Atelier B

Prouveur interactif

Manuel Utilisateur

version 3.7

ATELIER B
Prouveur interactif Manuel Utilisateur
version 3.7

Document établi par CLEARSY.

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

Table des matières

Chapitre 1

Introduction

Ce manuel décrit la méthode que nous conseillons pour les activités de preuve d'un projet avec les outils de l'Atelier B. Le *credo* de cette méthode est de décomposer ces activités de preuve en deux phases :

La phase de mise au point : analyse rapide des obligations de preuve et modification des composants B quand ces obligations révèlent des erreurs.

La phase de preuve formelle : preuve formelle complète des obligations de preuve. Les composants B ne sont plus modifiés.

Cette méthode n'est bien entendu pas la seule envisageable. Il y a deux utilisations possibles de ce manuel :

– vous pouvez le lire avant d'utiliser l'Atelier B
– vous pouvez aussi vous en servir au moment de la preuve de votre projet pour faire un suivi pas à pas des étapes correspondantes.

Attention, ce document ne remplace pas le manuel de référence du prouveur. Il ne contient pas une description analytique de chaque commande. Par exemple, le lecteur n'y trouvera pas la liste des mots clefs à employer dans la commande *ApplyRule*. Ce manuel cherche plutôt à guider l'opérateur pour savoir *quelle* commande appliquer et *pourquoi*. La lecture de ce manuel nécessite la connaissance du langage B et quelques notions sur l'Atelier B. La preuve est une activité difficile nécessitant une bonne disponibilité de l'information. Nous conseillons d'équiper tout poste de travail de preuve des documents suivants :

– **le manuel de référence du langage B**. Il contient la définition de chaque symbole mathématique, les propriétés essentielles et des exemples significatifs (de plus, l'équivalent ASCII de chaque opérateur y est indiqué).
– **le B-Book chapitres 1, 2 et 3**. Il contient la construction logique de toutes les notions mathématiques utilisées en preuve.
– **le manuel de référence du prouveur interactif** : il décrit la syntaxe des commandes de preuve.
– **le manuel utilisateur du prouveur interactif** (le présent manuel) : il indique la marche à suivre dans le cadre de la méthode de preuve que nous proposons.

En plus de ces documents, l'opérateur pourra utiliser la carte "mémo" du prouveur interactif pour un accès rapide aux commandes et le manuel d'utilisation des obligations de preuve qui indique comment interpréter chaque obligation de preuve.

Plan général

Le chapitre 2 est une introduction à la preuve formelle mathématique telle qu'elle est appliquée dans l'Atelier B. Si vous n'êtes pas familier avec les démonstrations mathématiques formelles, nous vous recommandons la lecture de ce court chapitre.

Le chapitre 3 est une présentation générale des outils de preuve de l'Atelier B. Il montre comment accéder à ces outils depuis les menus de l'interface, et explique comment vous pourrez intervenir pour piloter une démonstration qui n'aurait pas abouti automatiquement. Si vous n'avez jamais utilisé les outils de preuve, la lecture de ce chapitre est vivement recommandée.

Les chapitres 4, 5 et 6 définissent la méthode préconisée dans ce manuel pour conduire les activités de preuve. Le chapitre 4 montre la décomposition principale de la preuve en deux phases : la phase de mise au point du projet et la phase de preuve formelle proprement dite. Il indique comment savoir dans quelle phase on se trouve, et quand en changer. Le chapitre 5 décrit la phase de mise au point, et le chapitre 6 décrit la phase de preuve formelle. Dans ces deux chapitres 5 et 6, la phase de preuve concernée est décomposée en plusieurs activités. Lors du développement d'un projet en B, vous pouvez suivre dans ce manuel chacune des étapes de vos activités de preuve : vous disposerez ainsi immédiatement des remarques adaptées à chaque situation.

Le chapitre 8 est un recueil d'études de cas. Il vous permettra de voir sur des exemples les "astuces" utilisées en preuve interactive.

Chapitre 2

Rappels de preuve formelle

Le principe des méthodes formelles est d'utiliser des notions mathématiques pour représenter le comportement des programmes informatiques : c'est pourquoi on parle de modélisation formelle. Les notions mathématiques sont donc les éléments fondamentaux dont dispose l'utilisateur pour construire un modèle correspondant à ses besoins. Mieux il connaît ces notions, meilleure sera son utilisation du langage. Utiliser un langage formel permet d'exprimer des énoncés démontrables, et bien connaître ces notions mathématiques permet de conduire efficacement ces démonstrations.

Le langage B est fondé sur la théorie des ensembles. Cette théorie et toutes les notions qui en découlent sont construites dans le B-Book de J.R. Abrial, chapitres 1, 2 et 3. Si vous n'êtes pas familier avec ces notions mathématiques, nous vous recommandons vivement la lecture de ces trois chapitres qui vous donneront une connaissance *structurée* du sujet. D'autre part, le Manuel de référence du langage B donne pour chaque symbole sa définition et ses propriétés essentielles dans un format proche de celui d'un dictionnaire. Néanmoins, la connaissance de chaque symbole pris séparement ne remplace pas la compréhension des concepts mathématiques. Il est donc important d'étudier la construction de la théorie dont ils sont issus.

Nous allons donc exposer les différentes notions mathématiques dans l'ordre dans lequel elles sont construites. Attention : ce chapitre n'est qu'un résumé destiné à faciliter l'utilisation de l'Atelier B, il ne s'agit en aucun cas d'un cours de mathématique dont il n'a d'ailleurs pas la rigueur. Les notions suivantes sont présentées de manière intuitive et informelle, dans l'ordre de la construction du B-Book.

2.1 Les symboles

L'écriture mathématique est très riche en symboles inhabituels en informatique. Nous utilisons par exemple l'implication $\Rightarrow$, la surcharge $\Leftarrow$, etc. Ces symboles nécessaires pour une écriture synthétique des formules ne sont pas disponibles sur un clavier d'ordinateur. Pour cette raison ils sont représentés par des combinaisons de caractères ASCII : par exemple $\Rightarrow$ est représenté par `=>`.

Dans tous les documents il est préférable d'utiliser la notation symbolique qui facilite la lecture, plutôt que la notation ASCII. En particulier nous n'utiliserons pas la notation ASCII dans ce manuel. Pour chaque symbole, la correspondance ASCII est donnée dans le manuel de référence du langage B, vous en aurez besoin pour toute manipulation de l'Atelier B.

2.2 Le raisonnement formel

Un *raisonnement formel* consiste à *démontrer* un *énoncé* sous un ensemble d'*hypothèses* à l'aide d'une collection de *règles d'inférences*.

Par exemple, nous nous proposons de démontrer $8 > 0$ sous l'hypothèse $8 > 5$. Notre énoncé est donc $8 > 0$ et l'ensemble des hypothèses se réduit à $8 > 5$. Nous supposons que nous avons "tout oublié", c'est-à-dire que nous voulons utiliser seulement les règles et les hypothèses que nous présentons explicitement.Nous supposerons disposer des deux règles d'inférence :

si $5 > 0$, et si $8 > 5$, alors $8 > 0$ (règle 1)
$5 > 0$ est toujours vrai (règle 2)

En appliquant la règle 1 pour démontrer $8 > 0$, comme nous supposons savoir $8 > 5$ (c'est notre hypothèse), il ne nous reste plus qu'à démontrer $5 > 0$. Notre nouvel énoncé est donc $5 > 0$. Nous appliquons alors la règle 2 qui nous dit que $5 > 0$ est toujours vrai. L'application de cette règle ne produit pas de nouveau but, donc la preuve est finie.

Ces notions de preuve sont très intuitives et naturelles. Il est néanmoins utile de bien les saisir à partir des éléments que nous venons de voir, c'est-à-dire :

- démonstration d'un énoncé sous certaines hypothèses,
- collection des règles d'inférences autorisées.

Conventionnellement, nous représenterons l'ensemble de nos hypothèses par le mot **HYP**. Pour indiquer que nous ajoutons une hypothèse H à cet ensemble, nous écrirons **HYP**,H.

Que se passe t-il si l'une des hypothèses que nous supposons est toujours fausse ? Par exemple, doit on considérer que $8 < 0$ est valide sous l'hypothèse fausse $5 < 0$? Intuitivement, cela revient à s'interroger sur un cas impossible. La réponse peut sembler une affaire de conventions, il n'en n'est rien. La cohérence globale de la théorie nous impose de considérer que **Tout énoncé est VRAI sous des hypothèses fausses**. Nous verrons plus loin des exemples dans lesquelles cette nécessité apparait. Cette notion à première

vue très abstraite est souvent employée dans la mise en œuvre du langage B. La génération des obligations de preuve d'un composant B dans certains cas peut et doit produire des obligations de preuve contradictoires (voir paragraphe 5.8). Ces dernières sont JUSTES et participent à la preuve du composant.

2.3 Le calcul propositionnel

Intuitivement, une *proposition logique* peut être définie comme une affirmation vraie ou fausse. Par exemple, "la maison est blanche" est une proposition logique, car la question "cette phrase est-elle vraie ou fausse" a un sens. Par contre "la maison" n'est pas une proposition logique. Une proposition logique est désignée par le terme de *prédicat*.

Soient P et Q deux prédicats. On définit les notations suivantes :

- $P \wedge Q$ (P et Q)
- $P \Rightarrow Q$ (P implique Q)
- $\neg P$ (négation de P)

Ces notions sont utilisées en preuve formelle par les règles suivantes :

– pour démontrer $P \wedge Q$ sous les hypothèses **HYP** il suffit de démontrer P sous **HYP**, puis de démontrer Q sous les mêmes hypothèses.
– pour démontrer $P \Rightarrow Q$ sous les hypothèses **HYP** il suffit de démontrer Q sous les hypothèses **HYP** augmentées de l'hypothèse P, c'est-à-dire d'après nos conventions : **HYP**,P. Ceci est connu sous le nom de règle de *déduction*. On dit aussi que P "monte" en hypothèse.
– pour démontrer $\neg P$ sous les hypothèses **HYP**, nous disposons de la règle suivante : s'il existe un prédicat Q tel que sous les hypothèses **HYP**,P on puisse démontrer à la fois Q et $\neg Q$, alors $\neg P$ est démontré sous les hypothèses **HYP**. Intuitivement, en supposant P nous avons abouti à une contradiction.

Notons que si P est toujours faux, alors $P \Rightarrow Q$ est toujours vrai. Ceci découle de la règle de déduction et rejoint la remarque du paragraphe 2.2 sur les hypothèses fausses. Pour faciliter la manipulation des prédicats dont le statut vrai ou faux est connu, introduisons les notations suivantes :

- **btrue** est le prédicat toujours vrai ;
- **bfalse** est le prédicat toujours faux.

Il nous reste à introduire les deux dernières notations propositionelles, qui se définissent à partir des précédentes :

- $P \vee Q$ (P ou Q) est défini comme $\neg P \Rightarrow Q$.
- $P \Leftrightarrow Q$ (P équivalent à Q) est défini comme $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

La définition du "ou" (disjonction) nécessite quelques commentaires. Intuitivement, elle indique la chose suivante : dire que P ou Q est vrai revient à dire que si P est faux, Q est forcément vrai (traduction de $\neg P \Rightarrow Q$). Cette définition n'est pas symétrique en P et Q,

bien que l'on puisse démontrer que $\neg P \Rightarrow Q$ et $\neg Q \Rightarrow P$ soient équivalents, autrement dit que $P \lor Q$ est identique à $Q \lor P$. D'autre part, la définition de $P \lor Q$ est un exemple justifiant notre assertion que tout but est vrai sous des hypothèses fausses (voir paragraphe 2.2). En effet, considérons la proposition $\mathbf{btrue} \lor Q$. De façon à ce que la définition du *ou* corresponde à la notion naturelle, nous souhaitons que cette proposition soit toujours vraie. Autrement dit :

$$\mathbf{btrue} \lor Q \quad \Leftrightarrow \quad \mathbf{btrue}$$

D'après la définition du symbole $\lor$ cela s'écrit :

$$\begin{array}{lll} \mathbf{btrue} \lor Q & \Leftrightarrow & \neg(\mathbf{btrue}) \Rightarrow Q \\ & \Leftrightarrow & \mathbf{bfalse} \Rightarrow Q \end{array}$$

Il est donc nécessaire de considérer que $\mathbf{bfalse} \Rightarrow Q$ est toujours vrai.

Le lecteur pourra se reporter au B-Book pour avoir la liste des propriétés essentielles des opérateurs propositionnels. Nous citerons ici quelques propriétés moins fondamentales, mais choisies pour leur importance lors de l'utilisation de l'Atelier B :

– $(\mathbf{bfalse} \Rightarrow P) \Leftrightarrow \mathbf{btrue}$
– $(\mathbf{btrue} \Rightarrow P) \Leftrightarrow P$
– $(P \Rightarrow \mathbf{btrue}) \Leftrightarrow \mathbf{btrue}$
– $(P \Rightarrow \mathbf{bfalse}) \Leftrightarrow \neg P$

2.4 Les prédicats quantifiés

Afin d'exprimer les propriétés de nos composants écrits en langage B, nous aurons besoin de nouvelles notions. Par exemple, nous pourrions avoir à démontrer une propriété sur un indice de boucle :

$$indice \in 1 \,..\, 10 \Rightarrow indice < \mathsf{MAXINT}$$

Il nous manque encore beaucoup d'opérateurs pour cette écriture. Tout d'abord nous avons besoin de la notion de variable.

- **Variable** : tout identifiant non prédéfini, constitué avec certaines règles de lettres, chiffres et _ , est une variable [1].

Pour des raisons d'implantation de l'Atelier B, les variables à une lettre ne sont pas autorisées (ce sont des *Jokers*, voir paragraphe 6.2.2). La notion de variable nous permet d'introduire une notion essentielle, le *prédicat universellement quantifié*. Si v est une variable et P un prédicat, on a la construction suivante :

- $\forall v.P$ (lire pour tout v, P.)

[1] les règles syntaxiques précises définissant une variable sont données dans le manuel de référence du langage B

On dit que le prédicat P est *quantifié* par la *quantification universelle* $\forall v$. On dit aussi que la *portée* de la variable quantifiée v est le prédicat P. Donnons quelques exemples de prédicat quantifié :

$$\forall xx.(xx \in \mathbb{N} \ \wedge \ xx < 10 \ \Rightarrow \ xx < 100)$$
$$\forall var.(var = 10 \ \Rightarrow \ var < 100)$$

Remarquons que pour des raisons de typage, on impose que **tout prédicat universellement quantifié soit mis sous la forme** $\forall v.(P \ \Rightarrow \ Q)$ [2].

Une autre remarque essentielle est que **le nom de la variable quantifiée n'importe pas**. On dit que la variable quantifiée est une **variable muette**. Par exemple :

$\forall xx.(xx = 10 \ \Rightarrow \ xx < 100)$
est équivalent à
$\forall yy.(yy = 10 \ \Rightarrow \ yy < 100)$

La portée de la variable muette x dans $\forall x.P$ est le prédicat P uniquement. En particulier une variable de même nom peut être utilisée dans d'autres prédicats, sans conflit. Par exemple :

$$xx = 2000 \ \wedge$$
$$\forall xx.(xx = 10 \ \Rightarrow \ xx < 100)$$

Ce prédicat indique que la variable "externe" xx vaut 2000 et d'autre part que tout nombre égal à 10 est plus petit que 100. Il n'y a pas de confusion entre l'occurence de la variable "externe" et celle de la variable muette. Une telle écriture bien que correcte prête toutefois à confusion, il faut l'éviter.

Les règles d'inférence relatives aux prédicats universellement quantifiés sont légèrement plus complexes, car elles font appel à la notion de variable *non libre* dans une expression, notion que nous n'aborderons pas dans ce chapitre. La règle principale, restreinte aux prédicats de la forme $\forall x.(P \ \Rightarrow \ Q)$, est la suivante :

– Pour démontrer $\forall x.(P \ \Rightarrow \ Q)$ sous les hypothèses **HYP**, si la variable x n'est pas utilisée dans **HYP**, il suffit de démontrer Q sous les hypothèses **HYP**,P.

Cette règle dite *règle de généralisation* signifie que pour démontrer que Q est vrai pour toute variable x vérifiant P, il suffit de se donner une variable x vérifiant P et de faire la preuve de Q sous ces hypothèses. Il y a évidemment un problème si la variable x est déjà utilisée avec un autre sens dans les hypothèses ; il faut alors réécrire $\forall x.(P \ \Rightarrow \ Q)$ avec une autre variable. De telles réécritures de prédicats font intervenir la notion de *substitution* que nous ne développerons pas dans cette introduction mathématique.

[2] De plus, le contrôleur de types de l'Atelier B attend des prédicats quantifiés de la forme syntaxique $\forall v.(P \ \Rightarrow \ Q)$ où P est un prédicat typant les variables introduites

Chapitre 3

Introduction au prouveur

Cette partie est destinée aux utilisateurs du langage B et de l'Atelier connaissant les principes de la preuve, mais n'ayant pas encore utilisé le prouveur de l'Atelier B. Il s'agit d'une "visite guidée" qui permet de savoir où sont les fonctionnalités de preuve de l'Atelier B, et où se trouve leur documentation.

Résumé des notions principales :

la preuve sert à trouver des erreurs
prouver n'est pas programmer

la preuve peut être partiellement automatique
le prouveur ne sait pas démontrer qu'une obligation est fausse
la base de règles est l'ensemble des connaissances mathématiques du prouveur
les mécanismes de preuve choisissent les règles à utiliser
le cœur de preuve désigne la base et les mécanismes
les forces *Rapide, 0, 1, 2, 3* regroupent les mécanismes en niveaux

une obligation de preuve possède un état : *Proved, Unproved*
les commandes de preuve contrôlent le prouveur, en automatique comme en interactif
une PO possède un niveau de démonstration automatique
une PO possède une démonstration interactive

le schéma général de l'outil : un cœur de preuve, des commandes, un pilote automatique ou interactif
le menu de lancement du prouveur permet de choisir le mode de pilotage

la fenêtre de situation globale montre la liste des PO
la fenêtre de contrôle permet d'envoyer les commandes

le prouveur interactif communique avec son interface en mode ligne
utiliser les boutons de l'interface est équivalent à taper des commandes

orienter la preuve sans ajout de connaissance non validée
une démonstration manuelle peut n'employer que des règles validées
les règles manuelles sont des règles non validées

3.1 Approche pratique

3.1.1 Un exemple

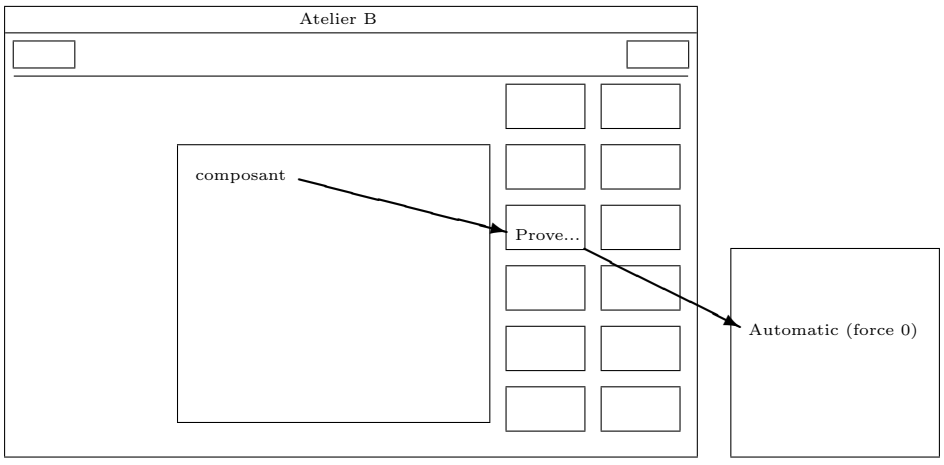
Il faut tout d'abord disposer d'un projet à prouver. Vous pouvez prendre l'exemple suivant.

MACHINE
DemoExample
VARIABLES
few, many
INVARIANT
few $\subseteq \mathbb{N} \wedge$
many $\subseteq \mathbb{N} \wedge$
few $\subseteq$ many
INITIALISATION
few,many := {1,2,3},{2,3,4}
END

Ce composant est volontairement faux : l'initialisation n'établit pas l'invariant parce que {1,2,3} n'est pas inclu dans {2,3,4}. Il produit donc une obligation de preuve fausse, ce qui nous permet de trouver l'erreur. En effet il ne faut jamais oublier que la preuve sert à trouver les erreurs dans les sources B. Cet exemple nous permettra quand même de faire une visite de l'outil de preuve. Rappelons également que l'activité de preuve ne sert qu'à valider le logiciel développé avec la méthode B, ce n'est aucunement une activité de programmation. Il nous paraît utile d'insister car l'opérateur est souvent tenté de considérer les preuves comme des programmes à corriger et à optimiser, état d'esprit qui cause beaucoup de perte de temps.

3.1.2 La preuve automatique

Lancez le contrôle de types et la génération d'obligations de preuve relatives à ce composant. Ensuite, lancez le prouveur automatique en force 0 :



Observez les messages qui s'inscrivent dans la fenêtre de lancement. Les + indiquent des preuves réussies, les - les preuves échouées. A la fin de la session, le prouveur imprime l'état final de preuve :

```
Proving DemoExample

  Proof pass 0, still 3 unproved PO

    clause Initialisation
        ++-

End of Proof
  Initialisation        Proved 2        Unproved 1
TOTAL for DemoExample   Proved 2        Unproved 1
```

Vous venez de lancer le *prouveur automatique* en *force 0* (nous expliquerons plus loin ces termes). Deux des obligations de preuve parmi les trois générées pour ce composant ont été démontrées automatiquement, vous n'avez plus à vous préoccuper de ces obligations ni de ce qu'elles vérifient. Si toutes les obligations de preuve de notre composant étaient ainsi automatiquement démontrées, il n'y aurait rien de plus à faire : le composant serait entièrement prouvé sans intervention de l'opérateur, le coût de la phase de preuve étant alors nul. L'opérateur pourrait néanmoins demander à voir une démonstration mathématique rédigée de chaque obligation de preuve pour faire certifier le logiciel produit, mais il ne le ferait sûrement pas pour toutes les obligations. En résumé, l'activité de preuve est parfois complètement automatique.

Dans notre cas il reste une obligation non démontrée, donc soit le composant est juste mais le prouveur n'a pas trouvé l'une des démonstrations, soit le composant est faux et l'obligation fausse restante localise l'erreur. Nous sommes bien entendu dans le deuxième cas puisque notre composant est volontairement faux ; l'obligation fausse restante doit évidemment découler de ce que $\{1, 2, 3\}$ n'est pas inclu dans $\{2, 3, 4\}$. Pourquoi le prouveur n'affirme t-il pas que cette obligation est clairement fausse ? En fait il se trouve qu'infirmer une obligation de preuve est un problème théoriquement beaucoup plus difficile que de la démontrer, car il faut *choisir* des valeurs vérifiant les hypothèses mais pas les conclusions. Pour cette raison il n'y a actuellement aucun outil automatique qui détecte les obligations de preuve fausses, en particulier le prouveur n'est pas prévu pour cela. Il nous faudra donc visualiser cette PO pour s'apercevoir qu'elle est fausse, ce que nous allons faire par la suite.

Comment la preuve automatique des deux premières obligations a t-elle pu se faire ? Nous allons maintenant présenter les notions fondamentales pour comprendre le fonctionnement du prouveur. Cela nous permettra de l'utiliser pour les démonstrations interactives.

- base de règles : c'est l'ensemble des règles qui constituent la connaissance mathématique du prouveur. Grossièrement, ces règles sont des instructions permettant au prouveur de transformer des formules. Par exemple, une règle indique que toute formule de la forme $a + b$ peut être remplacée par $b + a$ (commutativité de l'addition).
- mécanisme de preuve : Dans une situation donnée, plusieurs règles peuvent s'appliquer et le choix influe sur le cheminement de la démonstration. En reprenant l'exemple

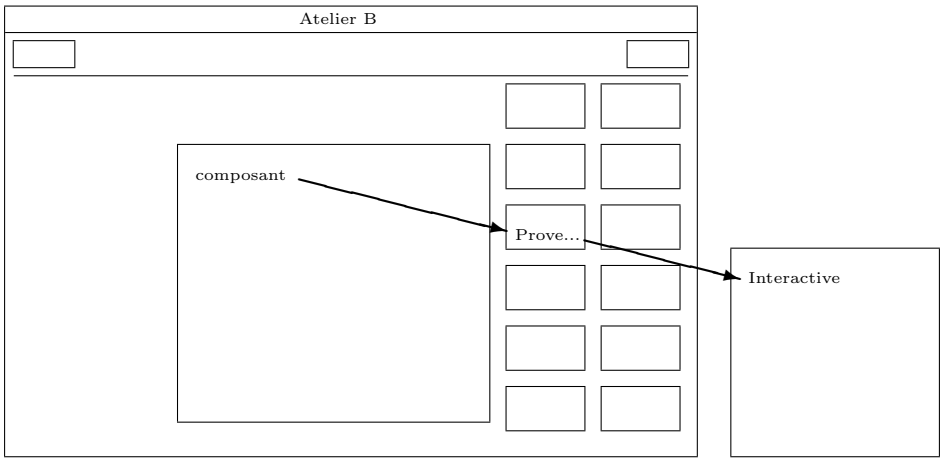
précédent, nous savons que $a + b$ peut être réécrit en $b + a$ mais cela ne nous dit pas s'il est bénéfique de faire cette transformation pour la démonstration en cours. Les mécanismes de preuve sont les procédures heuristiques qui permettent de faire de tels choix. Un exemple représentatif est le mécanisme de réduction des égalités, capable quand plusieurs variables sont égales entre elles de choisir un lot minimal de variables pour exprimer l'obligation de preuve.

– cœur de preuve : c'est un ensemble constitué de la base de règles mathématiques et des mécanismes de preuve. Les obligations de preuve qui ont directement abouti ont été démontrées par le cœur de preuve.
– force Rapide, 0, 1, 2, 3 : les mécanismes du cœur de preuve sont regroupés en ensembles compatibles qui sont les forces (voir paragraphe 4.2). Plus la force est élevée, plus la preuve est longue et risque de boucler, mais plus elle est puissante. La force 0 représente le meilleur compromis performance / rapidité des différentes forces disponibles, c'est elle qu'il faut utiliser en premier.

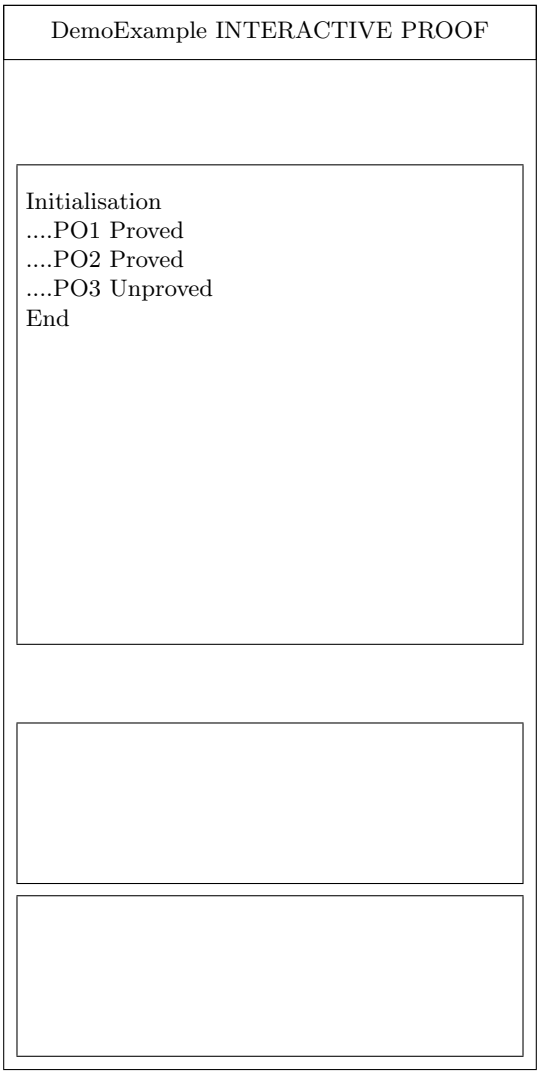
Ce que nous avons fait précédemment, c'est le lancement du cœur de preuve en force 0, c'est-à-dire avec les mécanismes de la force 0, sur chaque obligation de preuve. Les démonstrations réussies se résument à l'application de règles issues de la base de règles, choisies par les mécanismes de la force 0.

3.1.3 Le prouveur interactif

Pour examiner l'obligation de preuve restante, nous allons maintenant entrer dans le prouveur interactif. Sélectionnez votre composant et choisissez comme indiqué ci-dessous :



La fenêtre principale de l'Atelier B se range en icône, tandis que la fenêtre de situation globale du prouveur interactif apparaît. Cette mise en icône automatique de la fenêtre principale est justifiée par l'importance de l'activité de preuve dans un projet B : l'interface essaie de favoriser la concentration de l'opérateur en présentant la preuve comme une activité à part. La fenêtre qui apparaît a l'allure suivante :



Sur ce dessin, nous ne montrons que l'essentiel : le titre "DemoExample INTERACTIVE PROOF" et la liste des obligations de preuve (les différentes zones et boutons de cette fenêtre sont expliqués plus loin). Une première remarque est que l'état *Proved* ou *Unproved* de chaque obligation de preuve est enregistré, c'est bien sûr essentiel pour savoir quand la preuve est finie.

Il suffit de cliquer deux fois sur la ligne "PO3 Unproved" pour positionner la preuve interactive sur cette obligation de preuve. Deux autres fenêtres apparaîssent, l'écran a maintenant l'allure suivante :

Dans la zone du but, nous lisons au bout de la ligne : $1 \in \{2, 3, 4\}$ (le début est un commentaire). C'est le but faux qui dénonce l'erreur, et d'une manière assez analytique : c'est à cause de l'élément 1 que $\{1, 2, 3\}$ n'est pas inclus dans $\{2, 3, 4\}$. Attention, si vous avez choisi le mode ASCII par défaut, le but $1 \in \{2, 3, 4\}$ est affiché $1 : \{2, 3, 4\}$ car " :" est le symbole ASCII pour l'appartenance. Les symboles ASCII sont essentiels pour pouvoir saisir des formules sur un clavier traditionnel, même si l'interface peut afficher en police mathématique.

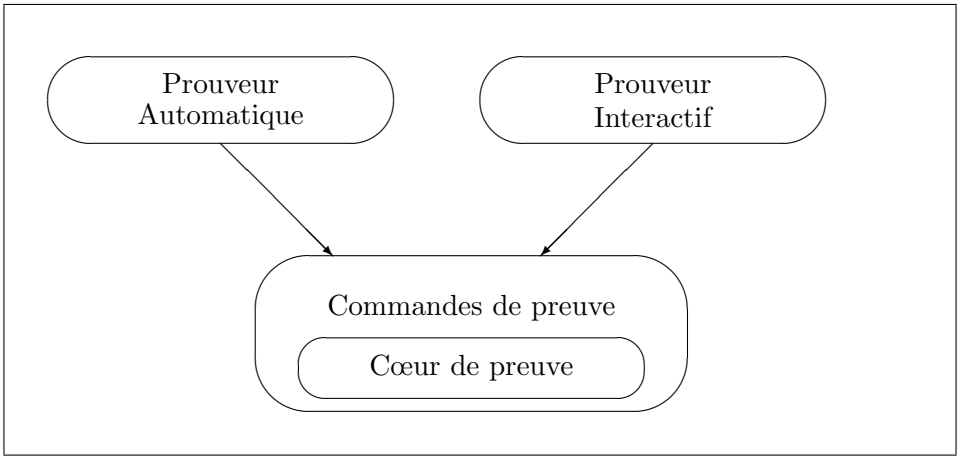
Dans la zone interactive, tapez `pr` et retour chariot derrière le marqueur `PRI>`. La preuve de cette obligation démarre, et échoue sur le but toujours faux **bfalse**. Par la commande `pr` vous avez lancé le cœur de preuve dans la force courante (force 0), ce qui revient à ce que nous avions déjà fait en mode automatique. Le cœur de preuve est toujours disponible en mode interactif, simplement ce n'est plus la seule commande possible. Il y a d'autres *commandes de preuve* possibles, qui permettent d'appliquer spécifiquement une règle, de faire de la preuve par cas, etc. Toutes ces commandes ont deux lettres, et sont désignées dans la documentation par un mot clef plus explicite. Il y a par exemple la commande *DoCases* : `dc`, ou bien *ApplyRule* : `ar`, et ainsi de suite.

– commandes de preuve : ce sont les commandes qui pilotent la preuve. Elles peuvent être soit des appels au cœur de preuve (commande *Prove*, `pr`), soit des actions directes de preuve (par exemple : appliquer une déduction, commande *Deduction*, `dd`). Les commandes de preuve applicables à chaque obligation de preuve sont mémorisées par l'outil, nous allons voir comment.

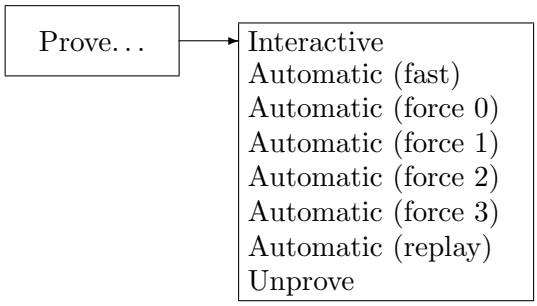
Ce que l'on désigne par *prouveur automatique*, c'est en fait le mode de pilotage du cœur de preuve dans lequel la commande *Prove* (`pr`) est essayée sur chaque obligation de preuve. La démonstration interactive au contraire, permet à l'opérateur de décider lui-même quelles commandes de preuve sont appliquées. La séquence des commandes qu'il a choisies pour démontrer une obligation est mémorisée avec l'état de preuve, c'est la *démonstration interactive*. Pour le mode automatique, il suffit de mémoriser la force maximale tentée pour chaque obligation, c'est le *niveau de preuve automatique* de l'obligation.

3.1.4 L'outil de preuve en général

Le fonctionnement global de l'outil de preuve (prouveur automatique et prouveur interactif) peut se comprendre sur le schéma suivant :



Quand vous utilisez le prouveur interactif, une interface vous permet d'envoyer des commandes vers le prouveur. L'une de ces commandes est *Prove* (`pr`), qui lance le cœur de preuve sur le but courant. Vous disposez ainsi à tout instant des mécanismes de preuve automatiques. Quand vous utilisez le prouveur automatique, toutes les obligations de preuve du composant sont traitées, soit en appliquant une commande `pr`, soit en rejouant les commandes enregistrées. Dans ce cas il s'agit des commandes enregistrées pour chaque obligation de preuve lors de la dernière session interactive. Le menu général de lancement de l'outil de preuve est donc le suivant :



Détaillons ce menu :

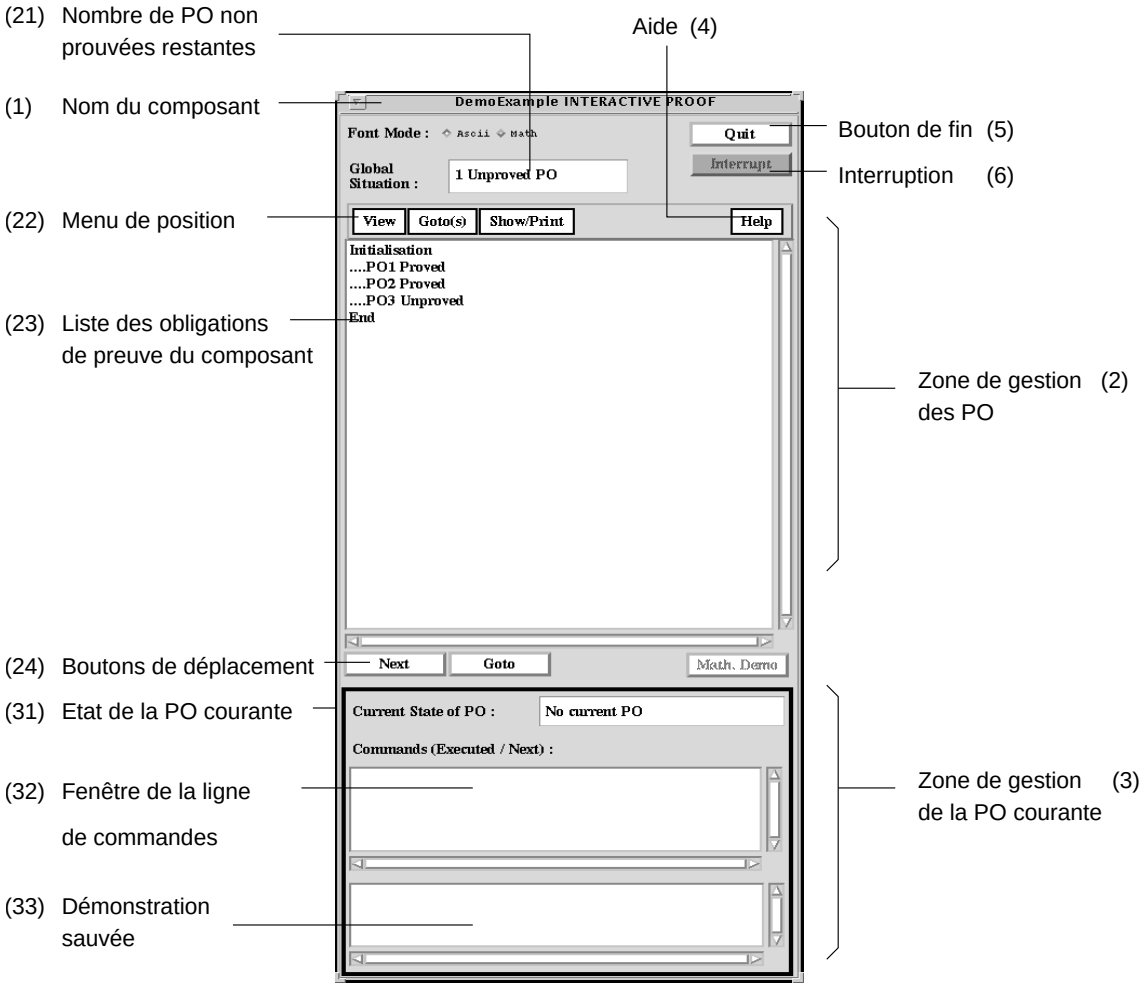
Interactive : lancement du prouveur en mode interactif. Les différentes commandes (déplacement entre les obligations de preuve, commandes de preuve ...) sont entrées par l'opérateur.

Automatic : lancement du prouveur en mode automatique sur toutes les obligations de preuve non prouvées du composant. Avec les options "force 0" à "force 3", les mécanismes du cœur de preuve sont essayés dans chacune des force successives de 0 à 3 jusqu'au chiffre indiqué. Avec l'option "fast" la force Rapide est employée seule. Avec ces options, la seule commande de preuve utilisée est donc `pr`. Par contre, avec l'option "replay" c'est la séquence de commandes enregistrées interactivement pour chaque lemme qui est rejouée.

Unprove : remet toutes les obligations de preuve du composant à l'état "Unproved".

3.1.5 Détail des principales fenêtres

Décrivons maintenant la fenêtre de situation globale du prouveur interactif, celle qui affiche la liste de obligations de preuve :



Les différentes parties de cette fenêtre sont les suivants :

1 Le nom du composant qui est rappelé dans la barre de label de la fenêtre.

2 La zone de gestion des PO regroupe tout ce qui concerne le composant vu dans son ensemble :

21 L'indicateur d'obligations de preuve restant à démontrer indique le nombre de lemmes non encore prouvés. Cette zone devient verte lorsque le composant est entièrement prouvé.

22 La barre des menus de position permet de régler l'affichage de la liste des obligations de preuve, d'utiliser les *Goto* spéciaux comme *GotoWithoutsave*, etc. Le bouton *Show/Print* permet d'imprimer ou de sauver dans un fichier les éléments de la preuve interactive.

23 La liste donne les obligations de preuve du composant triées par clause, avec leur indication d'état. Un double clic sur une obligation de preuve est équivalent à un *Goto* sur cette obligation de preuve.

24 Les boutons de déplacement principaux permettent le positionnement. Le bouton *Next* permet d'aller à la prochaine obligation de preuve non prouvée, le bouton

Goto permet l'accès à l'obligation de preuve désignée dans la liste. Le bouton *Mathematical Demo* permettra d'écrire dans un fichier la démonstration d'une obligation de preuve.

3 la zone de gestion de l'obligation de preuve courante regroupe tout ce qui est spécifique à la preuve en cours, c'est-à-dire l'obligation qui a fait l'objet du précédent *Goto*. Elle contient :

31 L'état de la PO courante, c'est-à-dire son état (Prouvée, non prouvée) dans la démonstration en cours et son état dans la démonstration sauvée.

32 La fenêtre de la ligne de commande contient toutes les commandes de preuve effectuées sur la PO courante, indentées en fonction de l'arbre de preuve.

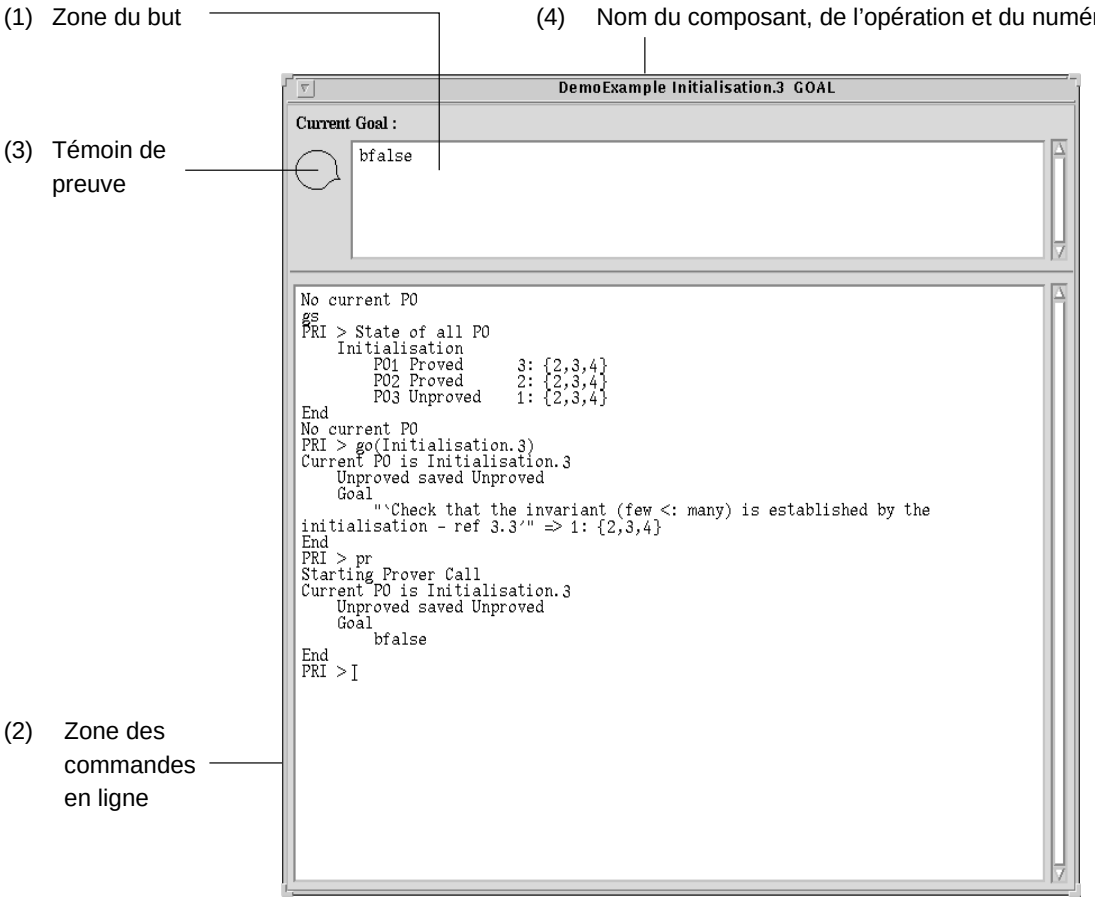
33 La démonstration sauvée contient la ligne de commande sauvée pour cette obligation de preuve.

