

**Atelier B**

# **LOGIC-SOLVER**

**Sémantique**

**version 1.4.1**



ATELIER B  
LOGIC-SOLVER Sémantique  
version 1.4.1

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>Coincidence de formules</b>	<b>1</b>
<b>2</b>	<b>Application d'une règle à une formule</b>	<b>3</b>
2.1	Règle de déduction . . . . .	3
2.2	Règle de ré-écriture . . . . .	4
<b>3</b>	<b>Preuve</b>	<b>5</b>
3.1	Preuve d'une formule . . . . .	5
3.2	Cas particulier des formules conjonctives . . . . .	6
3.3	Comment faire une preuve avec le Logic-Solver . . . . .	6
3.4	Éviter les preuves divergentes (1) . . . . .	7
3.5	Tactique par l'arrière . . . . .	7
3.6	Succès ou échec d'une preuve initialisée par <code>bcall</code> . . . . .	9
3.7	Recul . . . . .	10
<b>4</b>	<b>Hypothèses</b>	<b>11</b>
4.1	Déduction et hypothèses . . . . .	11
4.2	Éviter les preuves divergentes (2) . . . . .	12
4.3	Hypothèses dérivées . . . . .	12
4.4	Tactique par l'avant . . . . .	12
<b>5</b>	<b>Opérations</b>	<b>15</b>
<b>6</b>	<b>Gardes</b>	<b>17</b>

# Chapitre 1

## Coincidence de formules

On dit qu'une formule  $f$  coïncide avec une formule  $g$  si l'on peut obtenir  $f$ , en remplaçant, dans  $g$ , toutes les occurrences des mêmes jokers par certaines formules. On rappelle qu'un joker est une formule atomique composée d'une lettre simple. Un joker est donc une "variable de formule". Par exemple, la formule  $g$  suivante :

$$aa + (bb/ee - (cc + dd)*aa) - bb/ee$$

coïncide avec la formule  $f$  suivante

$$x + (y - z*x) - y$$

L'affectation de certaines formules à certains jokers s'appelle un filtre. Un filtre est donc une fonction partielle des jokers vers les formules. Appliquer un filtre à une formule  $g$ , consiste à remplacer chacun des jokers de  $g$ , figurant dans le domaine du filtre, par la formule correspondante. En résumé, une formule  $f$  coïncide avec une formule  $g$ , s'il existe un filtre, dont le résultat de l'application à  $g$  donne  $f$ . Dans le cas de la coïncidence précédente, on a le filtre suivant :

$$\{ \quad x \mapsto aa, \quad y \mapsto bb/ee, \quad z \mapsto cc + dd \quad \}$$



## Chapitre 2

# Application d'une règle à une formule

Pour faire des preuves formelles, on utilise des règles d'inférence. Dans cette section, nous allons définir ce que l'on entend par l'application d'une règle d'inférence à une formule. Le résultat de l'application d'une règle  $r$  à une formule  $f$  donne d'autres formules que l'on appelle les successeurs de  $f$ . Cet ensemble de successeurs peut être vide. À noter qu'une règle n'est pas toujours applicable à une formule. Lorsqu'on applique une règle  $r$  à une formule  $f$ , on dit que  $r$  *décharge*  $f$ . On dit aussi que  $r$  *produit* un certain nombre d'autres formules.

Nous allons maintenant expliciter dans quelles conditions une règle est applicable à une formule, et, dans l'affirmative, définir quel est le résultat de cette application. On rappelle qu'une règle a la forme générale suivante :

$$a_1 \quad \& \quad a_2 \quad \& \quad \cdots \quad \& \quad a_n \quad ==> \quad c$$

où  $a_1, a_2, \dots, a_n$  sont appelés les *antécédents* de la règle et où  $c$  est appelé le *conséquent* de la règle. Pour appliquer une règle à une formule, on doit distinguer deux cas suivant la nature du conséquent  $c$  de la règle.

