’un quantificateur existentiel apparaissant dans le but courant.

Si le but est de la forme :

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

alors le but devient :

$P(v_1, \dots, v_n)$

Cette commande n’est protégé ni contre le mauvais typage (voir chapitre 2.2 page 3) ni contre la mauvaise définition (voir chapitre 2.3 page 4) des valeurs avec lesquelles ont souhaite instancier les variables. Il faut donc faire attention à ne pas introduire d’expressions mal typées ou mal définies.

Ceci peut être vérifié *a posteriori* à l’aide de l’outil **mdelta** (cf. Manuel Utilisateur Version 1.0.).

Si la valeur d’une ou plusieurs variables n’est pas connue ou doit rester indéterminée, il est possible d’utiliser le mot-clé `_h`, afin de ne pas instancier les variables choisies. Par exemple, si le but est :

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

alors

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

transformera le but en

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

### Exemple

Soit la situation suivante :

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

L'opérateur peut remplacer la variable *kk* par une valeur judicieusement choisie.

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

Dans ce cas, le but courant est remplacé par le but avec la variable *kk* instanciée.

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

## 4.45 Search hypothesis

### RECHERCHE D'HYPOTHÈSES

#### Syntaxe

**sh**( $P, A$ )

**sh**( $P$ )

avec :

- $P$  est un ensemble de formules séparées par les opérateurs **\_and**, **\_or**, **\_not**
- $A$  est un ensemble de formules séparées par les opérateurs **\_and**, **\_or**, **\_not**.

#### Utilisation

Cette commande permet de rechercher, parmi les hypothèses, celles qui satisfont certains critères.

$P$  représente ce que les parties d'une hypothèse doivent vérifier pour que celle-ci soit sélectionnée.  $P$  doit être un ensemble de sous-formules séparées par les trois opérateurs spéciaux **\_and**, **\_or**, **\_not**.

Par exemple : si  $P$  est égal à

$(var1 \text{ \_and } var2) \text{ \_or } var3$

on sélectionne les hypothèses qui contiennent soit  $var1$  et  $var2$  à la fois, soit  $var3$ .

Les formules avec des jokers sont autorisées.

$A$  représente ce que l'hypothèse dans sa globalité doit vérifier, dans le même langage que précédemment.

Par exemple :

$(a = b) \text{ \_or } (a \Rightarrow b)$

sélectionne les hypothèses qui sont soit des égalités, soit des implications.

Attention !! A l'intérieur d'un **\_not**, l'usage de **\_and** et **\_or** n'est pas reconnu.

De même,  $A$  ne doit pas être équivalent à un terme du genre  $((a = b) \text{ \_and } (a \Rightarrow b))$ , sinon aucune hypothèse ne serait sélectionnée.

L'argument  $A$  peut être omis.

Si l'un des éléments de  $P$  est une variable, alors il faut que les hypothèses trouvées contiennent la variable en question.

Par exemple :

**sh**( $var$ )

sélectionne les hypothèses qui contiennent  $var$ , mais pas  $var\$i$  ou  $my\_var$  ( $var$  est considéré ici comme une formule et non comme une chaîne de caractères).

Une utilisation classique consiste à rechercher toutes les hypothèses qui font référence à une variable donnée.

## Exemple

Soit la situation suivante :

```
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)
```

On commence par rechercher toutes les hypothèses qui contiennent la variable *uu*.

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

Le résultat obtenu est :

```
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
```

Le message **match with a** est affiché car lorsque le paramètre *A* est omis, les hypothèses doivent coïncider avec le pattern très générique **a**.

On cherche ensuite les hypothèses qui font référence simultanément à *uu* et *tt*.

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

Le resultat obtenu est le suivant :

```
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
```

On cherche maintenant les hypothèses qui font référence à la variable *uu* ou qui contiennent l'expression *not(a)* (*a* est un joker).

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

Les hypothèses sélectionnées sont :

```
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
```

On cherche enfin les hypothèses qui font référence à la variable *uu* et dont la forme est *a : b* ou *a = b*.

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

Les hypothèses sélectionnées sont :

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

## 4.46 Submatch Goal

MATCH SUR UNE PARTIE DU BUT

### Syntaxe

**Sgoal**(**p** | **r** )

avec :

- **p** le pattern devant matcher avec le but
- **r** le pattern de remplacement

### Utilisation

Cette commande permet de rechercher un pattern dans le but, et est remplacée par le pattern de remplacement donné en paramètre. Si le pattern est trouvé, les jokers le composant sont instanciés, et **Sgoal** est remplacé par le pattern de remplacement.

Si le pattern n'est pas trouvé, la commande échoue.

Cette commande est principalement utilisée pour généraliser une suite de commande de preuve a plusieurs obligations de preuve

### Exemple

Soit le but suivant :

```
Hypothesis
  var1 = var2
Goal
  var1 : fonction[ens1]
```

Dans ce cas, la commande **sh(Sgoal( a : b | a))** va être remplacée par **sh(var1)** de la manière suivante :

- Le pattern **a : b** va matcher sur le but **var1 : fonction[ens1]**, avec pour valeurs "**a = var1**" et "**b = fonction[ens1]**"
- La commande **Sgoal(a : b | a)** va ensuite être remplacée par le pattern de remplacement, **a**, soit **var1**

De la même manière, la commande **eh(Sgoal( a : b | a))** va être remplacée par **eh(var1)**, permettant de réécrire le but sous la forme suivante :

```
Hypothesis
  var1 = var2
Goal
  var2 : fonction[ens1]
```

## 4.47 Show Proof

### AFFICHAGE DES COMMANDES DE PREUVE SAUVEGARDÉES

#### Syntaxe

**sp(o.i)**

avec : o.i désignant une obligation de preuve du composant en cours de preuve.

#### Utilisation

Cette commande permet d'afficher le niveau de force et les commandes de preuve sauvegardées de l'obligation de preuve o.i.

#### Exemple

L'obligation de preuve courante est *Initialisation.5*. L'opérateur effectue un travail de preuve et désire utiliser la preuve de l'obligation de preuve *Calcul.7* prouvée précédemment.

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

Cela conduit à l'affichage suivant :

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

## 4.48 Submatch Hypothesis

MATCH SUR UNE HYPOTHÈSE

### Syntaxe

**Shyp**(**p** | **r** )

avec :

- **p** le pattern devant matcher une hypothèse
- **r** le pattern de remplacement

### Utilisation

Cette commande recherche une hypothèse de la forme donnée par le pattern **p**, et est remplacer par le pattern de remplacement donné en paramètre. Si le pattern est trouvé, les jokers le composant sont instanciés, et **Shyp** est remplacé par le pattern de remplacement.

Si aucune hypothèse ne correspond au pattern **p**, la commande échoue. Si plusieurs hypothèses correspondent au pattern **p**, la plus récente est utilisée.

Cette commande est principalement utilisée pour généraliser une suite de commande de preuve a plusieurs obligations de preuve

### Exemple

Soit l'hypothèse suivante :

Hypothesis  
var1 : fonction[ens1]

Dans ce cas, la commande **ah(Shyp( a : b[c] | c < : dom(b)))** va être remplacée par **ah(ens1 < : dom(fonction))** de la manière suivante :

- Le pattern **a : b[c]** va matcher sur l'hypothèse **var1 : fonction[ens1]**, avec pour valeurs "**a = var1**", "**b = fonction**" et "**c = ens1**".
- La commande **Shyp(a : b[c] | c < : dom(b))** va ensuite être remplacée par le pattern de remplacement, **c < : dom(c)**, soit **ens1 < : dom(fonction)**

## 4.49 Save with question

SAUVEGARDE AVEC QUESTION

### Syntaxe

**sq**

### Utilisation

Cette commande permet de faire une sauvegarde du travail de preuve (ligne de commande) effectué sur les obligations de preuve courantes, si c'est utile.

S'il y a un risque de perte ou de régression, une confirmation est demandée à l'opérateur. L'opérateur qui travaille depuis longtemps sur la même obligation de preuve aura intérêt à utiliser la commande **sq** pour ne pas perdre son travail en cas de panne de courant.

