

Atelier B

Guide de rédaction de règles mathématiques

version 1.1



ATELIER B
Guide de rédaction de règles mathématiques
version 1.1

Document établi par CLEARSY

Cette documentation est placée sous License Creative Commons - Paternité (CC-By).



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 - Bat A
13857 AIX EN PROVENCE CEDEX 3
FRANCE

Tel : +33 (0)4 42 37 12 70
Fax 33 (0)4 42 37 12 71
Mail : contact@atelierb.eu

Table des matières

1	Introduction	1
2	Bibliographie	3
3	Terminologie	5
4	Introduction au langage de théorie	7
5	Comment écrire des règles mathématiques	13
6	Utilisation des mécanismes du prouveur	21
7	Gardes	25
A	Normalisation des expressions	27
B	Pièges à éviter	29
C	Gardes du langage de théorie	31

Chapitre 1

Introduction

Ce document est destiné aux utilisateurs avancés du prouveur qui ont besoin d'écrire des règles mathématiques afin de faciliter leur travail de preuve. En effet :

- la preuve en logique des prédicats du premier ordre est indécidable,
- la base de règles du prouveur est finie (environ 2800 règles)

Il peut donc s'avérer nécessaire d'ajouter des règles, dans un fichier Pmm ou dans un fichier PatchProver.

Nous attirons l'attention du lecteur sur les conséquences de l'ajout de règles inopportunes, qui peuvent induire un comportement indésirable du prouveur. En particulier, l'utilisation de règles fausses peut permettre la démonstration d'obligations de preuve fausses et la validation d'un développement logiciel invalide.

On attachera donc un soin tout particulier à la rédaction, la relecture et la validation des règles ajoutées.

Le lecteur trouvera ci-dessous des conseils pour écrire des règles correctes et dont la preuve sera facilitée.

Chapitre 2

Bibliographie

Repère	Référence	Titre
Doc[1]	1.1	Guide de rédaction de règles en langage de théorie
Doc[2]	1.8	Manuel de référence du langage B
Doc[3]	3.6	Manuel de référence du prouveur interactif

Chapitre 3

Terminologie

3.1 Abréviations

PO : obligation de preuve.

OPR : outils de preuve de règles.

TRADLT : traducteur de règles écrites en langage de théorie.

BDR : Base de règles du prouveur automatique de l'Atelier B.

3.2 Définition des termes employés

Obligation de preuve : prédicat logique fabriqué par l'Atelier B à partir d'un composant (machine, raffinement, implantation) écrit en langage B qu'il faut démontrer pour assurer la cohérence de ce composant.

Base de règles : ensemble de règles mathématiques écrites en langage de théorie nécessaires au prouveur pour mener à bien les démonstrations.

Pmm : fichier contenant des règles de l'utilisateur et permettant d'enrichir la BDR utilisée pour un composant.

PatchProver : fichier contenant des règles de l'utilisateur et permettant d'enrichir la BDR utilisée pour un projet.

Chapitre 4

Introduction au langage de théorie

Les règles mathématiques utilisées par le prouveur doivent être écrites en langage de théorie. Sans rentrer dans les détails de ce langage qui s'apparente au langage PROLOG, les paragraphes ci-dessous exposent les notions fondamentales permettant d'utiliser le langage de théorie à cette fin.

4.1 Qu'est ce qu'un joker ?

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 .

La seule manière de représenter une variable consiste à employer un joker.

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).

Par exemple, l'expression

$$a + bb*cc - d$$

contient les jokers `a`, `d` et les littéraux `bb`, `cc`.

On peut instancier `a` avec `ee+1` et `d` avec `3`. On obtient alors l'expression

$$ee + 1 + bb*cc - 3$$

4.2 Qu'est ce qu'une formule ?

Une formule est une expression ou un prédicat, qui peut utiliser

- les jokers,
- les littéraux et les nombres,
- les connecteurs logiques :
 - conjonction : `&`
 - disjonction : `or`
 - implication : `=>`

- équivalence : $\langle \Rightarrow \rangle$
 - négation : $\text{not}(a)$
 - les quantifications :
 - universelle : $!$
 - existentielle : $\#$
 - l'égalité : $=$,
 - l'appartenance : $:$,
 - l'inclusion : $<$:
 - les opérateurs arithmétiques, ensemblistes, booléens, relationnels, fonctionnels et de suite du langage B (voir Doc[2]).
- en respectant la syntaxe du langage B.

Par exemple,

$$0 = \langle aa \ \& \ 0 = \langle bb \ \Rightarrow \ 0 = \langle aa * bb$$

$$a = \text{TRUE} \ \text{or} \ b = \text{TRUE} \ \Rightarrow \ a : \text{BOOL}$$

sont des formules valides.

4.3 Coïncidence 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 :

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

4.4 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.

Par exemple, avec la règle

$$x=\text{FALSE} \Rightarrow \text{not}(x=\text{TRUE})$$

le but

$$\text{not}(\text{bool}(0 \leq aa*bb)=\text{TRUE})$$

est démontré si le but dérivé

$$\text{bool}(0 \leq aa*bb)=\text{FALSE}$$

l'est.

De même, la règle atomique

$$\text{not}(\text{BOOL}=\{\})$$

permet de démontrer immédiatement le but ayant même forme.

– déductive (forward)

Si A est de la forme

$$A_1 \wedge A_2 \wedge \dots \wedge A_n$$

si A_1 est une hypothèse qui vient d'être générée, et si

$$A_2 \wedge \dots \wedge A_n$$

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

Par exemple, avec la règle

$$\text{not}(a=b) \ \& \ a=c \Rightarrow \text{not}(b=c)$$

si l'hypothèse $\text{not}(xx=3)$ vient d'être générée et que l'hypothèse $xx=aa*bb-cc$ existe dans la pile des hypothèses, alors l'hypothèse $\text{not}(3=aa*bb-cc)$ est générée.

– 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 :

$$\begin{array}{ll} \text{btest}(p \leq q) & /* \text{ Si } p \text{ et } q \text{ sont des entiers } */ \\ \Rightarrow & /* \text{ et } p \leq q */ \\ \text{max}\{p\} \setminus \{q\} == q & /* \text{ alors } \text{max}\{p\} \setminus \{q\} \text{ se réécrit en } q */ \end{array}$$

peut s'appliquer sur le but :

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

pour le transformer en :

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

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

4.5 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 :

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

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.