### 2.1 Règle de déduction

Lorsque le conséquent  $c$  de la règle n'est pas de la forme  $g == d$ , on dit que la règle est une *règle de déduction*. Par exemple, la règle suivante est une règle de déduction :

$$x < z \quad \& \quad y < z \quad ==> \quad x+y < 2*z$$

Une règle de déduction  $r$  est applicable à une formule  $f$  lorsque  $f$  coïncide avec le conséquent de  $r$ . Il en résulte un certain filtre. Le résultat de l'application de  $r$  à  $f$  correspond alors l'application de ce filtre aux différents antécédents de  $r$ . Ce résultat peut être vide lorsque la règle  $r$  n'a pas d'antécédents. Ainsi, l'application de la règle précédente à la formule suivante :

$$aa+bb+cc < 2*(cc+dd)$$

produit les deux formules suivantes :

$$aa+bb < cc+dd \qquad cc < cc+dd$$

## 2.2 Règle de ré-écriture

Lorsque le conséquent  $c$  de la règle est de la forme  $g == d$ , on dit que la règle est une *règle de ré-écriture*. Dans une telle règle, la formule  $g$  s'appelle la *partie gauche* et la formule  $d$  s'appelle la *partie droite*. Par exemple, la règle suivante est une règle de ré-écriture (sans antécédents) :

$$x*(y+z) == x*y + x*z$$

Une telle règle  $r$  est applicable à une formule  $f$  lorsqu'il existe une sous-formule  $h$  de  $f$  qui coïncide avec la partie gauche du conséquent de  $r$ . Il en résulte un certain filtre. Le résultat de l'application de  $r$  à  $f$  correspond d'abord, comme dans le cas précédent, à l'application du filtre aux différents antécédents de  $r$ , s'il y en a. Le résultat comprend aussi la formule obtenue en remplaçant, dans  $f$ , la sous-formule  $h$  par le résultat de l'application du filtre à la partie droite du conséquent de  $r$ . Ainsi, l'application de la règle précédente à la formule suivante

$$aa + bb + cc*(ee+ff) < cc + dd$$

produit la formule suivante

$$aa + bb + (cc*ee + cc*ff) < cc + dd$$

S'il existe plusieurs sous-formules rentrant en coïncidence avec la partie gauche du conséquent d'une règle de ré-écriture, la sous-formule qui est choisie est celle qui est située la plus "à droite". Par exemple, si l'on désire appliquer la règle

$$a+b == b+a$$

à la formule

$$aa+bb+cc = cc+bb+aa$$

on voit qu'il y a quatre sous-formules qui coïncident avec la partie gauche, à savoir

$$aa+bb \qquad aa+bb+cc \qquad cc+bb \qquad cc+bb+aa$$

La sous-formule choisie est donc  $cc+bb+aa$ . La règle de ré-écriture produit alors la formule suivante :

$$aa+bb+cc = aa+(cc+bb)$$

# Chapitre 3

## Preuve

### 3.1 Preuve d'une formule

Etant donné un certain ensemble de règles, on dit qu'une formule  $f$  est formellement prouvée, à l'aide de cet ensemble, lorsque l'application répétée de ces règles à la formule initiale ainsi qu'à ses successeurs, puis aux successeurs de ceux-ci, et ainsi de suite, conduit à un ensemble vide. Autrement dit, la formule  $f$  est prouvée, par rapport à un certain ensemble de règles, lorsque elle-même et tous ses descendants sont déchargés par les règles en question.

La formule initiale, ainsi que ses descendants, sont appelés les *buts* de la preuve. On parle donc du but initial et des buts intermédiaires, qui apparaissent en cours de preuve. En fin de preuve, par définition, tous les buts sont prouvés.

Cette définition laisse une certaine indétermination puisque l'on ne précise ni l'ordre dans lequel les buts intermédiaires sont déchargés, lorsqu'il y en a plusieurs à prouver, ni l'ordre dans lequel les règles sont choisies pour décharger un but donné.