### Exemple

L'obligation de preuve courante est *Initialisation.1*. L'opérateur a effectué un travail de preuve et voudrait le sauvegarder, si c'est nécessaire.

```
PRI> sq
```

L'ancienne et la nouvelle ligne de commande ne permettent pas de prouver l'obligation courante. Une confirmation de sauvegarde est alors demandée :

```
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):
```

L'opérateur souhaite sauvegarder son travail.

```
Yes
```

L'obligation de preuve est sauvegardée :

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

## 4.50 Search rule

### RECHERCHE DE RÈGLES

#### Syntaxe

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

**sr**(T,CO)

**sr**

avec :

- T est une liste de théories séparées par des points (par exemple :  $t_1.t_2.t_3$ )  
 $t_i$  est soit un nom de théorie, pris parmi les théories de la base de règles, soit un mot-clef recouvrant plusieurs noms de théories de la base de règles.  
 Les mot-clefs sont les suivants :
  - **All** : toutes les théories de la base de règles, incluant la base de règles utilisateur
  - **Rewr** : les théories concernant les règles de réécriture
  - **Back** : les théories concernant les règles de déduction utilisées en tactique backward
  - **Fwd** : les théories concernant les règles de déduction utilisées en tactique forward
  - **Brpr** : les théories de la base de règles utilisateur
  - **Pmm** : les théories du fichier *.pmm*.
- CO permet de définir des critères de sélection des conséquents de règles. Il prend une des formes suivantes
  - **Goal** : les règles sélectionnées doivent se déclencher sur le but courant de l'obligation de preuve dans le contexte courant des hypothèses
  - **Goal2** : le conséquent des règles sélectionnées coïncide avec le but courant de l'obligation de preuve (c'est une contrainte moins forte que la sélection précédente)
  - **abs**(C),D : les règles sélectionnées doivent avoir un conséquent ayant même forme que C et contenant au moins une sous-formule ayant même forme que D.
- AN permet de définir des critères de sélection des antécédents de règles
  - **abs**(A),B : les règles sélectionnées doivent avoir un antécédent ayant même forme que A et contenant au moins une sous-formule ayant même forme que B.

#### Utilisation

Cette commande permet de rechercher une ou plusieurs règles dans l'ensemble des règles utilisées par le prouveur.

**CO** et **AN** sont des listes de formules et traduisent les critères que devront vérifier les règles sélectionnées.

**CO** identifie les contraintes liées au conséquent de règles et **AN** celles liées à l'antécédent. Les critères de recherche qui peuvent concerner l'antécédent et/ou le conséquent d'une règle, sont de deux types :

- **Le type absolu**, repéré par le préfixe **abs**, opère une sélection d'après la forme globale du conséquent de règle ou de l'antécédent

- **Le type relatif** opère une sélection suivant la forme d'une ou plusieurs sous formules du conséquent ou de l'antécédent

Dans le cas où la sélection ne concernerait que le conséquent de règle, AN sera omis.

Si l'opérateur recherche uniquement les règles s'appliquant sur l'obligation de preuve courante, il lui suffira d'entrer **sr**, la sélection s'effectuera alors parmi les règles sélectionnées par les mot-clefs Back et Rewr (**sr** équivaut à **sr(Back.Rewr, Goal)**).

Les critères C, D, A et B peuvent être omis. Si les critères A ou B sont présents, les critères C et D doivent l'être aussi (l'ordre des mot-clés est important). Dans ce cas, l'opérateur utilisera **abs(No), No** pour C, D afin d'exprimer qu'aucun critère ne s'applique sur le conséquent. La commande sera alors **sr(T, abs(No), No, abs(A), B)**.

De manière générale, les formules entrées sont formées d'expressions séparées par les opérateurs spéciaux **\_and**, **\_or** et **\_not**. **L'argument de \_not devra être mis entre parenthèses.**

Pour être sélectionnée, une règle devra donc contenir les expressions concernées suivant les indications données par ces opérateurs.

Par exemple, la formule qui suit permet de sélectionner les règles dont le conséquent est de la forme  $a=b$  (contrainte absolue), qui contient des surcharges, des unions, mais pas d'ensemble en compréhension :

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

**\_and** et **\_or** sont interdits à l'intérieur d'un **\_not** où leur emploi serait superflu. En effet, **\_not(X \_and Y)** équivaut à **\_not(X) \_or \_not(Y)**.

## Exemple

Par défaut, **sr** est équivalent à **sr(All,Goal)**.

```
PRI> sr
Searching rules which can be applied to the current goal in Rewr.Back
```

On recherche dans les théories SimplifyX et DifferenceX les règles dont le conséquent coïncide avec le but courant.

```
PRI> sr(SimplifyX.DifferenceXY, Goal2)
Searching rules matching with goal in : SimplifyX.DifferenceXY
```

Enfin il est possible de sélectionner les règles, en fonction de la forme générale et des expressions contenues dans les conséquents et dans les antécédents.

On recherche les règles dont le conséquent est de la forme  $a = b$  et qui contient au moins une formule de la forme  $\neg c$  :

```
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
```

On recherche les règles dont le conséquent est de la forme  $a = b$  et qui contient au moins une formule de la forme  $\neg c$ , et dont l'un des antécédents est de la forme  $a \geq b$  :

```
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
```

On recherche les règles dont l'un des antécédent est de la forme  $a \geq b$  :

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

On recherche les règles dont le conséquent est de la forme  $a \geq b$  :

```
PRI> sr(All, abs(a>=b))
Searching in All rules with filter
    consequent should match with a => b
```

On recherche les règles dont l'un des antécédents doit contenir au moins une formule de la forme  $not(a - b)$  :

```
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)
```

## 4.51 Simplify Set

SIMPLIFICATION DES EXPRESSIONS ENSEMBLISTES DU BUT COURANT

### Syntaxe

ss

### Utilisation

Cette commande déclenche un certains nombre de simplifications sur les expressions ensemblistes composant le but. Les mécanismes mis en place sont nettement plus puissants que les règles de la base de règles. Cette commande doit donc améliorer la simplification du but.

Cette simplification ensembliste utilise principalement trois outils.

- Le premier outil effectue les simplifications sur des expressions composées de valeurs littérales.
- Le deuxième outil travaille sur des termes quelconques.
- Le troisième outil essaye d'utiliser les informations de la pile d'hypothèses.

Le premier outil travaille sur les opérateurs ensemblistes et fonctionnels suivants :

- union ( $A \cup B$ )
- intersection ( $A \cap B$ )
- différence ensembliste ( $A - B$ )
- union généralisée ( $\text{union}(E)$ )
- intersection généralisée ( $\text{inter}(E)$ )
  
- inversion de relation ( $r^{-1}$ )
- domaine ( $\text{dom}(r)$ )
- co-domaine ( $\text{ran}(r)$ )
- identité ( $\text{id}(r)$ )
- restriction ( $u \triangleleft r$ )
- anti-restriction ( $u \triangleleft\!\!\triangleleft r$ )
- corestriction ( $r \triangleright v$ )
- anti-corestriction ( $r \triangleright\!\!\triangleright v$ )
- image ( $r[w]$ )
- évaluation de fonction ( $f(x)$ )
- surcharge ( $r \triangleleft q$ )
- produit direct ( $p \otimes q$ )
- composition ( $p; q$ )
- produit parallèle ( $p || q$ )
- première et deuxième projection ( $\text{prj}_1(E, F), \text{prj}_2(E, F)$ )
- produit cartésien ( $A \times B$ )
- cardinal ( $\text{card}(E)$ )

- transformation d'intervalle ( $a..b$ ) en énumération

De plus, il est capable de gérer l'ensemble **BOOL**, et certaines opérations sur l'ensemble

telle que l'appartenance à un ensemble.