4.5.1 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 :

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

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'antécédent. Ainsi, l'application de la règle précédente à la formule suivante :

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

produit les deux formules suivantes :

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

4.5.2 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 antécédent) :

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

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 coincidence 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 coincident avec la partie gauche, à savoir

$$aa+bb \quad aa+bb+cc \quad cc+bb \quad cc+bb+aa$$

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

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

4.6 Qu'est ce qu'une théorie ?

Une théorie consiste en un regroupement de règles, écrites en langage de théorie.

2 règles sont séparées par un $;$. Les règles ont pour nom $t.n$, avec

- t : nom de la théorie,
- n : index de la règle dans la théorie ¹.

Exemple :

```

THEORY th1 IS

  binhyp(a: B) &                               /* Règle th1.1 */
  binhyp(B<: C)
  =>
  a: C;

  btest(0 <= -t)                               /* Règle th1.2 */
  =>
  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").

Par exemple, lors de l'utilisation de la commande `ar(th1)` (voir Doc[3]), le prouveur va essayer d'appliquer la règle `th1.2` avant la règle `th1.1`. Dans le cas de cette commande, une règle `th1.n` ne s'applique que si aucune règle `th1.m` (avec $n < m$) ne s'applique. Lorsqu'une règle s'est appliquée, le prouveur recommence sa recherche à partir de la dernière règle de la théorie.

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

4.7 Preuve

4.7.1 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 application 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 but à prouver : le but initial est donc définitivement prouvé, à l'aide des règles proposées.

4.7.2 Cas particulier des formules conjonctives

Lorsqu'un but est de la forme suivante :

$$f_1 \quad \& \quad f_2 \quad \& \quad \dots \quad \& \quad f_n$$

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.

Chapitre 5

Comment écrire des règles mathématiques

5.1 Sens d'application d'une règle

Les règles se distinguent par leur sens d'application : en avant (Forward, génération d'hypothèses) ou en arrière (Backward, production de buts dérivés et résolution). Lorsque l'on écrit une règle, on doit donc indiquer, dans le nom de la théorie et/ou dans des commentaires, dans quel sens on doit utiliser la règle (avant, arrière). On veillera alors à ne pas regrouper les règles de nature différente dans une même théorie.

5.2 Restriction du domaine d'utilisation d'une règle

L'écriture de règles mathématiques repose sur l'utilisation de formules compatibles avec la syntaxe du langage B. Il est toutefois possible de limiter le domaine d'application d'une règle, afin qu'elle demeure valide, ou de la paramétrer, en ayant recours aux gardes du langage de théorie. Ces gardes sont décrites au chapitre 7.

Par exemple, pour la règle

```
bsearch((x: p..q),P,r) &
(SIGMA(x).(P & 0 <= -E | -E)<=SIGMA(x).(P & 0<=E | E))
=>
(0 <= SIGMA(x).(P | E));
```

la garde `bsearch` permet de s'assurer que le prédicat `P` contient bien le domaine de définition de la variable `x`, sous la forme d'un intervalle.

Pour la règle

```

band(binhyp(n<=size(a)) ,
btest(n>0)) &
(size(a) = 1 => b = c) &
(2<=size(a) => first(a) = c)
=>
(first(a<+{size(a)}|->b}) = c);

```

les 2 gardes permettent de s'assurer que la séquence a contient au moins 1 élément.

5.3 Règles d'équivalence

Le principe fondamental à respecter est d'écrire des règles d'équivalence. C'est à dire que pour la règle $a \Rightarrow b$, il est équivalent de démontrer a pour démontrer b . Cela signifie que la démontrabilité de b est préservée. Sinon le but produit peut être faux et empêcher la démonstration si la règle est appliquée sans contrôle.

Par exemple, la règle

```

0<=a &
0<=b
=>
0<=a*b

```

n'est pas d'équivalence. Elle ne permettra pas de démontrer le but $0 \leq a * b$ si $a < 0$ et $b < 0$.

Par contre, la règle

```

binhyp(0<=a) &
binhyp(0<=b)
=>
0<=a*b

```

est une règle d'équivalence.

Il est possible d'écrire des règles qui ne sont pas d'équivalence. Il faudra alors n'utiliser ces règles qu'en preuve interactive, lors d'application unitaire de ces règles. Il n'y a pas de risque de démontrer un but faux ; seulement d'aboutir à une preuve infaisable.

5.4 Règles de réécriture

Les réécritures ne doivent pas s'effectuer sans précaution à l'intérieur des prédicats quantifiés. Dans le cas des règles de réécriture possédant un antécédent, la réécriture n'est correcte que si les variables du contexte apparaissant dans la partie gauche du conséquent sont non libres au point de réécriture.

Par exemple, la règle

```

binhyp(x=0)
=>
(x == 0)

```

va transformer l'expression $\exists x1.P(x1)$ en $\exists 0.P(0)$ sous l'hypothèse $x1=0$. Le problème vient du fait que la variable x identifiée par la garde *binhyp* n'est pas forcément la même que celle au point de réécriture.

Les gardes *blvar*(Q) & Q\H empêchent les réécritures abusives au sein des prédicats quantifiés, les règles de réécriture qui nécessitent une protection par *blvar* en sont pourvues (prendre garde à l'instanciation de H).

Une écriture correcte de la règle précédente est

```

binhyp(x=0) &
bgetallhyp(H) &
blvar(Q) &
Q\H
=>
(x == 0)

```

5.5 Règles forward

Les règles forward sont des règles dont le mode d'application est bien différent de celui des règles backward. En pratique, ces règles sont peu utilisées, excepté dans le coeur du prouveur.

Une règle forward ne peut s'appliquer que lorsque des hypothèses viennent d'être générées et avant qu'elles ne soient montées dans la pile des hypothèses (avant qu'on puisse y accéder par *binhyp*).

Par exemple, l'application de la règle backward