À titre d'exemple, on se donne les règles suivantes :

$$x < z \quad \& \quad x < z \quad \Rightarrow \quad x+y < 2*z$$

$$a-a == 0$$

$$x < x+1$$

$$0 < x+1$$

On se propose maintenant de prouver la formule suivante :

$$aa + (bb-bb) < 2*(aa+1)$$

La première règle permet de décharger le but initial. Les deux buts intermédiaires suivants sont alors produits :

$$aa < aa+1 \qquad bb-bb < aa+1$$

Le premier but d'entre eux est déchargé par la troisième règle sans production d'aucun nouveau but. Il reste donc à prouver le deuxième but, qui est facilement déchargé par

applications de la deuxième puis de la quatrième règle (sans production de nouveaux buts dans chacun des deux cas). En fin de compte, il n'y a plus de buts à prouver : le but initial est donc définitivement prouvé, à l'aide des règles proposées.

### 3.2 Cas particulier des formules conjonctives

Lorsqu'un but est de la forme suivante :

$$f_1 \quad \& \quad f_2 \quad \& \quad \cdots \quad \& \quad f_n$$

la preuve de ce but est remplacé par celles de chaque formule  $f_1, f_2, \dots, f_n$ , qui deviennent donc de nouveaux buts intermédiaires.

### 3.3 Comment faire une preuve avec le Logic-Solver

Pour faire une preuve avec le **Logic-Solver**, il faut d'abord écrire une théorie, disons **ess**, dans un certain fichier "source", par exemple le fichier **ess** (le fait que les deux noms soient les mêmes n'a aucune importance). Cette théorie contient les règles à l'aide desquelles on se propose de faire la preuve :

```
THEORY ess IS

x < x+1;

0 < x+1;

a-a == 0;

x < z    &    y < z    =>    x+y < 2*z

END
```

On doit ensuite compiler ce fichier au moyen de la commande suivante :

```
krt -c ess
```

Cette compilation produit un certain fichier **ess.kin**. On doit enfin écrire dans un autre fichier, par exemple le fichier **ess.ex**, la formule que l'on veut prouver, en l'occurrence :

$$aa+(bb-bb) < aa+1$$

On lance alors la preuve de la façon suivante :

```
krt -b ess.kin ess.ex
```

Si la preuve échoue, l'exécution se termine avec un message. Par exemple, si l'on supprime la règle  $0 < x + 1$  de la théorie **ess**, la preuve échoue et le message suivant est imprimé par le **Logic-Solver** :

EXECUTION ABORTED ON GOAL:  $0 < aa + 1$

L'ordre dans lequel les buts sont prouvés par le **Logic-Solver** est l'ordre dans lequel ils apparaissent au fur et à mesure de l'application des règles. L'ordre dans lequel les règles sont choisies correspond à l'ordre inverse de celui dans lequel elles sont disposées dans la théorie **ess**. À noter que, lors de la preuve de chaque but, on reprend systématiquement la recherche d'une règle applicable en commençant par la dernière règle de la théorie **ess**.

### 3.4 Éviter les preuves divergentes (1)

Pour éviter qu'une preuve ne diverge, le **Logic-Solver** s'interdit d'appliquer une règle de déduction à une certaine formule  $f$  au cas où cette règle produirait un successeur identique à un ascendant de  $f$ . Dans le cas d'une règle de ré-écriture, le **Logic-Solver** agit comme ci-dessus en ce qui concerne les antécédents mais il recherche aussi une sous-formule qui ne produit pas de successeur identique à un ascendant de  $f$ . À noter que ces précautions ne garantissent pas qu'une preuve ne puisse pas diverger, elles suppriment seulement une cause certaine de divergence.

À titre d'exemple, on peut voir comment le **Logic-Solver** réussit à prouver la formule :

$$mm + nn + pp < bb + cc + aa$$

à l'aide de la théorie suivante :

THEORY **ess** IS

$$a + b + c == a + c + b;$$

$$mm + pp + nn < bb + aa + cc$$

END

Les buts suivants sont prouvés les uns après les autres :

$$mm + nn + pp < bb + cc + aa$$

$$mm + nn + pp < bb + aa + cc$$

$$mm + pp + nn < bb + aa + cc$$

On peut voir que, dans le deuxième but,  $mm + nn + pp < bb + aa + cc$ , la deuxième sous-formule possible,  $mm + nn + pp$ , est choisie de préférence à la première,  $bb + aa + cc$ . En effet cette dernière engendrerait, de nouveau, le premier but.