En raison de la complexité de l'algorithme, seuls les opérateurs ensemblistes sont reconnus par le deuxième outil :

- union ( $A \cup B$ )
- intersection ( $A \cap B$ )
- différence ensembliste ( $A - B$ )
- union généralisée ( $\text{union}(E)$ )
- intersection généralisée ( $\text{inter}(E)$ )

Le troisième outil reconnaît les opérateurs :

- union ( $A \cup B$ )
- intersection ( $A \cap B$ )
- différence ensembliste ( $A - B$ )
- inversion de relation ( $r^{-1}$ )
- domaine ( $\text{dom}(r)$ )
- co-domaine ( $\text{ran}(r)$ )
- identité ( $\text{id}(r)$ )
- restriction ( $u \triangleleft r$ )
- corestriction ( $r \triangleright v$ )
- image ( $r[w]$ )
- surcharge ( $r \triangleleft q$ )
- composition ( $p; q$ )

Remarquons une limitation importante de cette commande : le nombre maximum d'éléments d'un ensemble énuméré gérable par **ss** est fixé à dix en raison de la complexité de leur traitement.

### Exemple

Soit une obligation de preuve transformée par la commande **mp** en :

```
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
```

La commande **ss** va essayer de simplifier le but.

```
PRI> ss
Begin SimplifySet
```

et le nouveau but devient :

Goal

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

Cette nouvelle forme du but est effectivement plus simple. Sans avoir d'hypothèses sur les valeurs de **xx** et **yy**, il n'est pas possible de continuer les simplifications.

## 4.52 Step

EXÉCUTION DE LA COMMANDE SAUVEGARDÉE SUIVANTE

### Syntaxe

**st**  
**st(n)**

avec :

n vaut

- une valeur numérique indiquant le nombre de pas à faire
- **End** si l'on veut rejouer toute la preuve sauvée

### Utilisation

Cette commande permet d'exécuter la commande suivante de la ligne de commande sauvegardée.

Elle permet de ré-appliquer, pas à pas, les commandes interactives d'une session de preuve précédente.

La ligne de commande sauvegardée est constituée de commandes interactives qui ont été utilisées lors d'un travail de preuve interactive antérieur (sinon la ligne de commande sauvegardée ne contient que la commande **pr** (voir chapitre 4.38 page 100)). Lorsque l'on se positionne sur une obligation de preuve, aucune commande n'est exécutée au préalable. L'opérateur a alors la possibilité de rejouer le travail de preuve précédent (sauvegardé), grâce à la commande **st**, et/ou d'utiliser d'autres commandes interactives.

Le paramètre **n** permet d'appliquer plusieurs commandes sauvées en une seule fois.

Une valeur numérique permet d'effectuer le nombre de commande spécifié. Si le nombre est supérieur au nombre de commandes sauvées, la commande Step renvoie un message d'erreur.

**End** permet de rejouer toutes les commandes interactives sauvegardées, à partir de la position courante de rejeu de la ligne de commande sauvegardée.

### Exemple

Soit l'obligation de preuve dont la ligne de commande sauvegardée est :

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

Il y a 6 commandes interactives sauvegardées ( $ar(test.1, Fwd) \wedge dd \wedge dd \wedge ar(test.2, Once) \wedge pr \wedge pr$ ). Nous allons les rejouer les unes après les autres.

L'indicateur *Saved line pos* 1 indique que la prochaine commande ajoutée par l'intermédiaire de la commande **st** sera la commande n°1, c'est à dire  $ar(test.1, Fwd)$  car  $Force(0)$  est juste un indicateur de force.

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

La première commande sauvegardée s'applique :

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

La première commande  $ar(test.1, Fwd)$  a été rejouée. L'indicateur *Saved line pos* montre que la prochaine commande exécutée par la commande **st** sera la commande n°2.

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

La seconde commande sauvegardée s'applique :

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

Il est possible de rejouer toutes les commandes jusqu'à la dernière.

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

Les quatre dernières commandes sauvegardées sont alors exécutées :

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

La ligne de commande obtenue est donc :

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

Il n'y a plus de commandes à rejouer car l'indicateur de position vaut 7, c'est à dire qu'il pointe vers la fin de la liste de commandes sauvegardées.

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

## 4.53 Save without question

SAUVEGARDE FORCÉE DE LA LIGNE DE COMMANDE COURANTE

### Syntaxe

**sw**

### Utilisation

Cette commande permet de réaliser une sauvegarde forcée du travail de preuve effectué sur l'obligation de preuve courante.

### Exemple

Soit l'obligation de preuve Calcul.2 suivante qui a pour ligne de commande

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

L'obligation de preuve courante est alors sauvegardée (état, ligne de commande), grâce à la commande **sw**.

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

## 4.54 Try everywhere

APPLICATION D'UNE SUITE DE COMMANDES À UN ENSEMBLE D'OBLIGATIONS DE PREUVE

### Syntaxe

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

avec :

- *f* représente la ligne de commande que l'on veut essayer.
  - Soit *f* est une suite de commandes séparées par des & et encadrée par des parenthèses
  - Soit *f* est le nom d'une obligation de preuve. Dans ce cas, on utilise la ligne de commande **sauvegardée** de l'obligation de preuve *t.n*.
- *m* vaut :
  - **Append** : retenter les preuves en plaçant les commandes *f* après les commandes sauvegardées
  - **Prepend** : retenter les preuves en plaçant les commandes *f* avant les commandes sauvegardées
  - **Replace** : retenter les preuves en utilisant les commandes *f* au lieu des commandes sauvegardées
- *n* vaut :
  - **Loc** : se limiter aux obligations de preuve de l'opération (ou la clause) en cours de preuve
  - **Gen** : traiter toutes les obligations de preuve du composant
  - **List(L)** : traiter toutes les obligations de preuve de la liste *L* (liste d'identifiants d'obligations de preuve i.e. *Nom\_Operation . Indice\_Obligation*, séparées par des &)
  - **Patt(P)** : traiter toutes les obligations de preuve dont le but coïncide avec la formule *P*
  - **Op(O)** : traiter les obligations de preuve correspondant à l'opération (ou la clause) de nom *O*
  - **O[A..B]** : traiter les obligations de preuve de l'opération (ou la clause) de nom *O* comprises entre *O.A* et *O.B* avec *A* **strictement positif** et *B* **supérieur ou égal à A**
- *n'* vaut :
  - **List(L)**
  - **Patt(P)**
  - **Op(O)**
  - **O[A..B]**
- *n''* vaut :
  - **List(L)**
  - **Patt(P)**
  - **Op(O)**
  - **O[A..B]**
  - **O.I** : traiter l'obligation de preuve nommée *O.I* où *O* désigne une opération (ou

- une clause) et I un numéro d'obligation de preuve
- p vaut :
  - **All** : retraiter même les obligations de preuve prouvées
  - **Unproved** : se limiter aux obligations de preuve non prouvées

### Alias

**ta(f)** est équivalent à **te(f, Remplace.Gen.Unproved)**

### Utilisation

Cette commande permet d'essayer d'appliquer une démonstration à toutes les obligations de preuve du composant en ne mémorisant cette démonstration que sur les obligations de preuve qu'elle a permis de prouver. Généralement, l'opérateur aura préparé cette suite de commandes à l'occasion de la preuve de l'une des obligations de preuve. Différents modes sont proposés, pour pouvoir agencer la nouvelle démonstration par rapport à celles déjà existantes sur chaque obligation de preuve.

Des messages envoyés à l'opérateur indiquent quelles sont les obligations de preuve qui ont changé d'état.

La ligne de commande des obligations de preuve qui sont devenues prouvées est aussi modifiée. Seules les commandes qui auront été efficaces (action non nulle sur l'état de la preuve) seront sauvegardées.

La suite de commandes peut comprendre au plus une commande de force. L'emplacement de la commande de force est indifférent puisque toute la ligne de commande est exécutée à force constante.

**te(f)** est équivalent à **te(f, Replace.Loc.Unproved)**.

**te(f,n'')** est équivalent à **te(f,Replace.n'').Unproved)** pour n'' distinct de List(L) et O.I.

**te(f,List(L))** est équivalent à **te(f, m.List(L).All)**. De plus l'option Unproved n'est pas disponible avec List(L).