4 Le bouton d'aide lance la documentation en ligne du prouveur.

5 Le bouton de fin arrête le prouveur.

6 Le bouton d'interruption stoppe la dernière commande de preuve interactive. Son emploi le plus courant est l'interruption d'une commande *Prove* ou *ApplyRule* qui boucle. Ce bouton est invalidé quand le prouveur est en attente d'une commande opérateur (c'est le cas sur cette image).

Détaillons les éléments de la fenêtre centrale qui affiche le but et dans laquelle nous pouvons taper les commandes de preuve :



Les éléments de cette fenêtre sont :

1 La zone du but : elle contient le but courant affiché dans une fenêtre avec ascenseur horizontal. Cette zone se colore en vert lorsque la démonstration aboutit. Le témoin de preuve porte alors la mention *Proved*.

2 la zone des commandes en ligne : dans cette zone vous tapez toutes les commandes. Il y a quatre sortes de commandes :

- Les commandes d'action : ce sont les commandes de preuve proprement dites. Les plus courantes sont :
  - *Prove* (`pr`) : appel au cœur de preuve.
  - *AddHypothesis* (`ah`) : ajout d'une hypothèse, démontrable à partir des hypothèses courantes.
  - *ApplyRule* (`ar`) : utilisation directe d'une règle du prouveur ou ajoutée.
  - *DoCases* (`dc`) : déclenchement de preuve par cas.
  - *useEqualityinHypothesis* (`eh`) : utilisation d'une égalité en hypothèse.
  - *SuggestforExist* (`se`) : proposition pour un but de la forme $\exists x.P$.
  - *ParticularizeHypothesis* (`ph`) : instantiation d'une hypothèse de la forme $\forall x.P$.
  - *FalseHypothesis* (`fh`) : dénonciation d'une hypothèse contradictoire.
- Les commandes de position : sans permettre d'avancer la preuve, elles servent à reculer ou à rejouer des commandes enregistrées.

– Les commandes d'information : sans aucune action sur la preuve, elles permettent de chercher et d'afficher les informations nécessaires pour progresser dans la démonstration de la PO. Les commandes les plus importantes de ce type sont *Search Hypothesis*, qui permet la recherche d'une hypothèse en fonction d'un certain filtre, et *Search Rule*, qui permet la recherche d'une règle dans la base de règles.
– Les commandes de "finalisation" : généralisation d'une démonstration, interruption d'une preuve qui boucle, demande de la sortie d'une démonstration après succès, etc.

Tous les échanges entre le prouveur et son interface apparaîssent dans cette zone de commandes en ligne. Nous allons examiner de plus près cette notion essentielle dans le paragraphe suivant.

3 Le témoin de preuve : il porte la mention *Proved* quand la démonstration courante aboutit.

4 Le titre barreau de la fenêtre précise le nom du composant, le nom de l'opération dont est issue l'obligation de preuve et le numéro de cette dernière.

3.1.6 Echanges avec le prouveur interactif

L'outil de preuve interactive est formé de deux parties, le **prouveur interactif** proprement dit et son **interface homme-machine**.

Le prouveur effectue les commandes (commandes de preuve ou d'information). L'interface homme / machine vous affiche les résultats et transmet vos commandes au prouveur. Tout le dialogue avec le prouveur se ramène à des interactions de type commande vers l'outil / réponse de l'outil. Cette interaction en mode ligne est entièrement visible dans la zone des commandes en ligne. Par exemple, si vous appuyez sur le bouton *Next* de la fenêtre de situation globale, l'interface émet une commande `ne` (qui veut dire *Next*) vers le prouveur de la même manière que si vous aviez tapé `ne` dans la zone de commandes en ligne. Le prouveur effectue alors la commande, puis renvoie l'état courant sous la forme de lignes de texte que l'interface répartit dans ses fenêtres, tout en laissant une trace de la réponse, visible dans la zone des commandes en ligne.

L'interface pour la preuve interactive simule donc toujours un dialogue en mode ligne avec le prouveur, dialogue que l'opérateur peut avoir directement depuis la zone des commandes en ligne. Toutes les opérations peuvent être faites depuis cette zone (commande de preuve, positionnement...), mais il faut connaître la syntaxe de chaque commande. Tapez `help` pour obtenir la liste des commandes disponibles. Ces commandes sont toujours composées de deux lettres minuscules, qui sont les premières lettres des mots composant le mnémonique de la commande.

Par exemple :

La commande *Search Hypothesis* s'écrit `sh`

S'il n'y a qu'un seul mot dans le mnémonique, la commande est formée de ses deux premières lettres : par exemple, `ne` pour *Next* ou `qu` pour *Quit*. Ces commandes prennent souvent des arguments utilisant des mots clefs, comme `Goal`, `AllHyp` (des dialogues par boutons pourraient éviter d'avoir à employer ces mots clef).

3.2 Le principe de la preuve interactive

Comment les commandes évoquées précédemment peuvent-elles faire aboutir une preuve qui échoue lorsqu'elle est lancée en mode automatique ? Comment pourrez-vous piloter une preuve vers son succès avec ces commandes ? Ce sont les questions auxquelles nous allons répondre dans ce paragraphe. Nous montrons le principe de ce pilotage sur un exemple.

Soit à démontrer le lemme suivant :

$$
\begin{array}{l}
xx \in 1\,..\,10 \ \wedge \\
yy \in 2\,..\,10 \ \wedge \\
zz \in 3\,..\,10 \\
\Rightarrow \\
\mathsf{max}(\{xx, yy, zz\}) \leq 10
\end{array}
$$

Nous supposons que les mécanismes du cœur de preuve ne suffisent pas à démontrer ceci. Pour démontrer ce lemme, il faut faire trois cas suivant que le maximum est xx, yy ou zz. L'opérateur peut déclencher un premier cas par une commande *DoCases* :

$$\mathsf{dc}(\mathsf{max}(\{xx, yy, zz\}) = xx)$$

La preuve se poursuit alors pour $\mathsf{max}(\{xx, yy, zz\}) = xx$, puis pour $\mathsf{max}(\{xx, yy, zz\}) \neq xx$. Il est possible que ces deux cas soient directement démontrés par les mécanismes du cœur de preuve, c'est ce que nous supposerons, dans ce cas la preuve aboutit. **une action opérateur déclenchant une preuve par cas a suffi pour permettre la démonstration**.

Le but de cet exemple est de faire comprendre comment une interaction dans une preuve peut faire aboutir celle-ci par les mécanismes du cœur de preuve, sans que vous n'ayez introduit la moindre connaissance mathématique non validée. En fait, vous pilotez la preuve en ajoutant votre intuition, puis en relançant le prouveur jusqu'au nouvel échec ou jusqu'au succès. d'après une expression très imagée de F. Mejia, l'opérateur "joue au billard" avec les mécanismes du prouveur.

Il est clair qu'une bonne intuition de ce que les mécanismes vont faire est utile pour ce style de preuve interactive. Par exemple, soit à prouver :

$$
\begin{array}{l}
xx \in \mathbb{N} \ \wedge \\
yy \in \mathbb{N} \ \wedge \\
yy \leq 10 \ \wedge \\
xx + 1 - 8 \leq yy \\
\Rightarrow \\
xx + 1 - 8 \leq 10
\end{array}
$$

Les mécanismes du cœur de preuve peuvent échouer sur un tel lemme car ils cherchent d'abord à simplifier le but, qui devient $xx \leq 17$. Il est alors beaucoup plus difficile de faire le lien avec l'hypothèse clef $xx + 1 - 8 \leq yy$. Si l'opérateur voit cette simplification mal choisie, il décide d'agir avant d'appeler le cœur de preuve. Une action possible est de provoquer l'application de l'une des règles de la base du prouveur (nous verrons plus loin le format de ces règles et comment on les recherche par *SearchRule*). Supposons qu'il existe une règle "OrderXY.77" qui puisse démontrer notre PO. La commande pour l'appliquer

est *ApplyRule*, nous ne décrirons pas ici sa syntaxe ni celle de notre règle. La commande tapée serait par exemple :

```
ar(OrderXY.77,Once)
```

La preuve réussit. Nous avons alors affaire à une preuve effectuée de manière totalement manuelle, sans appel au cœur de preuve et sans ajout de règle.

Dans certains cas, il se peut que la règle spécifique nécessaire à la preuve ne soit pas dans la base du prouveur, et qu'aucune autre méthode de démonstration n'aboutisse. Il faut alors ajouter la règle en tant que **règle manuelle**.

Si des règles manuelles ont été utilisées pour la preuve d'un composant, cette preuve peut être fausse si certaines des règles sont fausses. Il s'agit alors d'une preuve sortant de la sphère sécuritaire du prouveur de l'Atelier B, mais la validation de cette preuve se ramène à la validation des règles manuelles, plus simple que la validation de la preuve elle-même. Il faut donc que le nombre et la complexité de ces règles soient petits devant la taille de la preuve initiale. En pratique cela s'obtient en utilisant les règles manuelles de manière occasionnelle, conjointement avec les mécanismes sécuritaires du prouveur. Les règles manuelles ne résolvent alors que des sous buts. Les règles manuelles sont écrites dans un langage appelé langage de théorie, dans le fichier `composant.pmm`. Ce fichier est totalement écrit par l'opérateur, l'outil ne le crée pas par défaut afin que son absence éventuelle valide la preuve.

Pour expliquer le principe de la preuve interactive, nous avons du aborder très rapidement les notions de règle, de commande d'application de règle, . . .sans les détailler. C'est l'objet du chapitre 6.

3.3 Conclusion

Vous savez maintenant faire fonctionner le prouveur dans l'Atelier B, et par quels principes les obligations de preuves dont la démonstration automatique échoue peuvent être démontrées interactivement. La suite de ce manuel insiste sur la méthodologie de la preuve plutôt que sur la présentation des différentes commandes. En effet, la preuve d'un projet B doit être conduite avec méthode.

Avant d'utiliser le prouveur interactif pour démontrer les obligations de preuve de votre projet, lisez le chapitre 4. En commençant directement par la preuve formelle vous risqueriez de dépenser beaucoup de temps sur des obligations de preuve non essentielles, et de découvrir ensuite des erreurs dont la correction invalide les preuves précédentes.

Les commandes du prouveur interactif sont décrites en détail dans le manuel de référence du prouveur. Il n'est pas nécessaire de lire cette référence entièrement pour pouvoir utiliser le prouveur, en effet les commandes seront présentées dans le chapitre 6 par ordre d'importance. Il suffit alors de consulter le manuel de référence suivant les besoins.

Chapitre 4

Méthode générale

4.1 Les phases de preuve

Quelles sont les activités de preuve dans le développement d'un projet informatique utilisant la méthode B et l'Atelier B ? Etudions ceci sur un exemple : soit un projet constitué d'une seule machine abstraite (la spécification) et de son implantation (le programme concret). Ce projet sera probablement réalisé de la manière suivante :

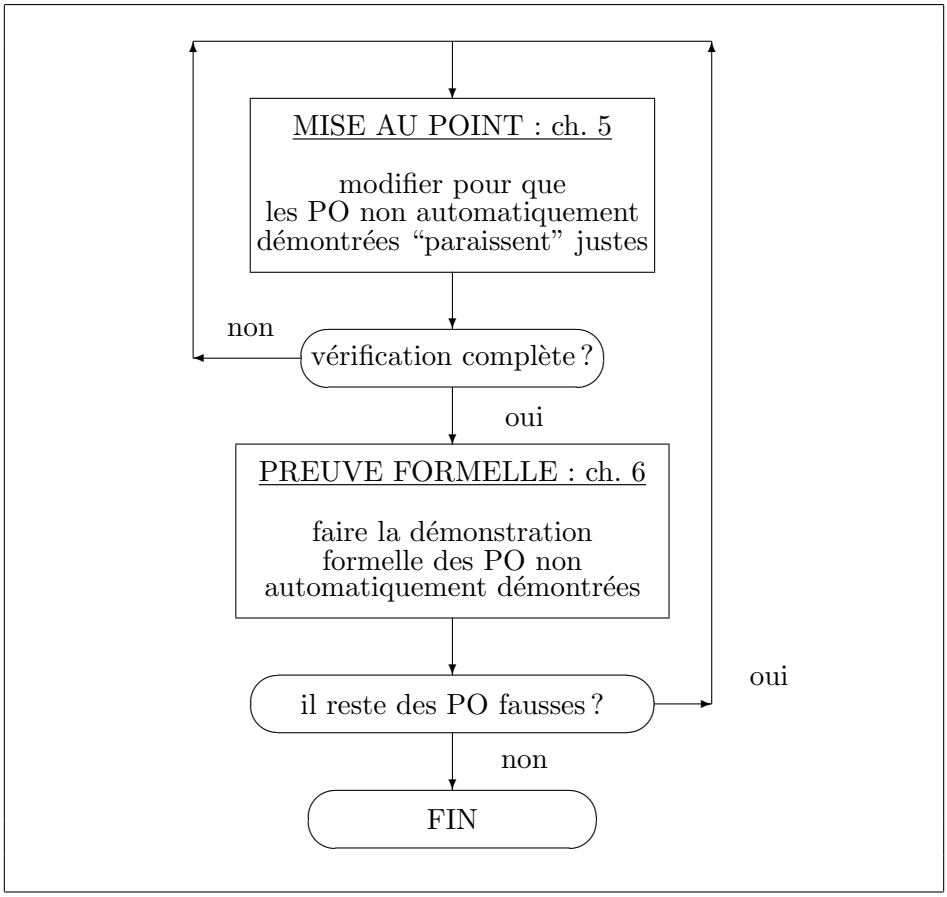
1. Ecrire la machine abstraite en fonction du cahier des charges ;
2. Contrôler la formalisation correcte du besoin ;
3. Lancer le prouveur automatique sur cette machine abstraite ;
4. S'il reste des obligations de preuve non automatiquement démontrées, contrôler rapidement qu'elles soient justes. Si certaines sont fausses, la machine abstraite est incohérente, il faut la corriger ;
5. Ecrire l'implantation ;
6. Relire cette implantation par rapport à la machine abstraite ;
7. Lancer le prouveur automatique sur l'implantation ;
8. S'il reste des obligations de preuve non démontrées, contrôler qu'elles soient justes. Si certaines sont fausses l'implantation n'est pas correcte, il faut la corriger ;
9. Faire la démonstration formelle des obligations de preuve restantes dans la machine abstraite et dans l'implantation à l'aide du prouveur interactif.

Dans le processus de développement ci-dessus, les étapes 3, 4, 7, 8 et 9 sont les étapes de preuve. Nous voyons que la preuve formelle complète est faite à la fin : il faut éviter les démonstrations longues tant que les composants risquent de devoir être modifiés. C'est pourquoi il y a deux phases bien distinctes dans l'activité de preuve en B : la **mise au point** des composants par vérification des obligations de preuve et la **preuve formelle finale**.

Cette distinction se retrouve toujours quelle que soit la méthode de développement utilisée. Notons que dans les étapes 3 et 7, il faut utiliser le prouveur automatique configuré pour

être assez rapide (force 0, voir paragraphe suivant), car nous devrons attendre qu'il termine pour passer à l'étape suivante.

Savoir si on se place en phase de mise au point ou en phase de preuve finale est essentiel. Cette méthode par phase peut se représenter par le schéma suivant :



Attention : ce que nous appelons "mise au point" désigne implicitement la mise au point *du point de vue preuve* : nous n'abordons pas dans ce manuel les méthodes générales pour écrire et contrôler des projets en langage B.

Faut-il avoir fini la phase de mise au point de tous les composants du projet avant de passer en phase de preuve formelle ? Faut-il finir complètement la mise au point d'un composant avant d'écrire le composant suivant ? Nous resterons volontairement imprécis sur ce sujet, qui dépend de la taille et de la structure du projet. Tout au plus peut on dire qu'il ne faut pas attendre d'avoir écrit tous les composants du projet avant d'aborder les problèmes de preuve, et qu'il ne faut pas entreprendre trop tôt la preuve formelle d'un composant.

Durant la phase de preuve formelle, on suppose ne plus avoir à retoucher les composants, sauf si une obligation de preuve supposée juste est en fait fausse. **Dans ce cas l'impact des modifications sur les démonstrations déjà faites peut provoquer des pertes de temps**. C'est pourquoi la phase de mise au point est très importante.

Les changements de phase de preuve sont des étapes délicates. Lors de ces changements, attention aux pièges suivants :

– **S'assurer que les composants ont bien leur forme définitive avant la phase de preuve formelle.** Il est en effet courant d'écrire les composants dans une version réduite ou incomplète pour une mise au point rapide, en prévoyant une étape de finition. Cette finition doit être faite avant la preuve formelle.
– **En phase de preuve formelle, s'assurer que toutes les obligations de preuve peuvent être présumées justes.** En effet, si une obligation de preuve fausse est découverte durant la phase de preuve formelle, l'opérateur est tenté de poursuivre cette phase après avoir modifié un composant, alors qu'il est impératif de refaire une phase de mise au point.

4.2 L'utilisation des forces du prouveur

L'opérateur ne s'attend jamais à ce qu'un ordinateur conçoive et réalise les programmes à sa place, parce que l'ordinateur ne peut pas deviner ce qu'il faut obtenir. Dans le domaine de la preuve au contraire, ce qu'il faut obtenir est clair : nous voulons des démonstrations des énoncés à prouver à partir d'un ensemble de règles connues. Il n'existe malheureusement pas d'algorithme qui produise la démonstration de tout énoncé correct, les démonstrateurs automatiques et en particulier celui de l'Atelier B appliquent donc un ensemble de *tactiques* plus ou moins heuristiques qui peuvent échouer ou aboutir. Si une démonstration est obtenue elle est correcte mais l'échec d'une tactique ne prouve pas que l'énoncé est faux.

Une différence importante entre la preuve et d'autres tâches plus classiques comme par exemple la conception de programmes est donc la possibilité d'aboutir par le travail automatique d'un ordinateur. Pour cette raison il est toujours souhaitable de faire travailler les prouveurs automatiques sur les projets à démontrer quel que soit le temps de calcul nécessaire, parallèlement au travail de preuve manuel.

Les tactiques employées en preuve automatique sont généralement d'autant plus coûteuses en temps de calcul qu'elles sont capables de trouver des démonstrations complexes. De plus les tactiques les plus complètes peuvent souvent provoquer des boucles infinies dans les démonstrations. C'est pourquoi les différentes tactiques du prouveur de l'atelier B ont été regroupées en *forces*. Les différentes forces sont les suivantes :

Force	*Temps indicatif par lemme*	*performance*
0	**toujours moins de 10 secondes**	70%
1	de quelques secondes à 2 ou 3 minutes	+1%
2	de quelques minutes à quelques dizaines de minutes	+3%
3	de quelques dizaines de minutes à plusieurs heures	+1%
"Rapide"	moins de trois secondes	30%

Les temps ci-dessus sont indicatifs, ils concernent surtout les premières obligations de preuve de chaque opération. En effet les obligations de preuve suivantes ont beaucoup d'hypothèses en commun avec les premières et le traitement de ces hypothèses est factorisé. Les performances sont très indicatives ; elles sont indiquées en pourcentage d'obligations de preuve démontrées sur un projet "standart" entièrement juste. Les performances des forces 1, 2 et 3 sont indiquées en gain par rapport à la force précédente parce que les

forces 1, 2 et 3 s'emploient toujours en séquence à partir de la force 0. Ainsi les forces les plus élevées ne peuvent traiter des lemmes démontrés dans une force inférieure, ce qui économise le temps de calcul et limite le risque de déclencher des boucles infinies. La force "Rapide" s'emploie seule.

La force 0 est considérée comme l'optimum entre l'efficacité et le temps de calcul. C'est cette force qui doit être utilisée pour tenter de démontrer les obligations de preuve avant même de les lire, afin de limiter leur nombre. Elles sont en effet très nombreuses, on compte en moyenne une obligation de preuve par ligne de code exécutable produite. La force "Rapide" n'a pas des performances suffisantes pour cet emploi. Les forces 1, 2 et 3 s'emploient plutôt en parallèle durant les phases de mise au point et de preuve formelle, en espérant que certaines obligations de preuve seront démontrées automatiquement avant d'avoir été traitées manuellement.

Les principes d'utilisation des forces du prouveur de l'Atelier B sont les suivants :

- **Employer la force 0** : ne jamais examiner une obligation de preuve avant d'avoir tenté de la démontrer avec le prouveur automatique en force 0.
- **"Occuper" les ordinateurs** : si vous disposez d'ordinateurs inemployés sur lesquels l'Atelier B est installé, il est toujours utile de lancer le prouveur automatique en force 1, 2 ou 3 sur ceux-ci pour démontrer des obligations de preuve justes de votre projet.
- **Ne pas attendre** : n'attendez pas que le prouveur automatique en force 1, et ou 3 termine le traitement de votre projet pour commencer les phases de mise au point ou de preuve formelle.

Le prouveur automatique est aussi employé en preuve interactive. Ceci peut sembler paradoxal, mais ce que nous appelons preuve interactive est en fait une preuve semi-automatique dans laquelle les actions de l'opérateur s'intercalent entre des appels au cœur de preuve. Il faut donc choisir la force utilisée également en preuve interactive, elle conditionne toutes les interventions du cœur de preuve dans les démonstrations manuelles. Dans la majorité des cas, il est conseillé d'employer la force 0 ; la force 1 est parfois utilisée aussi. Nous verrons ceci en détail dans le chapitre 6.

4.3 Les lemmes de bonne définition

Il est possible d'écrire des expressions qui ressemblent à des expressions mathématiques, mais qui n'ont pas de sens : par exemple $\mathsf{max}(\varnothing)$. Nous utilisons à tort l'opérateur max qui n'est défini que pour un ensemble d'entiers au moins non vide. De telles expressions posent des problèmes concernant la preuve automatique par règles d'inférence. Nous n'aborderons ces problèmes que du point de vue de leurs conséquences pratiques. L'outil **mdelta** permet de vérifier *a posteriori* la validité d'un projet B au sens de sa bonne définition (voir Outil mdelta - Manuel Utilisateur) en générant des lemmes de bonne définition.Dans la plupart des cas, ces lemmes de bonne définition sont triviaux, il suffit de les lire rapidement pour les vérifier. Si l'un de ces lemmes est faux alors :

- Soit il y a des expressions mal typées dans les composants en cours de preuve : en principe, cela provoque toujours l'apparition d'une obligation de preuve fausse, que le prouveur ne démontre pas.

– Soit l'une des expressions ajoutées en cours de preuve interactive est dépourvue de sens. Attention, dans ce cas la preuve n'est pas valide (il est néanmoins très rare qu'elle puisse aboutir).

Le contrôle de type des composants B écarte la plupart des possibilités d'expressions mal formées. Les problèmes restants relèvent de la preuve, ce sont :

– $\mathsf{card}(E)$ si E n'est pas un ensemble fini.
– les expressions $\mathsf{max}(E)$, $\mathsf{min}(E)$ si E est vide ou si E n'admet pas de maximum ou de minimum.
– les expressions $f(x)$ si x n'est pas dans le domaine de f, ou si f est une relation mais pas une fonction.
– les divisions par une expression potentiellement nulle.
– $\mathsf{size}(s)$, $tail(s)$, etc. si s n'est pas une séquence, par exemple $\mathsf{size}(\{2 \mapsto 3\})$.

Chapitre 5

La phase de mise au point

Les notions essentielles présentées dans ce chapitre sont les suivantes :

Méthode générale : après relecture et preuve en force 0 du composant, parcourir et examiner les PO.

Les moyens de visualisation d'une PO sont le PO viewer ou le prouveur interactif.
Avec le prouveur : choisir la PO, faire `dd`, utiliser les fonctions de recherche.

Le parcours des obligations doit conduire en premier vers les PO difficiles.
Il faut parcourir la liste des obligations en remontant.
On peut faire plusieurs phases de parcours des obligations : rapide, finale ou bien rapide, de simplification, finale.

L'examen d'une obligation de preuve se fait en cinq étapes : lecture du but, justification, sélection des hypothèses clefs, démonstration intuitive, notes et essais.
Lors de l'examen d'une obligation, le composant B doit être accessible.
Lecture du but : il faut l'interpréter et isoler la contrainte vérifiée.
Justification : utiliser le sens physique du composant.
Sélection des hypothèses : chercher en remontant et utiliser les fonctions de recherche du prouveur.
Démonstration intuitive : reprendre la justification et voir les règles employées.
Notes et essais : noter les simplifications envisagées et tenter une démonstration rapide.

La simplification des expressions d'un composant peut faciliter sa preuve.
Tout projet juste n'est pas forcément démontrable : sous une forme maladroite, le projet peut produire des preuves trop compliquées.
Redécouper un projet pour simplifier sa preuve.
Mettre les expressions sous la forme normalisée du prouveur.
Chercher à faire apparaître des égalités littérales.
Chercher à mettre les expressions arithmétiques sous forme canonique.

La preuve rapide d'une PO doit être tentée en se donnant une limite de temps.
Essayer en premier le prouveur de prédicats.
Ne pas tenter de démonstration de plus de 5 commandes.
Tenter de généraliser une démonstration rapide à d'autres obligations.

Les obligations avec des expressions complexes peuvent se lire en utilisant le prouveur comme simplificateur.
Les buts existentiels simples se traitent avec *SuggestforExists*, ils traduisent parfois un excès d'indéterminisme dans le composant.
Les buts existentiels abstraits traduisent l'expression imprécise d'une constante abstraite.
Les buts non découpés sont souvent dus à des disjonctions.

Si une obligation de preuve fausse est découverte, il faut corriger le composant avant de poursuivre.

Il faut vérifier que l'obligation est bien fausse, ce n'est pas évident.
Chercher un contre-exemple est un bon moyen de s'assurer qu'une obligation est fausse.
Reporter le contre-exemple dans le composant permet de localiser l'erreur.

5.1 La méthode générale de mise au point

La méthode générale de mise au point d'un composant par la preuve consiste à parcourir les obligations de preuve pour vérifier qu'elles sont toutes justes. A chaque obligation de preuve fausse découverte, le composant est modifié. De façon à limiter le nombre de lemmes à lire, la phase de mise au point doit être faite après le passage du prouveur en force 0. La force 0 du prouveur a été prévue pour cet usage, au contraire des forces plus élevées qui sont beaucoup plus coûteuses en temps. La force 0 est le compromis idéal entre la performance et le temps pour assurer la correction *à priori* des composants (voir paragraphe 4.2).

La phase de mise au point par la preuve ne doit commencer que si **il n'y a plus de corrections visibles par la seule lecture du composant**. Autrement dit, le composant doit avoir été relu avant la phase de mise au point par la preuve. En effet il n'est pas nécessaire de passer par le contrôle de très haut niveau que représente la preuve si les erreurs se détectent par simple lecture !

En résumé, la méthode générale en phase de mise au point est la suivante :

Après relecture du composant et **après avoir appliqué la preuve automatique en force 0** ;

- **Parcourir** les obligations de preuve restantes (voir paragraphe 5.3) ;
- **Examiner chaque obligation de preuve** en utilisant la méthode du paragraphe paragraphe 5.4.

Ces deux étapes seront examinées en détail dans la suite de ce chapitre. Pour l'instant, nous allons étudier les méthodes pratiques de visualisation d'obligations de preuve avec l'Atelier B.

5.2 Méthodes de visualisation des obligations de preuve

Pour faire défiler les obligations de preuve non démontrées par la force 0 du prouveur, il y a deux méthodes :

- **Par le PO Viewer** : utiliser le *PO Viewer* de l'Atelier (accessible par le menu `Status...` `Show/Print PO` de la fenêtre principale). Le *PO Viewer* est un simple visualisateur d'obligations de preuve, sans traitement.
- **Par le prouveur** : entrer dans le prouveur interactif, et pour chaque PO à visualiser faire `dd` (*Deduction*, pour monter les hypothèses locales) et `rp` (*ReducedPo*, visualisation réduite aux hypothèses qui ont un symbole en commun avec le but.

Le *PO Viewer* est d'accès plus rapide que le prouveur interactif. En effet, il ne contient aucune base de règle ou tactique de preuve, c'est seulement un afficheur : il est donc beaucoup plus léger que le prouveur. **L'usage du prouveur comme visualisateur de PO s'impose dans les cas suivants** :

- **Hypothèses complexes** : si la structure du composant est telle qu'il va y avoir plus de 200 hypothèses environ, et que les hypothèses intéressantes pour chaque obligation de preuve risquent de ne pas être regroupées dans les dernières. Dans ce cas les fonctionnalités de recherche du prouveur interactif seront utiles pour faire la sélection des hypothèses.
- **obligations de preuve nombreuses pour une opération** : si il y a plus de 100 PO pour une même opération, alors l'usage du *PO Viewer* risque d'être difficile car il visualise les obligations de preuve par opération en une fois. L'affichage produit contient donc trop d'informations.

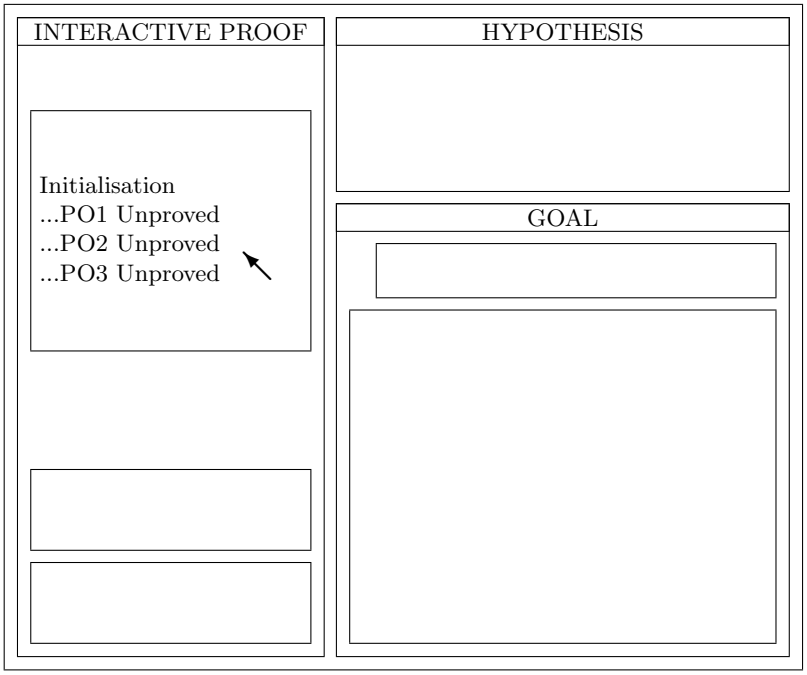
5.2.1 Visualisation avec le prouveur

Si vous utilisez le prouveur interactif pour visualiser des obligations de preuve, voici la méthode à employer :

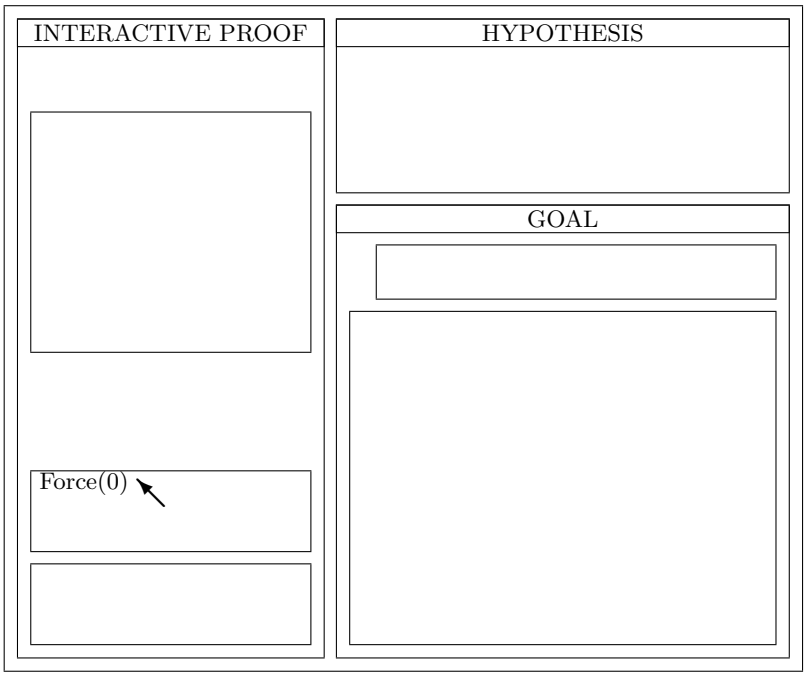
1. Accédez à l'obligation choisie
2. Faites monter les hypothèses locales
3. Utilisez les fonctions de recherche du prouveur interactif

Pour expliciter ce paragraphe, nous allons faire ces trois étapes en nous aidant de dessins qui représentent schématiquement l'écran de preuve. Sur ces dessins, seules les parties concernées seront représentées.

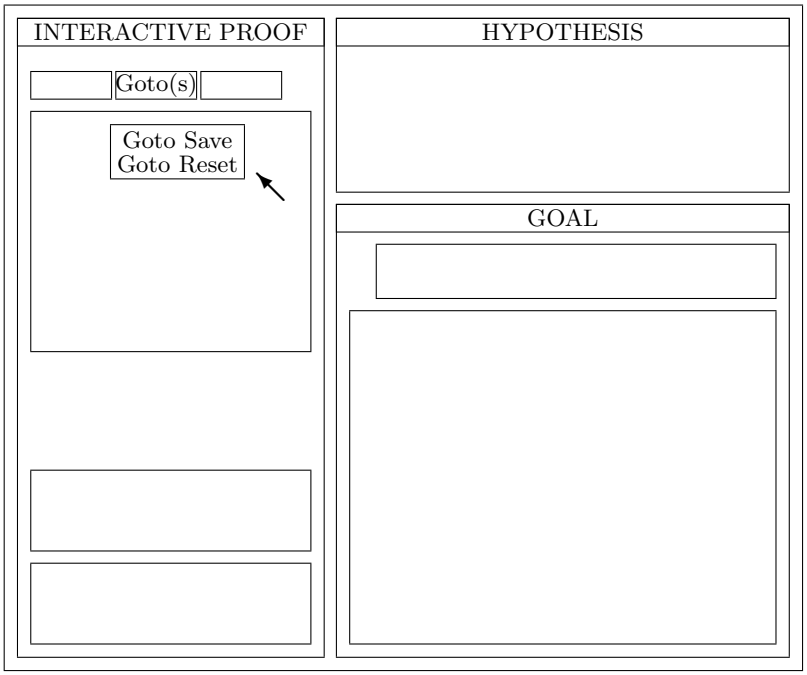
1. **Accédez à l'obligation choisie** : choisissez l'obligation de preuve à lire (ce choix est l'objet du paragraphe suivant) dans la liste de la fenêtre de situation globale, et cliquez deux fois dessus.



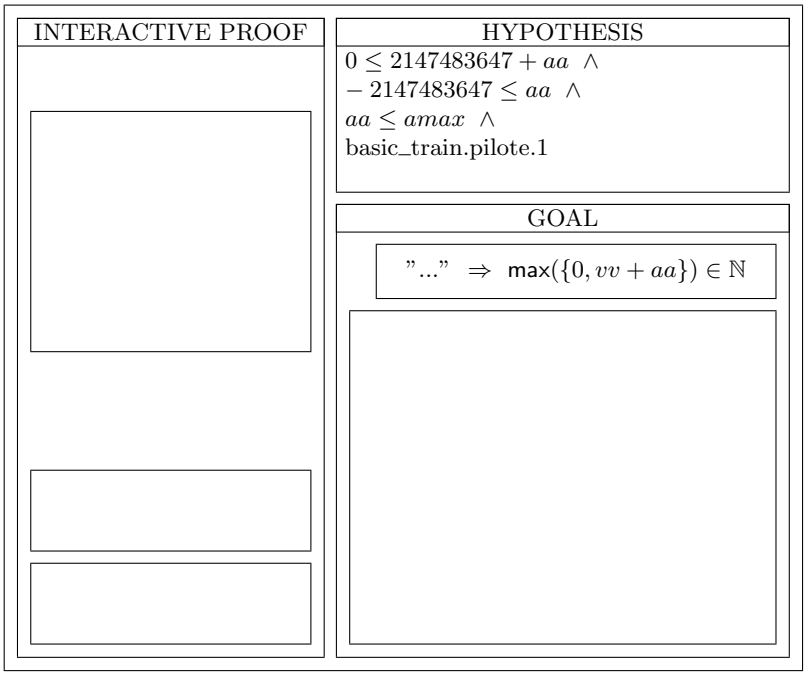
Rappelons que la force influe grandement sur le chargement d'une obligation (100 hypothèses se chargent en 1 ou 2 secondes en force 0, et en 1 minute en force 1 !). La force conseillée pour la lecture d'une PO est la force 0, car le temps d'accès est toujours négligeable et les hypothèses sont quand même largement simplifiées (en particulier : les variables égales entres elles ont été éliminées). Néanmoins, on peut utiliser la force rapide si on veut visualiser les hypothèses brutes. La force mémorisée sur chaque PO pour la démonstration interactive est bien sûr celle que vous avez choisie lors de la dernière session interactive sur cette obligation. Si c'est la première fois que vous l'accédez, c'est la force 0. Une fois la PO chargée, la force est affichée dans la zone de la ligne de commandes :



Si vous vous souvenez avoir sauvé une force élevée pour l'obligation à accéder, et que vous ne voulez pas que l'accès soit trop long, vous pouvez utiliser le bouton *GotowithReset* qui remet en force 0 :



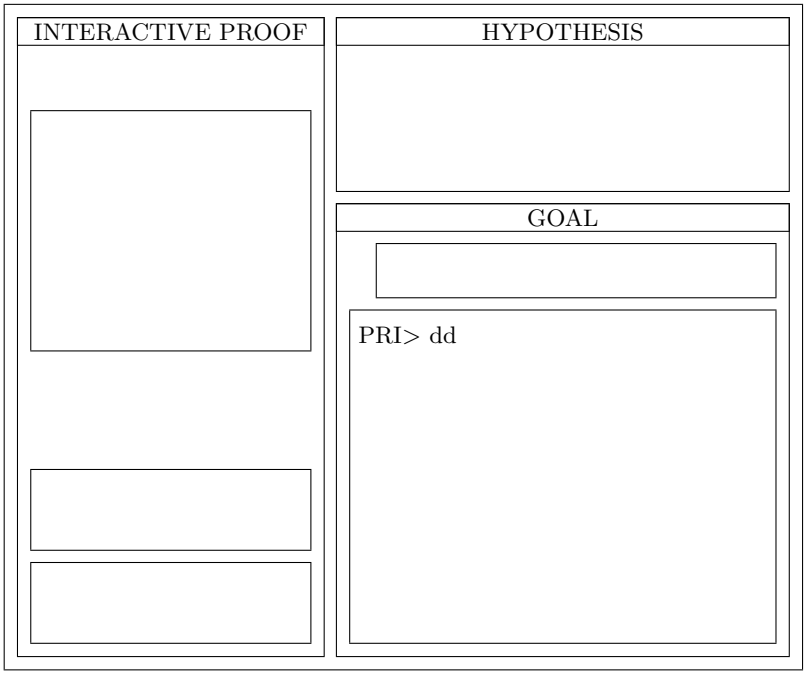
Maintenant, l'obligation de preuve est chargée ; les zones but, hypothèses et commandes sont renseignées. Le but apparaît avec les hypothèses locales, pour des raisons de logique de preuve. Voici un exemple d'affichage d'une obligation de preuve, dans lequel la seule hypothèse locale est un commentaire entre guillemets :



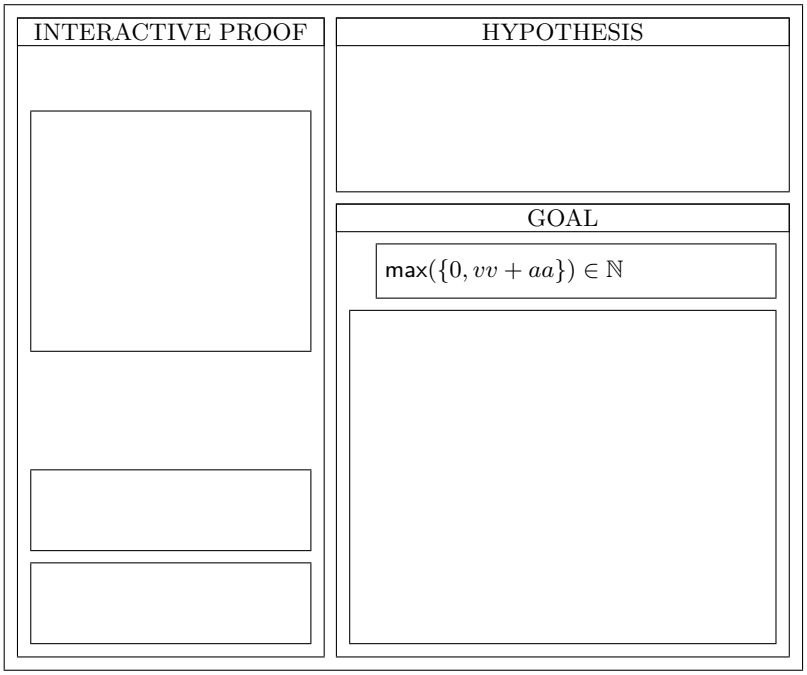
Notez l'hypothèse de localisation "basic_train.pilote.1". Nous sommes donc sur la première obligation de preuve de l'opération *pilote* du composant *basic_train*.