```

(not(X<=0 & Y<=0) => 0<=X & 0<=Y)
=>
(0<=X*Y);

```

sur le but

$$0 \leq (aa+bb) * (cc+3)$$

le nouveau but va être

$$\text{not}(aa+bb \leq 0 \ \& \ cc+3 \leq 0) \Rightarrow 0 \leq aa+bb \ \& \ 0 \leq cc+3$$

Le prouveur va alors, grâce au mécanisme de déduction, faire monter l'hypothèse. Le nouveau but va être

$$\text{not}(aa+bb \leq 0 \ \& \ cc+3 \leq 0)$$

dans la pile des hypothèses. C'est au moment de cette montée que les règles forward peuvent être appliquées.

Une règle forward doit être de la forme

$$A_1 \wedge A_2 \wedge \dots \wedge A_n \Rightarrow B_1 \wedge B_2 \wedge \dots \wedge B_p$$

si A_1 est une hypothèse qui vient d'être générée, et si

$$A_2 \wedge \dots \wedge A_n$$

sont dans la pile des hypothèses, alors les hypothèses B_1, B_2, B_n sont générées et montées dans la pile, si elles n'existent pas déjà.

Exemples :

```
(u\ / v = w)
=>
u: POW(w) & v: POW(w)
```

```
not(b: a) &
not(b..c/\a = {})
=>
not(b+1..c/\a = {})
```

La règle peut être complétée avec des gardes qui devront obligatoirement être placées APRES A_1 .

```
not(a:POW({x})) &
bgetallhyp(H) &
bfresh(zz,H,z)
=>
#z.(z:a & not(z=x))
```

5.6 Listes

La manipulation de listes doit être réalisée avec la plus grande précaution. Par exemple, la formule

[a]

peut correspondre à

[aa, bb, cc]

et à

[aa]

Il est donc important de savoir dans une règle si un joker représente une liste ou un littéral. Dans le cas ci-dessus, on utilisera la garde

```
bnot(bpattern(a, (u,v)))
```

pour ne retenir que les séquences ne contenant qu'un seul élément.

Certaines règles, qui agissent sur des listes, peuvent fonctionner par paire et être groupées. La première règle traite le cas général, lorsque la liste contient au moins 2 termes. La seconde règle est prévue pour le cas particulier d'une liste réduite à un élément. Cette dernière ne doit pas être appliquée sur une liste à plusieurs variables car elle produirait un but faux.

Par exemple, les règles du prouveur

```

bnot(bpattern(a, (u,v))
=>
([a]1 == a)

[a,b]1 == [a]1
```

permettent d'évaluer le premier terme d'une séquence.

5.7 Parenthésage

L'utilisation correcte du parenthésage des expressions et prédicats est nécessaire afin d'écrire des règles correctes et qui puissent s'appliquer. On trouvera ci-dessous des conseils quant à la bonne utilisation du parenthésage.

– Parenthésage des quantificateurs existentiels

Pour des raisons de compatibilité sous LINUX, il est nécessaire de surparenthéser les formules sous la portée d'un quantificateur existentiel.

#X. P doit s'écrire (#x. P) pour les utilisations sous LINUX.

– Prédicats quantifiés

Le parenthésage du prédicat quantifié n'est pas garanti. Par exemple, la règle suivante :

```

Antecedent => (!x.(x: E & P(x)))
```

est restrictive. Elle ne s'applique pas, par exemple, sur des buts du genre :

```
!xx. (xx : E & A(xx) & B(xx))
```

– Le parenthésage implicite dans les fonctions du langage de théorie peut être ambigu pour l'utilisateur. Par exemple,

```
bsearch (aa, aa => bb, C)
```

est implicitement parenthésé *bsearch*((aa, aa) => (bb, c)) par le prouveur.

On a aussi :

– La règle $a == b \& c$ est implicitement parenthésée en $((a == b) \& C)$.

– La règle $a => b => c$ est implicitement parenthésée en $((a => b) => C)$.

Le surparenthésage systématique des règles évite d'avoir des surprises avec le parenthésage implicite.

5.8 Instanciation de joker

Trois types de problèmes peuvent se produire avec les jokers

- Des jokers non instanciés dans l'antécédent de la règle, ou bien dans la partie droite d'une réécriture, provoquent l'apparition de jokers morts. Les jokers morts sont des jokers qui deviennent des identificateurs. Ils n'interviennent plus dans le pattern matching. Cela revient à introduire une nouvelle variable qui n'est pas définie par le code source.

Par exemple :

```

e: FIN(S) => card(e): NATURAL
```

s est un joker non instancié qui deviendra un joker mort.

- Le changement de nom est un cas particulier de joker non instancié qui se produit lorsque deux jokers distincts désignent le même objet mathématique

```

binhyp(s~: E +-> NATURAL) &
1<=size(f)
=>
(tail(s)~: E +-> NATURAL)

```

f représente manifestement la même suite que s .

- Le dernier problème concerne la confusion de nom qui se produit quand un même joker représente deux objets mathématiques différents. Dans ce cas, soit la règle est mal typée, soit la règle est correcte mais très restrictive.