### 3.5 Tactique par l'arrière

On peut aussi envisager de faire une preuve avec plusieurs théories. On peut, en effet, mettre plusieurs théories dans le fichier source, chaque théorie étant séparée de la suivante

par le symbole  $\&$  comme le demande la syntaxe du Langage de Théorie. On compile ensuite le fichier source et on lance la preuve comme il a été indiqué précédemment.

La preuve s'effectue d'abord à l'aide des règles de la première théorie comme nous l'avons vu plus haut. On ne commence à utiliser les règles des autres théories que lorsque l'on rencontre un but de la forme suivante :

$\text{bcall}(t:f)$

Dans ce but,  $t$  est une tactique (dite tactique “par l’arrière”) et  $f$  est une formule quelconque (qui, cependant ne contient pas elle-même de sous-formule correspondant à l’opération  $\text{bcall}$ ). Au lieu de tenter d’appliquer une règle à un tel but, **Logic-Solver** va l’interpréter directement, comme l’initialisation d’une nouvelle preuve, celle de  $f$ , à l’intérieur de la preuve en cours. Lorsque la preuve de  $f$  est achevée, on revient à la preuve en cours. La preuve de  $f$  est guidée par la tactique  $t$  comme on va l’expliquer maintenant.

On rappelle que les différentes formes de tactique sont les suivantes :

	Commentaire
$t;u$	$t$ et $u$ sont des tactiques
$t\sim$	$t$ est une tactique
$Id$	$Id$ est un nom de théorie

L’utilisation d’une tactique s’effectue de la façon suivante :

	Utilisation
$t;u$	On utilise la tactique $t$ puis la tactique $u$
$t\sim$	On utilise la tactique $t$ tant qu’elle réussit
$Id$	On applique <i>au plus</i> une règle de la théorie $Id$ en suivant l’ordre inverse des règles de cette théorie

Les conditions de réussite d’une tactique, conditions dont on a besoin pour utiliser une tactique de la forme  $t\sim$ , sont les suivantes :

	Condition de réussite
$t;u$	L'une des deux tactiques $t$ ou $u$ réussit
$t^\sim$	La tactique $t$ réussit au moins une fois
$Id$	Une règle de la théorie $Id$ est applicable

Par exemple, on peut prouver maintenant la formule  $aa+(bb-bb) < 2*(aa+1)$  en organisant les règles comme suit dans deux théories :

```

THEORY ess1 IS

  x < x+1;

  x < z    &    y < z    =>    x+y < 2*z;

  bcall((ess1~;ess2~): P) => P

END

&

THEORY ess2 IS

  0 < x+1;

  a-a == 0

END

```

### 3.6 Succès ou échec d'une preuve initialisée par bcall

On dit qu'une preuve, initialisée à l'aide de l'opération  $bcall(t:f)$ , réussit lorsque tous les descendants de  $f$  sont tous prouvés à l'aide de la tactique  $t$ . Inversement, on dit qu'une telle preuve échoue lorsque la tactique  $t$  échoue alors qu'il reste encore des descendants de  $f$  non prouvés.

### 3.7 Recul

Afin de pouvoir tenter d'autres preuves, et ce malgré l'échec d'une opération `bcall`, on généralise cette opération en introduisant la possibilité d'avoir plusieurs alternatives séparées par le symbole `|`, comme indiqué ci-dessous :

$$\text{bcall}( t_1:f_1 \mid \cdots \mid t_n:f_n )$$

Ainsi, au cas où la première alternative  $t_1:f_1$  échoue, on recule (c'est-à-dire que l'on supprime les descendants non prouvés de  $f_1$ ) et l'on tente la deuxième alternative  $t_2:f_2$ , et ainsi de suite jusqu'à trouver une alternative qui réussisse. On revient alors à la preuve interrompue momentanément par cette opération, preuve qui se poursuit dès lors normalement.