2. **Faites monter les hypothèses locales** : dans la présentation du prouveur interactif, les hypothèses locales apparaissent avec le but. Il faut les faire monter avec `dd` (*Deduction*) pour faciliter la lecture du but, ces hypothèses ne servant que pour la preuve formelle. Attention : ne pas sauver cette commande quand on quitte l'obligation de preuve. Le prouveur vous posera la question quand vous quitterez cette obligation, il suffira de répondre non.

   Pour faire monter les hypothèses locales, il suffit donc de taper `dd` dans la fenêtre de commande :



   Le but est maintenant isolé :

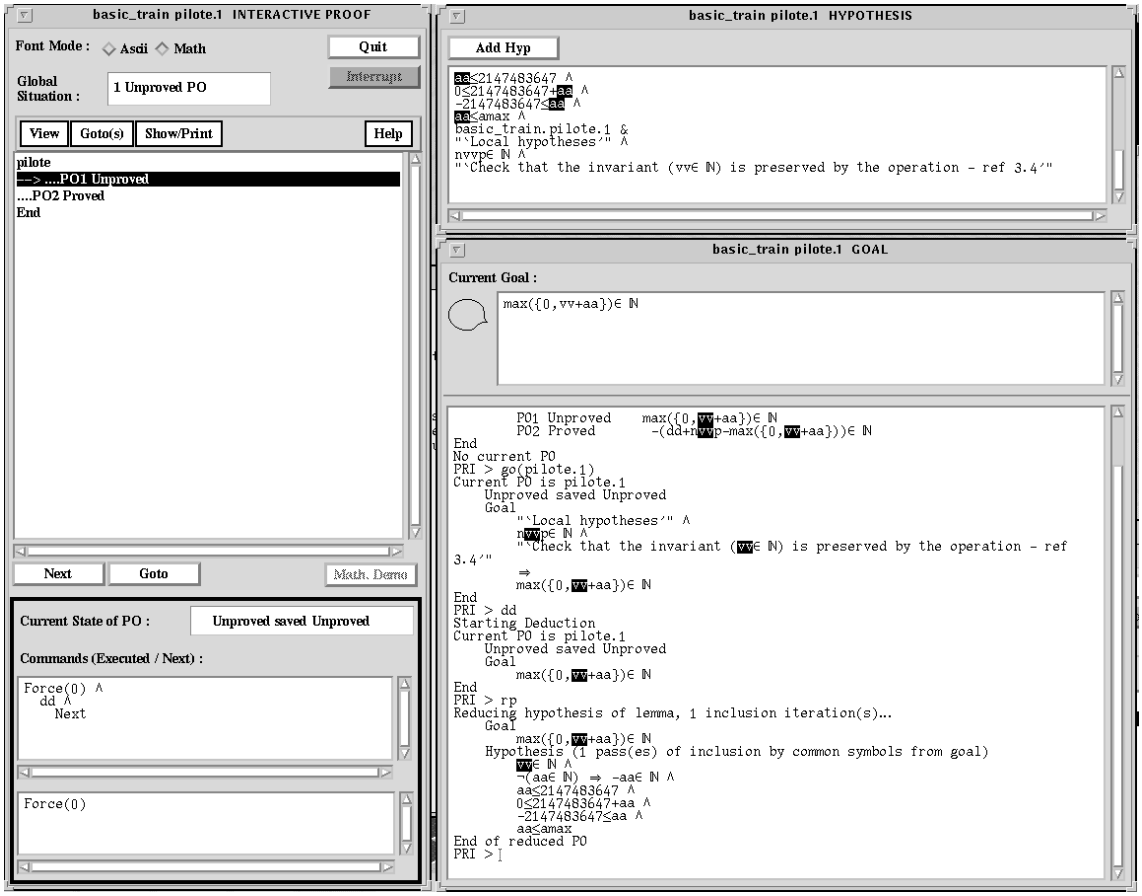


3. **Utiliser les fonctions de recherche du prouveur interactif** : Penser en particulier à :
   – la fonction `rp` (*ReducedPO*) qui permet de ne visualiser que les hypothèses ayant une variable en commun avec le but. Dans l'exemple précédent, nous devons démontrer que $\mathsf{max}(\{0, vv + aa\})$ est un naturel, ce qui est toujours vrai ; mais il est quand même souhaitable de contrôler la définition de aa et vv. En tapant `rp` dans la zone de commandes, nous obtenons :

```
PRI > rp
Reducing hypothesis of lemma, 1 inclusion iteration(s)...
  Goal
     max({0, vv + aa}) ∈ ℕ
  Hypothesis (1 pass(es) of inclusion by common symbols from goal)
     vv ∈ ℕ  ∧
     ¬(aa ∈ ℕ)  ⇒  −aa ∈ ℕ  ∧
     aa ≤ 2147483647  ∧
     0 ≤ 2147483647 + aa  ∧
      − 2147483647 ≤ aa  ∧
     aa ≤ amax
  End of reduced PO
PRI >
```

   Cette fois les deux premières hypothèses définissent vv comme un entier naturel et aa comme un entier relatif (cette dernière forme étant une normalisation du prouveur). Le lemme est donc juste.
   – la fonction `sh` (*SearchHypothesis*) qui permet de rechercher des hypothèses. Vous pouvez utiliser cette commande :
     – en mode simple : par exemple `sh(card(EE))` renvoie toutes les hypothèses qui contiennent $\mathsf{card}(EE)$. Dans notre exemple, nous pourrions rechercher les hypothèses qui concernent vv : `sh(vv)`, puis celles qui concernent aa : `sh(aa)`.
     – en mode multiple : par exemple `sh(aa _and vv)` pour les hypothèses qui contiennent à la fois `aa` et `vv` ;
     – avec des motifs : par exemple `sh(a+b)` pour les hypothèses qui contiennent des additions. Les variables à une lettre qui sont employées ici remplacent n'importe quelle formule, ce sont des *jokers*.
   – la fonction de sélection des fenêtres d'hypothèses et de commande du prouveur interactif : dans ces fenêtres, presser à la fois la touche de mise en majuscules de votre clavier et le bouton du milieu de la souris, puis choisir *Find All* dans le menu qui apparaît. Une zone de dialogue s'affiche, vous permettant de saisir une chaîne de caractères pour mettre en vidéo inverse toutes les occurrences de celle-ci. C'est très utile par exemple pour repérer rapidement toutes les occurrences d'une expression dans les hypothèses.

   Pour illustrer tout ceci, nous présentons ci-après l'écran de preuve correspondant à notre exemple de maximum entre 0 et $vv+aa$. De plus, nous avons utilisé la fonction de sélection précédemment décrite dans la fenêtre des hypothèses pour rechercher aa, et dans la fenêtre de commande pour rechercher vv.



5.3 Le parcours des obligations de preuve

Le bon choix de l'ordre de lecture des obligations de preuve sert à se diriger en premier vers celles qui ont le plus de chances d'être fausses. Leur correction précoce évite d'avoir à revérifier les obligations de preuves justes *à priori*, après modification du composant. Un mauvais choix de parcours des obligations de preuve conduit à s'apercevoir très tard des erreurs et oblige à relire plusieurs fois les mêmes obligations.

Les conseils essentiels sont :

1. **Commencer par les obligations de preuve qui semblent difficiles** : ce sont celles qui risquent le plus d'être fausses. Bien que cette démarche soit psychologiquement difficile car l'opérateur cherche souvent à se débarrasser d'un grand nombre de PO simples pour "avancer", elle nous paraît souhaitable. Pour trouver ces obligations difficiles :
   - **Parcourir les obligations de preuve d'une opération en remontant**, du plus grand numéro vers le plus petit. En effet, le générateur d'obligations de preuve engendre généralement les obligations de preuve les plus compliquées en dernier.
   - **Chercher les obligations de preuve qui concernent les parties com-**

pliquées de l'opération. En se guidant sur la structure du composant, on peut pressentir ce qui va poser problème.

L'ordre dans lequel les obligations de preuve apparaissent dans la liste dépend de la forme du composant. Néanmoins, les obligations de preuve concernant les préconditions des opérations appelées dans l'opération à prouver apparaissent généralement au début (même si ces appels sont à la fin de l'opération). D'autre part, dans une implantation, les obligations de preuve de non débordement des calculs intermédiaires sont au début (ces obligations servent à établir qu'aucun dépassement n'a lieu : par exemple pour xx := 2×v1-v2 il faut prouver 2×v1 ∈ INT et 2×v1-v2 ∈ INT).

2. **Faire plusieurs phases**. Il n'est pas forcément souhaitable de faire immédiatement l'examen complet de chaque obligation en les prenant une par une, souvent il vaut mieux faire plusieurs phases de mise au point. Particulièrement, lors de l'écriture des composants on fait souvent une mise au point réduite à l'interprétation des buts, complétée ensuite :
   - phase de développement : mise au point réduite à la seule interprétation des buts ;
   - phase finale : mise au point complète.

   Une autre méthode peut consister à mettre l'accent sur la recherche de la meilleure forme à donner aux expressions du composant pour faciliter la preuve, ce qui donne lieu à une phase séparée. En effet, la retouche des composants pour faciliter la preuve peut réduire énormément le nombre d'obligations de preuve non démontrées en force 0, et diminuer ainsi le coût global de la preuve. Par exemple, l'auteur connaît un cas où écrire $e_1 \in \{a, b\}$ plutôt que $\{a, b\} - \{e_2\} = \{e_1\}$ sachant que a et b sont des éléments de $\{e_1, e_2\}$ a fait passer le nombre d'obligations de preuves non démontrées en force 0 de 20 à 0. Le cycle avec retouche des composants est :
   - phase de développement : mise au point réduite à la seule interprétation des buts ;
   - phase de simplification : mise au point réduite à la recherche de la meilleure forme à donner aux expressions du composant pour faciliter la preuve, sans se concentrer sur la justesse des obligations de preuve ;
   - phase finale : mise au point complète.

L'interprétation des buts est l'une des cinq étapes de l'examen d'une obligation de preuve, que nous allons voir maintenant.

5.4 L'examen d'une obligation de preuve

Attention : avant de procéder à l'examen d'une obligation de preuve, assurez vous que vous avez choisi cette obligation de telle manière à commencer par celles qui ont le plus de chances de détecter des erreurs, comme expliqué au paragraphe précédent.

Dans la phase de mise au point, il faut obtenir le plus rapidement possible la démonstration intuitive de chaque obligation de preuve. Nous allons commencer par présenter les principes de l'examen d'une obligation de preuve ; puis nous verrons comment parcourir et comment visualiser ces obligations de preuve. La méthode d'examen d'une obligation de preuve est la suivante :

1. **interprétation du but** : examiner les différentes variables présentes dans le but

et retrouver le sens de chacune d'entre elles dans son interprétation physique (voir paragraphe 5.4.1) ;

2. **justification intuitive** : déterminer pour quelles raisons ce but doit être vrai dans le contexte du composant (voir paragraphe 5.4.2) ;
3. **sélection des hypothèses** : isoler dans l'obligation de preuve les hypothèses correspondantes à ces raisons (voir paragraphe 5.4.3) ;
4. **démonstration intuitive** : faire une démonstration intuitive de l'obligation de preuve réduite à ces hypothèses (voir paragraphe 5.4.4).
5. **notes et essais** : la démonstration intuitive faite peut donner des idées pour la démonstration formelle, sur les causes de l'échec de la démonstration automatique, etc. Dans cette dernière étape, on cherche à profiter de ces idées. Éventuellement, une démonstration formelle rapide sera recherchée et généralisée à d'autres obligations, permettant ainsi de réduire le nombre d'obligations à lire (voir paragraphe 5.4.5).

Cette liste décrit la méthode préconisée pour la phase de mise au point. Nous conseillons de suivre ces étapes pour chaque obligation de preuve, en s'aidant des paragraphes correspondants.

L'idée maîtresse de cette méthode en cinq étapes consiste à interpréter l'obligation de preuve dans le contexte du composant à prouver. On bénéficie ainsi de toute la démarche intellectuelle qui a été faite pour comprendre ou construire le composant, et qui deviendra à terme un ensemble de démonstrations rigoureuses.

Cette méthode nécessite la lecture du composant à prouver ainsi que celle des composants référencés. Nous conseillons donc de **garder en fenêtres icônifiées le fichier composant concerné et les composants associés**. Pour une machine abstraite, ce sont les machines vues ou incluses ; pour une implantation ce sont le raffinement supérieur et les machines importées [1]. Puisque la preuve automatique a pu décharger les obligations de preuve concernant des parties entières du composant, **il suffit de lire une partie restreinte du composant** : on se repérera au nom de l'opération (rappelée dans les barres de label des fenêtres du prouveur) et au but.

Dans les paragraphes qui suivent, nous allons évoquer rapidement l'existence de certaines commandes du prouveur interactif, sans faire une présentation générale de cet outil. En effet, pour la phase de mise au point il n'est pas forcément nécessaire de connaître l'utilisation complète de toutes les commandes interactives. Elles sont présentées en détail au chapitre 6 qui traite de la preuve formelle.

Les cinq étapes d'examen d'une obligation de preuve peuvent paraître longues quand elles sont exposées en détail comme dans ce qui suit. En fait avec de l'habitude chaque obligation se fait très rapidement, en une ou quelques minutes, sans séparer formellement chaque étape. Ces étapes servent seulement à attaquer les problèmes *dans le bon ordre*. Si une obligation de preuve semble fausse, vous avez probablement trouvé une erreur dans le source : suivez les conseils du paragraphe 5.8 pour trouver et corriger l'erreur.

[1] Notons que même quand le prouveur interactif est lancé, il est possible d'ouvrir les composants depuis l'Atelier B : il suffit d'ouvrir la fenêtre principale qui est en icône et de double-cliquer sur le composant.

5.4.1 Conseils pour l'interprétation des buts

La méthode générale pour interpréter un but est **d'isoler la partie concernée du composant**, généralement une seule ligne, et de voir **quelle contrainte on cherche à vérifier**. Il suffit généralement de se concentrer sur ces deux éléments pour comprendre la provenance d'un but.

Parfois, le but tient sur plusieurs lignes et sa lecture directe n'est pas souhaitable. Il faut alors utiliser les fonctionnalités du prouveur interactif, cela est expliqué au paragraphe 5.7.

Lors de la lecture d'un but, il y a un certain nombre de choses à savoir pour une compréhension facile :

- interprétation des variables avec $: confère le manuel d'interprétation des obligations de preuve. Sans paraphraser ce document, nous rappelons quelques repères simples et approximatifs :
  - variables vv : si on prouve un raffinement ou une implantation, variables du niveau plus abstrait.
  - variables vv$0 : dans le cas d'une variable modifiée plusieurs fois (boucle ou séquence), valeur initiale.
  - variables vv$1 : variables du raffinement ou de l'implantation qu'on est en train de prouver, ou d'une machine importée.
  - variables vv$2 : variables dans le corps d'une boucle.
  - variables vv$7777 : variables après toutes les modifications de l'opération (par exemple : après une boucle)
- opérations importées : pour générer les obligations de preuve d'une implantation le générateur de PO expanse textuellement les opérations importées. Donc il faut s'attendre à trouver dans ces obligations de preuve le code des opérations importées. Par exemple, si une opération spécifiée par `ANY xx WHERE ...` est utilisée, la variable intermédiaire `xx` peut apparaître dans les obligations de preuve de l'implantation utilisatrice. La remarque précédente est vraie à tous les niveaux de raffinement pour des opérations incluses.
- positionnement dans des cas : quand la spécification ou le raffinement contiennent des cas, le générateur de PO sépare les différents cas dans différentes obligations de preuve. Il est alors nécessaire de regarder les hypothèses locales, elles spécifient dans quels cas on se place et permettent de retrouver précisément la ligne concernée. Voici des exemples qui provoquent des séparations par cas :
  - ANY ... WHERE $P_1 \Rightarrow Q_1 \wedge P_2 \Rightarrow Q_2$ ... dans une spécification.
  - SELECT P_1 THEN S_1 WHEN P_2 THEN S_2 END dans une spécification.
  - IF ... THEN ... ELSE .. END dans une implantation.
  - CASE dans une implantation.
- élimination de variables : le générateur d'obligations de preuve élimine de lui-même les variables intermédiaires inutiles. Par exemple, pour VAR *vv* IN *vv* $\longleftarrow$ *op* si la spécification de *op* est *rr* $\longleftarrow$ *op* = *rr* :=*var*, c'est *rr* et non pas *vv* qui apparaît dans les obligations de preuve. Ceci peut surprendre car ce n'est pas la donnée manipulée dans le composant à prouver qui apparaît.
- ordre des obligations de preuve : Les obligations de preuve concernant les préconditions des opérations appelées dans la clause à prouver apparaissent généralement au début

(même si ces appels sont à la fin de la clause). Dans une implantation, les obligations de preuve de non débordement des calculs intermédiaires sont au tout début.
– commentaire du but : l'hypothèse qui apparaît en dernier dans la fenêtre des hypothèses après que vous ayez tapé `dd` est un commentaire qui explique la provenance du but. Ce commentaire contient une phrase explicative et une référence au manuel des obligations de preuve.

Exemple : supposons que le but de l'obligation de preuve 30 de l'opération *Commandes-Pompes* du composant *CmdPmp_1* soit :

$$(1\,..\,NB\_PUMP) \times \{FALSE\} \lhd$$
$$(indice\_l\$7777 + 1\,..\,NB\_PUMP \lhd \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)) =$$
$$(1\,..\,NB\_PUMP) \times \{FALSE\} \lhd \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)$$

Les quatre variables utilisées dans ce but sont NB_PUMP, l_wpok, $indice\_l\$7777$ et $on\$1$. On retrouve d'abord le sens de chacune des variables : NB_PUMP est le nombre de pompes à commander, l_wpok est un tableau qui indique pour chaque pompe si elle est OK ou non, $indice\_l\$7777$ est un indice de parcours de chaque pompe et $on\$1$ représente l'état d'un appareil de mesure. Nous consultons le composant, l'opération contient une boucle dont l'indice est $indice\_l$. La présence de \$7777 indique que cette obligation concerne l'état après la sortie de la boucle. Nous sommes donc en train de vérifier que cette boucle a bien construit ce qui était prévu dans la spécification. Le commentaire du but est :

Check that the invariant (pumpon = pumpon\$1) is preserved by the operation

Dans la spécification de cette opération, nous avons effectivement écrit :

$$pumpon := (1\,..\,NB\_PUMP) \times \{FALSE\} \lhd \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)$$

Ce qui indique que cette variable doit être construite à partir du tableau où toutes les pompes sont à FALSE, en ajoutant si $on = \mathsf{TRUE}$ le tableau l_wpok. L'interprétation du but est finie.

L'interprétation correcte du but nécessite la connaissance du composant et de ses niveaux adjacents. Nous conseillons de garder ces fichiers ouverts en icônes dans un coin de l'écran pendant les phases de preuve (Attention, ne pas les laisser affichés : les fenêtres de preuve doivent rester toutes visibles en juxtaposition). Il vaut souvent mieux faire une relecture de ces composants au début de la session de preuve. Le sens de beaucoup de PO apparaît alors clair sans même avoir à regarder à nouveau les fichiers.

Notons que lorsque le prouveur interactif est lancé, il est possible d'ouvrir les composants depuis l'Atelier B : il suffit d'ouvrir la fenêtre principale qui s'est icônifiée et de cliquer deux fois sur le composant.

Une fois que le sens du but est compris, nous pouvons chercher pourquoi il doit être vrai : c'est la justification intuitive.

5.4.2 Conseils pour la justification intuitive

- **Utiliser le sens physique** : raisonner à partir de ce que représentent physiquement les quantités et les expressions manipulées : par exemple $vv + (aa * tt)$ correspond à une nouvelle vitesse, $RESSOURCES - \{x_0\}$ est le nouvel ensemble de ressources disponibles... Ce n'est pas la preuve mathématique qui est recherchée à cette étape, mais une bonne interprétation du modèle abstrait.
- **Utiliser le raisonnement par cas et par contradiction** : très souvent on ne pense pas à ces deux artifices dans des raisonnements naturels. Ils s'appliquent généralement bien quand la justification directe n'aboutit pas.
- **Noter la justification intuitive** : Elle servira pour la démonstration intuitive (Noter très rapidement, au brouillon).

Exemple : reprenons notre précédent exemple. Pourquoi à la fin de la boucle, avons nous construit la valeur de *pumpon* prévue dans la spécification ? Simplement parce que chaque pompe a été contrôlée (utilisation du sens physique). C'est-à-dire que *indice_l*\$7777 qui représente la valeur de l'indice de boucle quand elle se termine, doit être telle que toutes les pompes sont contrôlées. En regardant l'opération, on voit que cet indice parcourt les numéros de pompes en décroissant, il est nul quand la boucle est finie. Donc l'intervalle $\$7777 + 1 \,..\, NB\_PUMP$ est égal à $1 \,..\, NB\_PUMP$, cet intervalle sert à restreindre la partie droite de l'expression qui est manifestement une fonction de $1 \,..\, NB\_PUMP$ dans BOOL. Cette restriction peut donc être enlevée, et le but devient une égalité littérale. Nous notons : remplacer l'indice par sa valeur pour éliminer la restriction dans le terme de gauche. Ici nous n'avons pas eu besoin d'un raisonnement par contradiction.

La phase de justification intuitive de l'obligation de preuve est peut être la plus importante : c'est à ce moment que l'on perçoit toutes les conséquences de la modélisation B choisie. Maintenant nous comprenons pourquoi l'obligation de preuve doit être vraie dans le contexte du composant. Ce contexte doit se retrouver dans les hypothèses, nous allons donc les examiner.

5.4.3 Conseils pour la sélection des hypothèses

Le point le plus important est le suivant :

- **lire les hypothèses en remontant**. Les hypothèses les plus significatives pour le but sont généralement dans les dix dernières. Les hypothèses de contexte du début de la liste sont généralement des propriétés de constantes n'ayant souvent rien à voir avec le but !

D'autre part, les conseils concernant la lecture d'un but s'appliquent aussi pour la lecture des hypothèses. Puisque la liste des hypothèses est souvent très longue, il faut **savoir ce que l'on cherche** avant toute lecture, c'est par la justification intuitive de l'obligation de preuve que la recherche des hypothèses significatives est dirigée. Si la recherche a lieu dans le prouveur interactif, utiliser au maximum les fonctions de recherche qui ont été décrites dans le paragraphe 5.2.1.

Lorsque les hypothèses significatives sont localisées, il faut **noter** les informations nécessaires pour les retrouver. Cela sera utile pour la phase de démonstration intuitive.

Exemple : continuons toujours l'exemple précédent. Nous cherchons :
– l'hypothèse indiquant que l'indice est nul ;
– l'hypothèse indiquant le type de *l_wpok*.
Nous pouvons essayer la commande *ReducePo* (`rp`). Si peu d'hypothèses sont sélectionnées, nous aurons trouvé celles que nous cherchons car elles contiennent au moins le nom de la variable concernée en commun avec le but. Effectivement, la commande `rp` retourne :

$$
\begin{array}{|l}
\hline
NB\_PUMP = 4 \land \\
l\_wpok \in 1 \mathinner{..} 4 \ \pfun\ \mathsf{BOOL} \land \\
\mathsf{dom}(l\_wpok) = 1 \mathinner{..} 4 \land \\
on\$1 \in \mathsf{BOOL} \land \\
on\$1 = \mathsf{TRUE} \ \Rightarrow\ off\$1 = \mathsf{FALSE} \land \\
on = on\$1 \land \\
indice\_l\$7777 = 0 \land \\
indice\_l\$7777 \in 0 \mathinner{..} NB\_PUMP \\
\hline
\end{array}
$$

Nous avons trouvé toutes les hypothèses utiles. Nous notons que les hypothèses utiles se trouvent par une simple commande `rp`.

5.4.4 Conseils pour la démonstration intuitive

Le but de la démonstration intuitive est d'éviter de croire à tort que certaines obligations de preuve sont justes, suite à des confusions facilement faites dans une justification intuitive. La méthode générale est la suivante :

– **transposer la justification intuitive** en utilisant les hypothèses sélectionnées
– **évoquer les règles utilisées** : il ne s'agit pas de se ramener à des règles existantes dans la base de règles ou dans le B-Book, mais seulement de voir quelles sont les règles qui sont implicitement utilisées dans la justification intuitive.
– **examiner ces règles** : car c'est au moment où la transformation utilisée a été sortie de son contexte en tant que règle générale que l'on voit toutes les conditions pour qu'elle soit valide.

Exemple : reprenons toujours l'exemple précédent. Les étapes de notre justification étaient :
– remplacer l'indice par sa valeur : la règle utilisée est la définition de l'égalité, sans problèmes.
– simplifier $0 + 1 \mathinner{..} NB\_PUMP$ en $1 \mathinner{..} NB\_PUMP$: règles de calcul entier.
– éliminer la restriction à $1 \mathinner{..} NB\_PUMP$: nous avons une expression de la forme

$$1 \mathinner{..} NB\_PUMP \lhd f$$

f étant défini par

$$f = \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)$$

la règle à appliquer est clairement :

$$f \in A \ \pfun\ B \ \Rightarrow\ A \lhd f = f$$

mais comment montrer que f est une fonction partant de $1 \,..\, NB\_PUMP$? Il est nécessaire de faire deux cas : $on\$1 = \mathsf{TRUE}$ ou FALSE.
  - si $on\$1 = \mathsf{FALSE}$: f est vide, l'élimination de la restriction est immédiate.
  - si $on\$1 = \mathsf{TRUE}$: $f = l\_wpok \rhd \{\mathsf{TRUE}\}$. Le raisonnement est un peu plus complexe, il fera intervenir des règles de typage d'une fonction co-restreinte. Dans une démonstration intuitive, il n'est pas nécessaire de tout développer, il suffit de faire apparaître les règles essentielles utilisées.
- une fois la restriction éliminée, nous avons deux termes littéralement égaux (règle employée : définition de l'égalité).

Au cours de ce processus, il faut faire attention aux pièges classiques que nous énumérons ci-après.

Pièges classiques

- **priorité $\Rightarrow$ et $\wedge$** : l'implication est moins prioritaire que la conjonction. Par exemple :
  $A \wedge$
  $B \Rightarrow C \wedge$
  D
  sera compris comme $(A \wedge B) \Rightarrow (C \wedge D)$. Pour éviter une mauvaise interprétation, mettre des parenthèses, par exemple, écrire :
  $A \wedge$
  $(B \Rightarrow C) \wedge$
  D
- **intervalles vides** : un intervalle peut être vide si sa borne de gauche dépasse celle de droite. Par exemple, $aa \in aa..bb$ n'est pas toujours vrai.
- **ajout de divergences à une fonction** : si *ff* est une fonction, l'objet obtenu en ajoutant des couples à *ff* n'est plus forcément une fonction. En effet l'un des couples ajoutés peut avoir le même élément de départ qu'un couple existant de *ff*; dans ce cas *ff* est transformée en une relation.
- **division entière** : en B, le symbole / désigne la division entière. Les règles de calcul sont beaucoup plus réduites que sur la division réelle. Par exemple, on n'a pas $(a + b)/c = a/c + b/c$ (contre exemple $a = b = 1$, $c = 2$). Ou encore, on n'a pas $0 \leq a/2 \Rightarrow 0 \leq a$ (contre exemple avec $a = -1$).
- **confusions de niveaux dans les fonctions de fonctions** : les fonctions de fonctions donnent souvent des prédicats plus compliqués que prévu. Par exemple, soit
  $f_0 \in \text{INDEX} \rightarrow (\text{NAT} \nrightarrow \text{NAT})$
  que l'on veut raffiner par
  $f_1 \in \text{INDEX} \rightarrow (\text{NAT} \rightarrow \text{NAT})$
  L'invariant de liaison n'est pas $f_0 = \mathsf{dom}(f_0) \lhd f_1$, mais

$$\forall x.(x \in \text{INDEX} \Rightarrow f_0(x) = \mathsf{dom}(f_0(x)) \lhd f_1(x))$$

- **extensions avec des variables** : Si les éléments d'un ensemble en extension ne sont pas des expressions littérales, il se peut que ces éléments soient égaux entre eux. L'oubli de ce fait conduit à des erreurs : par exemple, $\{xx, yy\} - \{zz\}$ n'est pas égal à $\{xx, yy\}$ (en particulier, si $zz = yy = xx$ c'est l'ensemble vide).

5.4.5 Notes et essais

Au cours des quatre étapes précédentes, l'opérateur imagine des simplifications sur les expressions qu'il examine et des débuts de démonstrations interactives. Avant de quitter une obligation de preuve, il est donc conseillé de :

- **Remettre en cause la forme des expressions**, en particulier lancer une fois le prouveur en force 0 pour voir quel est le premier but en échec. Cela aide à trouver des expressions simples qui aident le prouveur. Suivre les conseils du paragraphe 5.5.
- **Tenter éventuellement une démonstration rapide**. En cas de succès, l'obligation est éliminée et sa démonstration peut se généraliser, éliminant ainsi d'autres obligations. Attention : il ne faut surtout pas démarrer une phase de preuve formelle ! Suivre les conseils du paragraphe 5.6.

Dans l'exemple que nous avons traité à chaque étape, nous pouvons faire certaines remarques sur la forme de l'expression qui apparaît dans le but :

$$(1\,..\,NB\_PUMP) \times \{FALSE\} \lhd \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)$$

Il serait peut-être plus naturel de faire apparaître explicitement les deux cas suivant la valeur de $on\$1$. Nous aurions alors deux expressions plus simples :

$$(1\,..\,NB\_PUMP) \times \{FALSE\} \lhd (l\_wpok \rhd \{\mathsf{TRUE}\})$$

et

$$(1\,..\,NB\_PUMP) \times \{FALSE\}$$

En fait la première expression se simplifie puisque l_wpok est une fonction totale de $1\,..\,NB\_PUMP$ dans BOOL. Elle devient l_wpok tout seul !

Faut-il reporter ces simplifications : ce n'est pas évident. Mais si nous décidions de faire une phase de simplification des expressions, ces idées trouveraient sûrement leur emploi, il faut donc les noter.

Nous pouvons également tenter une démonstration rapide de cette obligation. L'examen de cette PO nous incite à faire deux cas suivant la valeur de $on\$1$. En fait, en démarrant cette preuve par cas avec une commande *DoCases* on aboutit directement. Avec de l'habitude, il est possible de pressentir cette démonstration interactive très vite. Malheureusement, elle ne se généralise pas aux autres obligations non démontrées du composant.

5.4.6 Admettre des obligations de preuve

Il est possible d'utiliser une règle d'admission (voir cette notion en 6.6.2) pour admettre des obligations de preuve. Cette méthode consiste à faire agir une règle manuelle démontrant artificiellement les obligations de preuve que l'on ne veut pas voir. Elle a deux avantages :

- les obligations de preuve ainsi éliminées sont réellement marquées comme prouvées, la commande *Next* par exemple, ne les atteint plus.
- les forces supérieures du prouveur pourront être lancées sur les obligations de preuve non écartées, sans perdre de temps sur celles qui sont admises.

Par contre, l'application de la règle d'admission est mémorisée comme démonstration manuelle des obligations de preuve concernées, ce qui peut être gênant en cas de modification du composant. En effet les obligations de preuve peuvent alors avoir changé, et l'application de la règle d'admission fait croire à tort qu'elles sont justes. L'utilisation d'une règle d'admission nécessite une connaissance suffisante du prouveur interactif, nous nous contentons ici de voir à quel moment cette méthode s'applique en phase de mise au point. Pour apprendre à utiliser les règles d'admission, lire le chapitre 6.

5.5 La simplification des expressions des composants B

Certains composants produisent trop d'obligations de preuve, ou bien produisent des obligations de preuve très compliquées. Il arrive même que le nombre d'obligations de preuve qu'il faudrait générer pour un composant soit tellement grand que le générateur sature sans aboutir. Dans tous ces cas c'est sur les composants eux-mêmes qu'il faut agir pour choisir la forme des expressions qui facilite la preuve.

D'autre part, simplifier les expressions des spécifications formelles permet souvent de mieux appréhender le problème, car le logiciel envisagé est alors mieux modélisé, dans une forme *qui facilite le raisonnement* comme elle facilite la preuve. L'esprit de la méthode B est de construire le logiciel comme la preuve qu'il satisfait le besoin considéré (lire la préface du B-Book).

Il doit être clair que **tout projet juste n'est pas forcément démontrable**. La preuve est une vérification de très haut niveau qui ne peut se faire que si certains impératifs de découpage sont respectés (*Divide and Conquer*). Un exemple classique de problème de cet ordre est l'examen des obligations de preuve concernant des opérations placées en séquence après un IF : ces PO sont dédoublées n fois, n étant le nombre de branches du IF.

La transformation des composants pour revenir à des expressions facilitant la preuve est un sujet méthodologique complexe, que nous n'aborderons que très partiellement ici. Nous allons simplement donner un certain nombre de "recettes".

5.5.1 Redécouper les composants

Les obligations de preuve trop nombreuses ou trop complexes traduisent souvent un mauvais découpage des composants. Les séquences d'instructions sont alors trop longues (voir ci-dessus), il y a des boucles imbriquées, etc. Le re-découpage est la première méthode à envisager pour diminuer la difficulté de la preuve. Penser en particulier à :

– Créer des machines importées pour séparer les parties complexes d'une implantation dans des opérations importées.
– Insérer des raffinements si la preuve d'une implantation est trop complexe.

5.5.2 Tenir compte des normalisations du prouveur

Le prouveur effectue un certain nombre de normalisations, d'une manière parfois partielle pour des problèmes de performance. En phase de mise au point, il peut être utile de changer les expressions du composant pour retrouver les normalisations du prouveur, ce qui peut influer de façon très importante sur ses performances. La méthode est la suivante :

1. **interroger le prouveur** : appliquer le prouveur sur l'obligation de preuve non démontrée, en force 0, pour examiner le but et les hypothèses en échec et voir sous quelle forme il a mis les expressions du composant (attention : faire `pr` sans faire `dd` avant, sinon les hypothèses locales montent non normalisées) ;
2. **normaliser** : mettre les expressions du composant dans les formes équivalentes données par le prouveur ;
3. **tester** : repasser le prouveur en force 0 (voir paragraphe 4.2) et constater le gain ou la perte.

En plus de ce procédé assez expérimental, nous allons donner ci-après un certain nombre de précautions à prendre systématiquement. Les normalisations du prouveur sont expliquées au chapitre "Normalisation" du manuel de référence du prouveur.

5.5.3 Rechercher les égalités littérales

Le prouveur étant mécanique, son principe de base est de faire coïncider des formules entre elles. Pour cette raison, il faut éviter de changer gratuitement la forme des expressions du composant. Par exemple, si dans l'invariant on a :

$$\{a \mapsto b\} \in f \Rightarrow P$$

Et que dans une précondition, on veut se placer dans le cas où $\{a \mapsto b\} \in f$, il ne faut surtout pas écrire :

PRE $a \in dom(f \rhd \{b\})$

Mais plutôt :

PRE $\{a \mapsto b\} \in f$

L'exemple ci-dessus est volontairement exagéré. Ce genre de problème peut néanmoins se produire lors de l'écriture d'un composant, surtout si ce dernier a subi beaucoup de modifications successives. Il faut aussi éviter de passer par des constantes intermédiaires qui vont obliger le prouveur à faire des remplacements. Utiliser plutôt la clause DÉFINITIONS qui fait un remplacement littéral. Par exemple, ne pas écrire :

CONSTANTS card_ens **PROPERTIES** card_ens = card(ens)

Mais plutôt :

DEFINITIONS card_ens == card(ens)

5.5.4 Rechercher les formes canoniques des expressions arithmétiques

Toutes les forces du prouveur utilisent le *Solveur arithmétique* qui met les expressions arithmétiques dans une forme canonique, avec un ordre constant des variables entre elles. Ainsi les chances de succès par correspondance littérale sont augmentées.

En force 0, pour des raisons d'efficacité seuls les prédicats entièrement arithmétiques ($a = b$ ou $a \leq b$) sont transformés par le *Solveur arithmétique*. Il se peut donc que la forme brute apparaisse dans l'obligation de preuve mélangée avec la forme canonique, produisant des asymétries gênantes. Par exemple :

$$jj + 1 \leq ii \;\Rightarrow\; ii..jj + 1 = \varnothing$$

sera vu par le prouveur comme :

$$1 + jj \leq ii \;\Rightarrow\; ii..jj + 1 = \varnothing$$

La normalisation du prouveur a provoqué l'inversion des termes jj et 1. La règle simple $i..j = \varnothing \Leftrightarrow j + 1 \leq i$ ne peut dons pas s'appliquer directement.

Il n'est pas possible de mettre toujours les expressions arithmétiques directement dans la forme canonique. Nous nous contenterons de conseiller de **mettre les constantes numériques à gauche** et de **classer les variables dans l'ordre d'apparition du composant**, c'est-à-dire l'ordre dans lequel ces variables apparaissent dans l'invariant.

5.6 La preuve rapide

Nous désignons par le terme *preuve rapide* les démonstrations interactives trouvées en quelques minutes au cours de la phase de mise au point. Il ne peut s'agir que de démonstrations très simples, mais elles évitent d'avoir à revenir sur l'obligation de preuve concernée et leur simplicité leur permet souvent de se généraliser à d'autres obligations. Nous allons décrire ceci en deux étapes : trouver une démonstration rapide et la généraliser.

5.6.1 Trouver une démonstration rapide

- **Se donner un délai limite** : il faut éviter de s'égarer dans une véritable preuve formelle, on se donnera donc une limite de temps variant entre 30 secondes et deux minutes suivant la complexité de l'obligation.
- **Essayer le prouveur de prédicats** : si l'obligation semble pouvoir se démontrer sans notions arithmétiques, vous pouvez essayer la commande *PredicateProver* (pp) qui lance

le prouveur de prédicats avec un temps limite de 60 secondes. Généralement, il vaut mieux faire cette commande après une déduction (`dd`) pour faire monter les hypothèses locales. Quelques variantes utiles :
  - `pp(30)` pour réduire le délai d'expiration à 30 secondes ;
  - `pp(rp.1)` ou `pp(rp.2)` qui permet d'appliquer le prouveur de prédicats sur l'obligation réduite avec une ou deux passes d'inclusion d'hypothèses à partir du but. En effet si vous avez plus d'une centaine d'hypothèses, la commande `pp` sera longue : le seul temps de traduction en prédicats quantifiés est déjà important.
- **Se limiter à 5 commandes** : si le prouveur de prédicats n'a pas déchargé l'obligation, tentez une démonstration à partir des commandes standard du prouveur interactif. Nous n'allons pas expliquer comment faire ici, c'est l'objet du chapitre 6. Nous dirons simplement que les deux commandes les plus fréquemment vues dans les démonstrations rapides sont *AddHypothesis* (`ah`) et *DoCases* (`dc`).

Si vous n'avez jamais utilisé le prouveur interactif, et si vous n'avez pas encore étudié le chapitre 6 sur la preuve formelle, vous ne ferez pas de démonstrations rapides lors de la mise au point de votre premier composant. Dans ce cas il peut être intéressant de faire la phase de preuve formelle juste après la mise au point de votre premier composant, dans un but didactique, même si vous aviez prévu une mise au point globale du projet entier. En tous cas ne commencez jamais une phase de preuve formelle avant d'avoir fait au moins la mise au point d'un composant.

5.6.2 Généraliser une démonstration

La généralisation d'une démonstration se fait par la commande *TryEverywhere* (`te`). Nous la décrivons ici car ce sont souvent les démonstrations rapides qui se généralisent.

- **Sauvez votre démonstration** avant de la généraliser par la commande *SaveWithoutquestion* (`sw`). Sinon vous risquez de la perdre lors de la généralisation.
- Utilisation de *TryEverywhere* :
  - `te(Op.30, Replace.Gen.Unproved)` : essayer d'appliquer la démonstration de l'obligation numéro 30 de l'opération *Op* à toutes les PO non prouvées du composant. Rappelons que le nom de l'opération et le numéro de l'obligation courante sont visibles dans les barres de titre des trois fenêtres du prouveur, et dans tous les messages de réponse de la zone de commandes.
  - `te((dd & pp), Replace.Gen.Unproved)` : essayer la démonstration consistant à faire une déduction et à lancer le prouveur de prédicats sur toutes les PO non prouvées du composant.

Pour plus d'informations sur la commande *TryEverywhere*, voir le manuel de référence du prouveur.

5.7 Les expressions complexes

Dans certains cas le but tient sur plusieurs lignes et sa lecture directe n'est pas possible. Il faut alors commencer sa démonstration avec le prouveur interactif pour le décomposer et le

rendre compréhensible. La présence d'un but complexe indique souvent qu'il faut changer la forme des expressions du composant pour rendre la preuve plus simple, en tous cas il faut quand même analyser le but avant de faire des modifications. Les buts complexes appartiennent généralement à l'une des trois catégories suivantes :
– Les buts existentiels particularisables : le but est de la forme $\exists x.P$ mais on peut trouver une valeur simple de x qui convient ;
– Les buts existentiels abstraits : la valeur de x qui convient est unique et complexe, généralement une λ-fonction ;
– Les buts non découpés : ce sont des buts qui mettent en échec l'algorithme de découpage du générateur d'obligations de preuve.

Nous allons étudier ces trois catégories. Dans les exemples qui illustrent la suite, nous n'utiliserons plus la *Deduction* (`dd`) pour faire monter les hypothèses locales mais la commande `pr(Red)` qui lance le cœur de preuve en mode réduit. En preuve interactive, on utilise `pr(Red)` dès qu'il faut faire avancer une démonstration sans la terminer ou charger des hypothèses, pour des raisons que nous verrons longuement dans le chapitre 6.