```

band(binhyp(r~: C +-> B)
binhyp(C: FIN(D))) &
(a: POW(r<+{a}))
=>
(a: FIN(r<+[a]))

```

les deux occurrences du joker a sont incompatibles du point de vue du typage.

Lors de l'écriture d'une règle, il faut faire particulièrement attention aux jokers. Le copier-coller est redoutable vis à vis de ce genre de problème. Il faut donc vérifier la cohérence de l'emploi des jokers.

Le nombre limité de jokers disponibles ne facilite pas la lisibilité des règles. Cependant, dans la mesure du possible, les conventions suivantes peuvent être suivies :

- f, g, h pour des fonctions,
- r pour une relation,
- A, B, C, D, E, F pour les ensembles,
- la même lettre en minuscule pour un élément de cet ensemble,
- s, t pour des suites,
- i, j, k, m, n, p pour des entiers,
- P, Q, R pour des prédicats,
- x, y, z pour des variables.

5.9 Ambiguïtés

Il faut se méfier de l'ambiguïté de certaines notations :

- - désigne de la même façon la soustraction arithmétique et la différence ensembliste,
- * désigne de la même façon la multiplication arithmétique et le produit cartésien,

Les règles pouvant entraîner des preuves fausses suite à des confusions entre opérateur ensembliste et opérateur arithmétique doivent être corrigées ou supprimées. Les expressions mal typées sont corrigées de façon à être correctement typées. Si cela est impossible, la règle est supprimée.

5.10 Remarques diverses

La sémantique exacte des gardes est définie en annexe. Il faut toutefois faire attention aux points suivants :

- les gardes $\text{btest}(a=b)$ et $\text{btest}(a/=b)$, contrairement aux autres formes de btest , ne vérifient pas que a et b sont des éléments de NAT . Attention, si $\text{btest}(a = b)$ implique $a = b$, en revanche $\text{btest}(a /= b)$ n'implique pas obligatoirement $a /= b$. En effet, si x et y sont deux jokers coïncidant avec une certaine expression, $\text{btest}(x /= y)$ réussit mais $x = y$ est vrai.

La garde $\text{btest}(a+b=a+b)$ échoue car $a+b$ n'est pas un identificateur.

- la garde bsubfrm sert principalement à guider la preuve. Cette garde ne doit pas être employée à d'autres fins dans les règles.
- la garde bsearch ne réalise qu'une seule extraction.

$\text{bsearch}(a, b, c)$: b doit être une liste d'au moins deux éléments, de la forme $x \text{ op } y$, où op est un opérateur binaire commutatif et associatif. Les caractères commutatif et associatif de l'opérateur sont primordiaux car ils donnent la même importance aux diverses occurrences de a dans b . Si op n'est pas commutatif et associatif, la traduction en langage mathématique n'est plus générale.

Chapitre 6

Utilisation des mécanismes du prouveur

Cette section présente les différents mécanismes du prouveur qui peuvent être utilisés lors de la rédaction de règles mathématiques. Ces mécanismes ont été validés et permettent de faciliter la phase de conception de règles. On trouvera pour chaque mécanisme :

- la syntaxe de l'appel du mécanisme,
- la sémantique associé à ce mécanisme,
- une description textuelle de ce mécanisme,
- un exemple de mise en oeuvre

6.1 Preuve par tentative

Forme	<code>bguard(attempt_to_prove: a_t_t_e_m_p_t__t_o__p_r_o_v_e(P Curr))</code>
Traduction	<code>P</code>

Ce mécanisme permet de lancer une sous-preuve sur P . Ce mécanisme réussit si P peut être démontré par le prouveur.

P doit être un prédicat correctement typé.

Exemples :

```
band(binhyp(f: s +-> (t --> u)) ,
bguard(attempt_to_prove: a_t_t_e_m_p_t__t_o__p_r_o_v_e(
    (t: POW(a) & u: POW(b)) | Curr))
=>
(f(x): a +-> b)
```

```
bguard(attempt_to_prove: a_t_t_e_m_p_t__t_o__p_r_o_v_e(
    (not(b) => a) | Curr))
=>
(a or b)
```

```
(n: a) &
not(n: b) &
bguard(attempt_to_prove: a_t_t_e_m_p_t__t_o__p_r_o_v_e(
    (a: POW(b)) | Curr))
=>
bfalse
```

La dernière règle est une règle Forward : si une hypothèse de la forme $n: a$ vient d'être générée, s'il existe une hypothèse $\text{not}(n: b)$ et si le prouveur peut démontrer $a: \text{POW}(b)$ alors l'hypothèse bfalse est générée.

6.2 Différence

Forme	<code>bguard(Fast_Prove_Difference: not(a = b))</code>
Traduction	<code>not(a = b)</code>

Ce mécanisme permet de démontrer l'inégalité de 2 termes. La réussite de ce mécanisme indique que a et b ne sont pas égaux.

Exemples :

```
bguard(Fast_Prove_Difference: not(a = b)) &
blvar(Q) &
Q\not(a = b)
=>
({a}<<|{b|->c} == {b|->c})
```

```
bguard(Fast_Prove_Difference: not(a = c)) &
blvar(Q) &
Q\{a = c}
=>
({a|->b}<+{c|->d} == {a|->b}\/{c|->d})
```

6.3 Relation d'ordre

Forme	<code>bguard(Fast_Prove_Order: m<=n)</code>
Traduction	<code>m<=n</code>

Ce mécanisme permet de démontrer que $m \leq n$. Si le mécanisme réussit, alors l'inégalité est vérifiée.

Exemple

```
bguard(Fast_Prove_Order: c<=d) &
(a = c) &
(b = d)
=>
(a..b = c..d)
```

6.4 Valeur positive

Forme	<code>bguard(Fast_Prove_Positif: a)</code>
Traduction	<code>0<=a</code>

Ce mécanisme permet de démontrer que la valeur d'une expression arithmétique est positive ou nulle.

Exemple :

```
binhyp(s: POW(a..b)) &
bguard(Fast_Prove_Positif: a)
=>
(s: POW(NATURAL))
```

6.5 Non-appartenance

Forme	<code>bguard(Fast_Prove_Distinction: not(x: s))</code>
Traduction	<code>not(x: s)</code>

Ce mécanisme permet de démontrer la non-appartenance de x à l'ensemble s .

Exemple :

```
Fast_Prove_Distinction(not(a: dom({c|->d})))
=>
not({a|->b} = {c|->d})
```

6.6 Appartenance à INTEGER

Forme	<code>bguard(Fast_Prove_Num: n)</code>
Traduction	<code>n: INTEGER</code>

Ce mécanisme permet de vérifier que n est un entier.

Exemple :