Si aucune alternative ne réussit, l'opération `bcall` échoue. On revient alors à la preuve interrompue par cette opération et cette preuve échoue aussi. On peut éventuellement reculer de nouveau et repartir sur une nouvelle alternative si cette preuve avait, elle-même, été initialisée par une opération `bcall`.

En cas d'échec de la première opération `bcall` qui ait été appelée, la preuve initialisée depuis le fichier d'entrée échoue à son tour et un message est émis comme on l'a vu plus haut.

A titre d'exemple, voici maintenant la définition de trois théories que l'on définit pour prouver la même formule que précédemment, à savoir,  $aa+(bb-bb) < 2*(aa+1)$ . On voit que la première alternative du `bcall` échoue, alors que la deuxième réussit :

```

THEORY ess1 IS
  x < x+1;

  x < z    &    y < z    =>    x+y < 2*z;

  bcall((ess1~;ess2~): P | (ess1~;ess3~): P) => P
END
&
THEORY ess2 IS
  a-a == 0
END
&
THEORY ess3 IS
  0 < x+1;

  a-a == 0
END

```

## Chapitre 4

# Hypothèses

### 4.1 Dédution et hypothèses

Lorsque la formule à prouver est de la forme suivante :

$$h_1 \ \& \ \cdots \ \& \ h_n \Rightarrow f$$

on peut, par application d’une règle pré-définie (intérieure au **Logic-Solver**), que l’on appelle la *règle d’implication*, produire directement le nouveau but  $f$ . Cette règle ne peut, toutefois, s’appliquer que lorsque la tactique courante correspond à l’identificateur pré-défini DED.

Par application de la règle d’implication, la formule  $h_1 \ \& \ \cdots \ \& \ h_n \Rightarrow f$  est donc coupée en deux. On ne perd cependant pas la sous-formule  $h_1 \ \& \ \cdots \ \& \ h_n$ . En effet, on dit que la preuve de  $f$  se poursuit maintenant *sous les hypothèses*  $h_1, \dots, h_n$ . Autrement dit, *tous* les descendants de  $f$  (y compris ceux issus d’une opération **bcall**) vont, eux-aussi, être prouvés sous ces mêmes hypothèses. Par contre, lorsque  $f$  sera définitivement prouvée, ces hypothèses n’auront plus cours. À noter que les descendants de  $f$  peuvent eux-mêmes produire de nouvelles hypothèses, et ainsi de suite. Cependant, une même formule ne peut se trouver plusieurs fois “en hypothèse”.

En définitive, on peut donc supposer que la preuve d’une certaine formule se fait toujours sous un certain ensemble (éventuellement vide) d’hypothèses.

La preuve d’une formule  $f$ , sous certaines hypothèses, s’effectue exactement comme nous l’avons vu précédemment pour les preuves sans hypothèses. La seule différence vient du fait qu’il existe maintenant une règle pré-définie (intérieure au **Logic-Solver**), que l’on appelle la *règle d’hypothèse*, qui est toujours applicable, qui veut qu’une formule  $f$  soit automatiquement prouvée si  $f$  figure parmi les hypothèses. C’est ce qui rend les hypothèses si intéressantes.

Par exemple, la théorie suivante :

```

THEORY ess IS
  x < x+1;

  a-a == 0;

  x<z & y<z => x+y<2*z;

  bcall((DED;ess~): P) => P
END

```

est suffisante pour prouver la formule suivante :

$$0 < aa+1 \Rightarrow aa + (bb-bb) < 2*(aa+1)$$

## 4.2 Éviter les preuves divergentes (2)

Dans le cas où de nouvelles hypothèses sont apparues entre temps, on ne s'interdit plus d'appliquer une règle à une certaine formule  $f$ , même si cette règle produit un successeur identique à un ascendant de  $f$ . En effet, il se peut que les hypothèses supplémentaires permettent maintenant de prouver la formule  $f$ .