5.7.1 Les buts existentiels particularisables

Montrons cette catégorie sur des exemples :

Exemple 1 : une valeur triviale à trouver. Soient les composants suivants :

MACHINE
ComplexGoal1
ABSTRACT_CONSTANTS
cc
PROPERTIES
cc ∈ 1..3 ⇸ NAT ∧
∀xx.(xx ∈ ran(cc) ⇒ xx mod 2 = 0) ∧
∀xx.(xx ∈ ran(cc) ⇒ xx mod 3 = 0) ∧
∀xx.(xx ∈ ran(cc) ⇒ xx mod 5 = 0) ∧
∀xx.(xx ∈ ran(cc) ⇒ xx mod 7 = 0)
END

IMPLEMENTATION
ComplexGoal1_1
REFINES
ComplexGoal1
END

Ce composant n'a artificiellement aucune opération, mais il représente le cas réel où une constante abstraite est utilisée pour spécifier le logiciel sans servir à l'implantation. Nous avons une obligation de preuve pour l'implantation, dont le but est :

$\exists cc.(cc \in 1\,..\,3 \;\pfun\; \mathsf{NAT} \;\wedge\; \forall xx.(xx \in \mathsf{ran}(cc) \;\Rightarrow\; xx \mathsf{mod} 2 = 0) \;\wedge$
$\forall xx.(xx \in \mathsf{ran}(cc) \;\Rightarrow\; xx \mathsf{mod} 3 = 0) \;\wedge\; \forall xx.(xx \in \mathsf{ran}(cc) \;\Rightarrow\; xx \mathsf{mod} 5 = 0) \;\wedge$
$\forall xx.(xx \in \mathsf{ran}(cc) \;\Rightarrow\; xx \mathsf{mod} 7 = 0))$

Nous voyons qu'il faut simplement prouver la validité de notre spécification en montrant qu'il existe bien une constante qui vérifie les propriétés annoncées. Ici il suffit de prendre $cc = \varnothing$. Nous utilisons la commande *SuggestforExists* (`se`), qui permet de suggérer de tenter la preuve avec une valeur particulière :

`pr(Red)` pour charger les hypothèses locales

`se({})`	suggestion : prendre $cc = \varnothing$
`pr`	relancer le cœur de preuve

Et la preuve aboutit directement. Si ce n'était pas le cas, il faudrait examiner les buts produits après la commande *SuggestforExists*, en utilisant éventuellement la technique du paragraphe 5.7.3. Ce type de but complexe n'indique pas que le composant soit à modifier.

Exemple 2 : forme trop indétermiste. Soient les composants suivants :

MACHINE
ComplexGoal2
OPERATIONS
xx ⟵ op = ANY ee WHERE
ee ⊆ NAT ∧
∀uu.(uu ∈ ee ⇒ uu mod 3 = 0)
THEN
xx :∈ ee
END
END

IMPLEMENTATION
ComplexGoal2_1
REFINES
ComplexGoal2
OPERATIONS
xx ⟵ op = xx := 9
END

La spécification ci-dessus signifie simplement que l'opération doit retourner un résultat divisible par 3, mais elle est exprimée sous une forme bien trop indéterministe. Nous obtenons le but suivant :

$\exists ee.(ee \subseteq \mathsf{NAT} \ \wedge$
$\forall uu.(uu \in ee \ \Rightarrow \ uu \mathsf{mod} 3 = 0) \ \wedge \ \exists (xx\$0).(xx\$0 \in ee \ \wedge \ 9 = xx\$0))$

Pour pouvoir lire le but plus simplement, nous demandons au prouveur de commencer la preuve en mode réduit : `pr(Red)`. Le but devient :

$\exists ee.(ee \subseteq \mathsf{NAT} \ \wedge \ \forall uu.(uu \in ee \ \Rightarrow \ uu \mathsf{mod} 3 = 0) \ \wedge \ 9 \in ee)$

Ce qui revient à prouver que 9 est dans NAT et que 9 est divisible par 3 ! En présence de ce type d'indéterminisme gratuit, il faut changer la spécification. Nous remplaçons la spécification de l'opération par :

$xx : (xx \in \mathsf{NAT} \ \wedge \ xx \mathsf{mod} 3 = 0)$

Cette fois-ci, tout se démontre en force 0 sans intervention manuelle.

Conclusions : Dans le cas où un but existentiel peut se résoudre par une valeur triviale ou une valeur particulière issue de la spécification, il faut mesurer si l'indéterminisme qui produit cet existentiel est bien nécessaire. Si c'est le cas, il faut démontrer l'obligation de preuve en utilisant la commande *SuggestforExists*, sinon il faut éliminer l'indéterminisme inutile.

5.7.2 Les buts existentiels abstraits

Ce sont les buts existentiels qui apparaissent par des constantes abstraites mathématiques. Supposons par exemple que vous avez besoin de la notion de factorielle dans une spécification. Vous pouvez décrire cette notion comme une constante abstraite :

ABSTRACT_CONSTANTS
fact
PROPERTIES
$\text{fact} \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$\forall nn.(nn \in \mathbb{N} \Rightarrow \text{fact}(nn+1) = (nn+1) \times \text{fact}(nn)) \wedge$
$\text{fact}(0) = 1$

Comme cette constante ne sera naturellement jamais implantée, il faudra prouver qu'elle existe. L'obligation concernée va apparaître assez tard dans le cycle de développement, lors de l'écriture de l'implantation du composant car c'est seulement à ce moment que l'Atelier B peut savoir que la constante ne sera pas implantée. Le but sera :

$\exists \text{fact}.(\text{fact} \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$\forall nn.(nn \in \mathbb{N} \Rightarrow \text{fact}(nn+1) = (nn+1) \times \text{fact}(nn)) \wedge$
$\text{fact}(0) = 1$

Un telle démonstration nous oblige à exprimer la factorielle sous forme d'une définition directe, puisque le seul moyen de prouver un but existentiel est de proposer une valeur. Ici :

$\text{fact} = \lambda nn.(nn \in \mathbb{N} \mid \prod(ii).(ii \in 1 \,..\, nn \mid ii))$

Comme nous sommes obligés d'écrire cette définition, autant la faire figurer dans le composant. Il est toutefois utile de laisser les propriétés précédemment écrites afin de les avoir en hypothèse pour la preuve, cela remplace la connaissance mathématique de la factorielle qui n'est pas dans le prouveur. Bien sûr, il faudra prouver ces propriétés. Nous pouvons écrire :

ABSTRACT_CONSTANTS
fact
PROPERTIES
$\text{fact} \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$\forall nn.(nn \in \mathbb{N} \Rightarrow \text{fact}(nn+1) = (nn+1) \times \text{fact}(nn)) \wedge$
$\text{fact}(0) = 1 \wedge$
$\text{fact} = \lambda nn.(nn \in \mathbb{N} \mid \prod(ii).(ii \in 1 \,..\, nn \mid ii))$

Dans ce cas nous devrons prouver les propriétés de la factorielle lors de l'implantation du composant, ou bien :

ABSTRACT_CONSTANTS
fact
PROPERTIES
$\text{fact} = \lambda nn.(nn \in \mathbb{N} \mid \prod(ii).(ii \in 1 \,..\, nn \mid ii))$
ASSERTIONS

$\text{fact} \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$\forall nn.(nn \in \mathbb{N} \Rightarrow \text{fact}(nn+1) = (nn+1) \times \text{fact}(nn)) \wedge$
$\text{fact}(0) = 1 \wedge$

Dans ce cas nous ferons la démonstration lors de la preuve du composant, c'est à dire immédiatement.

Conclusion : en présence d'un but existentiel abstrait, il faut toujours modifer le composant pour faire apparaître une définition explicite de la notion mathématique concernée.

5.7.3 Les buts non découpés

Les buts non découpés apparaissent quand l'algorithme de découpage du générateur d'obligations est mis en défaut. Il n'est pas forcément nécessaire de modifier le composant, de telles obligations pouvant être plus faciles à prouver qu'il n'y paraît. Dans la plupart des cas, le but non simplifié est de la forme $P \vee Q$. Pour lire de tels buts, il faut utiliser la commande `pr(Red)` pour faire monter les hypothèses locales en simplifiant le but.

(à compléter)

5.8 Les obligations de preuve qui semblent fausses

Lorsque le composant examiné comporte des erreurs, certaines obligations de preuve sont fausses. C'est ainsi que l'utilisation de la méthode B signale les erreurs telles que le non respect de la spécification, ou des propriétés invariantes. Nous proposons la méthode suivante en cas d'obligation de preuve fausse :

- **S'assurer que l'obligation de preuve est bien fausse** : quand une obligation de preuve semble fausse, il faut en avoir une assurance suffisante avant de commencer les modifications dans le composant. Les conséquences d'une modification faite à tort peuvent être très coûteuses.
- **Trouver la faute dans le composant** : il suffit de reporter dans le contexte du composant les raisons qui font que l'obligation de preuve est fausse. En particulier, si on dispose d'un contre-exemple, reporter les valeurs trouvées dans le composant.
- **Corriger** : l'impact d'une PO fausse varie du simple cas oublié à la remise en cause du modèle mathématique employé. C'est pourquoi la détection des obligations de preuve fausses doit être faite en continu ; il n'est pas raisonnable de supposer ne plus avoir à modifier les composants avant les phases de preuve. Les corrections suite à une PO fausse relèvent de la méthode B, nous nous bornerons aux conseils suivants :
  - se demander si, dans le sens physique attribué aux variables du modèle, les valeurs qui rendent l'obligation de preuve fausse sont possibles. Si ce n'est pas le cas, l'invariant est trop faible.
  - si on est amené à renforcer un invariant trop faible, s'assurer que la nouvelle propriété est toujours vraie, pour toutes les opérations. En effet il arrive souvent qu'une propriété physiquement vraie, oubliée dans l'invariant, ne soit pas conservée par des opérations dont le codage est biaisé par l'invariant trop faible.

- avant de modifier un invariant, s'assurer que la modification rend juste l'obligation de preuve fausse ! Ne pas faire de modification *à priori* pour "voir ce que cela donne" dans les PO.
- noter le raisonnement mathématique effectué. D'une manière générale, toutes les justifications recueillies lors de la mise au point du composant facilitent les phases de preuve ultérieures.

Nous allons maintenant examiner plus en détail comment éviter de considérer à tort des obligations de preuve comme fausses.

5.8.1 S'assurer que l'obligation de preuve est bien fausse

Pour éviter des modifications nuisibles dans le composant, il faut vérifier que l'obligation de preuve qui semble fausse l'est vraiment. Pour cela, il faut considérer les points suivants :

- **Vérifier que l'obligation de preuve n'est pas rendue vraie par la présence d'hypothèses contradictoires** : il est normal que de telles PO apparaissent, elles traduisent l'application rigoureuse de la théorie. Les buts de ces obligations de preuve n'ont pas à être vrais, il peuvent même être dépourvus de sens. Dans tous les cas, la contradiction dans laquelle on se place correspond à des branches du composant et de son niveau supérieur : c'est en se guidant sur ces branches qu'il faut chercher la contradiction.
- **Chercher un contre-exemple** : chercher des valeurs particulières des variables qui vérifient les hypothèses, mais pas le but. On fera attention aux points suivants :
  - chercher à utiliser les valeurs 0 et $\varnothing$, elles permettent souvent de fabriquer des contre-exemples simples.
  - si il y a beaucoup d'hypothèses, chercher le contre-exemple uniquement sur les variables qui interviennent dans le but. Ne pas chercher à contrôler si l'obligation de preuve est vraie par hypothèses contradictoire en cherchant le contre-exemple, c'est trop compliqué.
- **obligations de preuve dépourvues de sens** : il est parfaitement possible qu'une obligation de preuve soit dépourvue de sens. De telles obligations signalent bien une erreur dans le composant, mais elles peuvent surprendre en particulier parce qu'il n'y a pas clairement un contre exemple. Par exemple :

  $$uu \in \mathbb{N} \ \wedge \ uu \leq 10 \ \Rightarrow \ uu\$2 \in \mathbb{N}$$

  Un contre-exemple est $uu\$2 = \mathsf{TRUE}$ et $uu = 0$. La recherche de ce type de contre-exemple peut être déroutante. En particulier le contre-exemple précédent ne serait pas accepté par l'Atelier B à cause d'un problème de typage, il est néanmoins mathématiquement valide.
  Ceci est bien une obligation de preuve trahissant une erreur dans le composant, qu'il faut absolument modifier avant de poursuivre. Dans le cas d'obligations de preuve fausses par absence d'information sur une variable, penser à :
  - un invariant de boucle trop faible : Rappelons que dans un invariant de boucle, il faut au moins typer toutes les variables utilisées dans le corps de la boucle ;
  - un invariant de liaison manquant entre une variable abstraite et la ou les variables importées qui la réalisent.

Les obligations de preuve apparemment dépourvues de sens peuvent néanmoins être vraies par hypothèses contradictoires. Par exemple dans un IF..ELSIF sans ELSE, mais dont la conjonction des cas est toujours vraie, l'obligation de preuve correspondant au ELSE absent peut concerner une variable qui n'aura pas été initialisée, sous l'hypothèse contradictoire que la conjonction des cas est fausse. Cette obligation de preuve est correcte, elle ne détecte aucune faute dans le composant.

Chapitre 6

La phase de preuve formelle

Les notions essentielles présentées dans ce chapitre sont les suivantes :

Méthode générale : utliser autant que possible les appels au prouveur.
Utiliser `pr` ou `pp` pour terminer un but, `pr(Red)` pour avancer.
Démarrer en force 0 et avec une démonstration intuitive.

Les commandes se classent en commandes de déplacement, de visualisation et preuve de divers niveau.
Il est souhaitable de connaître l'existence de toutes les commandes (environ 35).
Le prouveur utilise des *règles d'inférence* basées sur la *coïncidence de formules*.
Les *jokers* sont les variables à une lettre, qui représentent n'importe quelle formule. Les *règles de réécriture* permettent de transformer une partie d'une expression.
Les *gardes* permettent d'effectuer des tests lors de l'application d'une règle.
Les règles manuelles doivent être écrites dans le fichier *pmm*, elles s'utilisent avec `pc` et `ar` et sont prouvées à l'aide de la commande `vr`.
Le prouveur de prédicats travaille en se ramenant à des prédicats quantifiés : c'est la commande `pp`.

L'interface du prouveur utilise plusieurs fenêtres prévues pour être juxtaposées et empilées.
L'opérateur peut se concentrer soit sur le choix d'une PO, soit sur la preuve.
Les composants B en cours de preuve doivent être facilement accessibles.
L'opérateur doit organiser son écran avec le prouveur et les composants en cours de preuve.
Le prouveur interactif peut également être employé en mode *batch*, dans un terminal.

La *ligne de commande* représente l'arbre de preuve, c'est un repère essentiel.

Les quatre tactiques simples de preuve sont :
– Utilisation de `pr` et `pp` ;
– Utilisation de l'ajout d'hypothèses et de la preuve par cas ;
– Recherche et utilisation de règles du prouveur ;
– Utilisation de règles manuelles.
La vérification finale de la preuve consiste à rejouer toutes les démonstrations.
Cette vérification se fait avec les option *Unprove* et Replay du menu de preuve.
Une *règle d'admission* permet d'admettre des buts, pour explorer la suite de la démonstration.
C'est une règle ajoutée dans le fichier pmm, dont le conséquent est un joker.
Les forces supérieures (1, 2 et 3) s'emploient parfois en preuve interactive.
La force 1 simplifie mieux les hypothèses, la force 2 ajoutes des hypothèses dérivées.
La trace de preuve permet d'analyser l'action d'une commande `pr`.

Il y a un tableau qui indique la commande de preuve à choisir en fonction de type de situation.
Il faut parfois refaire passer des hypothèses dans le prouveur, pour orienter le prouveur sur elles.
En particulier, des égalités apparues en hypothèse peuvent nécessiter cette opération pour être prises en

compte.
Pour mettre une hypothèse dans la forme exacte qui lui permettra de coïncider avec quelque chose, utiliser *AddHypothesis*.
Utiliser *AddHypothesis* plutôt qu'une règle par l'avant.
Utiliser *AddHypothesis* pour produire des expressions avec les bonnes parenthèses.

Contrôler le nombre de cas de la preuve en utilisant `pr(Red)` plutôt que `pr`.
Ne pas détruire de règle manuelle si cela peut introduire un décalage dans les numéros de règles.
Il faut sauvegarder la démonstration avant de changer de force en cours de preuve interactive.

6.1 Méthode générale

La phase de preuve formelle consiste à démontrer avec le prouveur interactif les obligations de preuve qui restent après la phase de mise au point (voir paragraphe 5). Pour être efficace dans ces preuves interactives, il faut tirer le meilleur parti possible des tactiques automatiques de preuve ; c'est-à-dire **utiliser autant que possible les appels au prouveur**. Ces appels se font par la commande *Prove* (`pr`) qui lance le cœur de preuve dans la force courante ; dans certains cas on essaie également la commande *PredicateProver* (`pp`) qui peut aussi décharger automatiquement le but.

Le cœur de preuve est prévu pour obtenir les meilleurs résultats possibles en preuve automatique, c'est pourquoi il tente des tactiques exploratoires avant de conclure à l'échec. Ces tactiques sont nuisibles en preuve interactive quand elles échouent, car elles conditionnent le but en échec. Le plus souvent, il s'agit de preuves par cas comme dans l'exemple ci-après :

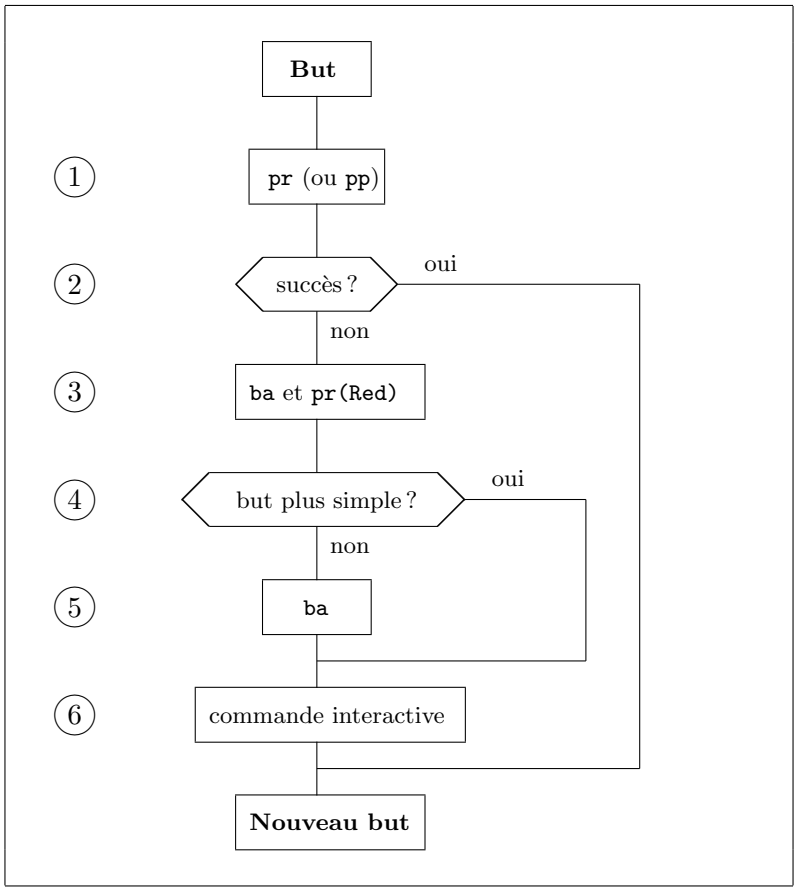
Soit à prouver une obligation de preuve de la forme :

$$H \ \wedge \ (p \ \Rightarrow \ q) \ \Rightarrow \ B$$

si le prouveur échoue dans la démonstration, il peut décider de faire deux cas : p et $\neg p$, pour tenter de tirer parti de l'hypothèse $p \ \Rightarrow \ q$. Si la preuve échoue toujours le but en échec peut être celui du cas p. L'opérateur poursuit la preuve interactive ; à son aboutissement seul le cas p aura été démontré et il restera à traiter l'autre cas. Ce comportement est particulièrement gênant quand le prouveur tente des preuves par cas qui ne servent pas à la démonstration, dupliquant alors gratuitement la preuve. Les preuves par cas ne sont signalées que par les hypothèses de cas, ce qui n'éveille pas facilement l'attention de l'opérateur. C'est pourquoi il est recommandé d'employer la commande `pr(Red)` qui lance le prouveur sans les tactiques de preuve exploratoires ; plus précisément :

– lancer `pr` ;
– si la branche de preuve courante aboutit, continuer sur le but suivant ;
– si la preuve échoue, reculer par *Back* (`ba`) et lancer `pr(Red)` pour avancer la preuve sans risque de cas exploratoires.

La méthode générale pour faire les démonstrations formelles est la suivante :



Ce schéma explique comment "avancer" dans la preuve en traitant chaque nouveau but qui se présente. Quand il ne reste plus de but à traiter, la preuve est finie. Détaillons les étapes :

Etape 1 Pour chaque nouveau but, il faut lancer le prouveur pour ne jamais perdre de temps s'il est automatiquement démontré. Il ne faut pas faire cette étape lorsqu'il est certain que le prouveur ne peut pas aboutir : c'est le cas en particulier du but de départ. Sa preuve échoue forcément puisque l'obligation de preuve n'a pas été démontrée automatiquement. Penser aussi à essayer la commande `pp` à cette étape, éventuellement précédée de `dd` pour faire monter des hypothèses (rappelons comme nous l'avons vu dans le chapitre précédent, que le prouveur de prédicat réagit généralement mieux si toutes les hypothèses sont montées).

Etape 2 Si la dernière branche de preuve est déchargée, nous avons obtenu un nouveau but. Sinon nous passons à l'étape 3.

Etape 3 La précédente commande peut avoir tenté des preuves par cas exploratoires, il faut donc revenir en arrière (`ba`) et utiliser une commande de preuve réduire pour progresser dans la preuve sans risque.

Etape 4 La commande précédente utilise les tactiques et les règles de la preuve automatique : il est possible que la direction prise ne soit pas celle souhaitée par l'opérateur. Nous devons donc juger à cette étape si il faut continuer à partir du but simplifié par la preuve réduite ou revenir en arrière. En pratique il suffit de juger la simplicité de ce nouveau but.

Etape 5 Retour en arrière si le but simplifé n'est pas jugé bénéfique.

Etape 6 C'est bien sûr l'étape la plus délicate : il s'agit de trouver la bonne commande interactive pour faire progresser la preuve vers son aboutissement. La suite de cette section traite principalement ce problème.

Par exemple, soit à démontrer l'obligation de preuve suivante :

$$
\begin{array}{l}
\text{"}\mathit{Local\ hypotheses}\text{"} \wedge \\
\mathit{ntt} \in \mathit{TACHES} \wedge \\
\neg(\mathit{ntt} \in \mathit{taches}) \wedge \\
\mathit{tt\$0} \in \mathit{taches} \cup \{\mathit{ntt}\} \wedge \\
\text{"}\mathit{Check\ that\ ...}\text{"} \\
\quad \Rightarrow \\
(\mathit{tconnait} \cup \{\mathit{ntt}\} \times \mathit{tconnait}[\{\mathit{tt}\}])[\{\mathit{tt\$0}\}] = \\
\quad \{\mathit{pass}((\mathit{tident} \lhd \{\mathit{ntt} \mapsto \mathit{tident}(\mathit{tt})\})(\mathit{tt\$0}))\}
\end{array}
$$

Volontairement, nous ne montrons que les hypothèses locales : telle que présentée ci-dessus, cette obligation est incomplète et donc indémontrable. Mais ce n'est pas notre propos de relire les hypothèses ici, normallement nous disposons d'une démonstration intuitive provenant de la phase de mise au point. C'est à partir de cette démonstration et non pas en relisant toute l'obligation que nous devons démarrer notre preuve formelle (pour cet exemple, que le lecteur nous fasse confiance pour la démonstration intuitive).

Nous sommes dans l'étape 1. Il est inutile d'essayer `pr` puisque l'obligation n'est pas démontrée automatiquement. Par contre, la présence d'unions et de relations nous suggère d'essayer le prouveur de prédicats : `pp`, précédé de `dd` pour séparer les hypothèses. Le prouveur de prédicats échoue. Nous devons donc conformément aux étapes 2 et 3 revenir de deux pas en arrière : `ba(2)` et fair un appel au cœur de preuve, mode réduit : `pr(Red)`. Cet appel charge les hypothèses et crée des hypothèses dérivées. Le but n'a pas changé.

$$
\begin{array}{l}
(\mathit{tconnait} \cup \{\mathit{ntt}\} \times \mathit{tconnait}[\{\mathit{tt}\}])[\{\mathit{tt\$0}\}] = \\
\quad \{\mathit{pass}((\mathit{tident} \lhd \{\mathit{ntt} \mapsto \mathit{tident}(\mathit{tt})\})(\mathit{tt\$0}))\}
\end{array}
$$

Nous sommes dans l'étape 4 : le but peut être considéré comme plus simple puisqu'il ne contient plus les hypothèses. Nous passons en étape 6 : choix de la commande interactive. La démonstration intuitive (que le lecteur nous fasse confiance) suggère de faire deux cas : $\mathit{tt\$0} = \mathit{ntt}$ ou $\mathit{tt\$0} \neq \mathit{ntt}$. La commande est donc :

```
dc(tt$0 = ntt)
```

Le but devient :

$$
\begin{array}{l}
\mathit{tt\$0} = \mathit{ntt} \ \Rightarrow \\
(\mathit{tconnait} \cup \{\mathit{ntt}\} \times \mathit{tconnait}[\{\mathit{tt}\}])[\{\mathit{tt\$0}\}] = \\
\quad \{\mathit{pass}((\mathit{tident} \lhd \{\mathit{ntt} \mapsto \mathit{tident}(\mathit{tt})\})(\mathit{tt\$0}))\}
\end{array}
$$

Nous sommes revenus dans l'étape 1 : il faut essayer `pr`, qui ne démontre toujours pas le but courant. Nous pourrions alors essayer `pp`, mais l'usage du prouveur de prédicat se fait plus rarement au milieu d'une preuve car il met une minute à échouer. Nous faisons donc l'étape 2 : `ba` et `pr(Red)`. Le but devient :

$$(tconnait \cup \{ntt\} \times tconnait[\{tt\}])[\{ntt\}] = \{pass(tident(tt))\}$$

Nous sommes dans l'étape 4. Comme le but s'est nettement simplifié, nous passons dans l'étape 6 : choix d'une nouvelle commande. Ici le nouveau but contient toujours des unions et des relations, peut être le prouveur de prédicats le démontre t-il. La commande choisie est `pp`, et le but devient :

$$\begin{array}{l} tt\$0 \neq ntt \ \Rightarrow \\ (tconnait \cup \{ntt\} \times tconnait[\{tt\}])[\{tt\$0\}] = \\ \quad \{pass((tident \lhd \{ntt \mapsto tident(tt)\})(tt\$0))\} \end{array}$$

Nous sommes passés dans le deuxième cas, qui se démontre pareillement (nous verrons dans ce chapitre comment savoir dans quel cas on se trouve). Nous refaisons les deux étapes précédentes, la preuve aboutit. L'arbre de preuve est le suivant :

```
Force(0) &
  pr(Red) &
    dc(tt$0 = ntt) &
      pr(Red) &
        pp &
      pr(Red) &
        pp &
  Next
```

Cet arbre est affiché dans l'une des zones de l'interface MOTIF, comme nous le verrons. La preuve est terminée en six commandes. Nous voyons sur cet exemple à quel point il est possible de s'appuyer sur `pr` et `pp` pour "débloquer" une démonstration, en utilisant l'organigramme présenté dans ce paragraphe. Il faut de plus considérer les points suivants :

- **La preuve interactive se fait en force 0**. Il est toujours possible d'utiliser des forces plus élevées en preuve interactive, mais cela se justifie rarement : rappelons qu'il faut environ une minute pour charger 100 hypothèses en force 1, contre moins d'une seconde en force 0. Le gain en performance, dans l'optique d'une démonstration interactive c'est-à-dire guidée par l'opérateur, ne justifie pas ce coût. On accède en interactif à une PO en la remettant en force 0 par `gr` (GotowithReset). Pour repasser en force 0 sur une obligation de preuve chargée, il suffit d'utiliser la commande `ff(0)`.
- **Démarrer avec une démonstration intuitive**. Dans la phase de mise au point, la PO concernée a du être visitée, une justification et une démonstration intuitive ont été faites. Il faut repartir de cet état. En principe, des notes auront été prises dans ce but en phase de mise au point. Il faut refaire le raisonnement intuitif.

La suite de ce chapitre est découpée de la manière suivante :

- Introduction à la preuve interactive : les notions essentielles d'utilisation de cet outil y sont expliquées. En particulier, les commandes sont présentées dans l'ordre d'importance.
- Utilisation de l'interface : le prouveur interactif fonctionne avec une interface Motif dont l'utilisation est présentée dans ce paragraphe.
- La ligne de commande : ce repère essentiel pour la preuve est détaillé dans ce paragraphe.
- Les tactiques simples de preuve.

– Utilisation avancée du prouveur.
– Les recettes de preuve : on y trouve en particulier le tableau des commandes à choisir en fonction de chaque type de situation.
– Les pièges à éviter : ce paragraphe est à lire impérativement avant une utilisation intensive du prouveur.

6.2 Introduction à la preuve interactive

Cette partie est destinée aux utilisateurs s'initiant à la preuve interactive avec l'Atelier B, mais qui connaissent le langage B, les principes mathématiques de preuve et les notations utilisées dans les obligations de preuve. Les notions présentées sont les suivantes :

– Les commandes du prouveur interactif ;
– Le langage des règles ;
– Le prouveur de prédicats.

6.2.1 Les commandes du prouveur interactif

Les explications qui suivent ne remplacent pas le manuel de référence du prouveur interactif, en particulier la syntaxe précise de chaque commande n'est pas décrite. Dans ce paragraphe, nous cherchons à donner les indications nécessaires pour penser à utiliser la bonne commande au bon moment. Si vous ne connaissez pas les commandes du prouveur interactif, la lecture du présent paragraphe vous est recommandée.

Nous avons classé les commandes dans différentes rubriques, classées par ordre d'importance pour une bonne utilisation du prouveur. Ces rubriques sont :

– les commandes de déplacement ;
– les commandes de lecture ;
– les commandes de preuve automatique ;
– les commandes de preuve sans ajout de règles ;
– les commandes d'ajout de règles protégées ;
– les règles manuelles ;
– la généralisation ;
– les commandes de preuves s'appliquant dans des cas particuliers ;
– les autres commandes.

Nous allons maintenant rentrer dans le détail de ces rubriques.

Les commandes de déplacement

Les commandes ci-après permettent de se déplacer dans les preuves. Il est difficile de leur donner un ordre d'importance, car elles sont toutes nécessaires pour utiliser le prouveur interactif.

– **Goto** (`go`) : permet de se positionner sur une obligation de preuve. Avec l'interface

Motif il est normalement plus simple de double-cliquer sur la PO dans la liste, cette commande est donc surtout employée en mode batch (voir paragraphe 6.3.2). Dans ce mode, elle est absolument nécessaire.
- **Next** (`ne`) : passe à la prochaine PO non prouvée.
- **Back** (`ba`) : annule l'effet de la précédente commande interactive ayant agit sur la preuve. Elle est beaucoup utilisée après un appel au cœur de preuve qui a échoué, conformément à la méthode décrite au début de ce chapitre. **Utiliser Back pour éliminer les commandes inutiles ou pour replacer une action interactive avant l'action courante**.
- **GotowithReset** (`gr`) : cette commande permet d'aller sur une obligation de preuve en remettant sa force à 0. Si il y a beaucoup d'hypothèses, le chargement d'une PO peut être long en particulier pour la première PO d'une opération où il comprend le chargement de toutes les hypothèses de contexte (1 minute par centaine d'hypothèses en force 1). **Utiliser GotowithReset quand il y a plusieurs centaines d'hypothèses, pour aller sur la première PO d'une opération**.
- **SaveWithoutquestion** (`sw`) : provoque la sauvegarde forcée de la ligne de commande. **Utiliser `sw` de temps à autre si la démonstration devient longue, en particulier avant un changement de force ou des retours en arrière répétés**.
- **Reset** (`re`) : permet de revenir au début de la démonstration d'une PO. Cela s'utilise soit pour relire le but initial (penser aussi à `lp`), soit pour reprendre la démonstration avec l'expérience de l'essai précédent (penser à sauvegarder auparavant, `sw`).
- **Step** (`st`) : la commande *Step* exécute la prochaine commande sauvegardée, et avance de un le curseur dans la démonstration sauvegardée. **Utiliser Step pour rejouer une preuve sauvegardée**.
- **Force** (`ff`) : permet de rejouer une démonstration dans une force différente. Souvent utilisée pour remettre à zéro la force d'une PO avant de commencer une démonstration interactive, cette commande devrait être réservée aux cas plus rares où l'opérateur veut tenter l'utilisation d'une force supérieure (voir paragraphe 6.8.3). Sinon utiliser plutôt *GotowithReset*.
- **Repeat** (`rr`) : répète la dernière commande entrée.
- **Quit** (`qu`) : quitte le prouveur interactif. Avec l'interface Motif, l'utilisateur peut aussi utiliser les boutons prévus à cet effet.

Les commandes de lecture

Les deux commandes suivantes permettent la lecture des obligations de preuve.

- **SearchHypothesis** (`sh`) : cette commande d'information permet la recherche de toutes les hypothèses qui vérifient un certain filtre. Son utilisation la plus courante est la recherche de toutes les hypothèses concernant une variable. **Utiliser SearchHypothesis dès que la recherche d'hypothèses dans la liste générale devient difficile**.
- **showReducedPo** (`rp`) : cette commande affiche l'obligation de preuve réduite, c'est-à-dire avec seulement les hypothèses qui ont un symbole en commun avec le but. C'est ce qu'il faut regarder en premier pour contrôler rapidement la justesse d'une PO ou d'un sous but. **Utiliser showReducedPo pour retrouver une démonstration intuitive, ou conjointement avec SearchHypothesis**.

Les commandes de preuve automatique

Présentons maintenant les deux commandes de preuve automatique :

- **Prove** (`pr`) : c'est l'appel au cœur de preuve. son usage doit être contrôlé, en particulier concernant les preuves par cas parasites (voir paragraphe 6.8.1). Mais il ne faut pas démontrer manuellement ce que le cœur de preuve sait démontrer. **Ne jamais oublier de tenter un appel au cœur de preuve sur un but avant de faire des commandes interactives pour le démontrer**.
- **PredicateProver** (`pp`) : cette commande appelle le prouveur de prédicats sur le lemme courant. Le prouveur de prédicats (voir paragraphe 6.2.4) agit souvent avec succès sur des buts composés d'expressions ensemblistes ou fonctionnelles. **Penser à PredicateProver si le but est uniquement composé d'expressions ensemblistes ou fonctionnelles**.

Les commandes de preuve sans ajout de règle

Les commandes suivantes sont celles qui permettent de faire aboutir des preuve sans ajout de règle manuelles ou contrôlées par le prouveur de prédicats.

- **AddHypothesis** (`ah`) : l'opérateur peut proposer de nouvelles hypothèses, qui seront ajoutées aux hypothèses existantes après démonstration de l'hypothèse proposée. Cette commande permet donc d'ajouter des hypothèses superfétatoires. Ces hypothèses sont saisies par l'opérateur qui peut ainsi introduire de l'information supplémentaire dans la preuve en apportant le bénéfice de son intuition et de son imagination. *AddHypothesis* est l'une des commandes de preuve les plus importantes, elle permet de diriger la preuve en faisant démontrer des buts intermédiaires partant des hypothèses afin de s'approcher du but principal. Une autre utilisation est de refaire passer les hypothèses dans le prouveur, voir paragraphe 6.7.2. **Penser à AddHypothesis dès qu'une nouvelle expression est nécessaire, ou si une hypothèse n'est pas complètement simplifiée**.
- **Deduction** (`dd`) : cette commande permet de faire monter l'hypothèse h si le but est de la forme $h \Rightarrow B$, sans passer par le cœur de preuve. **Penser à Deduction quand le prouveur monte une hypothèse sous une forme normalisée peu avantageuse**. Par contre, une hypothèse introduite de cette manière n'est pas traitée par certains mécanismes du prouveur, en particulier le simplificateur d'égalités.
- **DoCases** (`dc`) : cette commande permet la preuve par cas. L'opérateur peut donner un prédicat p en argument, la preuve se fait alors dans les cas p et $\neg p$; ou bien il peut donner le nom d'une variable v et un ensemble E fini de petit cardinal, la preuve se poursuit alors pour v étant chacun des éléments de E. Dans ce dernier cas, il faut d'abord établir $v \in E$. Il y a beaucoup de démonstrations qui ne peuvent se faire que par cas, ce que le prouveur ne détecte pas toujours. **Penser à DoCases si la démonstration intuitive se fait par cas**.
- **SearchRule** (`sr`) : permet de rechercher une règle dans la base de règles du cœur de preuve. **Utiliser SearchRule avant de tenter d'ajouter une règle**. S'il manque des hypothèses pour appliquer la règle trouvée, utiliser `ah` plutôt que d'écrire une règle manuelle de remplacement.
- **ApplyRule** (`ar`) : cette commande permet l'application d'une règle de la base du prouveur, trouvée par *SearchRule*. **Penser à ApplyRule dès qu'il est clair qu'une transformée mathématique simple permet de conclure**. En effet, il faut chercher à appliquer des règles les plus simples possibles, ces règles ont ainsi plus de chances de se trouver dans la base.

Les commandes pour les règles manuelles

Les commandes suivantes deviennent nécessaires si on est conduit à utiliser des règles manuelles :

- **ApplyRule** (`ar`) : cette commande permet l'application d'une règle. Il peut s'agir d'une règle de la base, trouvée par *SearchRule*, d'une règle ajoutée après preuve par le prouveur de prédicats (voir paragraphe 6.2.4) ou d'une règle manuelle. **Penser à ApplyRule dès qu'il est clair qu'une transformée mathématique simple permet de conclure**. En effet, il faut chercher à appliquer des règles les plus simples possibles, ces règles ont ainsi plus de chances de se trouver dans la base ou d'être démontrées par le prouveur de prédicats. Si la règle doit être ajoutée manuellement, sa validation sera facilitée si elle est simple.
- **PmmCompile** (`pc`) : cette commande permet la lecture et la compilation du fichier `composant.pmm` contenant les règles manuelles. **Utiliser PmmCompile pour prendre en compte une modification dans le fichier pmm**.

La généralisation

La commande suivante s'utilise après le succès d'une première obligation de preuve si l'opérateur désire généraliser sa démonstration à d'autre obligations.

- **TryEverywhere** (`te`) : essai de la généralisation d'une démonstration. Elle permet d'exécuter la preuve d'un ensemble d'obligations de preuve en modifiant ou remplaçant leurs lignes de commandes à partir d'une séquence de commandes passée en paramètre. **Utiliser TryEverywhere après le succès d'une démonstration, si d'autres PO semblent se prouver pareillement**.

Les cas particuliers

Les commandes suivantes s'appliquent dans des cas de preuve particuliers.

- **useEqualityinHypothesis** (`eh`) : cette commande permet de remplacer a par A dans le but ou les hypothèses lorsque $a = A$ ou bien $A = a$ est en hypothèse. Elle est utile dès que le prouveur ne fait pas un remplacement souhaitable, si l'égalité est une hypothèse dérivée (voir cette notion au paragraphe 6.7.2) ou si on veut réécrire une hypothèse antérieure à l'égalité, et qui n'a donc pas subi son action. **Penser à useEqualityinHypothesis dès qu'une égalité doit être utilisée dans le but ou une hypothèse**.
- **SuggestforExist** (`se`) : lorsqu'un but est de la forme $\exists x.P(x)$, l'opérateur peut proposer des valeurs x_0 pour les variables x. Le but devient alors $P(x_0)$, s'il est démontré le but existentiel initial est établi car on a exhibé une valeur qui vérifie le prédicat. En pratique c'est dans 90% des cas la seule manière de résoudre une PO existentielle. **Penser à SuggestforExist dès que le but est existentiel**. Les valeurs à proposer se choisissent en regardant d'où provient le but dans le composant, plus que par examen de ce but qui est souvent sur plusieurs lignes.
- **FalseHypothesis** (`fh`) : si l'une des hypothèses est contradictoire avec les autres, le but n'a plus d'intérêt : il suffit d'établir que les hypothèses courantes établissent la négation de cette hypothèse. L'opérateur peut donc indiquer une hypothèse qu'il croit contradic-

toire, le but devient alors la négation de cette hypothèse. Il n'est pas possible de retirer l'hypothèse concernée pour qu'elle n'apparaisse plus que dans le but, la commande `fh` ne le fait donc pas, mais ce n'est pas gênant : le lemme résultant est simplement vrai à la fois par le but et par hypothèses contradictoires. **Penser à FalseHypothesis si la preuve d'une PO vraie par hypothèses contradictoires échoue**.

– **Contradiction** (`ct`) : permet de faire de la preuve par contradiction. La négation du but monte en hypothèse et le but devient *bfalse*. Ceci est particulièrement utile lorsque le but est de la forme `not(p)`, car c'est alors p qui monte en hypothèse et la présence de *bfalse* en but incite les mécanismes du cœur de preuve à rechercher une contradiction. De plus, si le but est de la forme `not(a = b)`, la réécriture de a par b provoquée par l'hypothèse $a = b$ peut mettre très facilement en évidence la contradiction. **Penser à Contradiction si le but est de la forme `not(p)`, et particulièrement s'il est de la forme `not(a = b)`.**

Autres commandes

Enfin, les commandes suivantes sont plus rarement employées.

– **ParticularizeHypothesis** (`ph`) : pour une hypothèse de la forme $\forall x.(P(x) \Rightarrow Q(x))$, l'opérateur peut proposer une liste de valeurs x_0, il devra alors prouver le sous but $P(x_0)$ pour que l'hypothèse $Q(x_0)$ apparaisse. **Penser à ParticularizeHypothesis si le cœur de preuve n'arrive pas à instancier lui-même une hypothèse** $\forall x.(P(x) \Rightarrow Q(x))$. En particulier si $P(x_0)$ n'est pas directement en hypothèse, le prouver comme un sous but permet d'utiliser toutes les fonctionnalités du prouveur pour le faire apparaître.
– **ModusponensonHypothesis** (`mh`) : permet de générer Q si $P \Rightarrow Q$ et P sont en hypothèses. Le cœur de preuve le fait systématiquement, sauf pour des hypothèses déjà dérivées (voir paragraphe 6.7.2).
– **showLitteralPo** (`lp`) : cette commande permet d'afficher une obligation de preuve telle qu'elle est définie dans les fichiers du composant, sans même avoir à charger la PO concernée. Ceci est parfois utile pour retrouver plus facilement le lien entre une obligation de preuve et le source du composant.
– **GlobalSituation** (`gs`) : cette commande retourne la liste des obligations de preuve avec leur état et leur but. Avec l'interface Motif, l'opérateur ne devrait jamais avoir à l'utiliser ; elle n'est utile qu'en mode batch (voir paragraphe 6.3.2).
– **GotoWithoutsave** (`gw`) : rarement utilisée, cette commande permet de quitter une obligation de preuve sans aucune sauvegarde.
– **SavewithQuestion** (`sq`) : provoque la sauvegarde de la ligne de commande courante, en demandant confirmation à l'opérateur si il n'est pas clair que le remplacement de la démonstration sauvée par la démonstration courante soit bénéfique (par exemple : si aucune des deux démonstration n'aboutit).