```
bguard(Fast_Prove_Num: (x))
=>
(succ(x) == x+1 )
```


Chapitre 7

Gardes

Ce chapitre présente rapidement les différentes gardes accessibles pour l'écriture de règles mathématiques (pour plus de détails, voir le chapitre C). Pour chaque garde, on trouvera :

- sa syntaxe,
- éventuellement sa sémantique,
- une description textuelle de son fonctionnement.

Il faut noter que certaines gardes n'ont une sémantique mathématique que dans des cas particuliers (la garde `bsearch` par exemple).

Forme	<code>bgetallhyp(H)</code>
Traduction	H
Forme	<code>bgethyp(H)</code>
Traduction	H
Forme	<code>binhyp(H)</code>
Traduction	H

Ces gardes permettent de récupérer une ou plusieurs hypothèses dans la pile, en parcourant les dernières hypothèses générées en premier.

Forme	<code>bstring(s)</code>
Traduction	<code>s: STRING</code>

Cette garde vérifie que s est un élément de *STRING*.

Forme	<code>bpattern(P, (Q & R))</code>
Traduction	<code>P <=> (Q & R)</code>

Cette utilisation de cette garde détermine si P est un prédicat conjonctif.

Forme	bsearch(N, (P, Q), M)
Traduction	(P, Q) = (M, N)
Forme	bsearch(N, (P & Q), M)
Traduction	(P & Q) <=> (M & N)
Forme	bsearch(d, (a or b), c)
Traduction	a or b <=> c or d
Forme	bsearch(d, (a\ b), c)
Traduction	a\ b = c\ d
Forme	bsearch(d, (a/\b), c)
Traduction	a/\b = c/\d

La garde bsearch a une sémantique mathématique lorsque la liste dans laquelle s'effectue la recherche est une liste séparée par un opérateur binaire commutatif et associatif.

C'est le cas des opérateurs &, or, \| et \.

L'opérateur , possède ces propriétés dans une liste de variable quantifiées. Il faut donc vérifier que l'application de la règle ne sort pas de ce cadre.

Forme	btest(t)
Traduction	t

Pas de problème de traduction puisque cette garde ne réussit que lorsque t est vrai et de la forme $a \text{ op } b$.

Forme	bnum(n)
Traduction	n: NATURAL

Cette garde teste si n est un numérique positif inférieur à maxint.

Forme	band(P, Q)
Traduction	Traduction de P et traduction de Q

La sémantique de cette garde nécessite un traitement récursif de ses paramètres.

Forme	bvrb(x)
Variable	x
Forme	blvar(x)
Variable	x
Forme	x\ P
Variable	x
Condition	x\ P
Forme	bfresh(x, P, y)
Variable	y
Condition	y\ P

Ces gardes ont un rôle pour exprimer les relations entre les jokers représentant des variables et les jokers représentant des expressions.

Annexe A

Normalisation des expressions

Afin de limiter le nombre de règles dans la base de règles, des choix de normalisation ont été faits. Toute expression manipulée par le prouveur doit être normalisée. Aussi toutes les règles utilisateur, provenant des fichiers Pmm, sont normalisées lors de leur chargement. Par contre, les règles contenues dans le PatchProver doivent être normalisées, sinon elles peuvent induire un comportement anormal du prouveur.

Les formes normales retenues sont :

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

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 être 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 $a \text{ op } b$, 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
```

Annexe B

Pièges à éviter

B.1 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 :

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

...

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.

Annexe C

Gardes du langage de théorie

Ce chapitre ¹ présente les gardes (opérateurs) utiles pour l'écriture de règles. Ces gardes permettent de limiter le domaine d'application d'une règle, en utilisant des informations relatives au but et aux hypothèses.

Afin de simplifier le travail de validation a posteriori des règles, toutes les gardes du langage de théorie ne sont pas décrites ci-après.

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.

band	Conjonction de deux gardes
bfresh	Construction d'une variable fraîche
bgetallhyp	Obtention de toutes les hypothèses
bgethyp	Obtention des hypothèses principales
binhyp	Présence d'une formule en hypothèse
blvar	Liste des variables quantifiées
bmatch	Identité par coïncidence
bnot	Négation d'une garde
bnum	Test de nombre
bpattern	Coïncidence d'une formule
bsearch	Recherche dans une liste
bstring	Test de chaîne de caractères
bsubfrm	Recherche de sous-formule
btest	Comparaison numérique
bvrb	Test de variable

¹extrait du manuel de référence du Logic Solver

band(g_1, g_2)

Paramètres

g_1 : GARDE

g_2 : GARDE

Nature

Garde.

Utilisation pratique

Pour coordonner plusieurs gardes.

Évaluation

Plusieurs cas sont à envisager suivant la nature des gardes g_1 et g_2 :

- Si la garde g_1 est la garde `binhyp(H)`, ou la garde `binhyp(n, H)`, ou la garde `binhyp(m, n, H)`, et si, dans les deux derniers cas, n est un JOKER, alors la garde `band` est un SUCCES s'il existe une hypothèse h qui coïncide avec H et qui est telle que la garde g_2 soit un SUCCES.
- Si la garde g_1 est la garde `bsubfrm`, alors la garde `band` est un SUCCES s'il existe une instantiation de jokers par la garde `bsubfrm` (supposée être un SUCCES), qui est telle que la garde g_2 soit un SUCCES.
- Si la garde g_1 est la garde `bsearch`, alors la garde `band` est un SUCCES s'il existe une instantiation de jokers par la garde `bsearch` (supposée être un SUCCES), qui est telle que la garde g_2 soit un SUCCES.
- Si la garde g_1 est la garde `brule`, alors la garde `band` est un SUCCES s'il existe une instantiation de jokers par la garde `brule` (supposée être un SUCCES), qui est telle que la garde g_2 soit un SUCCES.
- Dans tous les autres cas, la garde `band` est un SUCCES si les deux gardes g_1 et g_2 , évaluées l'une après l'autre, sont des SUCCES. En cas d'ECHEC de la seconde, on ne revient pas sur la première comme dans les cas précédents.

Exemple

THEORY ess IS

```

    band(binhyp(aaa(x)), band(binhyp(ccc),binhyp(bbb(x))) ) &
    bcall(WRITE: bwritef("x: %\n",x))
=>
    H;

    bcall((DED~;ess):((bbb(1) => (aaa(2) & bbb(2) & ccc & aaa(1) => titi))))
=>
    coco

```

END

Résultat

x: 1

bnot(*g*)

Paramètres

g : GARDE

Nature

Garde.

Utilisation pratique

Pour simplifier l'usage des gardes en échec.

Évaluation

La garde `bnot` est un SUCCES, si la garde *g* est un ÉCHEC.