**te(f,O.I)** est équivalent à **te(f,O[I..I])**.

**te(f,m.n')** est équivalent à **te(f,m.n'.Unproved)** pour n' distinct de List(L).

**te(f,m.List(L))** est équivalent à **te(f,m.List(L).All)**.

### Exemple1

Soit le composant contenant la clause Initialisation, l'opération Calcull et sept obligations de preuve, toutes non prouvées. On affiche la situation grâce à la commande **gs** (voir chapitre 4.22 page 67) :

```
PRI> gs
```

On obtient :

```
State of all P0
  Initialisation
    P01 Unproved      1: 1..10
    P02 Unproved      1: 1..100
    P03 Unproved      not(e5 = e1)
    P04 Unproved      e1 = e5
  Calcul
    P01 Unproved      xx+yy-1: 1..100
    P02 Unproved      not(uu = e1)
    P03 Unproved      e1 = e5
End
```

L'opérateur tente d'appliquer la ligne de commande *dd & pr* pour toutes les obligations de preuve du composant, en remplaçant en cas de preuve réussie la ligne de commande existante par *dd & pr*. La ligne de commande existante est ignorée (argument *Replace*).

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

Les obligations de preuve *Initialisation.1*, *Initialisation.2*, *Initialisation.3*, *Initialisation.4* et *Calcul.1* sont prouvées et sauvegardées.

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

Le résultat de l'application de la ligne de commandes est alors affiché.

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

On vérifie, grâce à la commande **gs** (voir chapitre 4.22 page 67), que cinq obligations de preuve ont été effectivement prouvées.

```

PRI> gs
State of all P0
  Initialisation
    P01 Proved      1: 1..10
    P02 Proved      1: 1..100
    P03 Proved      not(e5 = e1)
    P04 Proved      e1 = e5
  Calcul
    P01 Proved      xx+yy-1: 1..100
    P02 Unproved    not(uu = e1)
    P03 Unproved    e1 = e5
End

```

## Exemple2

Soit le composant suivant contenant la clause Initialisation, l'opération Analyse et six obligations de preuve, toutes non prouvées. On affiche la situation grâce à la commande `gs` (voir chapitre 4.22 page 67) :

```
PRI> gs
```

On obtient :

```

State of all P0
  Initialisation
    P01 Unproved    ff: INTEGER +-> BOOL
    P02 Unproved    ff(1) = TRUE
    P03 Unproved    xx: dom(ff)
  Analyse
    P01 Unproved    a1: 1..10
    P02 Unproved    a1 = a2
    P03 Unproved    a2 <= 11
End

```

L'opérateur tente d'appliquer la ligne de commande `dd & pr` sur les obligations de preuve numéro 1 d'Initialisation et numéros 2 et 3 d'Analyse. Il utilise le mode par défaut soit Replace et All :

```

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

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

```

L'opérateur utilise ensuite la démonstration sauvegardée de l'obligation de preuve *Analyse.2* pour l'essayer sur *Analyse.1* :

```

PRI> te(Analyse.2,Analyse.1)

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

```

L'opérateur décide alors de tenter à nouveau la commande *dd* puis la commande *pr* sur les obligations de preuve non prouvées de la 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

```

On remarque donc que dans notre cas, en repartant d'obligations de preuve toutes non prouvées, il aurait été judicieux d'appliquer les commandes *dd* et *pr* sur toutes les obligations de preuve de la clause *Initialisation* par exemple :

```
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
```

Enfin on aurait aussi pu utiliser les commandes de preuve précédentes seulement sur les obligations de preuve dont le but coïncide avec la formule  $x : y$ , soit :

```
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
```

```
Analyse.1 transformed         Unproved --> Proved,   dd & pr
```

```
End TryEveryWhere
```

## 4.55 Proof by attempts

PREUVE PAR TENTATIVES

### Syntaxe

**tp(m)**  
**tp(m,n)**

avec :

- m vaut
  - Goal pour une preuve par tentative se basant sur la forme du but
  - Hyp pour une preuve par tentative se basant sur les hypothèses
- n est une valeur numérique indiquant le nombre maximum de tentatives à faire.

### Utilisation

Cette commande peut être utilisée de deux façon. La première méthode se base sur la forme du but et essaye de générer des hypothèses supplémentaires en se basant sur les règles qui pourraient s'appliquer. Cette méthode utilise des règles générées de manière automatique dites règles alpha. La deuxième méthode se base sur les hypothèses. Il s'agit des règles de preuve par tentatives classique du prouveur.

Dans les deux cas, une valeur numérique peut indiquer le nombre maximum de sous-preuves tentées. La valeur par défaut de ce paramètre est 20.

### Exemple

Si le but courant est :

```
Goal
aa <: xx\yy
```

L'application de la commande `tp(Goal,20)` donne le résultat suivant :

```
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
```

Les hypothèses générées sont des hypothèses qui vont servir à la preuve du but.

## 4.56 User Simplification

### UTILISATION DE RÈGLES DE RÉÉCRITURE UTILISATEUR

#### Syntaxe

**us(T)**  
**us(T | M)**

avec :

- T la tactique de réécriture :
  - t le nom d'une théorie utilisateur (du PatchProver ou d'un fichier Pmm) ne contenant que des règles de réécriture. Ces règles peuvent être gardées mais ne doivent pas avoir d'autre antécédent.
  - t.n, le nom d'une règle d'une théorie t de réécriture utilisateur,
  - t;U, où t est le nom d'une théorie utilisateur et U une tactique de réécriture,
  - t.n;U, où t.n est le nom d'une règle de réécriture utilisateur et U une tactique de réécriture.
- M le mode d'application des réécritures :
  - soit le mot-clé **\_Goal** (cas par défaut quand le mode M n'est pas spécifié)
  - soit le mot-clé **\_AllHyp**
  - soit le mot-clé **\_Hyp(h)** avec h correspondant à l'hypothèse choisie

#### Utilisation

Cette commande permet d'utiliser des règles de réécriture utilisateur (contenues dans le PatchProver et/ou dans un fichier Pmm), soit sur l'hypothèse h, soit sur toutes les hypothèses, soit enfin sur le but courant en limitant au maximum la consommation de mémoire.

Lorsqu'une tactique composée (c'est-à-dire une liste de tactique séparée par des point-virgules) est utilisée, on applique d'abord les règles spécifiées par la tactique la plus à gauche, puis celles des autres tactiques, en parcourant la liste de gauche à droite.

Si  $M = \mathbf{Hyp(h)}$ , le but devient :

$$H \Rightarrow B$$

où H est obtenu en appliquant les simplifications sur l'hypothèse h, si elle existe.

Si  $M = \mathbf{AllHyp}$ , le but devient :

$$H \Rightarrow B$$

où H est obtenu en appliquant les réécritures sur toutes les hypothèses.

Si  $M = \mathbf{Goal}$ , le but devient :

$$B'$$

où  $B'$  est obtenu en réécrivant le but courant B à l'aide des règles de simplification.

On essaie d'appliquer une règle de réécriture donnée tant qu'elle est susceptible de s'appliquer.

### Exemple

Soit les théories utilisateurs suivantes contenues dans le PatchProver ou dans un fichier Pmm :

```

THEORY Mes_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

```

Soit alors l'obligation de preuve suivante :

```

Hypotheses
...
aa : INTEGER &
bb : INTEGER &
6 <= (xx+2)*3 &
yy: {aa,bb}
...
Goal
xx: ENS => not((xx+yy)*2 : gg[{5}])

```

On peut réécrire le but en utilisant les réécritures de Mes\_Simplifications :

```
PRV> us(Mes_Simplifications|_Goal)
```

Cela donne donc le but :

```

Goal
xx: ENS => not({(xx*2 + yy*2) |-> 5} <: gg)

```

On peut aussi par exemple ne vouloir appliquer que la première règle de réécriture de Mes\_Simplifications. On doit alors expliciter la règle que l'on souhaite utiliser. Dans notre cas, le nom de la première règle de la théorie Mes\_Simplifications est *Mes\_Simplifications.1*.

```
PRI> us(Mes_Simplifications.1)
```

Le but devient :

```
Goal
  xx: ENS => not({(xx+yy)*2 |-> 5} <: gg)
```

On a bien appliqué la première règle seulement.

Il est aussi possible d'effectuer les simplifications pour toutes les hypothèses.