6.2.2 Les règles et leur usage

Une preuve mathématique formelle se fait en utilisant des règles mathématiques. Par exemple, on démontre :

$$p \in s \leftrightarrow t \wedge q \in s \leftrightarrow t \Rightarrow \mathsf{dom}(p) - \mathsf{dom}(q) \subseteq \mathsf{dom}(p - q)$$

en utilisant la définition de l'inclusion

$$\mathsf{dom}(p) - \mathsf{dom}(q) \subseteq \mathsf{dom}(p - q) \;\Leftrightarrow\; \forall x.(x \in \mathsf{dom}(p) - \mathsf{dom}(q) \;\Rightarrow\; x \in \mathsf{dom}(p - q))$$

puis la définition du domaine d'une relation, etc. Le mathématicien fait souvent des démonstrations sans citer explicitement toutes les règles qu'il utilise. Néanmoins, le choix des règles considérées comme autorisées est important pour savoir ce qu'on appelle une démonstration correcte. Il serait trop facile de prétendre que le lemme à démontrer est toujours une règle autorisée.

Dans une démonstration B, les règles autorisées sont celles de la théorie de construction du langage (voir B-Book) et celles de la base du prouveur. Comment de telles règles mathématiques peuvent-elles être traduites en quelque chose d'exécutable sur une machine informatique ? Nous allons introduire quelques notions nécessaires pour comprendre cela. Ces notions sont les fondements du langage de théorie sur lequel est basé le prouveur de l'Atelier B. Le langage de théorie est implanté par une couche logicielle appelée *Logic Solver*. Pour l'utilisateur du prouveur, il suffit de connaître les quelques notions ci-après pour pouvoir trouver une règle dans la base mathématique ou écrire une règle manuelle. La totalité du langage de théorie est définie dans le document LogicSolver.

Notion de coïncidence de formules

Expliquons cette notion sur un exemple : on dit que la formule :

$aa + (bb - cc) \times 12$

coïncide (match) avec le gabarit (template) :

$x + y \times z$

parce que le remplacement de x par aa, y par $bb - cc$ et z par 12 permet d'obtenir la formule à partir du gabarit. Dans un gabarit, les identificateurs ne comportant qu'une lettre ont un rôle particulier : elles remplacent n'importe quelle formule. On les appelle des **jokers**. Ainsi $aa + bb$ coïncide avec $aa + x$ ou $u + v$, mais pas avec $x + aa$. Attention aussi aux parenthèses implicites : $aa + bb + cc$ coïncide avec $x + y$ (x sur $aa + bb$ et y sur cc) car $aa + bb + cc$ est compris comme $((aa + bb) + cc)$ [1].

Notion de règle d'inférence

Expliquons cette notion sur un exemple : si le but à prouver est :

$\mathsf{bool}(xx - 2 \leq 10) = \mathsf{bool}(xx \leq 12)$

alors la règle d'inférence suivante peut s'appliquer :

[1] Les parenthèses dans une formule sont uniquement des modificateurs d'associativité : on considère par exemple qu'il n'y a aucune différence entre $3 + (3 \times 6)$ et $3 + 3 \times 6$.

$(a \Rightarrow b) \wedge$
$(b \Rightarrow a) \wedge$
$\Rightarrow$
$\mathsf{bool}(a) = \mathsf{bool}(b)$

car le but coïncide avec le gabarit $\mathsf{bool}(a) = \mathsf{bool}(b)$, avec le filtre suivant :

$a \rightsquigarrow xx - 2 \leq 10$
$b \rightsquigarrow xx \leq 12$

Ce gabarit $\mathsf{bool}(a) = \mathsf{bool}(b)$ placé après le symbole implique est appelé conséquent de la règle, tandis que $(a \Rightarrow b)$ et $(b \Rightarrow a)$ sont les antécédents. L'application de cette règle provoque le déchargement du but initial et l'apparition des sous buts suivant :

$a \Rightarrow b$ qui devient $(xx - 2 \leq 10 \Rightarrow xx \leq 12)$
$b \Rightarrow a$ qui devient $(xx \leq 12 \Rightarrow xx - 2 \leq 10)$

Certaines règles n'ont pas d'antécédent, dans ce cas le symbole implique est omis ; par exemple :

$\exists x.(x \in \mathsf{BOOL})$

Ces règles sont appelées règles feuilles ou terminales, car elles éliminent un but sans en produire de nouveau. Une preuve aboutit quand tous les buts ont disparu par l'action de règles feuilles.

Notion de réécriture

Une règle dont le conséquent est de la forme $E == F$ est une réécriture, elle ne fonctionne pas comme une règle normale. Elle s'applique s'il existe une sous formule dans le but qui coincide avec le gabarit E. le nouveau but est alors obtenu par remplacement de E par F. Par exemple, la règle :

$\neg\neg P == P$

transforme le but $\{uu \mid uu \in \mathbb{N} \wedge \underline{\neg\neg(uu = 0)}\} = \{0\}$ en $\{uu \mid uu \in \mathbb{N} \wedge \underline{uu = 0}\} = \{0\}$. La sous formule $\neg\neg(uu = 0)$ est localisée et transformée en $uu = 0$.

Chaque application d'une règle de réécriture ne transforme qu'une occurrence de la sous formule de gauche. Ainsi :

$aa == bb$

transforme la formule $aa + aa = 0$ en $aa + bb = 0$ pour la première application de la règle, puis en $bb + bb = 0$ pour la deuxième application (nous ne préciserons pas pourquoi c'est l'occurence de droite qui se transforme en premier, cela nous mènerait trop loin). Lors de l'application d'une règle par *ApplyRule*, une règle de réécriture se ré-applique toujours tant qu'elle transforme quelque chose (les modes *Once* et *Multi* ne s'appliquent pas aux réécritures). **Attention aux bouclages** : la règle

$x == x + 0$

provoque un bouclage. Par exemple sur le but $aa = bb$:

$aa = bb$
$aa = bb + 0$
$aa = bb + 0 + 0$
$aa = bb + 0 + 0 + 0$
etc.

Notons que cette règle est d'ailleurs fausse, car x peut coïncider avec des expressions non numériques.

Notion de garde

Il arrive souvent qu'une règle ait besoin de plus d'information que ce que contient le but pour s'appliquer. Par exemple, soit à prouver :

$xx \leq yy \ \wedge$
$yy \leq 5$
$\Rightarrow$
$xx \leq 5$

Quelle règle d'inférence permettrait de décharger ce lemme ? La règle suivante ne convient pas :

$x \leq y \ \wedge$
$y \leq z$
$\Rightarrow$
$x \leq z$

En effet, analysons comment elle s'appliquerait :

le but $xx \leq 5$ coïncide avec $x \leq z$ avec le filtre :
$x \rightsquigarrow xx$
$z \rightsquigarrow 5$
Les nouveaux buts sont produits :
$x \leq y$ produit le but $xx \leq y$
$y \leq z$ produit le but $y \leq 5$

Notons que le joker y qui n'est pas instancié reste tel quel dans ces nouveaux buts. Malheureusement, aucune variable du lemme à démontrer ne s'appelle y en une seule lettre, ces buts sont donc dépourvus de sens car ils utilisent une variable non définie. La règle considérée ne convient donc pas pour démontrer ce lemme.

Pour résoudre ces cas, il faut pouvoir écrire des règles qui vont rechercher directement dans les hypothèses : on utilise des fonctionnalités spécifiques du langage de théorie appelées gardes. Le langage de théorie propose une trentaine de gardes, dont la plupart ne sont pas intéressantes dans le cadre de la preuve interactive. Nous nous limiterons aux gardes exposées ci-après. Le principe d'une garde est d'effectuer une opération informatique après que le but ait coïncidé avec le conséquent de la règle, mais avant la génération de nouveaux buts. Cette opération informatique peut avoir différents effets, mais elle retourne toujours

VRAI ou FAUX. La règle ne s'applique que si le résultat est VRAI.

La liste des gardes utiles pour la preuve est la suivante :

– **binhyp** : permet de rechercher une hypothèse. Par exemple :
$$\begin{array}{l}\mathsf{binhyp}(x \leq y) \ \wedge \\ y \leq z \\ \Rightarrow \\ x \leq z\end{array}$$
Cette règle s'applique sur l'exemple suivant :
$$\begin{array}{l}xx \leq yy \ \wedge \\ yy \leq 5 \\ \Rightarrow \\ xx \leq 5\end{array}$$
Tout d'abord, le but $xx \leq 5$ coïncide avec $x \leq z$, x et z s'instancient sur xx et sur 5. Puis la recherche de $xx \leq y$ se fait en remontant dans les hypothèses : celles ci sont $yy \leq 5$ et $xx \leq yy$. La coïncidence se produit alors pour $xx \leq yy$ en instanciant y sur yy, la garde est VRAIE. Le but $yy \leq 5$ est alors produit. Attention, la recherche d'une hypothèse s'arrête à la première coïncidence. Si le lemme est :
$$\begin{array}{l}xx \leq yy \ \wedge \\ xx \leq uu \ \wedge \\ yy \leq 5 \\ \Rightarrow \\ xx \leq 5\end{array}$$
La règle s'applique toujours, mais c'est le but $uu \leq 5$ qui est produit. Celui-ci ne permet pas à la preuve d'aboutir.
– **band** : il s'agit d'un conjoncteur de gardes. $\mathsf{band}(g_1, g_2)$ provoque l'exécution de la garde g_1 pour tout les cas possibles tant que g_2 n'est pas vérifiée. Par exemple, la règle
$$\begin{array}{l}\mathsf{band}(\mathsf{binhyp}(x \leq y), \mathsf{binhyp}(y \leq z)) \\ \Rightarrow \\ x \leq z\end{array}$$
S'applique à l'exemple suivant :
$$\begin{array}{l}xx \leq yy \ \wedge \\ xx \leq uu \ \wedge \\ yy \leq 5 \\ \Rightarrow \\ xx \leq 5\end{array}$$
Tout d'abord, le but $xx \leq 5$ coïncide avec $x \leq z$, x et z s'instancient sur xx et sur 5. Puis la recherche de $xx \leq y$ se fait en remontant dans les hypothèses, la coïncidence se produit pour $xx \leq uu$ en instanciant y sur uu, le premier binhyp est VRAI. Mais il n'y a aucune hypothèse $uu \leq 5$ qui permettrait le succès du deuxième binhyp. Par l'action de band le parcours en remontant du premier binhyp se poursuit, la coïncidence a alors lieu sur $xx \leq yy$ qui permet le succès.
– **btest** : cette garde évalue une expression littérale de l'une des formes suivantes : $a = b$, $a \neq b$, $a < b$, $a > b$, $a \leq b$ et $a \geq b$. l'écriture ASCII correspondante est :
```
btest(a = b)
btest(a /= b)
btest(a< b)
btest(a> b)
btest(a <= b)
btest(a >= b)
```

Les quatre dernières formes qui concernent la relation d'ordre ne sont VRAIES que si a et b sont des entiers littéraux et positifs. Par exemple, la règle $btest(x > 0) \Rightarrow x \in \mathbb{N}$ permet de décharger les buts $3 \in \mathbb{N}$, $100 \in \mathbb{N}$, etc. Les formes $btest(a = b)$ et $btest(a \neq b)$ concernent l'égalité lexicale, elles testent si a et b sont lexicalement égaux. Par exemple, $btest(var1 = var2)$ est faux, même si il existe une hypothèse qui précise $var1 = var2$, mais $btest(aa = aa)$ est vrai. Attention, l'égalité lexicale ne s'applique qu'aux nombres ou aux identifiants. Par exemple, $btest(xx + yy = xx + yy)$ est faux car $xx + yy$ n'est ni un nombre, ni un identifiant.

6.2.3 L'écriture d'un fichier de règles manuelles

En dernier recours, l'opérateur peut introduire une règle manuelle, non démontrée, pour dépanner une preuve. Cette méthode ne doit être employée qu'après échec de la preuve par les commandes interactives, et échec de l'ajout de la règle désirée comme règle protégée par le prouveur de prédicats (commande *Validation of Rule*). La règle est alors écrite en langage de théorie dans un fichier. Ce fichier doit s'appeler `composant.pmm` (pmm : Proof Methods Manual), où `composant` est le nom sans extension du composant ; il doit se trouver dans le répertoire `bdp` correspondant au projet. Ce fichier n'est pas créé par défaut par l'Atelier, on peut ainsi s'assurer facilement de l'absence de règles manuelles pour un composant.

La syntaxe à respecter est la suivante :

- placer les règles dans des théories déclarées par `THEORY name IS liste de règles END` ;
- séparer les règles par des points-virgule ;
- séparer les théories par le caractère `&`.

Voici un exemple de fichier de règles :

```
THEORY MyRules IS

  binhyp(a<=b)
  =>
  b+1-1: a..b;

  a<=b
  =>
  a-1-b+1<=0

END

&

THEORY NatCalc IS

  a: NATURAL &
  b: NATURAL
  =>
  a+b: NATURAL

END
```

Notons que les règles de ce fichier ne sont pas forcément des équivalences. Par exemple, examinons la règle suivante :

```
a: NATURAL &
b: NATURAL
=>
a+b: NATURAL
```

Cette règle n'est pas une équivalence. Si elle est appliquée sur le but $1 + 2 \in \mathbb{N}$, elle produit bien les deux buts vrais $1 \in \mathbb{N}$ et $2 \in \mathbb{N}$ qui impliquent le but initial ; mais sur le but $1 + (-2) \in \mathbb{N}$ cette règle produit le but faux $(-2) \in \mathbb{N}$ et provoque l'échec de la preuve. Il s'agit donc d'employer de telles règles à bon escient.

Le fichier des règles manuelles d'un composant n'est pas lu automatiquement par le prouveur après chaque modification : ce serait trop lourd. Pour que les modifications soient prises en compte, il faut utiliser la commande *PmmCompile* (`pc`). Nous allons maintenant présenter un exemple complet d'utilisation :

Soit à démontrer le lemme suivant :

$$
\begin{array}{l}
\text{"}\textit{Local hypotheses}\text{"} \wedge \\
cc \in \mathbb{N} \wedge \\
cc \leq \textit{2147483647} \wedge \\
\text{"}\textit{Check that ...}\text{"} \\
\quad \Rightarrow
\end{array}
$$

$cc \bmod 3 \in 0 \mathinner{..} 100$

Le cœur de preuve ne contient aucune règle concernant le modulo, ce dont nous pouvons nous rendre compte par la commande *SearchRule* :

```
PRI > sr(All, (a mod b))
Searching in All rules with filter
    consequent should contain a mod b
Starting search...
Found 0 rule(s) for this filter.
```

Il est donc clair qu'il ne peut pas démontrer ce lemme. Le prouveur de prédicats (commande *PredicateProver*, `pp`) échoue également. La seule manière de résoudre ce problème est d'ajouter une règle, qui introduira la connaissance mathématique manquante. Il suffit d'ouvrir le fichier *composant.pmm* dans le répertoire de base de donnée projet pour ajouter la règle. Vous pouvez éditer ce fichier avec votre éditeur préféré, ou bien choisir l'option `Edit PO methods` dans le menu `Show/Print` de la fenêtre de situation globale du prouveur interactif. Ce choix provoque l'ouverture en édition du fichier de règles correspondant au composant en cours de preuve, de plus ce fichier est créé s'il n'existait pas.

Dans ce fichier de règle, nous tapons :

```
THEORY ModProps IS

  x: NATURAL
  =>
  x mod 3 : 0..2

END
```

Puis nous utilisons la commande *PmmCompile* (`pc`) pour charger le fichier dans le prouveur :

```
PRI > pc
Loading theory ModProps
```

Après avoir chargé les hypothèses locales par une commande *Deduction* (`dd`), nous allons utiliser la règle ajoutée dans notre démonstration. Cette règle ne convient pas directement pour le but $cc \bmod 3 \in 0 \mathinner{..} 100$, elle conviendrait pour le but $cc \bmod 3 \in 0 \mathinner{..} 2$. Nous allons donc ajouter cette dernière expression comme une nouvelle hypothèse (commande *AddHypothesis*, `ah`). Le prouveur nous demande de la démontrer, ce que nous pouvons faire avec la règle ajoutée, il faut employer la commande *ApplyRule* (`ar`). La règle produit le but $cc \in \mathbb{N}$ que le cœur de preuve démontre directement (commande `pr`), l'hypothèse nouvelle est alors acceptée et le but devient :

$cc \bmod 3 \in 0 \mathinner{..} 2 \;\Rightarrow\; cc \bmod 3 \in 0 \mathinner{..} 100$

Ce dernier but est directement démontré par le cœur de preuve, la démonstration est terminée. Le déroulement de cette démonstration est montré ci dessous tel qu'il apparaît dans la fenêtre de commandes du prouveur interactif :

```
PRI > ah(cc mod 3 : 0..2)
Starting Add Hypothesis
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        cc mod 3: 0..2
End
PRI > ar(ModProps.1,Once)
Starting Apply Rule
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        cc: NATURAL
End
PRI > pr
Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        cc mod 3: 0..2 => cc mod 3: 0..100
End
PRI > pr
Starting Prover Call
Current PO is Initialisation.1
    Proved saved Unproved
    Goal
        ...
End
```

La commande *ApplyRule* a été utilisée avec la syntaxe `ar(ModProps.1,Once)`, ce qui veut dire appliquer la règle numéro 1 de la théorie *ModProps* une seule fois (*Once* signifie *une fois*). C'est l'utilisation la plus commune de la commande *ApplyRule*, sauf pour les règles de réécriture où il faut employer `ar(Theorie.numero, Goal)` pour indiquer que la réécriture doit se faire dans le but. Pour plus d'informations sur la commande *ApplyRule* voir le manuel de référence du prouveur.

6.2.4 Le prouveur de prédicats

Le mécanisme sur lequel est basé le prouveur principal de l'Atelier B est l'inférence de règles. Autrement dit, une démonstration ne peut aboutir que s'il existe un enchaînement de règles qui fait disparaître tous les buts. Il n'y a aucune garantie théorique de complétude, de plus l'équilibre des règles entre elles est obtenu par des réglages empiriques.

Il existe d'autres méthodes de preuve. L'une de ces méthodes consiste à ramener toute preuve dans le langage élémentaire des prédicats quantifiés, puis à essayer différentes instanciations pour aboutir à une preuve par contradiction. Par exemple, voici la démonstration de $x \in A \ \wedge \ A \subseteq B \ \Rightarrow \ x \in B$:

Il faut rechercher une contradiction dans :
$x \in A \wedge$
$A \subseteq B \wedge$

$\neg(x \in B)$
Transformation en prédicats quantifiés :
$x \in A \wedge$
$\forall y.(y \in A \ \Rightarrow \ y \in B) \wedge$
$\neg(x \in B)$
En instanciant par x la deuxième hypothèse :
$x \in B$
Ce qui est en contradiction avec la dernière hypothèse.

Les performances de ce procédé peuvent être pressenties sur cet exemple : les chances d'aboutir sont grandes même si la démonstration est mathématiquement complexe, mais il faut un nombre réduit d'hypothèses pour éviter des problèmes de saturation liés à l'expansion importante lors de la transformée en prédicats quantifiés. Les choix d'instanciation possibles sont nombreux, ce qui donne des temps de preuve parfois longs. D'autre part ce procédé n'est pas très adapté aux démonstrations arithmétiques.

Le prouveur de prédicats de l'Atelier B est fondé sur ce principe, c'est lui qui a permis la validation de la base de règles mathématiques du prouveur principal. Il s'utilise par la commande *PredicateProver* que nous allons étudier.

La commande PredicateProver

En preuve interactive il est toujours possible de l'utiliser pour tenter de prouver une branche de preuve, c'est la commande *PredicateProver* (`pp`) qui lance le prouveur de prédicats sur le lemme courant, éventuellement allégé d'hypothèses, avec un temps maximum. Nous conseillons l'utilisation suivante :

Si le lemme est démontrable sans notions arithmétiques :		
S'il y a peu d'hypothèses	`pp`	preuve sur le lemme complet, maximum 60s
Si les hypothèses nécessaires sont celles qui ont un symbole commun avec le but	`pp(rp.1)`	preuve sur le lemme réduit, maximum 60s
Il y a un grand nombre d'hypothèses, mais celles nécessaires sont peu nombreuses	ramener les hypothèses intéressantes dans le but avec la commande `ah` et utiliser `pp(rp.0)`	preuve sur lemme avec hypothèses choisies, maximum 60s

Comme nous l'avons vu depuis le début de ce chapitre, la commande `pp` s'utilise seule ou au milieu d'une démonstration utilisant d'autres commandes. Nous allons présenter un court exemple de l'utilisation de *PredicateProver* avec des hypothèses réduites.

Soit à démontrer le lemme suivant :

$$EE \in \mathbb{F}(\mathbb{Z}) \wedge$$
$$\neg(EE = \varnothing) \wedge$$
$$cc \in EE$$
$$\Rightarrow$$
$$\mathsf{card}(EE \cup \{cc\}) = \mathsf{card}(EE)$$

Conformément au grand principe d'utilisation du prouveur interactif, nous commencons par faire un appel au cœur de preuve en mode réduit, puis nous examinons le but restant :

```
PRI > pr(Red)
Starting Prover Call
Current PO is Initialisation.2
    Unproved saved Unproved
    Goal
        card(EE)+1-1 = card(EE)
End
```

Comment se fait-il que le prouveur échoue sur ce but ? En fait il s'agit d'un problème de parenthésage, c'est-à-dire que l'expression $\mathsf{card}(EE) + 1 - 1$ est analysée comme l'expression $(\mathsf{card}(EE) + 1) - 1$, d'où l'absence de simplification. Sans chercher à analyser plus avant, nous nous doutons que le passage au prouveur de prédicats d'un tel but provoquera sa simplification lors de l'analyse plus complète qui permet la transformée en prédicats quantifiés. Il y a peut être beaucoup d'hypothèses, dans ce cas la commande `pp` peut être longue car toutes les hypothèses doivent être traduites. Aucune hypothèse n'est nécessaire pour démontrer ce but, nous pouvons donc utiliser `pp(rp.0)`.

```
PRI > pp(rp.0)
Starting Prover Predicate Call
Proved by the Predicate Prover
Current PO is Initialisation.2
    Proved saved Unproved
    Goal
        "‘Check that the invariant ...’" => card(EE\/{cc}) = card(EE)
End
```

La démonstration est finie, la zone du but devient verte et l'état courant est “Proved”.

6.2.5 La protection des règles manuelles

Puisque le prouveur de prédicats est bien adapté à la preuve de règles, il est naturel de l'employer pour démontrer les règles manuelles que l'opérateur ajoute.

Ceci a été implanté par la commande *Validation of Rule* (`vr`) qui permet de démontrer une règle à l'aide du prouveur de prédicat. Si l'opérateur n'ajoute que ce type de règles manuelles, la preuve est valide sans vérifications supplémentaires. Nous allons voir l'utilisation de ce principe sur un exemple.

Soit à démontrer le lemme suivant :

$$22 \in 20 \mathinner{..} 30$$
$$\Rightarrow$$
$$22 \in 1 \mathinner{..} 2 \ \lor \ 22 \in 20 \mathinner{..} 30 \ \lor \ 22 \in 300 \mathinner{..} 400$$

Bien sûr, ceci est évident. Il se trouve néanmoins que ce lemme échoue en force 0, car il n'y a pas de mécanisme pour reconnaître une hypothèse dans un but disjonctif. Comment

se débarrasser de ce but facile ? Une solution simple est d'ajouter une règle protégée avec la commande *Validation of Rule* (`vr`). Nous pouvons ajouter la règle $b \Rightarrow a \vee b \vee c$, par exemple dans la théorie NonEqui (la règle est, par exemple, NonEqui.1), dans le fichier Pmm du composant et en effectuer la démonstration en preuve interactive.

```
PRI > vr(Back, (b => (a or b or c)))
The rule was proved
```

Dans la commande ci-dessus, le mot clef `Back` indique qu'il s'agit d'une règle d'inférence normale, par opposition aux règles "par l'avant". Le second paramètre est l'énoncé de la règle. Le prouveur signale en retour que la règle a été démontrée avec succès par le prouveur de prédicats. Cette règle peut être employée avec une commande *ApplyRule* :

```
PRI > ar(NonEqui.1,Once)
Starting Apply Rule
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        22: 20..30
End
```

Il suffit alors d'un seul appel au cœur de preuve pour terminer la démonstration. Il n'y a aucune vérification complémentaire à faire sur cette règle ajoutée : elle a été démontrée par le prouveur de prédicats.

6.3 L'utilisation de l'interface du prouveur interactif

L'interface Motif du prouveur interactif permet un affichage Multi-fenêtre de la preuve. Nous allons voir comment l'utiliser.

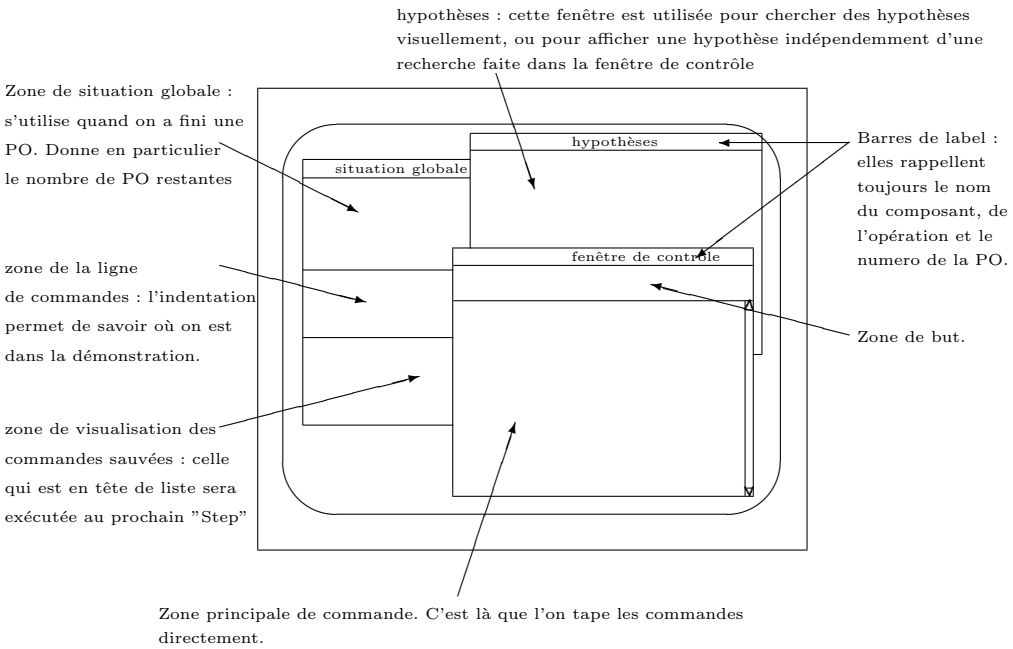
6.3.1 Organisation de l'écran

L'interface Motif du prouveur interactif est prévue pour faciliter le travail de l'opérateur dans une philosophie multi-fenêtre. c'est-à-dire que chaque sujet d'attention de l'opérateur est représenté par une fenêtre, les deux ou trois sujets en traitement étant en juxtaposition à l'écran et les sujets environnants étant rangés en icône sur les bords de l'écran. La juxtaposition de deux fenêtres débordant l'une sur l'autre permet de mettre plus d'informations à l'écran que ce que permet sa taille, la fonctionnalité Front / Back étant utilisée pour ramener immédiatement en avant la fenêtre lue.

Cette approche Multi-fenêtre nécessite une installation adéquate du système. En particulier, les commandes suivantes doivent être accessibles directement :

– icônifier ou désicônifier une fenêtre, à partir d'un endroit quelconque de sa surface ;
– faire passer une fenêtre devant ou derrière, à partir d'un endroit quelconque de sa surface.

La position normale du prouveur interactif est la suivante :

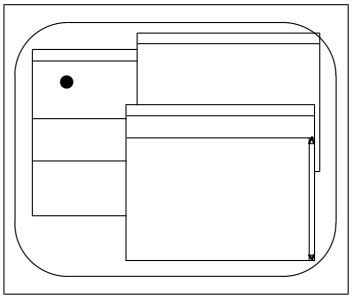


Il y a deux modes principaux d'utilisation au cours d'une session de preuve :

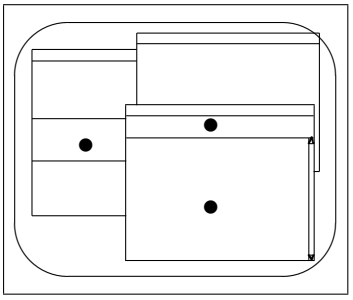
– **circulation entre les PO** : quand l'opérateur a fini une démonstration, il cherche quelle est la prochaine obligation de preuve à traiter, et combien il en reste. Son attention est alors portée sur la fenêtre de situation globale.

– **preuve** : quand l'opérateur fait une preuve, son attention est portée sur le but, sur la zone de commandes et sur la zone de la ligne de commandes qui lui permet de se positionner dans la démonstration.

Les zones d'attention dans ces deux modes sont montrés ci-après :



mode circulation

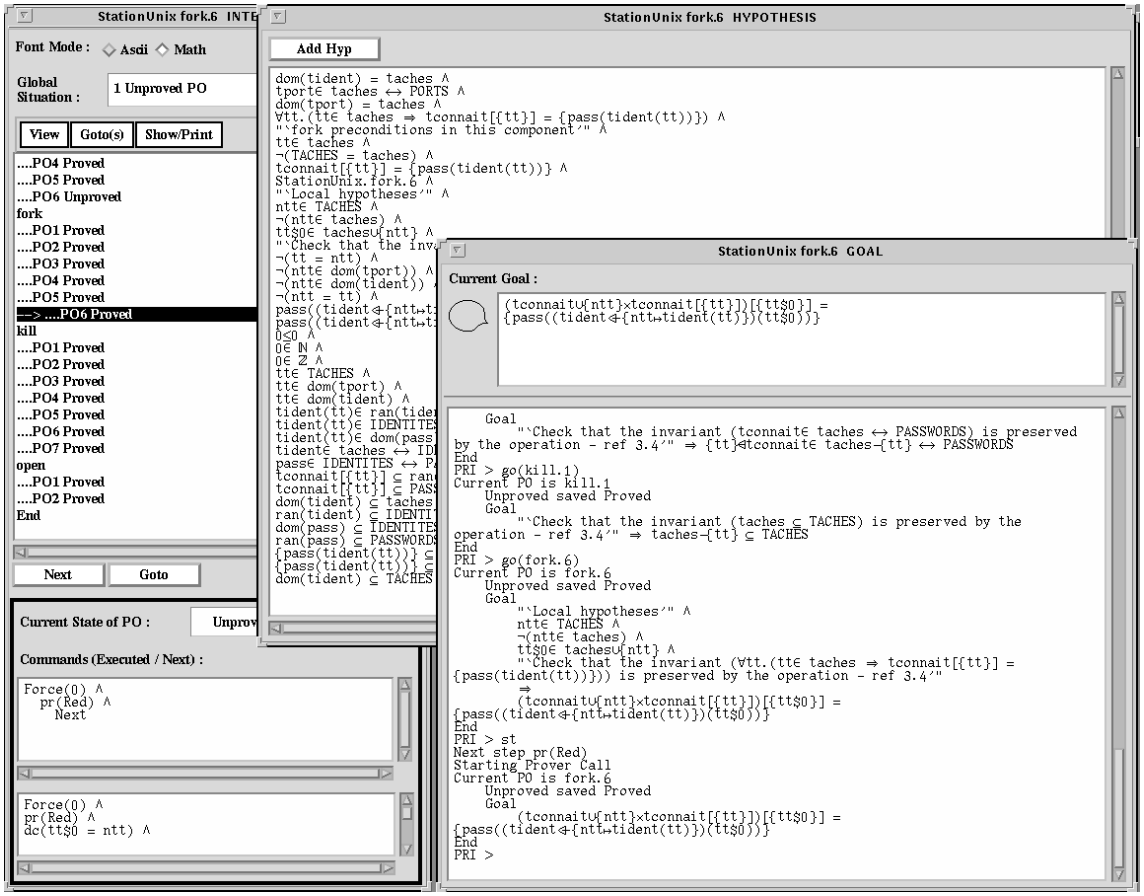


mode preuve

En mode circulation, seule la fenêtre de situation globale sert. L'opérateur regarde l'indicateur du nombre de PO non démontrées restantes qui est au dessus de la liste des PO. Il peut se déplacer dans les PO par les boutons qui sont autour de cette liste.

En mode preuve, l'opérateur fixe son attention essentiellement sur la zone but et sur la zone de commande. C'est dans la zone de commandes qu'il effectue ses recherches d'hypothèses (commande `sh`), la fenêtre des hypothèses étant plus réservée à une lecture rapide et moins dirigée. Après chaque commande de preuve, l'opérateur doit contrôler le nouvel état de preuve dans la zone de la ligne de commandes, à gauche (voir paragraphe 6.4). Durant une preuve, il arrive souvent que la lecture d'une partie du composant ou de l'un de ses niveaux adjacents soit nécessaire. C'est pourquoi il est conseillé d'avoir les fichiers correspondants en icônes sur un bord de l'écran. Lorsqu'une consultation est nécessaire, il suffit d'ouvrir ces fenêtres qui viennent devant celles du prouveur ; après lecture il faut les remettre en icône pour ne pas interférer avec l'affichage de preuve.

Ces considérations d'affichage peuvent paraître secondaires. En fait, elles influent sur la concentration de l'opérateur et ont un impact important sur l'efficacité obtenue. La méthode d'organisation de l'affichage que nous présentons n'est pas la seule possible, l'essentiel est **d'avoir une méthode d'affichage rationnelle**.



Nous montrons ci-dessus la disposition typique des fenêtres lorsque l'opérateur est concentré sur une obligation de preuve : la fenêtre de commande et du but est en avant, la zone de ligne de commande est visible à gauche. La fenêtre des hypothèses est intercalée de manière à ce que les plus récentes hypothèses soient visibles et proches par débordement à gauche, tout en gardant une taille de fenêtre suffisante. En appuyant sur une seule touche, l'opérateur peut ramener les hypothèses devant (nous ne dirons pas *quelle* touche : cela dépend de votre environnement).

6.3.2 Le prouveur interactif en mode "batch"

Comme tous les outils composant l'Atelier B, le prouveur interactif est accessible en mode "Batch". Rappelons que le mode Batch se lance par la commande `lanceBB` au lieu de `lanceAB`. Le prouveur automatique s'accède alors par la commande `pr(composant,force)` et le prouveur interactif par `br`. Le mode "Batch" est décrit dans le manuel utilisateur de l'Atelier B.

Après le lancement du prouveur interactif en mode "batch" (commande `br`), la preuve se déroule exactement comme depuis la fenêtre de commandes de l'interface Motif, mais les hypothèses et la ligne de commande sont uniquement dans les messages réponse du prouveur. Les difficultés d'utilisation du mode Batch par rapport à l'interface Motif sont :

– Comme il n'y a pas de fenêtre de situation globale, il faut utiliser la commande *Global-Situation* (`gs`) pour obtenir la liste des obligations de preuve.
– Les informations longues comme la liste des hypothèses défilent dans la fenêtre de commande au lieu d'être affichées dans des fenêtres séparées. Concernant les hypothèses, notons toutefois que lorsqu'il n'y a que des hypothèses en plus, le prouveur ne renvoie pas la liste complète mais seulement les nouvelles hypothèses.
– Le bouton d'interruption n'est pas disponible en mode Batch.

Par contre, l'utilisation du mode Batch permet d'employer un terminal non graphique et d'obtenir un affichage dans votre environnement de commande habituel (par exemple : affichage dans un `xterm`).

Nous allons présenter ci-après l'enregistrement d'une session de preuve en mode batch, correspondant à la démonstration qui a servi d'exemple au paragraphe 6.1. Pour les distinguer, les commandes entrées par l'opérateur seront soulignées. La commande pour lancer l'Atelier B en mode batch est `lanceBB`.

```
PRVB% lanceBB

Beginning interpretation ...

bbatch 1> spl

Printing Projects list ...

      LIBRARY
      passwd

End of Projects list
```

Nous n'insisterons pas sur les commandes de l'Atelier B en mode batch, celle-ci sont décrites dans le manuel utilisateur de l'Atelier B. Nous allons entrer dans le projet *passwd* :

```
bbatch 2> op passwd
bbatch 3> sml

Printing machines list ...

      StationUnix

End of machines list
```

Nous pouvons maintenant entrer dans une session de preuve interactive de la machine *StationUnix* :

```
bbatch 4> b StationUnix
No current PO
```

Nous retrouvons la présentation de la fenêtre de commandes du prouveur interactif. Une commande *GlobalSituation* permet de connaitre la liste des obligations de preuve :

```
PRI> gs

State of all PO
    Initialisation
        PO1 Proved      {} <: TACHES
        PO2 Proved      {}: {} <-> PASSWORDS
        PO3 Proved      {}: {} +-> IDENTITES
        PO4 Proved      dom({}) = {}
        PO5 Proved      {}: {} <-> PORTS
    login
        PO1 Proved      tconnait\/(tport~[{pp}]\/{ntt})*{ww}: ...
        PO2 Proved      tident<+{ntt|->pass~(ww)}: taches\/{ntt}...
        PO3 Proved      dom(tident<+{ntt|->pass~(ww)}) = taches\/{ntt}
        PO4 Proved      tport\/{ntt|->pp}: taches\/{ntt} <-> PORTS
        PO5 Proved      dom(tport\/{ntt|->pp}) = taches\/{ntt}
        PO6 Unproved    (tconnait\/(tport~[{pp}]\/{ntt})*{ww})[{tt}] ...
    fork
        PO1 Proved      tconnait\/{ntt}*tconnait[{tt}]: taches\/{ntt}...
        PO2 Proved      tident<+{ntt|->tident(tt)}: taches\/{ntt} +->...
        PO3 Proved      dom(tident<+{ntt|->tident(tt)}) = taches\/{ntt}
        PO4 Proved      tport\/{ntt}*tport[{tt}]: taches\/{ntt} <-> PORTS
        PO5 Proved      dom(tport\/{ntt}*tport[{tt}]) = taches\/{ntt}
        PO6 Unproved    (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] = ...
    kill
        PO1 Proved      taches-{tt} <: TACHES
        PO2 Proved      {tt}<<|tconnait: taches-{tt} <-> PASSWORDS
        PO3 Proved      {tt}<<|tident: taches-{tt} +-> IDENTITES
        PO4 Proved      dom({tt}<<|tident) = taches-{tt}
        PO5 Proved      {tt}<<|tport: taches-{tt} <-> PORTS
        PO6 Proved      dom({tt}<<|tport) = taches-{tt}
        PO7 Proved      ({tt}<<|tconnait)[{tt$0}] = {pass(({tt}<<|...
    open
        PO1 Proved      tport\/{tt|->pp}: taches <-> PORTS
        PO2 Proved      dom(tport\/{tt|->pp}) = taches
End
No current PO
```

Nous décidons de démontrer la sixième obligation de *fork* :

```
PRI> go(fork.6)

Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          Next
    Saved line pos 1
        Force(0) &
        dd
```

```
    Hypothesis
        "‘Component properties’" &
        pass: IDENTITES +-> PASSWORDS &
        pass~: PASSWORDS +-> IDENTITES &
        dom(pass) = IDENTITES &
        ran(pass) = PASSWORDS &
        TACHES: FIN(INTEGER) &
        not(TACHES = {}) &
        IDENTITES: FIN(INTEGER) &
        not(IDENTITES = {}) &
        PASSWORDS: FIN(INTEGER) &
        not(PASSWORDS = {}) &
        PORTS: FIN(INTEGER) &
        not(PORTS = {}) &
        btrue &
        "‘Component invariant’" &
        taches <: TACHES &
        tconnait: taches <-> PASSWORDS &
        tident: taches +-> IDENTITES &
        dom(tident) = taches &
        tport: taches <-> PORTS &
        dom(tport) = taches &
        !tt.(tt: taches => tconnait[{tt}] = {pa...
        "‘fork preconditions in this component’" &
        tt: taches &
        not(TACHES = taches) &
        tconnait[{tt}] = {pass(tident(tt))} &
        StationUnix.fork.6
    Goal
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt: ...
        =>
        (tconnait\/{ntt}*tconnait[{tt}])[{tt...
End
```

L'obligation de preuve est affichée, avec les autres informations d'état (en particulier : la ligne de commandes). Nous pouvons tenter une commande *Prove* :

```
PRI> pr

Starting Prover Call
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr &
```

```
        Next
Saved line pos 1
    Force(0) &
    dd
New Hypothesis since last command
    "‘Local hypotheses’" &
    ntt: TACHES &
    not(ntt: taches) &
    tt$0: taches\/{ntt} &
    "‘Check that the invariant (!tt.(tt: taches...
    not(tt = ntt) &
    not(ntt: dom(tport)) &
    not(ntt: dom(tident)) &
    not(ntt = tt) &
    pass((tident<+{ntt|->tident(tt)})(tt$0)): ran(pass) &
    pass((tident<+{ntt|->tident(tt)})(tt$0)): PASSWORDS &
    0<=0 &
    0: NATURAL &
    0: INTEGER &
    tt: TACHES &
    tt: dom(tport) &
    tt: dom(tident) &
    tident(tt): ran(tident) &
    tident(tt): IDENTITES &
    tident(tt): dom(pass) &
    tident: taches <-> IDENTITES &
    pass: IDENTITES <-> PASSWORDS &
    tconnait[{tt}] <: ran(tconnait) &
    tconnait[{tt}] <: PASSWORDS &
    dom(tident) <: taches &
    ran(tident) <: IDENTITES &
    dom(pass) <: IDENTITES &
    ran(pass) <: PASSWORDS &
    {pass(tident(tt))} <: ran(tconnait) &
    {pass(tident(tt))} <: PASSWORDS &
    dom(tident) <: TACHES &
    ntt|->tident(tt): TACHES*dom(pass) &
    ntt|->tident(tt): TACHES*IDENTITES &
    ntt|->tident(tt): TACHES*ran(tident) &
    pass(tident(tt)): ran(pass) &
    pass(tident(tt)): PASSWORDS &
    ntt = tt$0 &
    not(tt = tt$0) &
    not(tt$0 = tt) &
    tconnait[{tt$0}] <: ran(tconnait) &
    tconnait[{tt$0}] <: PASSWORDS
Goal
    tconnait[{tt$0}]\/{pass(tident(tt))} =
```

```
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

Notez toutes les nouvelles hypothèses dérivées qui ont été générées par le cœur de preuve. En mode batch, seules les nouvelles hypothèses sont affichées à chaque étape, sauf quand on recule dans l'arbre de preuve (par exemple à cause d'une commande *Back*, comme nous allons le voir à l'étape suivante. Nous reculons sur la commande `pr`, conformément à la méthode générale :

```
PRI> ba

Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        "‘Component properties’" &
        pass: IDENTITES +-> PASSWORDS &
        ...
        tconnait[{tt}] = {pass(tident(tt))} &
        StationUnix.fork.6
    Goal
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt: ...
        =>
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