Exemple

```
THEORY ess IS
```

```

    bcall(WRITE: bwritef("Salut \n"))
=>
    aaa(n);

    breade("Ecrire >>? (en instanciant les ? par des nombres): ",a>b) &
    bnot(btest(a>b)) &
    bcall(WRITE: bwritef("Hello \n")) &
    aaa(n+1)
=>
    aaa(n)/*(a,b)*;/

    bcall((ARI;ess)~: aaa(1))
=>
    coco
```

```
END
```

Résultat

```

Ecrire >>? (en instanciant les ? par des nombres): 3>5
Hello
Ecrire >>? (en instanciant les ? par des nombres): 4>9
Hello
Ecrire >>? (en instanciant les ? par des nombres): 5>1
Salut
```

$\text{bfresh}(v, f, V)$

Paramètres

v : VARIABLE

f : FORMULE

V : JOKER

Nature

Garde.

Utilisation pratique

Obtenir des variables fraîches.

Évaluation

La garde est toujours un SUCCES. Le JOKER V est instancié avec autant de variables que les variables de v . De plus, les nouvelles variables V ne sont pas libres dans f . Si v n'est pas libre dans f , alors V est identique à v .

Exemple

```
THEORY ess IS
```

```
  bfresh((xx,yy,zz),aaa+xx$0-yy*zz,V) &
  bcall(WRITE: bwritef("%\n",V))
```

```
=>
```

```
  coco
```

```
END
```

Résultat

```
xx$1,yy$1,zz$1
```

bgethyp(h) bgetallhyp(h)

Paramètres

h : FORMULE

Nature

Garde.

Utilisation pratique

Obtenir les hypothèses d'une preuve.

Évaluation

La garde est un SUCCES lorsque la preuve comporte des hypothèses et que la conjonction de ces hypothèses coïncide avec la formule h . Les hypothèses concernées dépendent de la garde :

- Avec la garde `bgethyp`, on n'obtient que les hypothèses principales.
- Avec la garde `bgetallhyp` on obtient les hypothèses principales ainsi que les hypothèses dérivées de ces hypothèses principales.

Dans tous les cas, les jokers non instanciés de h sont instanciés.

Exemple

```

THEORY ess IS

  bgethyp(H) &
  bgetallhyp(G) &
  bcall(WRITE: bwritef("Hypotheses principales:          %\n",H)) &
  bcall(WRITE: bwritef("Hypotheses princ. et derivees: %\n",G))
=>
  P;

  bcall((DED;ess),fwd: (aaa & bbb => ggg))
=>
  coco

END

&

THEORY fwd IS

  aaa => ccc;

  ccc & bbb => ddd & eee

END

```

Résultat

```
Hypotheses principales:          aaa & bbb
```

Hypotheses princ. et derivees: aaa & bbb & ccc & ddd & eee

$\text{binhyp}(h)$ $\text{binhyp}(n, h)$ $\text{binhyp}(m, n, h)$

Paramètres

h : FORMULE

n : FORMULE

m : NOMBRE

Nature

Garde.

Utilisation pratique

Pour accéder à une hypothèse.

Évaluation

La garde $\text{binhyp}(h)$ est un SUCCES lorsqu'il existe une hypothèse qui coïncide avec la formule h . Lorsqu'il y en a plusieurs, on choisit la dernière d'entre elles. Les JOKERS non instanciés de h sont instanciés.

Pour la garde $\text{binhyp}(n, h)$, on considère différents cas suivant la nature de n .

- Lorsque n est un NOMBRE, la garde est un SUCCES si l'hypothèse de rang n existe et coïncide avec la FORMULE h . Les JOKERS non instanciés de h sont instanciés.
- Lorsque n est un JOKER, la garde est un SUCCES s'il existe une hypothèse qui coïncide avec la FORMULE h . Lorsqu'il y a plusieurs hypothèse qui coïncident avec h , on choisit la dernière d'entre elles. Le JOKER n est alors instancié avec le numéro de l'hypothèse sélectionnée. Les JOKERS non instanciés de h sont instanciés.
- Dans tous les autres cas, la garde est un ECHEC.

L'évaluation de la garde $\text{binhyp}(m, n, h)$ correspond à ce que nous venons d'expliquer pour la garde $\text{binhyp}(n, h)$, à ceci près que, dans le cas où n est un JOKER, l'hypothèse sélectionnée est la dernière hypothèse, qui coïncide avec h , et dont le numéro est inférieur ou égal à i .

On notera la grande ressemblance entre ces deux dernières formes de la garde binhyp et les gardes `brule` et `blemma`.

Exemple

```
THEORY ess IS
```

```
  toto(i);
```

```
  binhyp(i,n,H) &
  bcall(WRITE: bwritef("hyp_%%: %%\n",n,H)) &
  toto(n-1)
```

```
=>
```

```
  toto(i);
```

```
  binhyp(n,H) &
  bcall(WRITE: bwritef("hyp_%%: %%\n",n,H)) &
  toto(n-1)
```

```
=>
  tata;

  binhyp(1,H) &
  bcall(WRITE: bwritef("hyp_1: %\n",H)) &
  tata
=>
  titi;

  binhyp(H+G) &
  bcall(WRITE: bwritef("hyp:  %\n",H+G)) &
  titi
=>
  P;

  bcall((DED;(ARI;ess)~):((aaa & bbb+ccc & ddd) => pp))
=>
  coco

END
```

Résultat

```
hyp:  bbb+ccc
hyp_1: aaa
hyp_3: ddd
hyp_2: bbb+ccc
hyp_1: aaa
```

blvar(*l*)

Paramètres

l : LISTE_DE_VARIABLES

Nature

Garde.

Utilisation pratique

Pour déterminer la liste des variables quantifiées au point de réécriture.

Évaluation

La garde est toujours un SUCCES.

S'il existe au moins une variable quantifiée au point de réécriture, *l* contient la liste de variables quantifiées. Sinon *l* vaut ?.

Par exemple, si le but courant est

$$(aa, bb, cc). 0 \leq aa \ \& \ 0 \leq bb \ \& \ 0 \leq cc \Rightarrow 0 \leq aa + bb + cc$$

et que la règle

`binhyp(x=0) &`

`blvar(Q) &`

`x \ Q`

`=>`

`x == 0`

s'applique, alors Q vaudra `(aa, bb, cc)`.

Cette garde est utilisée au sein de règles de réécriture. Elle permet de s'assurer que l'on ne confond pas une variable non quantifiée avec une variable quantifiée. Elle est souvent utilisée en conjonction avec la garde de non liberté `bnfree x \ E`.