```
PRI> us(Mes_Simplifications;Enum_Simp|_AllHyp)
```

Toutes les nouvelles hypothèses apparaissent comme antécédents du but courant :

```
Goal
6<=(xx*3 + xx*3) &
yy = aa or yy = bb
=>
xx: ENS => not((xx+yy)*2 : gg[{5}])
```

On peut enfin décider de ne simplifier qu'une hypothèse :

```
PRV> us(Mes_Simplifications|_Hyp(6<=(xx+2)*3))
```

On obtient alors :

```
Goal
6<=(xx*3 + xx*3)
=>
xx: ENS => not((xx+yy)*2 : gg[{5}])
```

## 4.57 Well Definedness

### AJOUT D'HYPOTHÈSES DE BONNE DÉFINITION

#### Syntaxe

**wd**  
**wd(h)**

avec :

— h est une hypothèse

#### Utilisation

Cette commande permet d'ajouter des hypothèses de bonne définition déduites de la bonne définition du but ou d'une hypothèse. Après son application, le but se présente sous la forme d'une implication, les hypothèses générées étant en antécédent de l'implication.

#### Exemple

Soit une obligation de preuve dont le but courant est :

```
toto = plus(nn)
```

L'application de la commande **wd**

```
PRI> wd
```

donne le but suivant :

```
plus : dom(plus) +-> ran(plus) &  
nn : dom(plus)  
=>  
toto = plus(nn)
```

L'antécédent de cette implication contient les hypothèses générées par la commande.

L'utilisateur peut ensuite effectuer le traitement qu'il désire sur ces hypothèses.



## Chapitre 5

# Paramétrage du Prouveur

Le prouveur utilise le mécanisme des ressources mis en place dans l'AtelierB. Celui-ci permet de spécifier des options de fonctionnement prises en compte lors de l'ouverture du projet B (cf. le paragraphe 2.6. "Paramétrer son AtelierB" du Manuel Utilisateur de l'AtelierB).

### 5.1 Temps de Coupure Paramétrable

*Ressource* : ATB\*PR\*Time\_Out.

*Valeur* : entier strictement positif.

*Signification* : temps de coupure en secondes.

*Valeur par défaut* : 300 (secondes).

Cette option permet de modifier la valeur du temps de coupure des prouveurs satellites PP (Prouveur de Prédicats) et ML (Prouveur MonoLemme) en Passe Configurable "**User\_Pass**" (voir chapitre 9 page 157) ou en phase de rejeu "**Replay**".

Cette option permet de tester en User Pass des tactiques de preuve utilisant de manière massive le prouveur de prédicats. Cette possibilité de modulation permet donc de lancer des preuves avec des temps de coupure (temps de calcul maximum autorisé avant arrêt du processus de preuve) faibles afin de pouvoir tester rapidement l'efficacité d'une telle tactique.

Une autre manière d'utiliser cette ressource est d'augmenter le temps de coupure de PP et ML lors d'une phase de rejeu (**prove replay**) d'un projet sur une machine lente : on est alors certain que les preuves réussies sur des machines rapides s'effectueront encore avec succès sur les machines plus lentes.

Exemple :

Si on dispose de la théorie **User\_Pass** suivante :

```
THEORY User_Pass IS
ff(0) & dd(0) & pp(rp.0)
END
```

et que l'on souhaite savoir si cette suite de commandes est efficace, il suffit de positionner la ressource `Time_Out` à une valeur faible, disons 10 secondes et de lancer la preuve automatique en mode `User_Pass`. Lorsque le temps de preuve avec `pp` exèdera ces 10 secondes, la preuve s'achèvera sur un échec.

## 5.2 Normalisation des formules $P \Rightarrow Q$ et $\neg P$

*Ressource* : `ATB*PR*Keep_Non_Simplified_Hypothesis`.

*Valeur* : `TRUE` ou `FALSE`.

*Signification* : On conserve les prédicats non simplifiés seulement si cette ressource vaut `TRUE`.

*Valeur par défaut* : `TRUE`.

Le principe de simplification de prédicats de la forme  $P \Rightarrow Q$  et  $\neg P$  consiste à transformer ces formules en  $P \wedge P' \Rightarrow Q$  et  $\neg(P \wedge P')$  (où  $P'$  est la forme simplifiée de  $P$ ). ce fonctionnement correspond à la valeur `TRUE` pour cette ressource.

A *contrario*, la valeur `FALSE` permet de ne conserver que les prédicats simplifiés *i.e.* de transformer les formules initiales en  $P' \Rightarrow Q$  et  $\neg P'$ .

Dans certains cas la présence simultanée du prédicat non simplifié et du prédicat simplifié peut empêcher l'application de mécanismes du prouveur. Par exemple, le prouveur ne peut effectuer un *Modus Ponens* sur les hypothèses  $P$ ,  $P'$  et  $(P \wedge P') \Rightarrow Q$  alors qu'il le fait sur les hypothèses  $P'$  et  $P' \Rightarrow Q$ .

Exemple :

Soit l'obligation de preuve suivante :

```
Goal
x1<=3-y1 => x1<=2+ii &
x1+y1<=3 &
=>
5+z1 = y1
```

Nous sommes dans le cas où la ressource est positionnée à la valeur **TRUE**. En effectuant successivement un `dd(0)` (pour faire monter dans la pile des hypothèses les deux hypothèses locales) puis `sh(x1)` pour visualiser les hypothèses contenant le terme  $x1$ , on obtient :

```
Found hypothesis List is
  x1+y1<=3 &
  0<=3-x1-y1 &
  0<=3-x1-y1 & x1+y1<=3 => x1<=2+ii
End of found hypothesis
```

Remarquons que l'hypothèse  $0 \leq 3 - x1 - y1$  (respectivement  $0 \leq 3 - x1 - y1 \wedge x1 \leq 3 - y1 \Rightarrow x1 \leq 2 + ii$ ) est la version simplifiée de la formule  $x1 + y1 \leq 3$  (respectivement

$x1 \leq 3 - y1 \Rightarrow x1 \leq 2 + ii$ ), on a donc conservé comme prévu les versions non simplifiées.

Puisque nous disposons des deux hypothèses  $x1 + y1 \leq 3$  et  $0 \leq 3 - x1 - y1$ , le *modus ponens* sur l'hypothèse  $0 \leq 3 - x1 - y1 \wedge x1 \leq 3 - y1 \Rightarrow x1 \leq 2 + ii$  devrait être possible :

```
Invalid argument / Inexistent a=>b Hypothesis in
mh(0<=3-x1-y1 & x1<=3-y => x1<=2+ii)
```

Le prouveur ne parvient pas effectuer le *Modus Ponens*.

On positionne la ressource à la valeur **FALSE**. En effectuant les mêmes commandes interactives sur la même obligation de preuve, on obtient :

```
Found hypothesis List is
  x1<=2+ii &
  0<=3-x1-y1 &
  0<=3-x1-y1 => x1<=2+ii
End of found hypothesis
```

Cette fois nous avons conservé uniquement les versions simplifiées des formules. Remarquons que l'on dispose aussi de l'hypothèse  $x1 \leq 2 + ii$  obtenue par un *modus ponens* déclenché automatiquement sur  $0 \leq 3 - x1 - y1 \Rightarrow x1 \leq 2 + ii$ . Le prouveur n'a pas été gêné par la présence d'hypothèses non simplifiées.

### 5.3 Paquetages de Règles Additionnelles

*Ressource* : ATB\*PR\*Use\_Rule\_Package.

*Valeur* : liste d'identificateurs de paquetages (ou de théories) séparées par des virgules.

*Signification* : liste des paquetages de règles additionnelles (*simplification*, *backward* et *forward*) à utiliser dans le cœur de preuve.

*Valeur par défaut* : le symbole "?".

Cette nouvelle fonctionnalité permet l'utilisation de paquetages de règles additionnelles. Ces paquetages de règles validées sont constituées de trois différentes catégories de règles : *simplification*, *backward* et *forward* (voir chapitre 2.4 page 5) et sont utilisées par le prouveur automatique comme des règles de la base de règles classique.

Dans la version actuelle, seul le paquetage **p1** a été ajouté. Pour accéder aux règles de simplification (respectivement, backward et/ou forward), il suffit de positionner la ressource **ATB\*PR\*Use\_Rule\_Package** à la valeur **s1** (respectivement **b1** et/ou **f1**). Si on veut utiliser toutes les règles de **p1**, il suffit alors de spécifier la valeur **p1** dans le fichier de ressource.

Positionner la ressource à la valeur "?" signifie que l'on n'utilise aucune règle du paquetage **p1**.

A terme, le but est de pouvoir utiliser dans le prouveur automatique plusieurs paquets de manière incrémentale.

Les règles du paquetage `p1` permettent de traiter des opérateurs du langage **B** qui n'étaient pas complètement couverts par les règles de la base native du prouveur : le modulo `mod`, le minimum `min`, le maximum `max`, la division entière `/`, la somme  $\sum$  et le produit  $\prod$ .

Exemple :

Soit le but  $xx \bmod tt \leq nn$  sous les hypothèses :