Pour écourter ce texte, nous n'avons pas fait figurer toutes les hypothèses. Notez l'évolution de la ligne de commandes sous la clause "`command line is ...`". Nous sommes revenus au début de la démonstration, nous pouvons essayer l'application du prouveur de prédicats :

```
PRI> dd & pp

Starting Deduction
Starting Prover Predicate Call
EXECUTION ABORTED ON GOAL: long, dumping in /tmp/lastfrm...

StationUnix.but interrupted
The Predicate Prover went out of time
Current PO is fork.6
```

```
    Unproved saved Unproved
    Command line is
        Force(0) &
          dd &
            Next
    Saved line pos 2
        Force(0) &
        dd
    New Hypothesis since last command
        ...
    Goal
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

Nous allons maintenant refaire rapidement la démonstration qui a été présentée au paragraphe 6.1. Nous nous remettons tout au début par une commande *Reset* :

```
PRI> re

Resetting PO
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        ...
    Goal
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt...
        =>
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

La démonstration précédemment vue peut commencer :

```
PRI> pr(Red)

Starting Prover Call
Current PO is fork.6
    Unproved saved Unproved
```

```
    Command line is
        Force(0) &
          pr(Red) &
            Next
    Saved line pos 1
        Force(0) &
        dd
    New Hypothesis since last command
        ...
    Goal
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End

PRI> dc(ntt = tt$0)

Starting Do Cases
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              Next
    Saved line pos 1
        Force(0) &
        dd
    New Hypothesis since last command

    Goal
        ntt = tt$0 =>
          (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
          {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End

PRI> pr(Red)

Starting Prover Call
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              pr(Red) &
                Next
    Saved line pos 1
        Force(0) &
        dd
```

```
    New Hypothesis since last command
        ntt = tt$0 &
        not(tt = tt$0) &
        not(tt$0 = tt) &
        pass(tident(tt)): ran(pass) &
        pass(tident(tt)): PASSWORDS
    Goal
        (tconnait\/{tt$0}*tconnait[{tt}])[{tt$0}] =
        {pass(tident(tt))}
End

PRI> pp

Starting Prover Predicate Call
Proved by the Predicate Prover
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              pr(Red) &
                pp &
              Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        ...
    Goal
        not(ntt = tt$0) =>
         (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
         {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End

PRI> pr(Red)

Starting Prover Call
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              pr(Red) &
                pp &
              pr(Red) &
                Next
    Saved line pos 1
```

```
        Force(0) &
        dd
    New Hypothesis since last command
        ...
    Goal
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass(tident(tt$0))}
End

PRI> pp

Starting Prover Predicate Call
Proved by the Predicate Prover
Current PO is fork.6
    Proved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              pr(Red) &
                pp &
              pr(Red) &
                pp &
          Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        ...
    Goal
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt...
        =>
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

Nous pouvons quitter le prouveur interactif et l'Atelier B. Notons que la commande `h` du mode batch permet d'obtenir la liste des commandes disponibles :

```
PRI> qu

PO fork.6 saved
bbatch 6> h

General commands :
```

```
(cd  ) change_directory
(ddm ) disable_dependence_mode
(erf ) edit_res_file
(eur ) edit_users_res
(edm ) enable_dependence_mode
(h   ) help
(hh  ) html_help
(hph ) html_prover_help
(hrb ) html_rules_base
(lsb ) list_sources_b
(lrf ) load_res_file
(pc  ) print_code
(pwd ) print_working_directory
(q   ) quit
(rs  ) restore_source
(srb ) show_rules_base
(v   ) version_print


Project level commands :
(add ) add_definitions_directory
(apl ) add_project_lib
(apr ) add_project_reader
(apu ) add_project_user
(arc ) archive
(crp ) create_project
(epr ) edit_project_res
(glfa) get_list_from_archive
(ip  ) infos_project
(op  ) open_project
(rdd ) remove_definitions_directory
(rp  ) remove_project
(rpl ) remove_project_lib
(rpu ) remove_project_user
(res ) restore
(sddl) show_definitions_directory_list
(sll ) show_libs_list
(spll) show_project_libs_list
(sprl) show_project_readers_list
(spul) show_project_users_list
(spl ) show_projects_list


Machine level commands (available after open_project) :
(aa  ) ada_all                (a   ) adatrans
(af  ) add_file               (ani ) animator
(b0c ) b0check                (b   ) browse
(c++a) c++all                 (c++t) c++trans
```

```
(cc  ) ccompile               (clp ) close_project
(cdf ) create_doc_framemaker  (cdi ) create_doc_ileaf
(cdr ) create_doc_rtf         (ct  ) ctrans
(dg  ) dep_graph              (e   ) edit
(lce ) edit_clc               (ep  ) edit_pmm
(fg  ) formula_graph          (gpx ) get_project_xref
(hiaa) hia_all                (hia ) hiatrans
(hg  ) homonymy_graph         (ic  ) infos_component
(lcm ) lchecker_mach          (lcp ) lchecker_project
(m   ) make_all               (nav ) navigator
(ocg ) op_call_graph          (pov ) po_view
(po  ) pogenerate             (ppf ) pov_print_framemaker
(ppi ) pov_print_ileaf        (ppl ) pov_print_latex
(ppr ) pov_print_rtf          (psf ) pov_show_framemaker
(psi ) pov_show_ileaf         (psl ) pov_show_latex
(psr ) pov_show_rtf           (ppfi) pretty_print_file
(pdi ) print_doc_ileaf        (pdl ) print_doc_latex
(pchk) project_check          (ps  ) project_status
(pr  ) prove                  (ppp ) prove_patchprover
(ppmm) prove_pmm              (r   ) remake
(rc  ) remove_component       (sn  ) set_native
(spp ) set_print_params       (sdl ) show_doc_latex
(sml ) show_machines_list     (svf ) show_vcg_file
(s   ) status                 (sg  ) status_global
(t   ) typecheck              (u   ) unprove


bbatch 7> clp
bbatch 8> q

End of interpretation (8 lines)
PRVB%
```

6.4 La ligne de commandes

Pour guider l'opérateur dans sa démonstration, le prouveur fournit une information qui représente l'arbre de preuve sous forme d'une liste indentée : la ligne de commandes. Cette information est visible dans la zone de la ligne de commande de la fenêtre de situation globale. L'indentation des commandes dans cette zone est capitale, elle représente les différentes branches de l'arbre de preuve. Nous allons expliquer son usage sur des exemples.

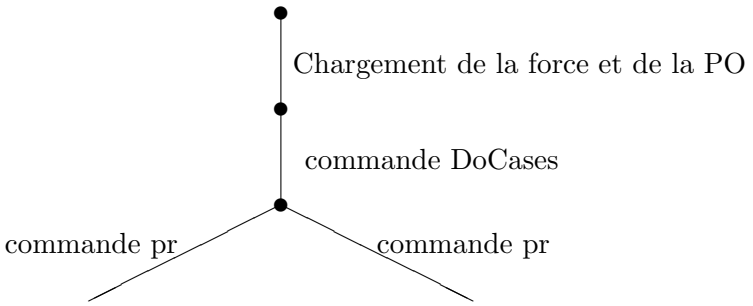
Supposons que l'affichage de la ligne de commande soit le suivant :

```
Force(0) &
  dc(var = TRUE) &
    pr &
    pr &
  Next
```

Cela indique que le déroulement de la preuve a été le suivant :

- le prouveur a été mis en force 0 ;
- dans cette force, l'opérateur a accédé à une PO. Cet accès n'est pas représenté, mais cela a créé un but sous l'action de la force 0.
- sur ce but initial, l'opérateur a lancé une commande *DoCases* (`dc`) pour faire deux cas : soit $var = \mathsf{TRUE}$ soit $\neg(var = \mathsf{TRUE})$.
- la commande *DoCases* (`dc`) a créé des buts fils, l'opérateur a fait un appel au cœur de preuve pour traiter le premier but fils.
- l'opérateur a encore fait un appel au cœur de preuve pour traiter le deuxième but fils.
- le mot clef `Next` n'est décalé que par rapport à `Force(0)`, ce qui signifie que le prochain but à traiter n'est plus un fils du but initial de la PO. Autrement dit, la preuve de cette PO est finie, ce qui est signalé par la couleur verte de la zone du but.

Dans cet exemple la démonstration aboutit. L'interface signale ce succès en colorant en vert pomme la zone du but, et en interdisant les autres commandes de preuve. L'arbre de preuve de notre exemple est :



Le lecteur l'aura compris, l'objectif en démonstration interactive est de ramener à gauche le mot clef `Next`. La position de ce mot clef permet de savoir si le but courant est un but fils du précédent. Un cas typique où le repérage par la ligne de commande est nécessaire est celui où une commande `pr(Red)` a créé plusieurs sous buts. Par exemple :

```
Force(0) &
  pr(Red) &
    ah(not(EE = {})) &
      pr &
      pr &
    Next
```

Ici l'opérateur a commencé par lancer le cœur de preuve en mode réduit. Celui-ci échoue

sur l'un de ses sous buts, l'opérateur l'aide en lui rajoutant une hypothèse (commande *AddHypothesis*), qu'il démontre par `pr`, un deuxième `pr` lui permet alors de démontrer le sous but qui bloquait. Il voit alors apparaître un nouveau but qui ne semble pas avoir de rapport avec le précédent : c'est normal puisqu'il s'agit du prochain sous but engendré par la commande `pr(Red)` initiale. Mais si l'opérateur n'est pas accoutumé à suivre sa démonstration dans la ligne de commande, il peut être surpris.

Dans notre exemple, l'appel au cœur de preuve s'est terminé lors de l'échec du premier sous but. Peut-être suffit-il de le relancer pour terminer la démonstration :

```
Force(0) &
  pr(Red) &
    ah(not(EE = {})) &
      pr &
      pr &
    pr &
  Next
```

6.5 Tactiques simples de preuve

Parmi toutes les diverses stratégies que l'opérateur utilise au cas par cas dans les démonstrations interactives, il y a des groupes de commandes qui reviennent souvent ensemble. L'usage conjoint de ces commandes semble être des sortes de *tactiques élémentaires* à partir desquelles les démonstrations complexes sont bâties.

Nous avons identifé quatre tactiques de base :

1. *Prove* et *PredicateProver*;
2. *AddHypothesis* et *DoCases*;
3. *SearchRule* et *ApplyRule*;
4. Règles manuelles et *ApplyRule*.

Ce découpage est bien entendu assez subjectif. Nous allons néanmoins présenter ces quatre tactiques.

6.5.1 Prouveur et prouveur de prédicats

Souvent, une démonstration peut être menée simplement en alternant les appels au cœur de preuve et au prouveur de prédicats. Normalement la méthode générale proposée dans ce chapitre devrait rendre implicite cette méthode, puisqu'elle stipule toujours l'essai des commandes `pr` et `pp` avant tout autre.

La commande `pp` met 60 secondes à échouer si elle ne permet pas de démontrer le but courant, ce qui peut être un peu long pour l'essayer systématiquement à chaque nouveau

but. C'est pourquoi il arrive fréquemment qu'on s'aperçoive après coup qu'un sous but pouvait être démontré par pp, alors qu'une autre démonstration plus complexe a été tentée. Le lecteur se réferera au paragraphe 6.2.4 qui donne des indices pour prévoir quand pp a des chances d'aboutir, et aussi comment lancer pp avec un délai plus faible ou sur des hypothèses réduites.

6.5.2 Ajout d'hypothèses et preuve par cas

Les commandes d'ajout d'hypothèse (*AddHypothesis*, ah) et de preuve par cas (*DoCases*, dc) permettent d'avoir une action déterminante sur la démonstration car elles donnent à l'opérateur la possibilité *d'introduire de nouvelles expressions.*

Un exemple d'ajout d'hypothèses

Soit à démontrer le lemme suivant :

$$
\begin{array}{l}
c1 \in 0 \mathinner{..} 100 \wedge \\
c2 \in 0 \mathinner{..} c1 \wedge \\
c3 \in 0 \mathinner{..} c2 \wedge \\
c4 \in 0 \mathinner{..} c3 \\
\Rightarrow \\
c1 + c2 + c3 + c4 \in 0 \mathinner{..} 400
\end{array}
$$

Le cœur de preuve en force 0 et le prouveur de prédicats échouent sur ce lemme. Conformément à la méthode générale, nous commençons par un appel au cœur de preuve en mode réduit :

```
PRI > pr(Red)

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        0<=400-c1-c2-c3-c4
End
```

Il s'agit maintenant d'obtenir des intervalles de variation qui ne dépendent plus des autres variables pour *c2* à *c4*, c'est-à-dire qu'il faut "décorréler" les variables. Nous allons suggérer au prouveur de faire cette opération, en commençant par *c2* qui ne dépends que de *c1* :

```
PRI > ah(c2<=100)

Starting Add Hypothesis
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        c2<=100
End
```

Le prouveur tente maintenant de démontrer ce premier résultat simple. Nous relançons le cœur de preuve :

```
PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        c2<=100 => 0<=400-c1-c2-c3-c4
End
```

Ce résultat intermédiaire a été prouvé directement, ce qui se voit par la réapparition du but initial sous la forme $H \Rightarrow B$ où H est l'hypothèse ajoutée. La commande `pr` ne termine toujours pas le lemme (l'essai n'est pas représenté). Nous pouvons faire monter cette hypothèse par une simple commande de déduction :

```
PRI > dd

Starting Deduction
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        0<=400-c1-c2-c3-c4
End
```

Continuons de même avec *c3* :

```
PRI > ah(c3<=100)

Starting Add Hypothesis
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        c3<=100
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        c3<=100 => 0<=400-c1-c2-c3-c4
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Proved saved Unproved
```

```
    Goal
        "‘Check ...’" => c1+c2+c3+c4: 0..400
End
```

Après l'ajout de l'hypothèse sur *c3*, le cœur de preuve a pu finir seul la démonstration comme nous le voyons dans la dernière commande `pr` (le nouvel état est `Proved saved Unproved`, ce qui veut dire que l'état courant est "Proved" et que nous n'avons pas encore sauvé). L'arbre de preuve final est :

```
  Force(0) &
    pr(Red) &
      ah(c2<=100) &
        pr &
        dd &
          ah(c3<=100) &
            pr &
            pr &
  Next
```

Dans cette démonstration, nous avons guidé le prouveur en lui proposant des expressions intermédiaires à démontrer. Nous avons pu ainsi faire aboutir cette démonstration, sans chercher aucune règle.

Un exemple de preuve par cas

Soit à démontrer le lemme suivant :

$$\begin{array}{l} \text{"‘}\mathit{Local\ hypotheses}\text{'"} \wedge \\ \mathit{xx} \in \{7,6,5,4,3,2,1\} \wedge \\ \text{"‘}\mathit{Check\ ...\ ref\ 3.3}\text{'"} \\ \Rightarrow \\ \mathit{xx} \in \{1,2,3,4,5,6,7\} \end{array}$$

Le cœur de preuve en force 0 et le prouveur de prédicats échouent sur ce lemme. En effet la seule manière de démontrer ceci est de vérifier l'égalité de chacun des ensembles $\{7,6,5,4,3,2,1\}$ et $\{1,2,3,4,5,6,7\}$, ce qui revient à faire sept cas sur la valeur de xx. De telles preuves par cas sont rarement déclenchées automatiquement, les preuves par cas multiples conduisant à des temps de preuve longs. Nous déclenchons cette preuve par cas par une commande *DoCases*, précédée par un appel réduit au cœur de preuve conformément à la méthode générale.

`PRI > pr(Red)`

```
Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 1
End
```

Inutile de chercher d'où vient le nouveau but généré, il faut clairement lancer la preuve par cas :

```
PRI > dc(xx, {7,6,5,4,3,2,1})

Do Cases on Enumerated {7}\/{6}\/{5}\/{4}\/{3}\/{2}\/{1}
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 7 => xx = 1
End
```

Nous avons utilisé la deuxième syntaxe de `dc` (voir manuel de référence du prouveur, c'est-à-dire `dc(v,E)` où `v` est une variable appartenant à l'ensemble enumérable `E`. Le prouveur doit alors démontrer $v \in E$, ici c'est directement une hypothèse, puis se place dans chaque cas $v = e_i$ pour chaque élément e_i de E. Dans notre exemple, chaque cas se démontre directement :

```
PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 6 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 5 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 4 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
```

```
        xx = 3 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 2 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 1 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Proved saved Unproved
    Goal
        "‘Local hypotheses’" &
        xx: {7,6,5,4,3,2,1} &
        "‘Check ... ref 3.3’"
        =>
        xx: {1,2,3,4,5,6,7}
End
```

Dans cet exemple, nous avons utilisé une seule commande de preuve *DoCases* pour déclencher une preuve avec plus de cas que ce qui est autorisé en preuve automatique. Les autres commandes ne sont que des appels au cœur de preuve. La preuve par cas est souvent utile dès que le lemme contient des ensembles énumérés, ou énumérables, ou des cas particuliers de valeur de variables.

6.5.3 Recherche et application de règles de la base

La base de règles de prouveur peut être utilisée en interactif, cela évite d'ajouter des règles dont la validité n'est pas sûre et qu'il faudra démontrer après. Quand le but courant semble pouvoir de simplifier par l'action d'une règle mathématique simple, on recherche la règle correspondante par la commande *SearchRule* et on l'applique par *ApplyRule*. Nous allons voir un exemple de ce procédé.

Considérons le lemme suivant :

$$
\begin{array}{l}
ff \in \mathbb{N} \pfun \mathbb{N} \land \\
ff[0 \upto 50] \subseteq 0 \upto 100 \land \\
ff[51 \upto 100] \subseteq 0 \upto 100 \\
\quad \Rightarrow \\
ff[0 \upto 50 \cup 51 \upto 100] \subseteq 0 \upto 100
\end{array}
$$

Rappelons que l'image fonctionnelle $f[E]$ désigne l'ensemble des images par f des éléments de E. L'image fonctionnelle d'une réunion d'ensembles est égale à la réunion des images fonctionnelles de chaque sous ensemble. Bien que cette règle semble claire, pour l'utiliser dans une démonstration il faudrait la démontrer rigoureusement. Heureusement, comme nous allons le voir cette règle existe dans la base de règles.

Cette règle permettrait de réécrire le but en :
$ff[0 \upto 50] \cup ff[51 \upto 100] \subseteq 0 \upto 100$
ce qui est plus directement lié aux hypothèses. Nous avons *l'intuition* que ce but plus simple serait démontré, pour le vérifier nous pouvons ajouter cette expression par une commande `ah` :

```
ah(ff[0..50]\/ff[51..100] <: 0..100)
```

Une simple commande `pr` démontre cette hypothèse. Nous avons utilisé la commande *AddHypothesis* pour tester si un but se démontre facilement. Nous pouvons revenir au début de l'obligation et rechercher la règle qui convient :

```
PRI > sr(Rewr,(f[a \/ b]))
```

Analysons la syntaxe de cette commande *SearchRule* : le mot clef `Rewr` indique que nous cherchons une règle de réécriture, afin de pouvoir transformer $ff[0 \upto 50 \cup 51 \upto 100]$ qui est une *sous formule* du but. Le filtre $f[a \cup b]$ indique que le terme de gauche de la réécriture doit contenir cette formule. La réponse du prouveur est :

```
Searching in Rewr rules with filter
    consequent should contain f[a\/b]
Starting search...
Rule list is
    SimplifyRelImaLongXY.6       (Backward)
        r[u\/v] == r[u]\/r[v]
End of rule list
```

La règle existe donc bien. Notons que les jokers r, u et v utilisés ne sont pas ceux que nous avons employés pour la recherche : le nom des jokers n'importe pas. En fait, une règle de la forme `r[u\/(f(x))] == ...` aurait également été retenue. Nous pouvons appliquer cette règle :

```
PRI > ar(SimplifyRelImaLongXY.6, Goal)
```

```
Starting Apply Rule
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
```

```
        ff[0..50]\/ff[51..100] <: 0..100
End
```

Finissons la démonstration :

```
PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Proved saved Unproved
    Goal
        "‘Check ... 3.3’" => ff[0..50\/51..100] <: 0..100
End
```

Dans cet exemple, nous avons vu comment chercher une règle dont l'idée nous est venue au lieu de l'écrire dans un fichier pmm. Cela permet d'être sûrs que la règle est valide.

La règle que nous avons utilisée appartient à une théorie dont le nom contient le mot `Long`. Les théories qui ont cette particularité regroupent des règles qui rallongent les expressions qu'elles transforment. Ces règles ne sont jamais utilisées par le prouveur automatique car elles présentent des risques de bouclage accrus. Elles sont donc uniquement réservées à un usage interactif.

6.5.4 Les règles manuelles

La dernière des techniques élémentaires de preuve est l'ajout de règles manuelles. Il faut éviter de l'employer si c'est possible, car les règles ajoutées devront être prouvées extérieurement.

La méthode pour ajouter et utiliser une règle manuelle est décrite au paragraphe 6.2.3, avec un exemple d'application. Le lecteur se reportera à ce paragraphe.

6.6 Utilisation avancée

Nous allons présenter maintenant divers éléments qui permettent une meilleure utilisation du prouveur interactif :

- La vérification finale de la preuve ;
- L'usage d'une règle d'admission ;
- L'optimisation des déplacements dans la preuve ;
- Le choix d'une force supérieure en preuve interactive ;
- La trace d'une démonstration.

6.6.1 La vérification finale de la preuve

En phase de preuve formelle, l'opérateur travaille sur chaque obligation de preuve une à une pendant un temps parfois long. Chaque obligation dont la preuve aboutit est marquée "Proved" dans les fichiers de gestion du composant et sa preuve n'est plus rejouée. Bien que ne pas rejouer les preuves déjà faites soit essentiel pour permettre un bon rendement, il y a des situations où des preuves ne peuvent plus être rejouées.

Par exemple, une démonstration peut aboutir en utilisant une règle ; si l'opérateur change cette règle lors de la démonstration d'une *autre* obligation de preuve il se peut que la démonstration initiale ne marche plus. Comme elle n'est pas rejouée, l'état de la PO reste quand même "Proved". De plus, en cours de preuve l'opérateur peut avoir ajouté des règles "provisoires", dont il n'est pas sûr. Si ces règles sont supprimées à la fin de la preuve, il faut vérifier que les démonstrations peuvent être refaites quand même. De tels rejeux de preuve se font à la fin, ou plus rarement de temps en temps sur des composants dont la preuve interactive prend plusieurs jours.

Pour effectuer ces rejeux de preuve, la méthode à suivre est la suivante :

1. Dans l'Atelier B, au niveau du projet (hors du prouveur interactif), choisir le composant concerné et sélectionner l'option *Unprove* dans le menu *Prove*. Ceci remet toutes les obligations dans l'état non prouvé.
2. Dans ce même menu *Prove*, choisir l'option *Automatic (Replay)*. Ceci provoque le rejeu de toutes les démonstrations telles qu'elles ont été enregistrées : c'est-à-dire qu'une PO démontrée en force 0 est rejouée en force 0, une PO démontrée en force 2 est rejouée en force 2, une PO démontrée en force 1 avec des commandes interactives est rejouée en force 1 avec ces commandes, etc.
3. Vérifier qu'on a bien retrouvé l'état de preuve de départ, avec le même nombre d'obligations prouvées.

Cette manipulation doit également être faite pour contrôler la preuve d'un projet lors de sa réception. Attention : ce peut être très long, le rejeu complet de quelques milliers d'obligations de preuve pouvant nécessiter beaucoup de temps machine.

6.6.2 L'usage d'une règle d'admission

Une règle d'admission est une règle manuelle qui permet de prouver n'importe quoi. Il suffit dans le fichier des règles manuelles `<composant>.pmm` d'ajouter :

```
THEORY Admis IS
  bcall(WRITE: bwritef(''A'')) => p
END
```

Le conséquent de cette règle est le joker `p`, qui coïncide avec n'importe quel but. Une telle règle est évidemment fausse. A la fin de toutes les phases de preuve, il faut la retirer et rejouer les démonstrations pour s'assurer qu'aucune d'entre elles ne l'utilise plus : voir paragraphe 6.6.1.

L'antécédent `bcall(...)` est une directive de langage de théorie qui permet d'écrire `A` comme "admis" juste avant le `+` qui signale le succès en preuve automatique, ce qui permet de trahir l'utilisation de ces règles lors d'un rejeu de preuve. On peut aussi écrire sur le terminal le but qui a été admis :

```
bcall(WRITE: bwritef(‘‘admis : %\n’’,p)) => p
```

L'usage d'une règle d'admission est vivement recommandé pour conduire efficacement la preuve. La règle d'admission s'emploie par *ApplyRule* : `ar(Admis.1,Once)`. Elle a deux usages principaux :

– le contrôle des cas : le cœur de preuve décide parfois de faire de la preuve par cas, pas toujours judicieusement. La règle d'admission permet de connaître ces cas. Par exemple :

```
pr &
  ar(Admis.1,Once) &
  ar(Admis.1,Once) &
  ar(Admis.1,Once) &
Next
```

Le premier appel au cœur de preuve avait fait trois cas. Voir paragraphe 6.8.1.

– la validation de l'utilité d'une hypothèse ajoutée : la commande *AddHypothesis* permet l'ajout d'une hypothèse démontrable à partir des hypothèses existantes. Cette démonstration peut être complexe, il vaut alors mieux être sûr de l'utilité de l'ajout avant de la faire. Par exemple :

```
ah(not(xx = e1)) &
  ar(Admis.1,Once) &
  pr &
Next
```

L'utilité de l'hypothèse $xx \neq e1$ étant ainsi établie, on peut revenir en arrière et faire sa démonstration.

L'usage de la règle d'admission oblige à faire des retours en arrière pour reprendre la démonstration, voir paragraphe 6.6.3 : il y a quelques précautions à prendre. La règle d'admission est parfois utilisée en phase de mise au point, voir paragraphe 5.4.6.

Pour des raisons de division des tâches, il peut être utile de créer des règles d'admission réduites, qui ne peuvent agir que sur certaines obligations de preuve. Par exemple :

```
THEORY Admis IS

  binhyp(C.operation1.n) &
  bcall(WRITE: bwritef(‘‘A’’))
  =>
  p;

  binhyp(C.operation2.3) &
  bcall(WRITE: bwritef(‘‘A’’))
  =>
  p

END
```

Ces règles s'appuient sur la présence d'une hypothèse non mathématique de la forme `composant.opération.numéro`, qui est systématiquement rajoutée par le prouveur lors du chargement d'une obligation de preuve. Cette hypothèse, dite hypothèse de repérage, est d'ailleurs affichée dans l'interface.

La théorie d'admission ci-dessu admet toutes les PO de l'opération opération1 et la PO numéro 3 de opération2. La directive `binhyp` du langage de théorie n'autorise l'application d'une règle que si la formule argument est trouvée dans les hypothèses. Cette théorie peut s'utiliser sans avoir à citer les numéros de règles grâce à la commande `ar(Admis)`, qui a pour effet d'appliquer toutes les règles de la théorie `Admis` tant que l'une d'elles s'applique.

Attention aux décalages de numéros : il est important de mettre les règles d'admission dans une théorie séparée. Ainsi il sera possible de les retirer à la fin sans que les autres règles perdent leur numéro, ce qui détruirait les démonstrations manuelles utilisant ces règles. En effet, l'utilisation d'une règle par une commande *ApplyRule* s'effectue en *désignant la règle par son numéro d'ordre dans sa théorie d'appartenance*. Supprimer des règles dans une théorie risque de décaler ces numéros (voir paragraphe 6.8.2).

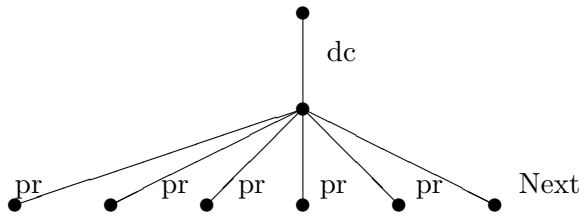
6.6.3 Le déplacement dans la preuve

Le système de pilotage de la preuve permet les retours en arrière. Il est ainsi possible de faire des démonstrations très longues, à chaque nouvelle commande le prouveur poursuit à partir de l'état précédent sans tout rejouer, et en cas d'erreur il suffit de faire des retours en arrière. Cette conduite de preuve est basée sur les fonctionnalités natives du *Logic Solver*.

Le système gestion de preuve du *Logic Solver* sous-jacent a une influence sur la performance des déplacements. En particulier, les retours en arrière sur des branches de preuves terminées est plus long. Par exemple :

```
dc(xx,{e1,e2,e3,e4,e5,e6}) &
  pr &
  pr &
  pr &
  pr &
  pr &
  Next
```

Ceci correspond à l'arbre de preuve :



Si l'opérateur utilise la commande *Back* (`ba`), le système doit retrouver le but précédent de la preuve par cas, lequel a été déchargé par la commande `pr`. Le *Logic Solver* retient tous les buts précédents de la branche courante de preuve uniquement. Retenir les branches de preuve terminées prendrait beaucoup trop de mémoire. Donc la seule manière de retrouver ce but est de remonter jusqu'à la commande *DoCases* non comprise, puis de se débarrasser des buts intermédiaires en refaisant les appels au cœur de preuve. Il y a donc 4 appels au cœur de preuve à faire, on voit le problème si chaque appel prend une minute ! Dans cet exemple précis, il vaudrait mieux éliminer les buts intermédiaires par admission au lieu de rejouer leur démonstration, mais on conçoit aisément qu'une telle optimisation n'a rien de général.

Autrement dit, une commande *Back* peut provoquer le rejeu de parties entières de démonstration pour recréer un but disparu. Ces rejeux sont d'ailleurs dénoncés par les messages de début de chaque commande rejouée : `Starting Prover Call...` etc. Par contre, la commande *Back* est immédiate si il n'y a rien à recréer, c'est-à-dire si elle remonte à un but dont le but courant est le fils. Dans l'exemple précédent, il est immédiat de remonter juste avant la commande *DoCases*.

La commande `ba(Node)` permet de remonter au plus proche but parent, elle est toujours immédiate car elle ne rejoue aucune commande. Dans l'exemple précédent `ba(Node)` permet de remonter directement avant la commande *DoCases*. En fait, une commande `ba(Node)` remonte toujours au précédent niveau d'indentation dans la ligne de commande. Par exemple, si l'état avant la commande est le suivant :

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

Pour trouver le nouvel état après une commande `ba(Node)`, il suffit de placer `Next` à la place de la plus proche commande d'indentation directement inférieure en remontant :

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

Dans cet exemple, l'opérateur aurait pu taper cinq fois `ba`, ou faire `ba(5)` pour obtenir le même résultat. Ce serait une très mauvaise méthode, beaucoup plus longue.

6.6.4 Le choix d'une force supérieure

Le choix d'une force supérieure pour faire une tentative de preuve en fonction de la forme d'une PO est toujours très spéculatif. En effet, il n'est pas question de prévoir exactement ce que va faire le cœur de preuve dans telle ou telle force : il faudrait pour cela connaître toutes les règles et tous les mécanismes. De telles tentatives sont donc guidées par l'intuition et l'expérience.

Les forces supérieures sont les forces 1, 2 et 3. Chacune contient tout les mécanismes de la force précédente, ce qui n'est pas le cas de la force 0 dont les mécanismes ne sont pas systématiquement dans la force 1. Les forces supérieures ont également des constantes de dimensionnement croissantes : la force 2 a le droit de créer plus de sous cas que la force 1, de dériver plus d'hypothèses, etc.

Pour se guider dans le choix d'une force supérieure, il est raisonnable de ne retenir que les spécificités de chaque force. Celles-ci sont :

- **Force 1** : les hypothèses sont traitées par passage dans le prouveur. Elle sont donc plus décomposées qu'en force 0, par exemple toute hypothèse de la forme $x \in a \,..\, b$ sera systématiquement transformée en $a \leq x$ et $x \leq b$.
- **Force 2** : des hypothèses spécifiques sont créés en fonction des expressions trouvées dans le but. Par exemple, si le but contient un produit $a \times b$ avec $0 \leq a$ et qu'il existe une variable c telle que $c \leq b$, alors l'hypothèse $a \times c \leq a \times b$ est ajoutée.
- **Force 3** : les règles à tentatives sont employées. Par exemple, si le but à prouver est $a \leq b$ et si $a \leq c$ est en hypothèse, alors une sous-preuve sera lancée pour tenter de démontrer $c \leq b$.

6.6.5 La trace de preuve

Nous désignons par trace de preuve les informations fournies lors d'une démonstration qui permettent de savoir comment elle s'est faite. Cette trace est particulièrement utile avec l'emploi de prouveurs automatiques, car elle constitue la justification de la démonstration.

Il y a deux sortes de traces de preuve qui ont été envisagées dans le cadre de l'Atelier B : la production de la démonstration mathématique rédigée d'une obligation démontrée, ou bien la trace pas à pas d'un appel au cœur de preuve. Ce dernier type de trace sert à comprendre les nouveaux buts qui peuvent apparaître après un appel au cœur de preuve. Sur un exemple, nous allons examiner la trace d'un appel au cœur de preuve.

Avant d'aborder l'exemple, insistons sur le fait que *le traçage est rarement employé pour faire aboutir une preuve*. En effet, le plus souvent il est inutile de comprendre exactement ce qu'a fait le dernier appel au cœur de preuve, l'essentiel étant de voir comment poursuivre sur le prochain but. Dans la plupart des cas, une trace produite au cours d'une démonstration non aboutie est une perte de temps. C'est pourquoi nous avons choisi dans l'exemple qui va suivre de tracer une obligation qui se démontre par un seul appel au cœur de preuve en force 1 ; la trace de preuve est ainsi présentée dans son vrai rôle d'information après coup.

Soit à démontrer le lemme suivant :

$$
\begin{array}{l}
PORTS \in \mathbb{F}(\mathbb{Z}) \wedge \\
PORTS \subseteq \mathbb{Z} \wedge \\
\neg(PORTS = \varnothing) \wedge \\
taches \subseteq TACHES \wedge \\
tport \in taches \ \leftrightarrow \ PORTS \wedge \\
\mathsf{dom}(tport) \subseteq taches \wedge \\
\mathsf{ran}(tport) \subseteq PORTS \wedge \\
\mathsf{dom}(tport) \subseteq TACHES \wedge \\
\mathsf{dom}(tport) = taches \wedge \\
tt \in taches \wedge \\
tt \in \mathsf{dom}(tport) \wedge \\
tt \in TACHES \wedge \\
\neg(TACHES = taches) \\
\Rightarrow \\
\{tt\} \lhd tport \in taches - \{tt\} \ \leftrightarrow \ PORTS
\end{array}
$$

Pour obtenir la trace de la démonstration, il suffit de lancer le prouveur interactif, d'accéder à cette PO (on est alors en force 1) et de taper :

```
pr(Ru.Goal.None, File)
```

Il s'agit d'une commande `pr` avec des arguments pour indiquer que nous désirons une trace de preuve présentant les buts et les règles, présentés sur le terminal et recopiés dans un fichier. L'affichage de la trace est le suivant :

```
Starting Trace in mode Ru.Goal.None , File
```

```
Starting Prover Call
  After deduction, goal is now
        {tt}<<|tport: taches-{tt} <-> PORTS
```

La déduction permet de faire monter les hypothèses locales.

```
  Attempt to prove
        not(tt: taches)
```

Le but contenant des expressions qui se simplifient si *tt* n'est pas dans *taches*, le prouveur tente de démontrer cette proposition.

```
  Obvious goal tt: taches  is discharged because in hypothesis.
  HiddenPredicate mecanism is transforming goal
       not(tt: taches)
  in
       bfalse
  Obvious goal StationUnix.kill.5  is discharged because in hypothesis.
  Obvious goal "‘Check ...4’"  is discharged because in hypothesis.
  After deduction, goal is now
        bfalse
  Attempt to prove
        bfalse
  fails.
```

La démonstration de $\neg(tt \in taches)$ échoue. En effet, le prédicat $tt \in taches$ est en hypothèse, il est donc remplacé par **btrue** dans la proposition à prouver. C'est le mécanisme *HiddenPredicate* qui fait ce remplacement. Avant de conclure à l'échec, les deux dernières hypothèses : `StationUnix.kill.5` et `"‘Check ...4’"` ont été repassées dans le prouveur pour tenter de les simplifier.

En résumé, le prouveur a fait un cycle complet pour s'apercevoir que la proposition $\neg(tt \in taches)$ est fausse, et qu'il n'est donc pas possible de simplifier le but en l'utilisant. Dans une telle trace de preuve, tout ce que le prouveur a tenté est tracé. De plus les noms des mécanismes utilisés sont cités ; à moins d'être spécialiste des mécanismes du prouveur il n'est pas question de connaitre tous ces noms. Nous conseillons donc de ne pas s'attarder sur chaque étape dans la trace, l'essentiel est de comprendre *globalement* ce qui s'est passé. Continuons la lecture :

```
  By applying atomic rule InRelationXY.1,
       dom(a): POW(s) &
       ran(a): POW(t)
       =>
       a: s <-> t
  the goal {tt}<<|tport: taches-{tt} <-> PORTS  is now
       dom({tt}<<|tport) <: taches-{tt}
   and ran({tt}<<|tport) <: PORTS
```

Une règle s'est appliquée, nous avons maintenant les deux buts ci-dessus.

```
Attempt to prove
      not(tt: taches)
Obvious goal tt: taches  is discharged because in hypothesis.
HiddenPredicate mecanism is transforming goal
     not(tt: taches)
in
     bfalse
Obvious goal "‘Check ...4’"  is discharged because in hypothesis.
After deduction, goal is now
      bfalse
Attempt to prove
      bfalse
fails.
```

Le prouveur a de nouveau tenté de prouver $\neg(tt \in taches)$.

```
Goal
     dom({tt}<<|tport) <: taches-{tt}
is simplified in
     dom(tport)-{tt} <: taches-{tt}
```

Le but a été simplifié par une règle de réécriture. Dans le mode de trace que nous avons choisi, ces règles de réécriture ne sont par tracées : on peut obtenir ces traces supplémentaires en utilisant la commande `pr(Ru.Goal.None, File, Simpl)`. Attention, cela produit des traces longues.

```
By applying atomic rule InPOWXY.24,
     a: POW(c\/b)
     =>
     a-b: POW(c)
the goal dom(tport)-{tt} <: taches-{tt}  is now
     dom(tport) <: taches-{tt}\/{tt}

Goal
     dom(tport) <: taches-{tt}\/{tt}
is simplified in
     dom(tport) <: taches
Obvious goal dom(tport) <: taches  is discharged because in
     hypothesis.
```

Le premier des deux sous buts est déchargé. Nous passons maintenant au deuxième :

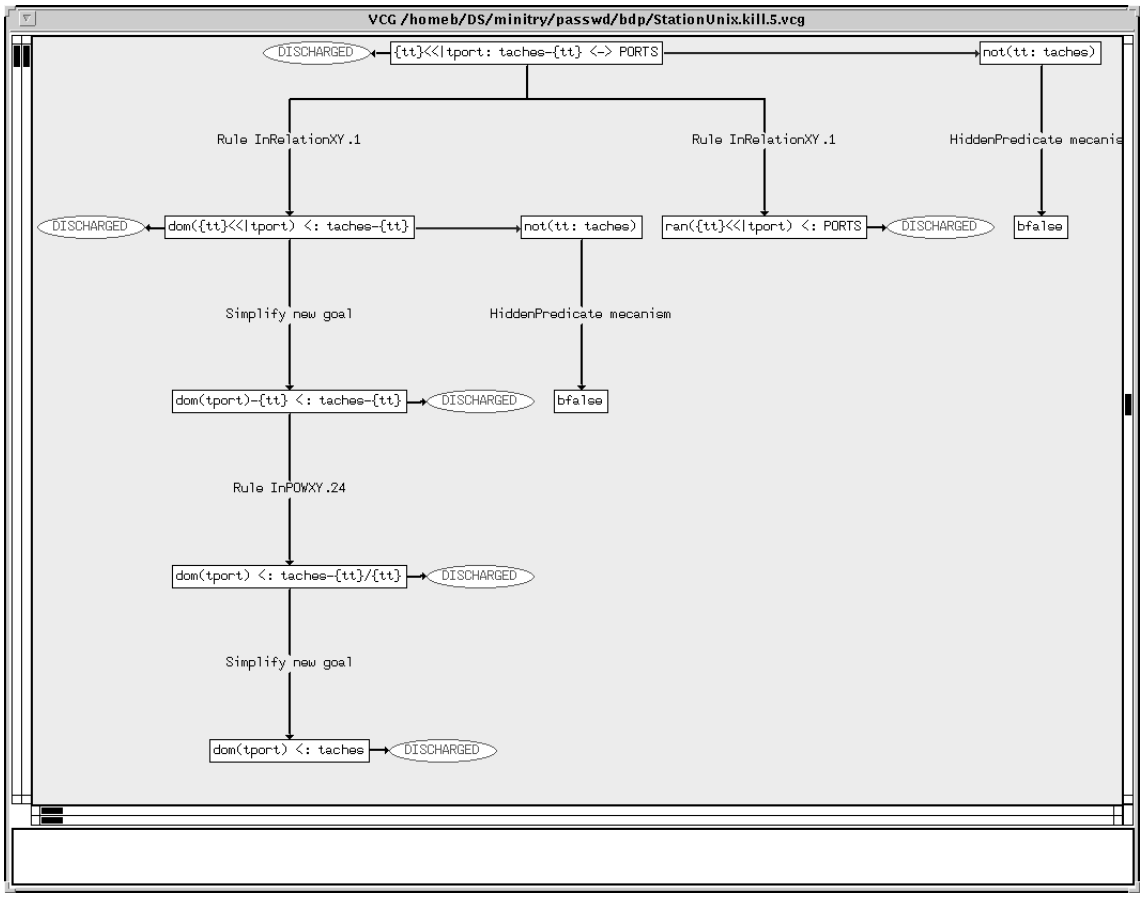
```
By applying atomic rule InPOWLeavesXY.35,
     binhyp(ran(r): POW(b))
     =>
     ran(a<<|r): POW(b)
the goal ran({tt}<<|tport) <: PORTS is discharged.