`? \ E` est toujours vrai quelque soit E.

Exemple

THEORY ess IS

`bnot(blvar(?))`

`=>`

`qq == pp;`

`!qq.(qq+1>qq)`

`=>`

`%ii.(ii: NAT | uu) = oi$5;`

`blvar(t$i)`

`=>`

`(t$i) == ii;`

`blvar(Q) &`

`Q \ !yx.(yx < yx + oo) &`

```

bcall(WRITE: bwritef("free Q is %\n",Q)) &
%(tt$0).(tt$0: NAT | uu) = oi$5
=>
!yx.(yx<yx+oo);

blvar(Q) &
Q\ (aa+2) &
bcall(WRITE:bwritef("aa transforme en oo\n"))
=>
aa == oo;

blvar(Q) &
Q\ (yx+2) &
bcall(WRITE:bwritef("yx transforme en za\n"))
=>
yx == za;

!yx.(yx<yx+aa)
=>
!(zz$0,uu,hh$9).(zz$0+uu+hh$9>0);

blvar(Q) &
bcall(WRITE:bwritef("Q is %\n",Q))
=>
vv == hh;

!(zz$0,uu,vv$9).(zz$0+uu+vv$9>0)
=>
!yy.(yy<yy+1);

blvar(Q) &
bcall(WRITE:bwritef("Q is %\n",Q))
=>
xx == yy;

!xx.(xx < xx+1)
=>
coco

END

```

Résultat

```

Q is xx
Q is xx
Q is xx
Q is zz$0,uu,vv$9
Q is zz$0,uu,vv$9
aa transforme en oo

```

free Q is ?

EXECUTION ABORTED ON GOAL: !pp.(pp+1>pp)

bmatch(x, p, q, e)

Paramètres

x : VARIABLE

p : FORMULE

q : FORMULE

e : FORMULE

Nature

Garde.

Utilisation pratique

Pour vérifier que l'on peut instancier une formule quantifiée.

Évaluation

Pour que la garde soit un SUCCES, il est tout d'abord nécessaire que la formule p ne contienne pas de quantificateurs. Il faut ensuite qu'il existe une formule f telle que le remplacement de x par f dans p soit identique à la formule q . Il faut, enfin, que cette formule f coïncide avec e , supposée non instanciée (la plupart du temps, e est un simple joker). Les jokers non instanciés de e sont instanciés.

Exemple

THEORY ess IS

```

    binhyp(!x.(H=>P)) &
    bmatch(x,P,Q,E) &
    bcall((SUB;WRITE):
    bwritef("P: %\nx: %\nE: %\n[x:=E]P: %\nQ:          %\n",P,x,E,[x:=E]P,Q))
=>
    Q;

    ( !(xx$1,yy).(qq(xx$1,yy) => pp(xx$1,ff(xx$1),yy))
    =>
        pp(aa,ff(aa),bb)
    )
=>
    ccc;

    bcall((ess;DED;ess):ccc)
=>
    coco

```

END

Résultat

```

P: pp(xx$1,ff(xx$1),yy)
x: xx$1,yy
E: aa,bb

```

$[x:=E]P: pp(aa, ff(aa), bb)$
 $Q: pp(aa, ff(aa), bb)$

bpattern(*f*, *g*)

Paramètres

f : FORMULE

g : FORMULE

Nature

Garde.

Utilisation pratique

Savoir si une formule coïncide avec une autre.

Évaluation

La garde est un SUCCES si la formule *f* coïncide avec la formule *g*, supposée non instanciée. Les jokers non instanciés de *g* sont instanciés.

Exemple

THEORY ess IS

```
    bnot(bpattern(p,q)) &
    bcall(WRITE: bwritef("ECHEC"))
```

=>

```
    titi(p,q);
```

```
    bpattern(aaa+bbb,a+b) &
    bcall(WRITE: bwritef("%  %\n",a,b)) &
    titi(aaa+bbb,k+1)
```

=>

```
    coco
```

END

Résultat

aaa bbb

ECHEC

$$\text{bsearch}(p, l, r) \quad \text{bsearch}(p, l, r, s)$$

Paramètres p : FORMULE l : FORMULE_NON_ATOMIQUE r : FORMULE s : FORMULE**Nature**

Garde.

Utilisation pratique

Pour rechercher, et éventuellement modifier, un élément dans une liste.

Évaluation

La formule l est de la forme $l_1 \text{ op} \cdots \text{op} l_i \text{ op} \cdots \text{op} l_n$ où n supérieur ou égal à 2 et où op est un OPÉRATEUR_BINAIRE. On doit considérer deux cas suivant la forme de la garde :

- La garde $\text{bsearch}(p, l, r)$ est un SUCCES lorsqu'il existe une sous formule l_i qui coïncide avec p et lorsque la formule, que l'on obtient en enlevant de l la sous-formule l_i , coïncide avec la formule r .
- La garde $\text{bsearch}(p, l, r, s)$ est un SUCCES lorsqu'il existe une sous formule l_i de l qui coïncide avec p , et lorsque la formule, que l'on obtient en enlevant de l la sous-formule l_i et en la remplaçant par la formule s dûement instanciée, coïncide avec la formule r .

Les jokers non instanciés de p , r et s sont instanciés.

Exemple

THEORY ess IS

```

bsearch((a-{x}), (aaa \ (bbb-{xx}) \ ccc \ (ddd \ {xx}) \ eee), r, a) &
bsearch((b\{x}), r, s, b) &
bsearch((b\{x}), (aaa \ (bbb-{xx}) \ ccc \ (ddd \ {xx}) \ eee), h) &
bcall(WRITE: bwritef("r: %\na: %\nb: %\ns: %\nh: %\n", r, a, b, s, h))
=>
coco

```

END

Résultat

```

r: aaa\bbb\ccc\ddd\eee
a: bbb
b: ddd
s: aaa\bbb\ccc\ddd\eee
h: aaa\bbb-{xx}\ccc\eee

```

bstring(*f*)

Paramètres

f : FORMULE.

Nature

Garde.

Utilisation pratique

Pour tester qu'une formule est une CHAINE_ENTRE_GUILLEMETS.