## 4.3 Hypothèses dérivées

Comme on vient de le voir, plus les hypothèses de la preuve d'une formule sont nombreuses, plus on a de chances de voir cette formule effectivement prouvée. Afin d'augmenter les chances de succès d'une preuve, on va donc essayer de lui ajouter de nouvelles hypothèses, et ce, en dehors, évidemment, de la règle d'implication déjà vue à la section précédente. L'idée consiste à appliquer certaines règles "par l'avant" comme nous allons l'expliquer maintenant. De telles règles ont la forme générale suivante :

$$h \ \& \ a_1 \ \& \ \cdots \ \& \ a_n \Rightarrow c_1 \ \& \ \cdots \ \& \ c_m$$

On tente d'appliquer une telle règle au moment précis où une certaine formule  $f$  va devenir une hypothèse. Pour que la règle soit applicable, un certain nombre de conditions doivent être remplies. Tout d'abord, la formule  $f$  doit coïncider avec la formule  $h$ . Il en résulte un certain filtre. On applique ensuite ce filtre aux formules  $a_1, \dots, a_n$ . Si toutes les formules résultantes sont *déjà* des hypothèses, alors la règle est applicable. Et les formules obtenues en appliquant le filtre à  $c_1, \dots, c_m$  deviennent, à leur tour, de nouvelles hypothèses, dite dérivées. À noter que seules les hypothèses dérivées qui ne sont pas déjà des hypothèses sont effectivement générées. Si aucune hypothèse nouvelle ne peut être générée, on considère que la règle n'est pas applicable.

## 4.4 Tactique par l'avant

De façon à guider l'application des règles permettant de produire des hypothèses dérivées, on introduit la notion de tactique "par l'avant". Une tactique par l'avant fonctionne exactement de la même façon qu'une tactique par l'arrière. Les tactiques par

l'avant sont définies elles-aussi dans les opérations `bcall`. Pour cela, les alternatives de ces opérations peuvent prendre la forme plus générale suivante :

$$t_1, t_2 : f$$

Ici,  $t_1$  est la tactique par l'arrière (toujours présente) et  $t_2$  est la tactique par l'avant (facultative). Dans cette alternative, la preuve de  $f$  doit donc s'effectuer avec la tactique par l'arrière  $t_1$  et avec la tactique par l'avant  $t_2$ . Cette dernière est déclenchée chaque fois qu'une nouvelle hypothèse est créée, qu'il s'agisse d'une hypothèse issue de l'application de la règle d'implication ou qu'il s'agisse d'une hypothèse dérivée.

À titre d'exemple, on se donne maintenant les théories suivantes :

```

THEORY ess IS
  x < x+1;

  a-a == 0;

  x < z    &    y < z    =>    x+y < 2*z;

  bcall((DED;ess~),avt: P) => P
END
&
THEORY avt IS
  x <= y    =>    x < y+1
END

```

On peut constater que le but

$$0 \leq aa \quad \Rightarrow \quad aa + (bb-bb) < 2*(aa+1)$$

est prouvable grâce à la règle située dans la théorie `avt`. En effet, cette règle transforme l'hypothèse  $0 \leq aa$  en l'hypothèse  $0 < aa+1$  qui permet de décharger le dernier but.



## Chapitre 5

# Opérations

Certaines formules sont interprétées directement par le **Logic-Solver**. Ces opérations ne peuvent effectivement s'exécuter que si la tactique courante est une certaine tactique pré-définie. Certaines opérations demandent qu'une certaine pré-condition soit établie pour pouvoir effectivement s'exécuter.