End of trace
```

```
Current PO is kill.5
    Proved saved Proved
    Goal
        "‘Check ...4’" => {tt}<<|tport: taches-{tt} <-> PORTS
End
```

La démonstration est terminée. Ces traces de preuve permettent de comprendre les démonstrations faites, mais ne constituent pas une rédaction mathématique pour les raisons suivantes :

– Les traitements préliminaires sur les hypothèses ne sont pas tracés. Dans notre exemple, l'hypothèse $\mathsf{ran}(tport) \in \mathbb{P}(PORTS)$ a servi. Cette hypothèse ne provient pas directement du composant B, il s'agit d'une hypothèse dérivée par les traitements préliminaires qui n'ont pas été tracés.
– Les tentatives inutiles figurent dans la trace.
– La trace part du but initial et le décompose dans l'ordre d'application des règles. Une démonstration mathématique au contraire, part des hypothèses et arrive à la conclusion par démonstration successive de résultats intermédiaires.

Le mode de trace est néanmoins déterminant pour faire l'analyse d'une preuve. Il est également possible d'afficher la trace de preuve dans un graphe : après avoir produit une trace dans un fichier (mode `File` de la commande `pr` précédente), utiliser l'option *Show Proof Tree* du menu *Show/Print* dans la fenêtre de situation globale du prouveur. L'affichage produit pour notre exemple est le suivant :



6.7 Les recettes de preuve

Il existe un certain nombre de méthodes adaptées à des situations de preuve, qui peuvent faciliter le succès. Ces méthodes consistent souvent à savoir conduire la preuve vers des mécanismes du prouveur, pour éviter d'avoir à faire soi-même la preuve. Un tel ensemble de méthodes est toujours perfectible, avec la pratique l'opérateur se constituera ses propres techniques. Dans ce chapitre, nous tenterons de regrouper le plus grand nombre possible de recettes efficaces.

6.7.1 Les commandes par situation

Le choix de la "bonne" commande pour faire progresser une preuve n'est pas donné par des lois systématiques (sinon ces lois auraient été intégrées aux tactiques du cœur de preuve). Nous devons donc nous contenter de donner des listes de commandes à considérer en fonction des différentes situations possibles. C'est l'objet du tableau suivant. En cours de preuve, l'opérateur pourra chercher à identifier la situation courante dans la colonne de gauche, puis essayer les commandes correspondantes.

recherche d'une démonstration intuitive	`sh` (*SearchHypothesis*), `rp` (*showReducedPo*)
à essayer en premier	`pr` (*Prove*), `pp` (*PredicateProver* surtout pour un but ensembliste ou fonctionnel)
expression clef manquante	`ah` (*AddHypothesis*), `eh` (*useEqualityinHypothesis*), repassage d'hypothèses dans le prouveur (voir paragraphe 6.7.2), application de règles de réécriture.
la démonstration intuitive se fait par cas	`dc` (*DoCases*)
ce qu'il faut faire s'exprime par une règle simple	`sr` (*SearchRule*), `vr` (*Validation of Rule*), `ar` (*ApplyRule*)
des hypothèses devraient se simplifier entre elles	refaire passer la conjonction de ces hypothèses dans le prouveur (voir paragraphe 6.7.2)
but existentiel, de la forme $\exists x.P$	`se` (*SuggestforExists*)
hypothèses contradictoires	`fh` (*FalseHypothesis*)
faire monter une hypothèse sans le cœur de preuve	`dd` (*Deduction*)
but de la forme `not(p)`	`ct` (*Contradiction*)
la démonstration à faire ressemble à une autre	`te` (*TryEverywhere*)
hypothèse clef de la forme $\forall x.(P(x) \Rightarrow Q(x))$	`ph` (*ParticularizeHypothesis*)
hypothèse $P \Rightarrow Q$ non exploitée	`mh` (*ModusponensonHypothesis*)
avant toute commande risquée	`sw` (*SaveWithoutquestion*)

6.7.2 Refaire passer des hypothèses dans le prouveur

La commande *AddHypothesis* : `ah(h)` s'utilise souvent avec le paramètre h correspondant textuellement à une hypothèse qui existe déjà. Dans ce cas, le résultat cherché n'est évidemment pas de rajouter l'hypothèse, mais simplement de la faire redescendre dans le but qui devient $h \Rightarrow B$, B étant le but courant. Au prochain appel au cœur de preuve, h recevra le traitement privilégié des hypothèses locales, en fait c'est un moyen d'attirer

l'attention du prouveur sur cette hypothèse.

Ceci est particulièrement utile si h était une hypothèse dérivée. Les hypothèses dérivées sont celles qui sont produites après normalisation, par des règles par l'avant, par exemple :

$$a = \mathsf{FALSE} \ \wedge \ a = \mathsf{bool}(P) \ \Rightarrow \ \neg(P)$$

Cette règle produit l'hypothèse dérivée $\neg P$ si $a = \mathsf{FALSE}$ et $a = \mathsf{bool}(P)$ sont en hypothèse.

Les hypothèses générées de cette façon ne bénéficient pas du même traitement que les autres, pour des raisons d'efficacité (risque de bouclage).

Il suffit donc parfois de refaire passer une hypothèse dans le prouveur pour faire aboutir directement une démonstration. La commande *AddHypothesis* est optimisée pour ce cas là : `ah(h)` si h est déjà en hypothèse crée directement le but $h \ \Rightarrow \ B$, sans demander la démonstration de h. Il faut penser à faire repasser des hypothèses par le prouveur dans les cas suivants :

- si une égalité importante semble être une hypothèse dérivée. Attention en particulier aux égalités générées par instanciation de $p \ \Rightarrow \ a = b$. Il faut alors refaire passer la nouvelle hypothèse $a = b$ dans le prouveur.
- si plusieurs prédicats se simplifient entre eux (par exemple : $\neg p \wedge \forall x.(x \in EE \wedge p \ \Rightarrow \ q)$) faire passer la conjonction de ces prédicats dans le prouveur. Cette tactique permet de faire agir le mécanisme de simplification des conjonctions / disjonctions, car celui-ci est lancé sur tout nouveau but.
- si une hypothèse dont la normalisation n'est pas terminée pourrait faire aboutir la démonstration (exemple : $\neg\neg P$ non simplifié).

L'interface du prouveur interactif est optimisée pour permettre la sélection rapide d'une hypothèse pour la refaire passer dans le prouveur :

1. Pacer le curseur a gauche de l'hypothèse choisie dans la fenêtre des hypothèses, et "tirer" vers le bas pour noircir une zone commençant par cette hypothèse ;
2. Presser le troisième bouton de la souris : la sélection se cadre automatiquement sur l'hypothèse choisie ;
3. Désigner le bouton *Add Hyp* (en haut de la fenêtre) et appuyer sur *OK* : la commande `ah` voulue est générée automatiquement.

6.7.3 Instancier $p \ \Rightarrow \ q$ si p est "presque" en hypothèse

Si $p \ \Rightarrow \ q$ est en hypothèse, et si une hypothèse est équivalente à p à une normalisation près, alors le prouveur n'aura pas produit q à cause de la différence de forme sur p. Pour le faire, nous conseillons la méthode suivante :

- faire un ajout de l'hypothèse p telle qu'elle apparaît dans $p \ \Rightarrow \ q$.
- démontrer cette hypothèse. Cette preuve devrait se faire facilement, le prouveur commençant par normaliser le but, ce qui devrait ramener p dans sa forme en hypothèses.
- le but initial B devient $p \ \Rightarrow \ B$; faire `dd` pour monter p en hypothèse tel quel.
- utiliser `ph` pour générer q

6.7.4 Penser “ah” plutôt que règle par l’avant

Les règles “par l’avant” sont des règles particulières qui permettent de générer de nouvelles hypothèses à partir d’hypothèses existantes. Par exemple, la règle $a \in \{b\} \Rightarrow a = b$ peut générer l’hypothèse dérivée $xx = 3$ à partir de l’hypothèse $xx \in \{3\}$. Dans ce manuel, nous n’abordons pas l’étude de ces règles dont l’usage est plus rare en interactif et qui provoquent facilement des bouclages.

Quand une règle par l’avant semble manquer, c’est-à-dire quand une hypothèse dérivée par une règle simple serait souhaitable, il est souvent meilleur d’ajouter cette hypothèse par *AddHypothesis*. Dans la plupart des cas, le prouveur devrait réussir à démontrer cette hypothèse qui dérive simplement des hypothèses courantes ; il suffit alors de la faire monter par `dd` ou `pr`. Dans l’exemple précédent, si d’aventure l’hypothèse $xx = 3$ venait à manquer alors que $xx \in \{3\}$ est en hypothèse, il suffirait d’ajouter $xx = 3$ par une commande *AddHypothesis*.

L’usage spécifique d’une règle est ainsi évité, le temps de recherche de cette règle est économisé.

6.7.5 Problèmes de normalisation (parenthèses)

Il arrive assez souvent que l’opérateur désire faire des transformations qui imposent de passer par des formes à l’encontre des normalisations du prouveur. Il peut alors se produire une sorte de conflit entre l’opérateur et le cœur de preuve, chacun défaisant au pas suivant ce que l’autre a fait. Nous allons exposer des solutions pour éviter ces conflits. Par exemple, supposons que l’opérateur désire transformer l’hypothèse :

$$\mathsf{dom}(gg \lhd \{xx \mapsto aa\} \lhd \{xx \mapsto bb\}) = EE$$

On suppose que la simplification des deux surcharges sur le même élément n’a pas été faite parce que le prouveur n’a pas regroupé les deux termes de droite. L’opérateur a trouvé la règle suivante par une commande *SearchRule* :

$$\{a \mapsto b\} \lhd \{a \mapsto c\} == \{a \mapsto c\} \quad \text{(SimplifyRelOveXY.13)}$$

Mais il ne peut pas l’appliquer car le parenthésage implicite de l’hypothèse concernée ne le permet pas. Plutôt que de chercher une autre règle pour changer les parenthèses, nous conseillons la méthode suivante :

- `ah(` $\mathsf{dom}(gg \lhd (\{xx \mapsto aa\} \lhd \{xx \mapsto bb\})) = EE$ `)`
- `pr` : démonstration immédiate car le cœur de preuve remet la formule sous la forme qui est en hypothèse.
- `ar(SimplifyRelOveXY.13,Goal)` : notez l’application en mode “Goal” qui signifie “but”, l’hypothèse ajoutée étant encore dans le but. Ce mode est réservé aux règles de réecriture.

Ainsi une transformation qui nécessite d’éviter les normalisations du prouveur peut-elle être faite.

6.8 Les pièges à éviter

L'usage du prouveur interactif comporte malheureusement quelques écueils, que nous allons voir maintenant.

6.8.1 Le contrôle des preuve par cas

Après chaque appel au cœur de preuve il faut contrôler que la preuve ne s'est pas placée dans un système de cas multiples inutiles. Si cela se produit vous risquez d'avoir à répéter la suite de la démonstration dans plusieurs branches de preuve, ce qui est long.

Pour contrôler le nombre de cas, le plus efficace est d'utiliser une règle d'admission (voir paragraphe 6.6.2). Si le cœur de preuve démarre des preuves par cas abusives, utiliser la commande `pr(Red)` à la place de `pr`, ce mode propre à la force 0 empêche les preuves par cas automatiques.

6.8.2 Les numéros de règles manuelles

Les règles d'un fichier de règles manuelles ne doivent pas être supprimées si il y a des règles placées après dans la même théorie, qui sont utiles. En effet ces règles changent alors de numéro d'ordre, ce qui fait que les adressages par *ApplyRule* sont décalés.

Par exemple, supposons qu'un fichier.pmm contienne trois règles :

```
THEORY MyRules IS
  r1;
  r2;
  r3
END
```

Si l'opérateur s'aperçoit à la fin de la preuve du composant que `r1` ne sert pas, il peut être tenté de la supprimer. Mais dans ce cas les commandes concernant la troisième règle (`ar(MyRules.3,...)`) sont refusées (message `MyRules.3 : no such rule`) et celles concernant la deuxième règle n'utilisent plus la bonne règle. Ces commandes ont pu être mémorisées comme démonstration d'obligations de preuve, ces dernières sont considérées comme prouvées. On ne s'aperçoit du problème que si l'état de ces obligations de preuve revient à non prouvé et que l'on tente de rejouer la preuve.

Pour cette raison il est toujours souhaitable de **Remettre à l'état non prouvé toutes les obligations de preuve d'un composant et rejouer les démonstrations**. Pour faire ceci, utiliser l'option `Unprove` du menu `Prove` de la fenêtre principale, puis l'option `Prove (replay)` du même menu. Il faut le faire quand les modifications sur les règles manuelles deviennent importantes, et en tous cas à la fin de la preuve d'un composant. Cette précaution n'est inutile que si aucune règle manuelle n'a été employée.

Il est néanmoins souhaitable de pouvoir éliminer une règle inutile pour limiter le travail de validation des règles manuelles. Nous conseillons de remplacer la règle par une règle inapplicable, par exemple :

```
THEORY MyRules IS
  empty_rule;              /* places 1 et 2 vides */
  empty_rule;
  (a - a/b) == (a mod b)  /* règle numéro 3 */
END
```

Dans l'exemple ci-dessus le mot `empty_rule` ne peut coïncider avec un but mathématique valide.

6.8.3 Le changement de force en cours de preuve

Normalement, une démonstration manuelle se fait en force 0 (voir paragraphe 6.1). Il se peut toutefois qu'au cours d'une démonstration interactive l'opérateur ait l'intuition que la difficulté de la PO correspond à une particularité de l'une des forces du prouveur, voir paragraphe 6.6.4. Il faut alors essayer la force en question avec un simple appel au cœur de preuve, sans perdre la démonstration interactive en cas d'échec.

La commande `ff` permet de se replacer dans la même position dans une force différente. Par exemple, si la démonstration actuelle est :

```
Force(0) &
  pr &
    ar(OrderXY.63,Once) &
      pr &
        Next
```

La commande `ff(1)` à cette étape indique qu'il faut refaire le chargement des hypothèses en force 1, puis refaire toutes les commandes. Le premier appel au cœur de preuve a un effet très différent en force 1, c'est pourquoi il est peu probable que le but de sortie convienne pour l'application de la règle OrderXY.63. Donc la commande *ApplyRule* va être refusée, l'état final sera :

```
Force(1) &
  pr &
    pr &
      Next
```

La commande *ApplyRule* est perdue ! C'est pourquoi il faut impérativement sauver la liste des commandes avant de changer de force. La méthode à suivre est la suivante :

– sauver la démonstration en force 0 par `sw`

– revenir au début de la démonstration par `re`
– passer en force supérieure : commande `ff(x)`
– prouver en force supérieure : tenter un appel au cœur de preuve (`pr`), éventuellement reprendre des commandes de l'ancienne démonstration (commande `st`)
– en cas d'échec, revenir en force 0 par `re`, `ff(0)` et refaire la démonstration précédente par `st(End)`

D'autre part, si une démonstration interactive en force 1 est sauvegardée sur une obligation de preuve, l'accès interactif à cette obligation provoque un passage en force 1. Il faut donc éviter de sauvegarder inutilement en force 1 pour ne pas ralentir sans raison les accès.

6.8.4 Les problèmes de chargement

vérifiez les messages de chargement : après avoir chargé le prouveur interactif et avoir accédé à la première PO non prouvée (commande *Next*), il est possible de remonter au début de la fenêtre de commandes pour voir les messages de chargement d'un éventuel fichier de règles manuelles ou prouvées par le prouveur de prédicats : ce sont les messages `Loading theory` . Les erreurs éventuelles sont citées à ce niveau. Juste après ces messages, il y a aussi la trace de la commande `gs` émise par l'interface pour obtenir la liste des obligations.

Chapitre 7

Indications Utiles pour la preuve

7.1 Poursuite de la preuve en fonction de la forme du but

Dans la plupart des cas, la forme du but n'est pas suffisante pour déterminer le type de démonstration à appliquer. Ce sont souvent, en fait, les hypothèses locales qui permettent de déterminer le type de démonstration à appliquer.

On peut toutefois établir une classification des buts et associer des commandes à tenter pour chacun d'eux :

- $\boxed{\exists x.P}$: en général, la commande `se` (*SuggestForExist*) est obligatoire, à moins que l'on ait un but du style
  $$\exists xx.(xx = 0)$$
  Dans ce cas, la commande `mp`/`pr`/`pp` réussit.
- $\boxed{P \Rightarrow Q}$: il est possible d'utiliser :
  - `dd` pour monter directement P dans la pile des hypothèses, sans traitement (simplification, normalisation) sur celles-ci.
  - `dd(0)` pour monter P dans la pile des hypothèses, après normalisation et application des simplifications de la force 0.
  - `mp`, `pr` pour simplifier le but et résoudre,
  - `pp` pour résoudre,
  - `pp(rp.0)` si Q peut se démontrer avec uniquement les hypothèses P.
- $\boxed{\forall x.(P(x) \Rightarrow Q(x))}$: il est possible d'utiliser :
  - `mp`, `pr` pour simplifier le but en $P(y) \Rightarrow Q(y)$ avec y variable(s) fraîche(s), et le résoudre.
  - `pp` pour résoudre,
  - `pp(rp.0)` si le but peut se démontrer sans hypothèses.
- $\boxed{A \vee B}$: il est possible d'utiliser :
  - `mp`, `pr` pour simplifier le but et résoudre.
  - `pp` pour résoudre.
  - `ar(SplitOr.1, Once)` si l'on veut orienter la preuve vers $not(b) \Rightarrow a$,
  - `ar(SplitOr.2, Once)` si l'on veut orienter la preuve vers $not(a) \Rightarrow b$.
- $\boxed{A \wedge B}$: il est possible d'utiliser :

  - `mp`, `pr` pour simplifier le but (décomposer) et résoudre.
- $\boxed{a \in E}$: il est possible d'utiliser :
  - `ss` pour simplifier tout prédicat ensembliste,
  - `mp`, `pr` pour simplifier le but, et le résoudre.
  - `pp` pour résoudre,
  - `pp(rp.0)` si le but peut se démontrer sans hypothèses.
- $\boxed{a = b}$: il est possible d'utiliser :
  - `ap`, `pr`, `mp`, `pp` pour résoudre
  - `eh`, si a et b apparaissent dans d'autres égalités
- $\boxed{a < b,\ a \leq b,\ a > b,\ a \geq b}$: il est possible d'utiliser :
  - `ap`, `pr`, `mp`, `pp` pour résoudre
- $\boxed{bfalse}$: il est possible d'utiliser :
  - `fh(H)` avec H hypothèse contradictoire
- $\boxed{P(x)}$ avec $x \in E$ et E ensemble énuméré ou intervalle réduit

  - `dc(x, E)` pour tenter de démontrer $P(x)$ pour toutes les valeurs possibles de x.

Toutes ces indications sont générales et n'entendent pas prendre en compte toutes les spécificités des projets de preuve imaginables. N'oubliez pas avant de vous lancer de la preuve interactive de vérifier que l'obligation de preuve que vous voulez démontrer est juste.

7.2 Quand et comment utiliser le prouveur de prédicats

7.2.1 Réduction du nombre d'hypothèses

Le prouveur de prédicats s'utilise lorsque `mp` et/ou `pr` ont échoué. Ces 3 prouveurs ont des domaines qui se recouvrent partiellement. Contrairement à `mp` ou `pr`, `pp` ne fonctionne correctement qu'avec un jeu réduit d'hypothèses.
On peut estimer que le temps de calcul suit une loi exponentielle en fonction du nombre d'hypothèses. Aussi il n'est pas envisageable d'utiliser `pp` tel quel pour démontrer une obligation d'un projet, surtout si le composant en question voit et/ou importe d'autres composants.

Il faut alors réduire le nombre d'hypothèses. Trois techniques sont utilisables :
- sélection des hypothèses ayant un symbole en commun avec le but.
  La principale limitation est que l'on ne peut pas contrôler finement la sélection et que si un des symboles du but apparaît dans beaucoup d'hypothèses, `pp` va s'avérer inefficace.
  Exemple : `pp(rp.1)` pour sélectionner les hypothèses qui ont un symbole en commun avec le but.
- sélection des d'hypothèses en fonction de leur origine.
  Les hypothèses sont sélectionnées par paquets, sans qu'il soit possible d'en retirer.
  Exemple : `pp(rp(loc+inv))` pour sélectionner les hypothèses locales et l'invariant du composant.
- sélection manuelle des hypothèses.

On sélectionne les hypothèses par la commande `ah(H)`. Les hypothèses sont déjà présentes dans la pile des hypothèses. Le but B devient

$$H \Rightarrow B$$

Lorsque toutes les hypothèses sont incluses dans le but de cette manière (le but est de la forme

$$H_n \Rightarrow (H_{n-1} \Rightarrow (... \Rightarrow B)...))$$

on fait `pp(rp.0)`.

Par ailleurs, il est important de ne pas oublier d'inclure les hypothèses de bonne définition du lemme à démontrer. Ces hypothèses permettent souvent la démonstration du lemme en question.

Par exemple, si le but contient l'expression

$$f(x)$$

on ajoutera les 2 hypothèses

$$x \in dom(f)$$
$$f \in A \nrightarrow B$$

7.2.2 Limitation du temps de calcul

Il est possible de régler le temps de coupure de `pp` (temps de calcul maximum alloué à `pp`). Une bonne valeur du temps de coupure est 10 s. Cette valeur permet, par la commande `te` (*TryEveryWhere*), de tester rapidement `pp` sur de nombreuses obligations de preuve. Utiliser de manière systématique une valeur plus élevée (par exemple 300 s) est envisageable mais le gain en taux de preuve restera certainement modeste. Ce temps de coupure n'est pas utilisable en preuve automatique, lors de l'utilisation de la User_Pass.

7.2.3 Utilisation de pp à bon escient

L'utilisateur est parfois amené à se poser la question : dois-je utiliser `pp` ou ajouter une règle ?

Il n'y a pas de réponse absolue à cette question. Ce sera à l'utilisateur de décider en final laquelle des deux options choisir, en fonction de son expérience et de ses préférences. Il est toutefois possible de fournir des éléments de réponse pour chaque approche :

- une démonstration interactive pure (sans ajout de règle manuelle) ne nécessite pas de vérification supplémentaire, mais peut exiger la réalisation de nombreux pas de preuve, et se traduire par une durée de preuve importante. Il ne faut pas perdre de vue que ce travail de preuve peut s'avérer coûteux si des modifications des modèles B entraînent des pertes de démonstration par modification des obligations de preuve.
- l'ajout d'une règle permet souvent de gagner du temps en preuve interactive, surtout si la base de règles est insuffisante pour traiter le but à démontrer. Cette règle ajoutée devra par contre être soigneusement démontrée par la suite (de préférence par une personne différente du rédacteur) afin d'éviter de démontrer des obligations de preuve fausses et invalider ainsi le développement en cours.

7.2.4 Utilisation dans les tactiques de preuve

`pp(rp.0)` peut être utilisé dans la `User_Pass`. Il faut alors introduire dans le fichier `PatchProver` (localisé dans le répertoire bdp du projet) ou dans le fichier `Pmm` du composant, la théorie `User_Pass` puis indiquer une tactique de preuve par ligne.

Soir, par exemple, le composant *tests*, dont le fichier pmm associé contient la théorie User_Pass :

```
THEORY User_Pass IS
    mp;
    pp(rp.0) ;
    dd(0) & ap
END
```

Cette théorie définit les démonstrations interactives que l'opérateur désire voir essayées. On voit que `pp(rp.0)` sera essayé après `dd(0) & ap`.

Chaque ligne, du bas de la théorie vers le haut, sera alors appliquée tant qu'il reste une obligation de preuve non prouvée.

On obtient alors, par sélection du bouton *Prove - User_Pass*, une trace d'exécution comme suit :

```
Loading theory User_Pass
Proving tests
  Proof pass User_Pass.1, still 8 unproved PO
    clause Initialisation
    ---
    clause AssertionLemmas
    -++-
    clause op
    -

  Proof pass User_Pass.2, still 6 unproved PO
    clause Initialisation
    ---
    clause AssertionLemmas
    +-
    clause op
    -

  Proof pass User_Pass.3, still 5 unproved PO
    clause Initialisation
    ++-
    clause AssertionLemmas
    +
    clause op
    -
```

On s'aperçoit que :
– `dd(0) & ap` (`User_Pass.1`) permet de démontrer 2 obligations de preuve,
– `pp(rp.0)` (`User_Pass.2`) permet de démontrer 1 obligation de preuve,
– `mp` (`User_Pass.3`) permet de démontrer 3 obligations de preuve,
Par contre, le paramètre de coupure du prouveur de prédicats ne peut plus être utilisé dans ces conditions.

Il est toutefois possible de superviser le temps de calcul du prouveur de prédicats et l'empêcher de passer trop de temps sur certaines obligations de preuve. Il faut pour cela que l'opérateur, après avoir lancé la preuve automatique *Prove - User Pass*, appuie sur le bouton *Interrupt - Next PO* à chaque fois qu'il considère que le temps de preuve est trop long.

7.3 Application de règles manuelles

Pour l'application de règles manuelles, plusieurs conseils peuvent être utiles :

- ne pas oublier de normaliser correctement les règles :
  - les règles des fichiers Pmm sont normalisées par le prouveur au chargement,
  - les règles du fichier PatchProver doivent être normalisées par le rédacteur de ces règles avant toute utilisation.
  - $a < b$ est normalisé par le prouveur en $a + 1 \leq b$. Aussi la garde $\mathsf{btest}(a < b)$ sera réécrite en $\mathsf{btest}(a + 1 \leq b)$ et ne réussira jamais.
- dans le cas d'expressions complexes, n'hésitez pas à surparenthéser les termes de votre règle.
- vérifiez que vous n'utilisez que des jockers (variables à une seule lettre).
  La règle :
  $$\mathsf{binhyp}(xx = 0) \Rightarrow xx \bmod 2 = 0$$
  ne s'appliquera que si le but est strictement $xx \bmod 2 = 0$, mais pas s'il s'agit de $yy \bmod 2 = 0$.
- vérifiez que vous n'introduisez pas de jockers "morts" (non instanciés) dans votre règle.
  En pratique, tous les jockers apparaissant dans le conséquent d'une règle Backward sont instanciés. Il faut s'assurer, dans le cas du remplacement d'un but par un but équivalent, que le but généré est complètement instancié.
  Par exemple, la règle :
  $$\mathsf{binhyp}(H) \ \wedge \ (h \ \Rightarrow \ B) \Rightarrow B$$
  va produire, pour le but
  $xx = 0$
  le but dérivé incohérent
  $b \Rightarrow xx = 0$
  Autre exemple, la règle :
  $$\mathsf{binhyp}(b) \Rightarrow b$$
  est fausse car le lemme mathématique associé
  $$(b \ \Rightarrow \ B)$$
  est faux.
- si le but est de la forme
  $H \Rightarrow Q$
  et que l'on désire utiliser les hypothèses H pour résoudre Q, il faut d'abord les faire monter dans la pile des
  grace à la commande `dd` ou `dd(0)`.
- une règle backward (en arrière) est de la forme :
  $$garde(s) \ \wedge \ sous - but(s) \Rightarrow but$$
  qui signifie que but est transformé en $sous - but$ si la $garde$ est vraie.
  $garde(s)$ et $sous - but(s)$ peuvent être inexistants. On parle alors de règle atomique. Par exemple :
  $$a - a = 0.$$
- une règle de réécriture est de la forme :

$$garde(s) \ \wedge$$
$$sous - but(s)$$
$$\Rightarrow$$
$$formule1 == formule2$$

qui signifie que $formule1$ est transformé en $formule2$ si la *garde* est vraie et si le $sous - but$ est démontré.

$garde(s)$ et $sous - but(s)$ peuvent être inexistants. On parle alors de règle atomique. Par exemple :

$$a + 1 - 1 == a.$$

Les gardes sont là pour restreindre le champ d'application de la règle aux cas où elle est juste. Les gardes les plus utiles sont :

- $\mathsf{binhyp}(H)$: vraie si H est en hypothèse,
- $\mathsf{btest}(a \ op \ b)$: vraie si $a \ op \ b$ est vrai (op doit appartenir à $=,<,>,\leq,\geq$ et a, b doivent être des identificateurs B ou des numériques)
- $\mathsf{bnot}(G)$: vraie si G est faux.
  Attention ! pour $bnot(btest(a = b))$, le test ne porte que sur la valeur littérale de a et b. Donc cette garde peut réussir si a et b sont littéralement différents mais être identiquement valués.
  On pourrait avoir par exemple
  $\mathsf{binhyp}(a = 1) \wedge \mathsf{binhyp}(b = 1)$
  évalué à vrai. Attention donc.
- $\mathsf{bnum}(a)$: vrai si a est un entier naturel plus petit que $MAXINT$.
- $\mathsf{bgoal}(G)$: vrai si le but courant est de la forme G.
- $x \backslash P$: vraie si x n'a pas d'occurrence libre dans P.
  $x \backslash (x + 3)$ est faux
  $x \backslash (\forall x.(x \in E \ \Rightarrow \ P(x)))$ est vrai
- $\mathsf{blvar}(Q)$: Q est instancié avec la liste des variables actuellement quantifiées.

Ces 2 dernières gardes sont nécessaires pour la rédaction de règles de réécritures non atomiques. En effet, il faut faire attention à ne pas capturer de variable.
Par exemple, si l'on fait appel à un binhyp, il faut vérifier que les variables instanciées sont non libres dans la liste des variables quantifiées grace à la combinaison des gardes $\mathsf{blvar}(Q)$ et $Q \backslash x$.
Par exemple, soit l'obligation de preuve fausse :

$$xx \in NAT \ \wedge$$
$$xx = 0 \ \wedge$$
$$xx + 1 = 1 \ \wedge$$
$$xx + 2 = 2$$
$$\Rightarrow$$
$$\forall xx.(xx + 1 \in \mathbb{N} \ \Rightarrow \ xx + 2 \in \mathbb{N})$$

et la règle

$$\mathsf{binhyp}(a + b = c)$$
$$\Rightarrow$$
$$(a + b == c)$$

qui remplace $a + b$ par c.
L'application de cette règle sur le but va transformer celui-ci en

$$\forall xx.(1 \in \mathbb{N} \ \Rightarrow \ 2 \in \mathbb{N})$$

qui est vrai. L'erreur vient de la confusion entre la variable xx dans la pile des hypothèses et la variable quantifiée xx (muette).
La règle devrait être gardée comme suit :

$$\begin{array}{l}\mathsf{binhyp}(a+b=c)\ \wedge\\ \mathsf{blvar}(Q)\ \wedge\\ Q\backslash(a,b,c)\\ \Rightarrow\\ (a+b==c)\end{array}$$

Concernant ce type de règle, il faut bien faire attention à ce que l'on fait.
Par exemple, la règle

$$\begin{array}{l}\mathsf{binhyp}(a=b)\ \wedge\\ \mathsf{blvar}(Q)\\ \Rightarrow\\ a==b\end{array}$$

est fausse. En effet, le symbole \$ est vu comme un opérateur. Donc $aa\$0$, qui est un identificateur parfaitement acceptable, peut être décomposé en aa, \$ et 0.
Supposons que les hypothèses suivantes sont dans la pile des hypothèses :

$$\begin{array}{l}aa=1\\ 0=xx\end{array}$$

En appliquant la règle ci-dessus 2 fois, il devient possible de transformer l'identificateur $aa\$0$ en $1\$xx$!

Il faut dans ce cas manipuler le prédicat complet. Par exemple, la règle

$$\begin{array}{l}\mathsf{binhyp}(a=b)\ \wedge\\ \mathsf{blvar}(Q)\\ \Rightarrow\\ (0leqa==0leqb)\end{array}$$

est parfaitement valide.

7.4 Ajout de règles utilisateur

Il est possible qu'une règle ne s'applique pas lors de l'utilisation de la commande `ar`. Il faut savoir que la recherche de règles pouvant s'appliquer sur le but (commande `sr`) propose des règles pouvant EVENTUELLEMENT s'appliquer. Il faut ensuite vérifier que les gardes de ces règles sont effectivement évaluées vraies.

Une règle peut ne pas s'appliquer car :
– elle est mal normalisée (règle utilisateur). Il faut alors la corriger.
– les gardes ne se déclenchent pas
– une hypothèse n'existe pas,au préalable, et vérifier que les gardes soient vraies.
– le but n'a pas exactement la forme du conséquent de la règle :
  – il faut voir si une autre règle ne peut pas s'appliquer,

- le but peut être réécrit sous une autre forme acceptable, qui permettra l'application de la règle.
  Imaginons que le but contienne l'expression E alors que la règle attend l'expression E'. Il est possible de d'ajouter l'hypothèses $E = E'$ (`ah(E=E')`).
  Cette égalité peut être démontrée par `pp(rp(0))`, `ss`, `mp` ou `pr`. Une fois cette égalité démontrée, elle est montée dans la pile des hypothèses (commande `dd`).
  Cette égalité est ensuite appliquée au but (`eh(E)`). La règle peut alors être appliquée.
- vérifier de ne vous être pas trompé en ajoutant une hypothèse (mauvaise écriture). Dans ce cas, il faut revenir en arrière jusqu'à cet ajout d'hypothèse, qu'il faut corriger puis repartir en avant jusqu'au point où l'on s'est arrêté dans la démonstration interactive.

De manière systématique, il faut vérifier la bonne écriture des règles que l'on ajoute. Pour cela, on utilise la commande `sr(theorieutilisateur,a)`. Toutes les règles utilisateurs contenues dans la théorie *theorieutilisateur* seront alors affichées.

Par exemple, la commande `sr(tt,a)` permet d'afficher toutes les règles de la théorie *tt* :

```
PRI > sr(tt,a)
Searching in tt rules with filter
    consequent should contain a
Starting search...
Rule list is
    tt.1
        binhyp(s: seq(T))
        =>
        (size(s) = 0 == s = {})
End of rule list
```

Il ne faut pas oublier que les règles ajoutées dans un `pmm` en cours de preuve interactive ne sont prises en compte qu'après avoir exécuté la commande `pc`.
De même, n'oubliez pas que si vous supprimez une règle au milieu d'une théorie, l'ordre des règles va être modifié et donc la commande `ar(regle.n, Once)` peut ne plus s'appliquer car du fait du décalage des règles, la règle *regle.n* n'est plus celle que l'on croit. De plus, si vous ajoutez/retirez/modifiez une ou plusieurs règles de votre théorie, la commande `ar(theorieutilsateur)` peut ne plus s'appliquer.

D'une manière générale, lors de modifications de vos théories utilisateurs, il est prudent de déprouver le composant considéré puis de lancer un *Prove - Replay*, pour vérifier qu'il n'y a pas de régression de preuve.

Pour le fichier `PatchProver`, n'oubliez pas que si vous le modifiez en cours de preuve, il ne sera pas rechargé même si vous quittez le prouveur interactif. Il faut pour cela quitter le projet puis rouvrir le projet et lancer à nouveau le prouveur interactif.

On ajoute une règle lorsque les prouveurs et solveurs n'arrivent pas à résoudre ou à simplifier de manière satisfaisante le but courant. Il peut s'agir d'une proposition logique simple mais contenant des expressions complexes, qui vont gêner la mise en œuvre efficace des heuristiques de `pp`.
Dans ce cas, on ajoute une règle correspondant au but, dont les expressions complexes

auront été squelétisées (on remplace ces expressions par des variables). La règle doit toujours être simple car elle devra être démontrée par la suite.

Lors de l'ajout de règles, il faut se poser certaines questions quant à la rédaction de la règle :
– ma règle n'est elle pas trop spécifique ?
– faut-il écrire une règle complexe ou plusieurs plus simples ?
– y a t'il une règle de la base de règles qui peut être utilisée sous une forme un peu différente ?

7.5 Faciliter la preuve en ajoutant des informations dans le modèle B

Il est possible d'ajouter des informations dans le modèle B qui peuvent faciliter le travail de preuve. Il s'agit des clauses ASSERTIONS, PROPERTIES et ASSERT.

Ces 3 clauses ont des portées différentes :
– substitution, opération pour ASSERT,
– composant pour ASSERTIONS et PROPERTIES,
PROPERTIES permet de caractériser les constantes utilisées (typage + expression de propriétés).
ASSERTIONS permet de caractériser les variables du composant (une assertion doit pouvoir être démontrée sous l'hypothèse que l'invariant et les assertions précédentes sont vraies).

Il y a différentes manières d'exprimer ces propriétés. Certaines sont facilement démontrables, d'autres moins.
En fait, l'ajout d'une assertion (A) correspond à ajouter systématiquement l'hypothèse A à toutes les obligations de preuve du composant.

L'ajout de telles propriétés est une conséquence du travail de preuve et nécessite une bonne expérience du prouveur et de son fonctionnement ; ceci afin d'éviter d'ajouter des assertions qui ne seront d'aucune utilité car mal exprimées ou ne modifiant pas le chemin de preuve.

Il faut toutefois faire attention lors de la modification de ces assertions car il est possible de provoquer des régressions de preuve.

Supposons qu'une obligation de preuve P soit démontrée de manière interactive sous les hypothèses H et les assertions A. Que se passe t'il si une assertion B est ajoutée ?

En toute logique, si

$$H \ \wedge \ A \ \Rightarrow \ P$$

est vrai, on a obligatoirement

$$H \ \wedge \ A \ \wedge \ B \ \Rightarrow \ P$$

L'obligation de preuve est donc toujours démontrée et sa démonstration interactive est conservée.

Si maintenant une assertion est supprimée car jugée inutile, soit l'assertion B, et que l'on avait

$$H \ \wedge \ A \ \wedge \ B \ \Rightarrow \ P$$

alors on ne peut plus conclure a priori sur

$$H \ \wedge \ A \ \Rightarrow \ P$$

Le générateur d'obligations de preuve considère que cette obligation de preuve n'est plus prouvée mais n'efface pas la démonstration interactive sauvegardée. Dans ce cas, le déclenchement du prouveur automatique en mode *Prove - Replay* permet de reprouver cette obligation de preuve.

Supposons que cette suppression d'assertion est couplée à d'autres modifications du composant qui induisent une modification du nombre d'obligations de preuve pour le composant considéré. Dans ce cas précis, la démonstration interactive précédente peut être considérée comme perdue car associée désormais à une autre obligation de preuve (à cause du décallage des obligations de preuve) qui a a priori bien peu de chance d'être démontrée par cette démonstration interactive.

7.6 Utilisation de la commande Do Cases

La commande `dc` est indispensable lorsque :
– $P(x)$ doit être démontré avec $x \in E$ (E est un intervalle) (commande `dc(x,E)`)
– $P(x)$ doit être démontré sous les hypothèses $(A \Rightarrow Q(x)) \wedge (not(A) \Rightarrow R(x))$, et Q et R permettent de résoudre $P(x)$. `dc(A)`
Cette commande doit être appliquée si `mp`/`pr`/`pp` n'a pas réussi ou ne réussira visiblement pas à résoudre.
Attention ! La commande `pr` peut déclencher des preuves par cas, en fonction du but et de certaines hypothèses. Ces preuves par cas peuvent ne pas être fondées et nécessiter de prouver plusieurs fois le même but (du au fait que les heuristiques de pr sont générales et peuvent être sous-optimales dans certains cas).

Exemple : soit l'obligation de preuve

```
ETATS = {E0, E1, E2} &
xx: ETATS &
"‘Local hypotheses’" &
xx = E0 => xx$1: {E0,E1} &
xx = E1 => xx$1: {E0,E1,E2} &
xx = E2 => xx$1: {E1,E2} &
"‘Check that the invariant (xx: ETATS) is preserved by
the operation - ref 3.4’" &
=>
xx$1: ETATS
```

On exécute d'abord mp afin que les hypothèses soient montées dans la pile des hypothèses. Le but devient :

```
PRI > mp
Starting Prover Call
        xx$1: ETATS
```

En examinant les hypothèses, on s'aperçoit que le domaine de $xx\$1$ est dépendant de la valeur de xx. Il faut donc déclencher une preuve par cas pour xx décrivant toutes les valeurs de l'ensemble énumérés $ETATS$.

```
PRI > dc(xx,ETATS)
Do Cases on Enumerated {E2,E1,E0}
        xx = E2 => xx$1: ETATS
```

Le premier but $xx \in ETATS$ a été trivialement démontré par le prouveur. Les 3 cas vont maintenant être générés. Il faut résoudre le premier d'entre eux :

$$xx = E2 \Rightarrow xx\$1 \in ETATS$$

On peut réaliser un appel au prouveur automatique (`mp` ou `pr`). On peut aussi réaliser un appel au prouveur de prédicats (`pp(rp.0)`), à condition d'ajouter l'hypothèse locale

$$xx = E2 \Rightarrow xx\$1 \in \{E1, E2\}$$

avec la commande `ah` :

```
PRI > ah(xx$1: {E1,E2})
Starting Add Hypothesis
        xx$1: {E1,E2} => xx$1: ETATS
```

L'appel au prouveur de prédicats permet de démontrer ce premier but puis de passer au second cas.

```
PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
        xx = E1 => xx$1: ETATS
```

On ajoute l'hypothèse

$$xx = E1 \ \Rightarrow \ xx\$1 \in \{E0, E1, E2\}$$

puis on active le prouveur de prédicats.

```
PRI > st
Next step ah(xx$1: {E0,E1,E2})
Starting Add Hypothesis
        xx$1: {E0,E1,E2} => xx$1: ETATS

PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
        xx = E0 => xx$1: ETATS
```

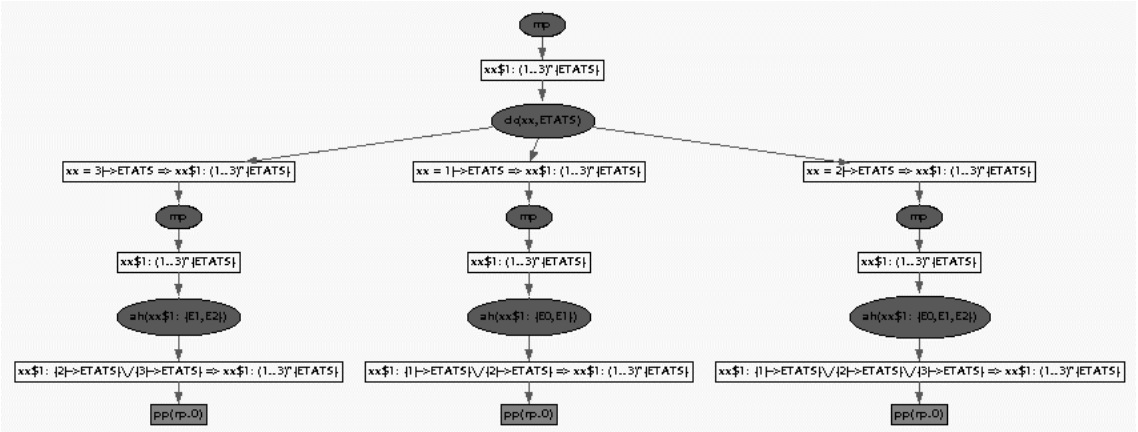
Le second sous-but est prouvé. Le troisième sous-but est alors à démontrer. Il suffit d'ajouter l'hypothèse

$$xx = E0 \ \Rightarrow \ xx\$1 \in \{E0, E1\}$$

puis d'exécuter à nouveau le prouveur de prédicats afin de démontrer le dernier but.