Évaluation

SUCCES si la formule *f* est une CHAINE_ENTRE_GUILLEMETS. On rappelle qu'une CHAINE_ENTRE_GUILLEMETS est une suite de caractères commençant par un GUILLEMET et se terminant par un GUILLEMET. On peut trouver un GUILLEMET dans la chaîne à condition qu'il soit précédé du caractère l'ANTL_SLASH.

Exemple

THEORY ess IS

```
bcall(WRITE: bwritef("test3: ECHEC\n"))
=>
test3;
```

```
bstring(aa+bb) &
bcall(WRITE: bwritef("test3: SUCCES\n"))
=>
test3;
```

```
bcall(WRITE: bwritef("test2: ECHEC\n")) &
test3
=>
test2;
```

```
bstring(aaa) &
bcall(WRITE: bwritef("test2: SUCCES\n")) &
test3
=>
test2;
```

```
bcall(WRITE: bwritef("test1: ECHEC\n")) &
test2
=>
test1;
```

```
bstring("Bonjour \"Monsieur\"") &
bcall(WRITE: bwritef("test1: SUCCES\n")) &
test2
```

```
=>
  test1;

  bcall(ess: test1)
=>
  coco

END
```

Résultat

```
test1: SUCCES
test2: ECHEC
test3: ECHEC
```

$$\text{bsubfrm}(g, d, p, q) \quad \text{bsubfrm}(g, d, p, (q, v))$$

Paramètres

g : FORMULE

d : FORMULE

p : FORMULE

q : FORMULE

v : FORMULE

Nature

Garde.

Utilisation pratique

Tester la présence d'une sous-formule dans une formule et la remplacer par une autre.

Évaluation

Pour que la garde soit un SUCCES, il est d'abord nécessaire qu'il existe une sous-formule f de p qui coïncide avec g (supposée non instanciée). On considère ensuite la formule p' , obtenue en remplaçant, dans p , la sous-formule f par d , dûement instanciée. Il faut que cette formule p' coïncide avec q , supposée non instanciée (la plupart du temps, q est un simple joker).

Dans le cas de la deuxième forme de la garde, on considère enfin la liste des variables quantifiées dont dépend la sous-formule f . S'il n'y a pas de telles variables, on considère, par convention, la liste formée du seul symbole ?. Il faut que cette liste coïncide avec v , supposée non instanciée (la plupart du temps, v est un simple joker).

Les jokers non instanciés de g , q et, éventuellement, v sont instanciés.

Exemple

THEORY ess IS

```

    bsubfrm(x/:s,not(x:s),#(y,z).!(xxx$1,x,bbb).(x/:s => aaa),(q,v)) &
    bcall((SUB;WRITE): bwritef("q: %\nv: %\n",q,v))
=>
    coco

```

END

Résultat

```

q: #(y,z).!(xxx$1,x,bbb).(not(x: s) => aaa)
v: xxx$1,x,bbb,y,z

```

$btest(m \text{ op } n)$

Paramètres

m : FORMULE

n : FORMULE

op : OPÉRATEUR_DE_COMPARAIISON

Nature

Garde.

Utilisation pratique

Pour comparer deux nombres.

Évaluation

La garde est un SUCCES lorsque m et n sont deux NOMBRES reliés par la relation spécifiée. On rappelle que les OPÉRATEURS_DE_COMPARAIISON sont les suivants :

- Égal : =
- Différent : / =
- Inférieur : <
- Inférieur ou égal : <=
- Supérieur : >
- Supérieur ou égal : >=

Lorsque l'opérateur est celui d'égalité ou d'inégalité, la garde est aussi un SUCCES lorsque m et n sont deux IDENTIFICATEURS reliés par la relation spécifiée.

Exemple

```
THEORY ess IS

    btest(bb/=aa) &
    bcall(WRITE: bwritef("test3: SUCCES\n"))
=>
    test3;

    bnot(btest(8=aa)) &
    bcall(WRITE: bwritef("test2: ECHEC\n")) &
    test3
=>
    test2;

    btest(8=8) &
    bcall(WRITE: bwritef("test1: SUCCES\n")) &
    test2
=>
    coco
```

END

Résultat

test1: SUCCES

test2: ECHEC

test3: SUCCES

bvrb(f)

Paramètres

f : FORMULE.

Nature

Garde.

Utilisation pratique

Pour tester qu'une formule est une VARIABLE.

Évaluation

SUCCES si la formule f est une VARIABLE. On rappelle qu'une VARIABLE est soit une LETTRE, soit un IDENTIFICATEUR ne commençant pas par un UNDERSCORE, soit l'une des deux possibilités précédentes suivi d'un DOLLAR et d'un nombre inférieur à 10000, soit, enfin, une liste constitués d'éléments distincts et appartenant aux trois possibilités précédentes.

Exemple

```
THEORY ess IS
```

```
    bcall(WRITE: bwritef("test5: ECHEC\n"))
=>
    test5;

    bvrb(a+b) &
    bcall(WRITE: bwritef("test5: SUCCES\n"))
=>
    test5;

    bcall(WRITE: bwritef("test4: ECHEC\n")) &
    test5
=>
    test4;

    bvrb(_a) &
    bcall(WRITE: bwritef("test4: SUCCES\n")) &
    test5
=>
    test4;

    bcall(WRITE: bwritef("test3: ECHEC\n")) &
    test4
=>
    test3;

    bvrb(a$10000) &
    bcall(WRITE: bwritef("test3: SUCCES\n")) &
```

```
test4
=>
test3;

bcall(WRITE: bwritef("test2: ECHEC\n")) &
test3
=>
test2;

bvr(b(a,a,bbb) &
bcall(WRITE: bwritef("test2: SUCCES\n")) &
test3
=>
test2;

bcall(WRITE: bwritef("test1: ECHEC\n")) &
test2
=>
test1;

bvr(b(aaaa_3,xxx,xx$2,y) &
bcall(WRITE: bwritef("test1: SUCCES\n")) &
test2
=>
test1;

bcall(ess: test1)
=>
coco
```

END

Résultat

```
test1: SUCCES
test2: ECHEC
test3: ECHEC
test4: ECHEC
test5: ECHEC
```