```
tt<=nn &
1<=tt &
tt:INT1 &
xx<=nn &
xx:INT1 &
1<=nn &
nn:INT1
```

Lorsque la ressource *n'est pas* positionnée à la valeur `p1`, la commande `pr` échoue. Par contre, cette commande réussit lorsque la ressource vaut `p1`.

## 5.4 Base de règles utilisateur

*Ressource* : `ATB*PR*BRPR_Path`.

*Valeur* : le chemin vers le fichier contenant les règles utilisateur.

*Signification* : Fichier contenant des règles utilisateur chargées automatiquement au démarrage du prouveur

Les fichiers `.pmm` ((voir chapitre 6 page 151)) permettent d'ajouter des règles pour la preuve d'un composant. La base de règles utilisateur est un mécanisme similaire permettant d'ajouter des règles utilisateur de manière globale dans l'Atelier B. Le fichier référencé par la ressource `ATB*PR*BRPR_Path` contient un ensemble de théories chargées automatiquement par l'Atelier B lors du démarrage du prouveur.

En outre, les trois théories `BRPR_Backward`, `BRPR_Simplify` et `BRPR_Forward` peuvent être définies. Ces théories sont utilisées directement par le prouveur automatique, et peuvent permettre d'améliorer le taux de preuve automatique.

La théorie `BRPR_Forward` doit être de la forme `THEORY BRPR_Forward IS Tac(T) END`, et les théories `BRPR_Backward` et `BRPR_Simplify` contiennent des règles de preuve.

## 5.5 Nombre Maximum d’Instanciation d’Hypothèses Quantifiées Universellement

*Ressource* : ATB\*PR\*Max\_Number\_Of\_Universal\_Hypothesis\_Instantiation.

*Valeur* : un quadruplet d’entiers littéraux positifs (séparés par des virgules).

*Signification* : Nombre maximum d’application du mécanisme **GenAny** pour chacune des forces de preuve.

*Valeur par défaut* : (100, 200, 1000, 10000).

Cette ressource permet de limiter le nombre d’application des règles forward **GenAny** sur des hypothèses quantifiées universellement en fonction de la force de preuve utilisée : ainsi la première valeur du quadruplet correspond au nombre d’application maximum des règles de GenAny par hypothèse quantifiée universellement pour la force 0, le deuxième pour la force 1, le troisième pour la force 2 et le dernier pour la force 3.

Le mécanisme **GenAny** du prouveur permet de particulariser des hypothèses de la forme  $\forall(X).(P(X) \Rightarrow Q)$  (où  $P(X)$  est un prédicat vérifié par  $X$ ) en fonction d’hypothèses  $P(X_i)$ , i.e. de générer les hypothèses  $[X := X_i]Q$  pour chacun des  $X_i$  vérifiant  $P$ .

Cette option permet donc à l’utilisateur de modifier le nombre maximum d’application des règles GenAny sur des hypothèses quantifiées universellement et ce, pour chaque force de preuve séparément.

Lorsque un composant B comporte un certain nombre d’hypothèses quantifiées universellement, il est en général intéressant de limiter le nombre d’application des règles de GenAny sur chacune des hypothèses : cela permet d’éviter de générer trop d’hypothèses dont peu d’entre elles serviront à la preuve proprement dite et ainsi d’éviter une explosion combinatoire.

## 5.6 Trace de règles utilisateur

*Ressource* : ATB\*PR\*Trace\_User\_Rules.

*Valeur* : TRUE ou FALSE.

*Signification* : Indique si l’application d’une règle utilisateur doit afficher un message.

*Valeur par défaut* : FALSE.

Dans le cas où cette ressource est positionnée à TRUE, un message est affiché à chaque fois qu’une règle utilisateur est appliquée. Ce message est de la forme suivante : **Applying user rule  $T.n$** , où  $T.n$  correspond au nom de la règle utilisateur.  $T$  correspond au nom de la théorie ou la règle est définie, et  $n$  a la position de cette règle dans la théorie.

Utiliser cette ressource peut permettre de déterminer quelles sont les règles utilisateur utilisées.



## Chapitre 6

# Proof Manual Method : ajout de règles utilisateur

Le prouveur automatique dispose de mécanismes généraux de preuve, reposant sur une base de règles dont la portée n'est pas universelle.

Ces mécanismes ont été conçus pour résoudre une large gamme d'obligations de preuve, principalement les obligations de preuve peu complexes. Le prouveur a donc un pouvoir de résolution limité. Pour permettre de traiter les cas de preuve les plus difficiles, il est possible, outre d'orienter la preuve par l'intermédiaire de commandes interactives (**ah**, **dd**, **ph**, **se**, ...) et d'utiliser des règles manuelles. Ces règles peuvent correspondre à des manques de la base de règles. Elles permettent aussi de décharger une obligation de preuve, lors du premier pas de preuve interactive, en incorporant l'obligation de preuve (sous forme "jokerisée") dans le fichier *pmm* du composant.

Ce fichier se situe dans le même répertoire que le composant, a pour racine le nom du composant et pour extension *pmm*.

Il est écrit en langage de théorie<sup>1</sup>.

Les règles contenues dans le fichier *.pmm* sont chargées par la commande **pc** (voir chapitre 4.35 page 90) et appliquées par la commande **ar** (voir chapitre 4.5 page 23).

Lors de l'accès au fichier *.pmm*, le prouveur affiche un message d'acceptation du fichier ou un message d'erreur.

Les règles doivent obligatoirement être équipées du système de trace (voir chapitre 10 page 161), si l'on veut utiliser le système de trace lors des démonstrations et obtenir l'arbre de preuve. Si des règles *pmm* sont appliquées et ne sont pas tracées, le comportement du module de génération d'arbre de preuve n'est plus garanti.

### Attention !

Alors que toutes les autres fonctionnalités du démonstrateur interactif sont totalement protégées, cette possibilité d'application de règles écrites manuellement ne l'est pas.

Il est possible d'entrer une règle fautive, provoquant ainsi des démonstrations fausses.

---

1. voir le *Manuel de référence du Logic Solver*

Si aucune règle manuelle de ce type n'a été utilisée, alors la validité du démonstrateur (automatique + interactif) suffit à assurer la validité de la preuve, quelles que soient les commandes interactives appelées.

Si par contre, des règles manuelles ont été ajoutées, alors il faudra s'assurer de ces règles. L'emploi d'un démonstrateur de règles pourra être préconisé pour cette tâche ; mais il est clair que l'esprit dans lequel est fait le démonstrateur interactif est d'éviter l'emploi de ces règles manuelles.

## Chapitre 7

# Patchprover : ajout direct de règles dans le prouveur

Ce système s'utilise de la manière suivante :

- Créer un fichier de nom PatchProver dans le répertoire “base de donnée du projet” (*bdp*) concerné
- Programmer en langage de théories dans ce fichier, sachant que :
  - Les règles de la théorie *PatchProverB<sub>i</sub>* où *i* est la force seront appliquées par la force *i* AVANT les règles et les mécanismes du prouveur.
  - Les règles de la théorie *PatchProverA<sub>i</sub>* où *i* est la force seront appliquées par la force *i* APRES les règles et les mécanismes du prouveur (juste avant l'échec).
  - Les règles de la théorie *PatchProverH<sub>i</sub>* sont appliquées sur la formule conjonctive de chaque paquet d'hypothèses qui est chargé en force *i*.
  - La force *Rapide* n'est pas équipée du *PatchProver*.
- Dans *PatchProverB<sub>i</sub>*, B signifie “Before” et dans *PatchProverA<sub>i</sub>*, A signifie “After”.
- Ces théories sont déclarées vides dans le prouveur :
  - PatchProverH0 PatchProverH1 PatchProverH2 PatchProverH3
  - PatchProverB0 PatchProverB1 PatchProverB2 PatchProverB3
  - PatchProverA0 PatchProverA1 PatchProverA2 PatchProverA3
- AUCUNE NORMALISATION N'EST FAITE DANS CE FICHER. Donc ne jamais utiliser les formes de gauche du tableau de normalisation. En particulier : dans les notations internes du prouveur, {e} doit toujours être un singleton. Sinon le comportement ultérieur n'est pas garanti.
- Toutes ces théories sont des backward. Les théories *PatchProverH<sub>i</sub>* doivent fonctionner comme des réécritures. En effet, elles sont appelées sur chaque paquet d'hypothèses par