<code>bcall</code>	Lancement d'une sous-preuve
<code>bcall1</code>	Lancement d'une sous-preuve en mode trace 1
<code>bcall2</code>	Lancement d'une sous-preuve en mode trace 2
<code>bcatl</code>	Concaténation de chaînes de caractère
<code>bclean</code>	Nettoyage d'une théorie
<code>bclose</code>	Fermeture d'un fichier
<code>bcompile</code>	Compilation dynamique d'une théorie
<code>bcrelr</code>	Création d'une suite de règles ou de lemmes
<code>bcrel</code>	Création d'un lemme
<code>bcrer</code>	Création d'une règle
<code>bctrule</code>	Modification des compteurs de règle d'une théorie
<code>bdef1</code>	Positionnement du système de trace 1
<code>bdef2</code>	Positionnement du système de trace 2
<code>bdump</code>	Stockage de théories en binaire
<code>bflat</code>	Applatissement d'une liste
<code>b fwd</code>	déclenchement de la tactique "forward"
<code>bhalt</code>	Arrêt d'une preuve
<code>blen</code>	Taille d'une théorie
<code>bload</code>	Chargement de théories en binaire
<code>bmark</code>	Marquage d'un lemme
<code>bmodr</code>	Modification d'une règle
<code>bnewv</code>	Création d'une variable
<code>bnlmap</code>	Distribution sur un sous-but
<code>bnmap</code>	Distribution sur un sous-but
<code>bpop</code>	Suppression de la dernière règle d'une théorie
<code>bprintf</code>	Écriture formatée sur un fichier
<code>brecompact</code>	Recompactage de la mémoire
<code>bresetcomp</code>	Nettoyage de la zone de compilation
<code>bresult</code>	Résultat de l'exécution d'un bguard
<code>brev</code>	Renversement d'une liste
<code>bsetmode</code>	Changement du mode de preuve

<code>bshell</code>	Echappement vers le shell
<code>bslmap</code>	Distribution sur un but
<code>bsmap</code>	Distribution sur un sous-but
<code>bstatistics</code>	Informations internes du kernel
<code>btac</code>	Création d'une théorie
<code>bwait</code>	Mise en attente
<code>bwritef</code>	Écriture formatée sur le fichier standard
<code>bwritem</code>	Écriture formatée sur le fichier menu

## Chapitre 6

# Gardes

Une garde est un antécédent spécial dans une règle. En fait, chaque garde d'une règle est interprétée directement avant que la règle ne soit (ou non) effectivement appliquée. Donc une garde ne donnera jamais lieu à la création d'un successeur. L'interprétation d'une garde peut réussir ou échouer. Pour qu'une règle soit effectivement appliquée, il faut que toutes ses gardes réussissent.

<code>\</code>	Non liberté
<code>band</code>	Conjonction de deux gardes
<code>bappend</code>	Ouverture d'un fichier en ajout
<code>bconnect</code>	Ouverture d'un fichier au début
<code>bfresh</code>	Construction d'une variable fraîche
<code>bfork</code>	Lancement d'une tâche fille
<code>bgetallhyp</code>	Obtention de toutes les hypothèses
<code>bgethyp</code>	Obtention des hypothèses principales
<code>bget</code>	Lecture d'un fichier
<code>bgetd</code>	Lecture d'un fichier sans cpp
<code>bgetresult</code>	Résultat d'un bguard
<code>bguard</code>	Lancement d'une sous preuve
<code>bguardi</code>	Lancement d'une sous preuve interruptible
<code>bgoal</code>	Obtention du but
<code>bident</code>	Test d'identificateur
<code>bidentb</code>	Test d'identificateur B
<code>binhyp</code>	Présence d'une formule en hypothèse
<code>binter</code>	Test des interruptions
<code>bkid</code>	Obtention du pid
<code>blemma</code>	Obtention d'un lemme
<code>blenf</code>	Taille d'une formule
<code>blent</code>	Taille d'une théorie
<code>blident</code>	Test de liste d'identificateur
<code>blvar</code>	Liste des variables quantifiées
<code>bmatch</code>	Identité par coïncidence
<code>bmask</code>	Masquage des interruptions
<code>bnot</code>	Négation d'une garde
<code>bnum</code>	Test de nombre
<code>bpattern</code>	Coïncidence d'une formule
<code>breade</code>	Lecture depuis le terminal

<code>breadf</code>	Lecture depuis le terminal
<code>brule</code>	Obtention d'une règle
<code>bsearch</code>	Recherche dans une liste
<code>bsparemem</code>	Taille de la mémoire libre
<code>bstring</code>	Test de chaîne de caractère
<code>bsubfrm</code>	Recherche de sous-formule
<code>btest</code>	Comparaison numérique
<code>bUpident</code>	Test d'identificateur
<code>bunify</code>	Unification de formules
<code>bunmask</code>	Démasquage des interruptions
<code>bvrb</code>	Test de variable