```
bguard((PatchProveri~;RES): bresult(H), Q)
```

On peut créer d'autres théories, faire des bcall et des bguard, etc... Pour sortir des messages, utiliser

```
bcall(WRITE: bwritef(...))
```

- ATTENTION : L'USAGE DE PATCHPROVER EST RESERVE A L'UTILISATEUR CONNAISSANT LE LANGAGE DE THEORIES. CE N'EST PAS UNE MÉTHODE SECURITAIRE. PatchProver n'est lu qu'au démarrage du prouveur automatique

ou interactif. Après avoir modifié ce fichier, il vaut mieux dé-prouver toutes les obligations de preuve et lancer une preuve en rejeu.

- Si PatchProver contient des erreurs de syntaxe, il n'est pas pris en compte et n'a donc pas d'influence sur la preuve.

Les règles et les appels de mécanismes doivent obligatoirement être équipés du système de trace (voir chapitre 10 page 161), si l'on veut utiliser le mécanisme de trace lors des démonstrations et obtenir l'arbre de preuve. Si des règles du PatchProver sont appliquées et ne sont pas tracées, le comportement du module de génération d'arbre de preuve n'est plus garanti.

## Chapitre 8

# User Simplification : théories de simplification de l'utilisateur

Cette fonctionnalité permet à l'utilisateur d'employer des théories de simplification particulières, dans le cadre de la réalisation de règles utilisateur. Ces théories, ne contenant que des règles de réécriture, contenues dans le PatchProver et/ou dans le fichier Pmm associé, sont utilisables à l'aide de la garde en Langage de Théorie :

**bguard(UserSimpX : UserSimpG(T | B), R)**

où **T** est la **tactique** de preuve, **B** une formule que l'on veut simplifier et **R**, un **joker** (syntaxiquement : une unique lettre), qui reçoit le résultat de l'application des règles de la tactique **T** sur la formule **B**.

Les tactiques de preuve représentent l'ordre d'application des règles de simplifications. Leur syntaxe est la suivante :

**Tactique ::= T | T.n | T; Tactique | T.n; Tactique**

où *T* représente un nom de théorie de réécriture et *n* un entier (donc *T.n* dénote le nom d'une règle de la théorie *T*). Ainsi quand la tactique est réduite à un nom de théorie, on tentera l'application de toutes les règles de réécriture la composant. Quand la tactique est un nom de règle, seule la règle de réécriture correspondante sera utilisée. Enfin, si la tactique est de la forme **U ; V** où **U** et **V** sont des tactiques, on exécutera d'abord la tactique **U** puis **V**.

**EXEMPLE**

Soit par exemple les règles de réécriture suivantes contenues dans le PatchProver :

```

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

```

On peut utiliser ces règles au niveau d'autres règles utilisateur de manière simplifiée et optimisée pour réduire la consommation mémoire à l'aide de la théorie prédéfinie *UserSimpX* :

```

THEORY Preuve_Admise 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

```

## Chapitre 9

# User Pass : utilisation de passes configurables

### 9.1 Présentation

Il est possible de définir des tactiques de preuve et de les utiliser en preuve automatique. Ces tactiques de preuve sont constituées de lignes de commandes interactives. Chaque ligne de commandes sera testée sur toutes les PO restant à prouver. L'appel au prouveur automatique se fait en “Automatic User Pass”.

Les tactiques de preuve sont définies :

- soit dans le fichier PatchProver (voir chapitre 7 page 153)
- soit dans le fichier pmm (voir chapitre 6 page 151) associé à chaque composant à prouver

Elles doivent être contenues dans la théorie **User\_Pass**. Si la théorie **User\_Pass** est définie à la fois dans le fichier PatchProver et dans le fichier pmm, seule la théorie contenue dans le PatchProver sera prise en compte et on obtiendra alors le message suivant :

```
Theory User_Pass Not Loaded because name clashes with native Theories
```

Un exemple de théorie **User\_Pass** est donnée ci-dessous :

```
THEORY User_Pass IS

  dd(0) & pr(Red);
  dd(1) & pr(Red) & dd(1) & pr(Red);
  dd(1) & tp(Goal,10)

END
```

La première ligne de commandes utilisée sera donc :

```
dd(0) & pr(Red)
```

Sur les PO non prouvées après application de cette première ligne de commande, la ligne de commandes suivante sera appliquée :

dd(1) & pr(Red) & dd(1) & pr(Red)

Enfin, pour les PO restantes, c'est la dernière ligne de commandes qui sera utilisée :

dd(1) & tp(Goal,10)

## 9.2 Filtres pour User\_Pass

Il est possible de filtrer les tactiques de preuve de la manière suivante :

- Utilisation du mot-clef **Operation** : Si l'on souhaite utiliser certaines commandes de preuve uniquement sur les obligations de preuve (non prouvées) d'une opération (ou clause)  $o$ , il suffit de rajouter dans la ligne de commande de la théorie User\_Pass, le mot-clef **Operation(o)**,
- Utilisation du mot-clef **Pattern** : Si l'on souhaite utiliser certaines commandes de preuve uniquement sur les obligations de preuve (non prouvées) dont le but sans les hypothèses locales coïncide avec une formule  $f$ , il suffit de rajouter dans la ligne de commande de la théorie User\_Pass, le mot-clef **Pattern(f)**.

La position du filtre dans la liste des commandes n'a aucune incidence.

Il est aussi possible de combiner les deux filtres précédents pour n'utiliser des commandes que sur les obligations de preuve d'une opération (ou clause) donnée et dont les buts coïncident avec une certaine formule.

Soit par exemple la théorie User\_Pass suivante :

THEORY User\_Pass IS

```
Operation(op0) & dd(0) & pr(Red);
Pattern(x=y) & dd(1) & pr(Red) & dd(1) & pr(Red);
Operation(op1) & Pattern(x:X) & dd(1) & tp(Goal,10)
```

END

La première ligne de commandes utilisée sur les obligations de preuve non prouvées de l'opération op0 sera donc :

dd(0) & pr(Red)

Sur les PO non prouvées après application de cette première ligne de commande dont le but coïncide avec la formule  $x = y$ , la ligne de commandes suivante sera appliquée :

dd(1) & pr(Red) & dd(1) & pr(Red)

Enfin, pour les PO non prouvées restantes de l'opération op1 et dont le but coïncide avec la formule  $x \in X$ , c'est la dernière ligne de commandes qui sera utilisée :

dd(1) & tp(Goal,10)

L'avantage de l'utilisation des filtres est de ne pas tenter inutilement l'application de commandes sur des PO dont on sait que cela ne donnera rien (en particulier à cause de la forme de leurs buts).



# Chapitre 10

## Système de trace

### 10.1 Description

Le système de trace permet de suivre l'application des règles de la base de règles du prouveur, les simplifications du but effectuées lors de l'appel de mécanismes, les preuves par cas et par tentatives, la génération, la simplification et la montée des hypothèses dérivées.

L'équipement des règles se fait de la manière suivante :

— règle Backward atomique

La règle originale n'a pas d'antécédent et est de la forme

La règle équivalente équipée du système de trace est :

$$\text{bcall1(AtomicRule(premiere\_regle))} \Rightarrow Q$$
*premiere\_regle* est le nom de baptême de la règle, qui est indépendant de la théorie d'accueil de la règle.

— règle Backward non-atomique

La règle originale est de la forme

La règle équivalente équipée du système de trace est :

$$\text{bcall1(BackwardRule(Deuxieme\_regle))} \ \& \ P \Rightarrow Q$$

— règle Forward

La règle originale est de la forme

La règle équivalente équipée du système de trace est :

$$P \Rightarrow Q \ \& \ \text{bcall1(ForwardRule(Troisieme\_regle))}$$

Pour l'équipement de mécanismes (PatchProver), on tracera l'entrée et la sortie du mécanisme. Si le code d'appel du mécanisme MECA est le suivant :

```
Traitement
=>
MECA(I, 0);
```

avec I correspondant aux données d'entrée du mécanisme MECA et O correspondant aux données de sortie, le mécanisme équivalent équipé du système de trace sera :

```
Traitement &
bcall1(SimplifyNewH(I,0))
=>
MECA(I, 0);
```

s'il s'agit d'un mécanisme transformant ou générant des hypothèses, et

```
Traitement &
bcall1(SimplifyNewG(I,0))
=>
MECA(I, 0);
```

s'il s'agit d'un mécanisme transformant le but.

## 10.2 Utilitaire

On trouvera en annexe le code source d'un programme permettant d'équiper les règles d'un fichier avec le système de trace, de manière automatique.

La mise en oeuvre de ce programme est la suivante :

1. Le programme doit être compilé (fichier *equipe.src*)

```
krt -c equipe.src equipe.kin
```

en ayant au préalable copié la table des symboles depuis `AB/press/lib/Bsym/B_ST` dans le répertoire contenant le fichier `equipe.src`.

2. Il faut créer un fichier *equipe.ex* contenant le prédicat **Equipe**, paramétré par le nom du fichier à équiper et la liste éventuelle des théories Forward (Par défaut, on considère que les théories contiennent des règles Backward).

Par exemple, si le fichier *equipe.ex* contient :

```
Equipe('test.src')
```

les règles contenues dans le fichier "test.src" seront équipées du système de trace en mode backward.

Si le fichier *equipe.ex* contient :

```
Equipe('test.src' | (toto , titi, tutu))
```

les règles contenues dans le fichier "test.src" seront équipées du système de trace en mode backward, sauf celles des théories toto, titi et tutu qui seront équipées en mode forward.

3. Le programme doit être exécuté

```
krt -b equipe.kin equipe.ex
```

Exemple d'utilisation
-----------------------

Si le fichier *equipe.ex* contient :

```
Equipe('test.src' | SensAvant)
```

et que le fichier *test,src* contient :

```
THEORY truc IS
```

```
toto  
=>  
tata;
```

```
titi  
=>  
machin;
```

```
toto;
```

```
titi
```

```
END
```

```
&
```

```
THEORY SensAvant IS
```

```
titi  
=>  
toto;
```

```
tutu  
=>  
tata
```

```
END
```

le lancement du programme *equipe*

```
krt -b equipe.kin equipe.ex
```

donnera :

```
THEORY truc IS
bcall1(BackwardRule(truc.1)) &
toto
=>
tata

;
bcall1(BackwardRule(truc.2)) &
titi
=>
machin

;
bcall1(AtomicRule(truc.3))
=>
toto

;
bcall1(AtomicRule(truc.4))
=>
titi

END

&

THEORY SensAvant IS
titi
=>
toto &
  bcall1(ForwardRule(SensAvant.1))

;
tutu
=>
tata &
  bcall1(ForwardRule(SensAvant.2))

END
```

# Chapitre 11

## Liste des commandes disponibles

### LISTE DES COMMANDES PAR THEME

catégorie	sous-catégorie	commande	signification	page
Construction des preuves	Choix du niveau de preuve	ff	Change force	58
	Appel des prouveurs	pr	Prove	100
		ml	Mono Lemma Prover	86
		mp	MiniProof	83
		pp	Predicate Prover	96
		ap	Arithmetic Prover	20
		ss	Simplify Set	125
		mc	ModelChecking	79
		tp	Proof by attempts	138
	Application d'une règle	ar	Apply rule	23
		us	User Simplification	140
	Réécriture	ae	Abstract Expression	15
		aq	Abstract predicate	22
		eh	User equality in hypothesis	54
	Cas particuliers de règle d'inférence	ct	Contradiction	41
		cts	Special contradiction	43
		fh	False hypothesis	60
		dc	Do cases	44
		dcS	Special do cases	47
		se	Suggest for exist	113
	Opérations sur les hypothèses	dd	Deduction	49
		ch	Create Hypothesis	40
		ph	Particularize hypothesis	94
		mh	Modus ponens hypothesis	84
		ah	Add hypothesis	18
		wd	Well definedness	143

catégorie	sous-catégorie	commande	signification	page
Recherche et visualisation d'information	Recherche dans la base	sr	Search rule	122
	Recherche dans le composant	gs	Global situation	67
		sp	Show Proof	119
	Recherche dans l'OP	sh	Search hypothesis	115
		la	Logical Analysis	73
		dt	Display Term	52
		gt	Graphical Trace	70
		lp	Show literal PO	77
		rp	Show reduced PO	109
		cg	Current Goal	39
	Recherche dans les commandes	help	Help	72
Navigation dans les OP		ba	Back	30
		re	Reset	108
		ne	Next	89
		pv	PreviousPO	106
		go	Goto	64
		gr	Goto with reset	66
		gw	Goto without save	71
Répétition de commandes		rr	Repeat	111
		bb	Loop	33
		te	Try everywhere	132
		st	Step	128
		fw	Forward	62
Sauvegarde des OP		sw	Save without question	131
		sq	Save with question	121
Théories utilisateur		pc	Pmm compile	90
Quitter		qu	Quit	107

## LISTE DES COMMANDES PAR ORDRE ALPHABETIQUE

Commande	Signification	page
ae	Abstract Expression	15
ah	Add hypothesis	18
ap	Arithmetic Prover	20
ar	Apply rule	23
aq	Abstract Predicate	22
ba	Back	30
bb	Loop	33
cg	Current Goal	39
ch	Create Hypothesis	40
ct	Contradiction	41
cts	Special Contradiction	43
dc	Do cases	44
dc	Special do cases	47
dd	Deduction	49
dt	Display Term	52
eh	Use equality in hypothesis	54
ff	Change force	58
fh	False hypothesis	60
fw	Forward	62
go	Goto	64
gr	Goto with reset	66
gs	Global situation	67
gt	Graphical Trace	70
gw	Goto without save	71
help	Help	72
la	Logical Analysis	73
lp	Show literal PO	77
mc	ModelChecking	79
mh	Modus ponens hypothesis	84
ml	Mono Lemma Prover	86
mp	MiniProof	83
ne	Next	89
pc	Pmm compile	90
ph	Particularize hypothesis	94
pp	Predicate Prover	96
pr	Prove	100
pv	PreviousPO	106
qu	Quit	107
re	Reset	108
rp	Show reduced PO	109
rr	Repeat	111
se	Suggest for exist	113
sh	Search hypothesis	115
sp	Show Proof	119
sq	Save with question	121
wd	Well definedness	143

Commande	Signification	page
sr	Search Rule	122
ss	Simplify Set	125
st	Step	128
sw	Save without question	131
te	Try everywhere	132
tp	Proof by attempts	138
us	User Simplification	140

## Chapitre 12

# ANNEXE

### Programme d'équipement du système de trace

Voici un programme, écrit en langage de théorie, qui permet d'équiper automatiquement du système de trace, les règles d'un fichier.

'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