```
PRI > ah(xx$1: {E0,E1})
Starting Add Hypothesis
        xx$1: {E0,E1} => xx$1: ETATS

PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
```

L'arbre de preuve de la démonstration est le suivant :



7.7 Application : premier exemple

Soit les composants

```
MACHINE
    M1
VISIBLE_VARIABLES
    tab
INVARIANT
    tab: 0..7 --> 0..1
INITIALISATION
    tab := (0..7) * {0}
OPERATIONS
    op =
        BEGIN
        tab: (
            tab: 0..7 --> 0..1 &
            tab(0) = 0 &
            tab(1) = 1 &
            tab(2) = 1 &
            tab(3) = 0  &
            tab(4) = 1 &
            tab(5) = 0 &
            tab(6) = 1 &
            tab(7) = 0
            )
        END
END
```

et

```
IMPLEMENTATION
    M1_i
REFINES
    M1
INITIALISATION
    tab := (0..7) * {0}
OPERATIONS
    op =
        BEGIN
            tab(0) := 0 ;
            tab(1) := 1 ;
            tab(2) := 1 ;
            tab(3) := 0 ;
            tab(4) := 1 ;
            tab(5) := 0 ;
            tab(6) := 1 ;
            tab(7) := 0
        END
END
```

L'état du projet, après avoir exécuté la preuve des 2 composants en force 0, est le suivant :

```
Project status
+-----------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| COMPONENT | TC | POG | Obv | nPO | nUn | %Pr | BOC |  C  | Ada | C++ |
+-----------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| M1        | OK | OK  |   3 |   2 |   0 | 100 |  -  |     |     |     |
| M1_i      | OK | OK  |   7 |  18 |   8 |  55 |  -  |  -  |  -  |  -  |
+-----------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| TOTAL     | OK | OK  |  10 |  20 |   8 |  60 |  -  |  -  |  -  |  -  |
+-----------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Après avoir démarré le prouveur interactif, la commande **gs** permet de connaître la forme des obligations de preuve restant à démontrer :

```
PO9 Unproved    tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0}: 0..7 +-> 0..1
PO11 Unproved   (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(0) = 0
PO12 Unproved  (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(1) = 1
PO13 Unproved  (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(2) = 1
PO14 Unproved  (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(3) = 0
PO15 Unproved  (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(4) = 1
PO16 Unproved  (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
```

```
                  <+{5|->0}<+{6|->1}<+{7|->0})(5) = 0
    PO17 Unproved  (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                  <+{5|->0}<+{6|->1}<+{7|->0})(6) = 1
```

7 obligations de preuve ont même forme. La recherche de règles de la forme

$$(f < +g)(x) = c$$

par la commande `sr(All, ((f<+g)(x)=c))` ne permet pas de découvrir de règle directement applicable. Plutôt que de réaliser 7 preuves interactives successives, l'ajout d'une règle de simplification peut être envisagée. Le fichier M1_i.pmm contient maintenant :

```
THEORY func IS
    bcall1(BackwardRule(func.1)) &
    bnum(a) &
    bnum(c) &
    bnot(btest(a=c)) &
    (f(c) = d)
    =>
    ((f<+{a|->b})(c) = d)
END
```

Cette règle doit maintenant être compilée et chargée en mémoire grâce à la commande `pc`. On vérifie ensuite que cette règle a bien été chargée en mémoire grâce à la commande SearchRule :

```
PRI > pc
Loading theory func
PRI > sr(func,a)
Searching in func rules with filter
    consequent should contain a
Starting search...
Rule list is
    func.1      (Backward)
        bnum(a) &
        bnum(c) &
        bnot(btest(a = c)) &
        f(c) = d
        =>
        (f<+{a|->b})(c) = d
End of rule list
```

Elle peut maintenant être utilisée, notamment au sein d'un appel étendu au prouveur automatique. Dans ce cas particulier, la théorie $func$ va être utilisée en complément des règles et mécanismes natifs du prouveur. Le mode trace du prouveur est utilisé afin de visualiser l'action de la règle $func.1$.

Il faut noter d'ailleurs que si la règle n'était pas équipée du système de trace (terme $bcall1(BackwardRule(func.1))$, son déclenchement ne serait pas visible dans ce mode.

```
PRI > pr(Tac(func), Ru.Goal.None)
```

```
  Starting Trace in mode Ru.Goal.None , NoFile , NoSimpl
Starting Prover Call
  After deduction, goal is now
        (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
        <+{5|->0}<+{6|->1}<+{7|->0})(0) = 0
  By applying atomic rule func.1,
       bnum(a) &
       bnum(c) &
       bnot(btest(a = c)) &
       f(c) = d
       =>
       (f<+{a|->b})(c) = d
  the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
        <+{5|->0}<+{6|->1}<+{7|->0})(0) = 0  is now
       (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
        <+{5|->0}<+{6|->1})(0) = 0
```

La règle s'est bien appliquée. Le but est simplifié. La preuve continue avec les applications successives de cette règle :

```
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
          <+{5|->0}<+{6|->1})(0) = 0  is now
          (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
          <+{5|->0})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
          <+{5|->0})(0) = 0  is now
          (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1})(0) = 0  is now
          (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0})(0) = 0  is now
          (tab$1<+{0|->0}<+{1|->1}<+{2|->1})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1})(0) = 0  is now
          (tab$1<+{0|->0}<+{1|->1})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1})(0) = 0  is now
          (tab$1<+{0|->0})(0) = 0
```

puis le dernier but est démontré par le prouveur natif :

```
Goal (tab$1<+{0|->0})(0) = 0  is discharged.
```

La démonstration est alors sauvegardée :

```
PRI > sw
PO op.11 saved
```

Enfin, puisque 7 des 8 obligations de preuve restantes se ressemblent, cette démonstration est appliquée (commande TryEveryWhere) à toutes les obligations de preuve non prouvées :

```
te(op.11)
Begin TryEveryWhere
-++++++
Summary
op.17 transformed   Unproved --> Proved,   pr(Tac(func))
op.16 transformed   Unproved --> Proved,   pr(Tac(func))
op.15 transformed   Unproved --> Proved,   pr(Tac(func))
```

```
op.14 transformed   Unproved --> Proved,   pr(Tac(func))
op.13 transformed   Unproved --> Proved,   pr(Tac(func))
op.12 transformed   Unproved --> Proved,   pr(Tac(func))
End TryEveryWhere
```

Considérons maintenant la dernière obligation de preuve non prouvée :

```
tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
    <+{5|->0}<+{6|->1}<+{7|->0}: 0..7 +-> 0..1
```

Encore une fois, aucune règle du prouveur ne permet de simplifier efficacement le but. Une nouvelle règle est ajoutée, après avoir vérifié sa validité grâce aux outils de preuve de règles :

```
PRI > vr(Back, (b: A+->B & f: A+->B => (f<+b): A+->B))
The rule was proved
```

Cette règle est alors ajoutée au fichier M1_i.pmm, dans la théorie simpl cette fois :

```
THEORY simpl IS
    bcall1(BackwardRule(func.1)) &
    f: A +-> B & b: A +-> B
    =>
    f<+b : A +->B
END
```

Cette règle est alors chargée en mémoire :

```
PRI > pc
Loading theory func
Loading theory simpl
```

On vérifie que cette règle est bien chargée en mémoire :

```
PRI > sr(simpl, a)
Searching in simpl rules with filter
    consequent should contain a
Starting search...
Rule list is
    simpl.1      (Backward)
        f: A +-> B &
        b: A +-> B
        =>
        f<+b: A +-> B
End of rule list
```

La règle *simpl*.1 peut maintenant être utilisée, notamment au sein d'un appel étendu au prouveur automatique. Dans ce cas particulier, la théorie *simpl* va être utilisée en complément des règles et mécanismes natifs du prouveur. Le mode trace du prouveur est utilisé afin de visualiser l'action de la règle *simpl*.1.

```
PRI > pr(Tac(simpl), Ru.Goal.None)
```

```
  Starting Trace in mode Ru.Goal.None , NoFile , NoSimpl

Starting Prover Call
  After deduction, goal is now
        tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
        <+{5|->0}<+{6|->1}<+{7|->0}: 0..7 +-> 0..1
  By applying atomic rule simpl.1,
       f: A +-> B &
       b: A +-> B
       =>
       f<+b: A +-> B
  the goal tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
       <+{5|->0}<+{6|->1}<+{7|->0}: 0..7 +-> 0..1  is now
       tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
       <+{5|->0}<+{6|->1}: 0..7 +-> 0..1
   and {7|->0}: 0..7 +-> 0..1
```

Le but va alors être successivement décomposé par action de la règle *simpl*.1 et des règles natives du prouveur.

7.8 Application : second exemple

Soit le nouveau composant M1_i, en remplacement du précédent :

```
IMPLEMENTATION
    M1_i
REFINES
    M1
INITIALISATION
    tab := (0..7) * {0}
OPERATIONS
    op =
        BEGIN
            tab(0) := 0 mod 2 ;
            tab(1) := 1 mod 2;
            tab(2) := 1 mod 2;
            tab(3) := 0 mod 2;
            tab(4) := 1 mod 2;
            tab(5) := 0 mod 2;
            tab(6) := 1 mod 2;
            tab(7) := 0 mod 2
        END
END
```

La seule différence par rapport à l'exemple précédent est l'apparition, uniquement démonstrative ici, du modulo qui rend le rejeu des démonstrations inefficace car, après rejeu, il reste toujours 8 obligations de preuve non prouvées. Pour se ramener au cas précédent, il suffirait d'ajouter en hypothèse de toutes ces obligations de preuve les prédicats

$$0 \bmod 2 = 0$$
$$1 \bmod 2 = 1$$

puis d'utiliser ces égalités pour remplacer le terme de gauche par le terme de droite dans chacune des obligations de preuve.

Une solution consiste à ajouter des prédicats en assertions :

```
ASSERTIONS
    0 mod 2 = 0;
    1 mod 2 = 1
```

Ces 2 assertions se démontrent facilement en preuve interactive (`pp(rp.0)`).

La démonstration des obligations de preuve non prouvées se fait par utilisation de ces 2 égalités puis exécution de la démonstration interactive de l'exemple précédent.

La démonstration des 7 obligations de preuve ayant même forme se fait par une seule commande TryEveryWhere :

```
te((eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))), Replace.Loc.Unproved)
```

On vérifie alors que ces obligations de preuve sont effectivement démontrées :

```
Begin TryEveryWhere
-+++++++
Summary
op.25 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.24 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.23 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.22 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.21 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.20 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.19 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
End TryEveryWhere
```

Chapitre 8

Études de cas

Dans ce chapitre, nous allons présenter quelques preuves interactives. Après chacune de ces preuves, nous résumerons l'esprit de la méthode employée.

8.1 Preuve simple par contradiction

Nous allons présenter un exemple de preuve interactive sans aucune règle manuelle. Le but initial est **bfalse**, ce qui indique que la preuve ne peut être vraie que par hypothèses contradictoires. Le prouveur échoue parcequ'il ne sait pas isoler l'hypothèse contradictoire, nous allons voir comment la lui indiquer.

"'Previous components invariants'" $\wedge$
ens $<$ $\in$ NAT $\wedge$
card(ens) $\leq$ 3 $\wedge$
"'Component invariant'" $\wedge$
v1$1 $\in$ $\mathbb{N}$ $\wedge$
v1$1 $\leq$ 2147483647 $\wedge$
v2$1 $\in$ $\mathbb{N}$ $\wedge$
v2$1 $\leq$ 2147483647 $\wedge$
v3$1 $\in$ $\mathbb{N}$ $\wedge$
v3$1 $\leq$ 2147483647 $\wedge$
taille$1 $\in$ $\mathbb{N}$ $\wedge$
taille$1 $\leq$ 2147483647 $\wedge$
taille$1 = card(ens) $\wedge$
taille$1 = 0 $\Rightarrow$ ens = $\varnothing$ $\wedge$
taille$1 = 1 $\Rightarrow$ ens = {v1$1} $\wedge$
taille$1 = 2 $\Rightarrow$ ens = {v1$1,v2$1} $\wedge$
taille$1 = 3 $\Rightarrow$ ens = {v1$1,v2$1,v3$1} $\wedge$
btrue $\wedge$
"'enter preconditions in previous components'" $\wedge$
ee $\in$ $\mathbb{N}$ $\wedge$
ee $\leq$ 2147483647 $\wedge$
lll_1.enter.30 $\wedge$
"'enter preconditions in this component'" $\wedge$
"'Local hypotheses'" $\wedge$
$\neg$(taille$1 = 0) $\wedge$
$\neg$(taille$1 = 1) $\wedge$
$\neg$(taille$1 = 2) $\wedge$
$\neg$(v1$1 = ee) $\wedge$
$\neg$(v2$1 = ee) $\wedge$
$\neg$(v3$1 = ee) $\wedge$
taille$1 = 3 $\wedge$
card(ens $\cup$ {ee}) $\leq$ 3 $\wedge$
"'Check that the invariant (ov$1 = ov) is preserved by the operation, ref 8'"
$\Rightarrow$
bfalse

Comme *taille$1 = 3*, nous savons que *ens* = {*v1$1,v2$1,v3$1*}. D'autre part, *taille$1* = card*(ens)* donc les trois *vi$1* sont différents. Or *ee* est différent de chaque *vi$1*, donc card*(ens*$\cup${*ee*}*)* = *4*. C'est donc l'hypothèse card*(ens*$\cup${*ee*}*)* $\leq$ *3* qui est contradictoire. Le but initial qui apparaît dans le prouveur interactif contient les hypothèses locales :

"'Local hypotheses'" $\wedge$
$\neg$(taille$1 = 0) $\wedge$
$\neg$(taille$1 = 1) $\wedge$
$\neg$(taille$1 = 2) $\wedge$
$\neg$(v1$1 = ee) $\wedge$

$\neg$(v2\$1 = ee) $\wedge$
$\neg$(v3\$1 = ee) $\wedge$
taille\$1 = 3 $\wedge$
card(ens $\cup$ {ee}) $\leq$ 3 $\wedge$
"'Check that the invariant (ov\$1 = ov) is preserved by the operation, ref 8'"
$\Rightarrow$
bfalse

Il y a deux méthodes pour charger ces hypothèses :

– la commande *Deduction* (`dd`) : les hypothèses sont alors directement chargées sans intervention du prouveur. En particulier, le prouveur ne peut pas réduire les identifiants utilisés au minimum comme il le fait sur les autres hypothèses.
– la commande *Prove* (`pr`) : les hypothèses sont alors chargées par le prouveur qui enchaine sur la preuve. Le but initial est donc transformé.

Dans notre cas le but **bfalse** ne peut pas se transformer. Nous pouvons donc charger les hypothèses par la commande *Prove*, en utilisant l'option `Red` pour éviter le déclenchement de preuves par cas intempestives.

```
PRI > pr(Red)
Starting Prover Call
```

Le but devient :

bfalse

L'hypothèse contradictoire card*(ens*$\cup${*ee*}*)* $\leq$ *3* c'est transformée, elle apparaît maintenant sous deux formes différentes. Sans même chercher à comprendre ces nouvelles formulations, nous vérifions facilement qu'elles sont toujours contradictoires si card*(ens*$\cup${*ee*}*)*. Il suffit alors d'indiquer l'une de ces formes au prouveur, par la commande *FalseHypothesis* :

```
PRI > fh(0<=2-card(ens)+card(ens /\ {ee}))
Starting False Hypothesis
```

Le but devient :

$\neg$(0 $\leq$ 2-card(ens)+card(ens $\cap$ {ee}))

C'est-à-dire que la preuve se résume maintenant à démontrer la négation de cette formule. Pour des raisons de cohérence de la preuve, il est impossible de supprimer une hypothèse, c'est pourquoi la formule est toujours en hypothèse. Ce n'est pas gênant car l'hypothèse P n'est pas employée pour démontrer $\neg P$. Nous pouvons tenter la preuve :

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

Le but devient :

0 $\leq$ -card({v1\$1} $\cap$ {v2\$1})-card({v1\$1,v2\$1} $\cap$ {v3\$1})
-card({v1\$1,v2\$1,v3\$1} $\cap$ {ee})

C'est bien compliqué ! Il faut encore aider le prouveur avant de lancer cette preuve. Revenons en arrière :

```
PRI > ba
```

Le but redevient :

$$\neg(0 \leq 2\text{-}\mathsf{card}(\text{ens})+\mathsf{card}(\text{ens} \cap \{\text{ee}\}))$$

Relançons la preuve en mode réduit, pour que le but soit simplifié sans preuves par cas ou remplacements exploratoires. Nous appliquons ainsi précisément la méthode décrite au chapitre 6.1.

```
PRI > pr(Red)
Starting Prover Call
```

Le but devient :

$$0 \leq \text{-}3+\mathsf{card}(\text{ens})\text{-}\mathsf{card}(\text{ens} \cap \{\text{ee}\})$$

Comment aider la preuve à cette étape ? L'expression *ens* $\cap$ {*ee*} représente l'ensemble vide, une telle simplification dans le but est sûrement bénéfique .Nous pouvons essayer d'attirer l'attention du prouveur sur ceci :

```
PRI > ah(ens /\ {ee} = {})
Starting Add Hypothesis
```

Le but devient :

$$\text{ens} \cap \{\text{ee}\} = \varnothing$$

Tentons de démontrer ceci :

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

Le but devient :

$$\text{ens} \cap \{\text{ee}\} = \varnothing \;\Rightarrow\; 0 \leq \text{-}3+\mathsf{card}(\text{ens})\text{-}\mathsf{card}(\text{ens} \cap \{\text{ee}\})$$

La preuve de cette nouvelle hypothèse a donc réussi. Grâce à elle la preuve principale va peut-être aboutir :

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

Le but devient :

$$0 \leq \text{-}\mathsf{card}(\{\text{v1\$1}\} \cap \{\text{v2\$1}\})\text{-}\mathsf{card}(\{\text{v1\$1,v2\$1}\} \cap \{\text{v3\$1}\})$$

Ce n'est pas suffisant. Revenons en arrière pour remplacer `pr` par `pr(Red)` conformément à la méthode générale :

```
PRI > ba
```

Le but devient :

$$\text{ens} \cap \{\text{ee}\} = \varnothing \;\Rightarrow\; 0 \leq \text{-3+}\mathsf{card}(\text{ens})\text{-}\mathsf{card}(\text{ens}\cap\{\text{ee}\})$$

```
PRI > pr(Red)
Starting Prover Call
```

Le but devient :

$$3 \leq \mathsf{card}(\text{ens})$$

Ce but est évident puisque card*(ens)* = *taille$1* et que *taille$1* = *3*. Le prouveur échoue car il ne remplace pas l'expression card*(ens)* par *taille$1*. Sans chercher à comprendre pourquoi, il suffit de le faire menuellement :

```
PRI > eh(card(ens),taille$1,Goal)
Starting use Equality in Hypothesis
```

Le but devient :

$$3 \leq \text{taille\$1}$$

Tentons la preuve :

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

Le but initial réapparaît en vert, la preuve est terminée. L'arbre de preuve est le suivant :

```
Force(0)
  pr(Red)
    fh(0<=2-card(ens)+card(ens /\ {ee}))
      pr(Red)
        ah(ens /\ {ee} = {})
          pr
          pr(Red)
            eh(card(ens),taille$1,Goal)
              pr
  Next
```

8.2 Preuve arithmétique avec divisions

L'exemple présenté ici est une preuve concernant les divisions entières et nécessitant de connaître l'intervalle de variation du reste d'une telle division. Nous nous placerons dans le cas où le prouveur n'a aucune connaissance mathématique sur les restes. La preuve se fera donc avec ajout de règles manuelles, nous allons voir comment l'usage des fonctionnalités permet de réduire ces règles au minimum. Considérons l'obligation de preuve suivante :

"'Included,imported and extended machines invariants'" ∧
c1$1 ∈ 0..150 ∧
c2$1 ∈ 0..150 ∧
nmesure$1 ∈ ℕ ∧
ntransfert$1 ∈ ℕ ∧
btrue ∧
0 ≤ c1$1 ∧
c1$1 ≤ 150 ∧
0 ≤ c2$1 ∧
c2$1 ≤ 150 ∧
"'Previous components invariants'" ∧
c1$1 ∈ 0..120 ∧
c2$1 ∈ 0..120 ∧
c1$1-c2$1 ∈ -1..1 ∧
"'Component invariant'" ∧
c2 = c2$1 ∧
c1 = c1$1 ∧
nmesure$1 = ntransfert$1 ∧
ncycle = ntransfert$1 ∧
c1$1 ≤ 120 ∧
c2$1 ≤ 120 ∧
0 ≤ 1+c1$1-c2$1 ∧
-1 ≤ c1$1-c2$1 ∧
0 ≤ 1-c1$1+c2$1 ∧
c1$1-c2$1 ≤ 1 ∧
equi_1.cycle.20 ∧
"'Local hypotheses'" ∧
nc1 ∈ 0..150 ∧
nc2 ∈ 0..150 ∧
c1$1-nc1 ∈ -4..4 ∧
c2$1-nc2 ∈ -4..4 ∧
nc1+nc2 ≤ c1$1+c2$1 ∧
nc1-(nc1-nc2)/2 ∈ 0..150 ∧
nc2+(nc1-nc2)/2 ∈ 0..150 ∧
nmesure$1+1 ∈ ℕ ∧
ntransfert$1+1 ∈ ℕ ∧
"'Check operation refinement, ref 11'"
⇒
nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) ∈ -1..1

Il est facile de vérifier intuitivement que cette obligation de preuve est juste. En effet, si l'expression considérée était un calcul dans l'ensemble des nombres réel nous aurions :

nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) = 0

Si (nc1-nc2) est divisible par 2, le calcul entier est indentique au calcul réel, il donne donc 0. Si (nc1-nc2) n'est pas divisible par 2, le reste est 1 donc l'expression considérée vaut 1 ou -1. Ceci n'est pas une démonstration bien entendu, mais une manière de comprendre *pourquoi* cette obligation de preuve est juste. C'est la "démonstration intuitive" évoquée dans ce document.

Commençons la preuve formelle. Après chargement de l'obligation de preuve, la zone d'affichage du but contient la formule suivante :

```
"'Local hypotheses'"  ∧
nc1 ∈ 0..150  ∧
nc2 ∈ 0..150  ∧
c1$1-nc1 ∈ -4..4  ∧
c2$1-nc2 ∈ -4..4  ∧
nc1+nc2 ≤ c1$1+c2$1  ∧
nc1-(nc1-nc2)/2 ∈ 0..150  ∧
nc2+(nc1-nc2)/2 ∈ 0..150  ∧
nmesure$1+1 ∈ ℕ  ∧
ntransfert$1+1 ∈ ℕ  ∧
"'Check operation refinement, ref 11'"
⇒
nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) ∈ -1..1
```

En effet les hypothèses locales ne sont pas encore chargées. La première chose à faire est de lancer la preuve pour voir où elle s'arrête, conformément à la méthode décrite au paragraphe 6.1. Il suffit de lancer la commande `pr`, et d'observer le but qui apparaît :

$$0 \leq 1\text{-nc1+nc2+2}\times((\text{nc1-nc2})/2)$$

Nous constatons que le prouveur cherche à démontrer que l'expression après simplification est inférieure ou égale à 1. Ceci est une moitié de la preuve initiale consistant à démontrer l'appartenance à -1..1 : il y a donc un autre but à démontrer après, établissant que l'expression initiale est supérieure à -1.

Nous supposons que le prouveur ne contient aucune règle sur les restes de divisions entières (par exemple : nous l'avons constaté par examen de la base de règle, avec la commande *SearchRule* `sr`). Il y a donc une connaissance mathématique à amener, dont les deux parties de la preuve devraient bénéficier. C'est pourquoi il est préférable de revenir au début de la démonstration avant de procéder à cet ajout de connaissance, ce que nous faisons par la commande *Reset* (`re`).

Les règles minimales *à priori* suffisantes pour faire cette démonstration sont les suivantes :

```
THEORY IntDiv IS

  b*(a/b) == a - (a mod b);

  a: NATURAL &
  b: NATURAL &
  not(b = 0)
  =>
  (a mod b) : 0..(b-1);

  a: NATURAL &
  b<=0 &
  not(b = 0)
  =>
  (a mod b) : 0..(1-b);

  a<=0 &
  b: NATURAL &
  not(b = 0)
  =>
  (a mod b) : (1-b)..0;

  a<=0 &
  b<=0 &
  not(b = 0)
  =>
  (a mod b) : (b-1)..0

END
```

Nous avons présenté ces règles en langage de théorie, dans le format qui permet de les écrire dans un fichier *composant.pmm*. Ce format nécessite quelques commentaires :

– Pour indiquer qu'une variable doit être un entier positif, nous écrivons $a \in \mathbb{N}$ plutôt que $a \in \mathbb{Z} \ \wedge \ 0 \leq a$ car $\mathbb{N}$ fait partie des symboles de base du prouveur, contrairement à $\mathbb{Z}$.
– Pour indiquer qu'une variable doit être un entier négatif, nous écrivons simplement $a \leq 0$, sans préciser $a \in \mathbb{Z}$. En effet si a n'est pas un entier, par exemple $a = \mathsf{TRUE}$, alors l'expression $a \ \mathsf{mod} \ b$ est mal typée et n'a pas pu apparaître dans une obligation de preuve provenant d'un composant dont le contrôle de type est correct. Nous évitons ainsi le symbole $\mathbb{Z}$.

Le modulo que nous avons décrit ici est l'extension à $\mathbb{Z} \times \mathbb{Z}_1$ de la définition sur $\mathbb{N} \times \mathbb{N}_1$:

"Si a et b sont deux entiers naturels et si b est non nul, alors il existe un et un seul couple (q, r) tel que $a = bq + r$ et $r < b$. Par définition : $q = a/b$ et $r = a \ \mathsf{mod} \ b$."

Cette définition peut être étendue à $\mathbb{Z} \times \mathbb{Z}_1$ de manière non ambigüe de telle manière que $a = b \times (a/b) + (a \ \mathsf{mod} \ b)$ reste vrai et que les règles de simplification des signes d'un

quotient soient naturelles. C'est cette définition du modulo qui est utilisée dans l'Atelier B. Elle est constituée de cette égalité et des règles d'appartenance du reste, nous l'avons donc entièrement décrite dans `IntDiv`.

La première règle : `b*(a/b) == a - a mod b` n'a pas une forme quelconque. Nous avons choisi une règle de réecriture qui permet d'éliminer une division en fabriquant un modulo, ce qui nous permettra de remplacer les divisions dans le but ou dans une hypothèse. De plus, l'écriture choisie est $b \times (a/b)$ au lieu de $(a/b) \times b$ parceque le solveur arithmétique met les coefficient les plus simples en premier. Ecrire une telle règle directement dans sa forme optimale n'est pas chose facile, cela nécessite de l'expérience. Néanmoins il est toujours possible d'améliorer la forme de la règle quant elle s'avère peu pratique.

Nous introduisons ces règles dans le fichier ".pmm" du composant concerné, puis nous chargeons ce fichier en utilisant la commande *PmmCompile* (`pc`). Comment utiliser cette connaissance mathématique dans notre exemple ? Pour nous a vaut $nc1 - nc2$ et b vaut 2. b est donc positif, mais il faut faire deux cas suivant le signe de $nc1 - nc2$; il faut faire ces cas le plus tot possible et de toutes manières avant que le but soit divisé en deux sous-buts.

Il ne serait pas logique de faire les deux cas $nc1 - nc2 \leq 0$ et $nc1 - nc2 \geq 0$ à partir du but courant puisque $nc1$ et $nc2$ sont définis dans les hypothèses locales. Celles-ci sont encore dans le but, il n'est donc pas possible de les utiliser avant leur montée. Nous avons deux moyens de faire monter les hypothèses locales :

- la commande *Deduction* (`dd`) : les hypothèses sont alors directement chargées sans intervention du prouveur. En particulier, le prouveur ne peut pas réduire les identifiants utilisés au minimum comme il le fait sur les autres hypothèses.
- la commande *Prove* (`pr`) : les hypothèses sont alors chargées par le prouveur qui enchaine sur la preuve. Le but initial va donc être divisé en deux comme nous l'avons vu plus haut.

Pour éviter d'avoir à faire les deux cas $nc1 - nc2 \leq 0$ et $nc1 - nc2 \geq 0$ pour deux sous-buts, nous sommes obligés d'utiliser `dd`. L'option `pr(Red)` de la commande `pr` ne nous permet pas d'éviter l'apparition des deux sous-buts dans ce cas, car il ne s'agit pas d'une tentative de preuve par cas du prouveur mais bien d'une règle de base pour prouver $x \in u \,..\, v$: il faut démontrer $u \leq x$ puis $x \leq v$. Ces considérations peuvent être faites en effectuant rapidement quelques essais de la commande `pr` à partir du but précédent ; il n'est pas nécéssaire d'étudier la programmation interne du prouveur.

La première commande que nous allons utiliser est donc `dd` :

```
PRI > dd Starting Deduction
```

Le but devient :

$$\text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

Les hypothèses locales sont maintenant à la fin de la liste affichée dans la fenêtre des hypothèses.

Pour faire deux cas suivant le signe de $nc1 - nc2$, nous disposons de la commande *DoCases* (`dc`). Rappelons les deux formes de cette commande :

– `dc(P)` : permet de se placer dans les cas P et $\neg P$;
– `dc(v,E)` : preuve pour v valant chacun des éléments de E.

Nous devons utiliser la première forme. Dans une preuve par cas, il est toujours indiqué de commencer par le cas le plus difficile : ici c'est probablement le cas où $nc1 - nc2$ est négatif. En effet les preuves sur des nombres négatifs sont souvent plus difficiles. Nous choisissons donc $nc1 - nc2 \leq 0$ pour P.

Le lecteur aura remarqué que nous utilisons de préférence le symbole $\leq$, plutot que $\geq$, $<$, $>$. En effet $\leq$ est un symbole de base du prouveur, les autres symboles étant ramenés à lui. Malgré la présence d'un système automatique de normalisation, il est conseillé d'utiliser ces symboles de base en priorité. La liste de ces symboles est indiquée dans le Manuel de référence du prouveur.

```
PRI > dc(nc1-nc2<=0)
Starting Do Cases
```

Le but devient :

$$\text{nc1-nc2} \leq 0 \;\Rightarrow\; \text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

L'hypothèse de preuve par cas est une hypothèse locale, elle apparaît dans le but. Les considération précédentes sur le chargement des hyptohèses locales étant toujours valides, nous chargeons cette hypothèses par `dd`.

```
PRI > dd
Starting Deduction
```

Le but devient :

$$\text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

Nous n'avons déchargé aucun but jusque là, l'arbre de preuve est donc une montée régulière. Nous pouvons le contrôler dans la zone de ligne de commande (fenêtre de situation globale) dont l'affichage est le suivant :

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        Next
```

Nous nous sommes placés dans le cas où $nc1 - nc2$ est négatif. Nous pouvons maintenant introduire l'intervalle de variation du reste de la division par 2 de $nc1 - nc2$ en utilisant les règles ajoutées. Nous ne savons pas encore *comment* cet élément sera utilisé, mais il est clairement nécessaire.

Le moyen le plus simple pour indiquer cet intervalle de variation est l'ajout d'une hypothèse : (nc1-nc2) `mod` 2 $\in$ -1..0. Le prouveur ne saurait pas démontrer cette nouvelle hypothèse puisque nous avons vu qu'il n'a pas les règles nécessaires, c'est donc avec la règle correspondante de la théorie `IntDiv` que nous allons le faire. Cette règle est :

```
a<=0 &
b: NATURAL &
not(b = 0)
=>
(a mod b) : (1-b)..0;
```

Pour pouvoir utiliser cette règle, il faut que le but à prouver ait exactement la forme du conséquent de la règle. Nous allons donc ajouter l'hypothèse non simplifiée (nc1-nc2) mod $2 \in 1\text{-}2..0$ qui correspond précisément à notre règle. Les simplifications nécessaires seront faites après.

```
PRI > ah((nc1-nc2) mod 2: 1-2..0)
Starting Add Hypothesis
```

Le but devient :

$$(\text{nc1-nc2}) \bmod 2 \in 1\text{-}2..0$$

Il faut maintenant appliquer notre règle, qui est la quatrième de la théorie `IntDiv`. Pour de telles règles sans réecritures, la commande *ApplyRule* (`ar`) possède les modes `Once` (application une seule fois) et `Multi` (application tant que possible). Ici ces deux modes sont équivalents parceque la règle ne se réapplique pas sur ses antécédents.

```
PRI > ar(IntDiv.4,Once)
Starting Apply Rule
```

Le but devient :

$$\text{nc1-nc2} \leq 0$$

Il s'agit bien du premier antécédent instancié de la règle. Essayons de le démontrer par un simple appel au prouveur :

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

Le but devient :

$$2 \in \mathbb{N}$$

C'est le deuxième antécédent, le but précédent a été déchargé. Continuons :

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

Le but devient :

$$\neg(2 = 0)$$

C'est le troisième antécédent, le but précédent a été déchargé. Continuons :

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

Le but devient :

$$(\text{nc1-nc2}) \bmod 2 \in \text{1-2..0} \;\Rightarrow\; \text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

Nous retrouvons le but précédent l'ajout d'hypothèse, sous l'hypothèse voulue. Cette partie de la démonstration a profité à la fois de la règle ajoutée et des fonctionnalités de preuve automatique, ce qui nous permet de ne pas s'attarder sur les parties démontrables automatiquement. La zone de ligne de commande contient l'arbre de preuve suivant :

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        ah((nc1-nc2) mod 2: 1-2..0) &
          ar(IntDiv.4,Once) &
            pr &
            pr &
            pr &
          Next
```

Le mot clef `Next` indenté directement sous la commande `ah` indique que le but actuel est un sous-but produit par cette commande. Rappelons que `ah(H)` produit deux sous-buts à partir d'un but B : le sous-but H puis le sous-but $H \;\Rightarrow\; B$. Nous sommes donc sur ce deuxième sous-but, en effet `Next` est le deuxième mot clef indenté sous `ah`, après `ar`.

La règle manuelle a servi à fabriquer une nouvelle hypothèse, il s'agit donc d'une sorte de génération par l'avant. Il aurait été possible de fabriquer une règle *forward* pour obtenir le même résultat, mais celle-ci serait moins simple. Le procédé utilisant l'ajout d'hypothèse nous permet d'exploiter facilement une règle écrite uniquement en fonction de considérations mathématiques.

Nous avons introduit l'intervalle de variation du reste, mais le prouveur ne peut pas encore aboutir sans utiliser la première règle de la théorie `IntDiv` :

```
b*(a/b) == a - (a mod b);
```

Si nous utilisons la commande `pr`, il est clair que le but courant ne va pas être déchargé. Il est néanmoins utile que le prouveur simplifie la nouvelle hypothèse et commence la preuve en simplifiant le but. Pour faire tout ceci sans démarrer des preuves par cas exploratoires, nous pouvons utiliser `pr(Red)` :

```
PRI > pr(Red)
Starting Prover Call
```

Le but devient :

$$0 \leq \text{1+nc1-nc2-2}\times((\text{nc1-nc2})/2)$$

Comme prévu, le prouveur a simplifié le but et éliminé l'intervalle en produisant deux sous buts. L'hypothèse a bien été simplifiée et chargée, elle apparaît en bas de la liste d'hypothèses. Nous devons maintenant utiliser la première régle de `IntDiv`. Il s'agit d'une règle de réecriture, pour laquelle la commande *ApplyRule* (`ar`) possède les modes `Goal` (réecriture dans le but) et `Hyp`, `AllHyp` et `Hyp(h)` (réecriture dans les hypothèses). Nous l'appliquons au but :

```
PRI > ar(IntDiv.1,Goal)
Starting Apply Rule
```

Le but devient :

$$0 \leq 1+\text{nc1-nc2-(nc1-nc2-(nc1-nc2) } \texttt{mod}\ 2)$$

Maintenant il ne manque *à priori* plus rien pour que la preuve de ce sous but puisse aboutir. Nous pouvons donc tenter un appel au prouveur complet :

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

Le but devient :

$$\text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \leq 1$$

Il s'agit du deuxième sous-but nécessaire pour démontrer l'appartenance à l'intervalle $-1\,..\,1$. Nous voyons que ce but n'a pas été simplifié : en effet le prouveur s'est arrêté avec l'échec du premier sous-but.

L'arbre de preuve est maintenant le suivant :

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        ah((nc1-nc2) mod 2: 1-2..0) &
          ar(IntDiv.4,Once) &
            pr &
            pr &
            pr &
          pr(Red) &
            ar(IntDiv.1,Goal) &
              pr &
            Next
```

Le mot clef `Next` est le deuxième à être indenté immédiatement sous `pr(Red)`, ce qui indique qu'il s'agit du deuxième sous but généré par cette commande, celui qui correspond à la deuxième borne de l'intervalle. Nous savons comme précedemment que la preuve de ce but ne pourra pas aboutir sans utiliser la première règle de `IntDiv`. Pour procéder à sa simplification, il faut donc utiliser `pr(Red)` au lieu de `pr` qui tenterait des preuves par cas nuisibles.

```
PRI > pr(Red)
Starting Prover Call
```

Le but devient :

$$0 \leq \text{1-nc1+nc2+2}\times((\text{nc1-nc2})/2)$$

Comme précedemment, nous utilisons la première règle de `IntDiv` :

```
PRI > ar(IntDiv.1,Goal)
Starting Apply Rule
```

Le but devient :

$$0 \leq \text{1-nc1+nc2+(nc1-nc2-(nc1-nc2) mod 2)}$$

Tout est pret, nous pouvons lancer la preuve automatique :

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

Le but devient :

$$\neg(\text{nc1-nc2} \leq 0) \;\Rightarrow\; \text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

Ce but est le deuxième cas la preuve : si $nc1 - nc2$ est positif. Nous avons donc fini la moitié de la preuve principale. l'arbre de preuve est affiché dans la zone ligne de commande :

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        ah((nc1-nc2) mod 2: 1-2..0) &
          ar(IntDiv.4,Once) &
            pr &
            pr &
            pr &
          pr(Red) &
            ar(IntDiv.1,Goal) &
              pr &
            pr(Red) &
              ar(IntDiv.1,Goal) &
                pr &
      Next
```

Le mot clef `Next` est le deuxième indenté immédiatement sous la commande `dc`, il s'agit donc du deuxième cas cette preuve par cas. L'hypothèse locale $\neg(nc1 - nc1 \leq 0)$ indique que nous supposons maintenant $nc1 - nc2$ positif.

Nous sommes revenus en dessous de la commande `ah` par laquelle nous avions indiqué l'intervalle de variation du reste. Cet intervalle de variation n'est effectivement plus le

même ; nous devons donc le préciser à nouveau dans le cas où $nc1 - nc2$ est positif. La règle qui convient est :

```
a: NATURAL &
b: NATURAL &
not(b = 0)
=>
(a mod b) : 0..(b-1);
```

Comme précédemment, nous devons introduire cet intervalle de variation avant l'apparition des deux sous-buts. Il faut déjà charger l'hypothèse locale $\neg(nc1 - nc1 \leq 0)$ actuellement dans le but, elle est en effet nécessaire pour préciser l'intervalle de variation du reste. Nous devons faire ce chargement sans utiliser le prouveur :

```
PRI > dd
Starting Deduction
```

Le but devient :

nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) ∈ -1..1

Nous procédons comme pour le premier cas de la preuve :

```
PRI > ah((nc1-nc2) mod 2: 0..2-1)
Starting Add Hypothesis
```

Le but devient :

(nc1-nc2) mod 2 ∈ 0..2-1

La règle qui convient est la deuxième de la théorie `IntDiv` :

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

Le but devient :

nc1-nc2 ∈ ℕ

Nous essayons naturellement de décharger ce but par un appel au prouveur (`pr`). Malheureusement, cette preuve échoue malgré la présence de l'hypothèse clef $\neg(nc1 - nc2 \leq 0)$. Cette hypothèse n'a manifestement pas été utilisée ; c'est l'occasion d'employer le procédé consistant à faire repasser des hypothèses dans le prouveur (voir paragraphe 6.7.2).

```
PRI > ah(not(nc1-nc2<=0))
Starting Add Hypothesis
```

Le but devient :

¬(nc1-nc2 ≤ 0) ⇒ nc1-nc2 ∈ ℕ

L'hypothèse existant déjà telle quelle, il n'est pas demandé de refaire sa démonstration. Nous pouvons relancer la preuve :

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

Le but devient :

$$2 \in \mathbb{N}$$

Il s'agit bien de l'antécédant suivant de la deuxième règle de la théorie `IntDiv`. Le simple fait de refaire passer l'hypothèse clef dans le prouveur lui a permis de démontrer le but précédent. Continuons :

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

Le but devient :

$$\neg(2 = 0)$$

Continuons :

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

Le but devient :

$$(\text{nc1-nc2}) \bmod 2 \in 0..2\text{-}1 \;\Rightarrow\; \text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-}1..1$$

L'ajout de l'hypothèse indiquant l'intervalle de variation du reste est terminé. Il reste à appliquer la règle `(a/b) == a - (a mod b)` après simplification du but :

```
PRI > pr(Red)
Starting Prover Call
```

Le but devient :

$$0 \leq 1+\text{nc1-nc2-}2\times((\text{nc1-nc2})/2)$$

puis :

```
PRI > ar(IntDiv.1,Goal)
Starting Apply Rule
```

Le but devient :

$$0 \leq 1+\text{nc1-nc2-(nc1-nc2-(nc1-nc2)} \bmod 2)$$

Nous pouvons tenter de décharger ce but :

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

Le but devient :

$$\text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \leq 1$$

Le but qui apparaît correspond à la deuxième borne de l'intervalle, nous le traitons de manière analogue :

```
PRI > pr(Red)
Starting Prover Call
```

Le but devient :

$$0 \leq \text{1-nc1+nc2+2}\times\text{((nc1-nc2)/2)}$$

Puis :

```
PRI > ar(IntDiv.1,Goal)
Starting Apply Rule
```

Le but devient :

$$0 \leq \text{1-nc1+nc2+(nc1-nc2-(nc1-nc2) mod 2)}$$

Nous pouvons tenter la preuve de ce dernier sous-but :

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

Avec la disparition du dernier sous-but, le but initial de l'obligation de preuve réapparaît coloré en vert : la preuve est finie. Il suffit de quitter l'obligation de preuve pour provoquer la sauvegarde de la démonstration. L'arbre de preuve final est le suivant :

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        ah((nc1-nc2) mod 2: 1-2..0) &
          ar(IntDiv.4,Once) &
            pr &
            pr &
            pr &
          pr(Red) &
            ar(IntDiv.1,Goal) &
              pr &
            pr(Red) &
              ar(IntDiv.1,Goal) &
                pr &
      dd &
        ah((nc1-nc2) mod 2: 0..2-1) &
          ar(IntDiv.2,Once) &
            ah(not(nc1-nc2<=0)) &
              pr &
            pr &
            pr &
          pr(Red) &
            ar(IntDiv.1,Goal) &
              pr &
            pr(Red) &
              ar(IntDiv.1,Goal) &
                pr &
  Next
```

Les deux cas principaux de la preuve apparaîssent clairement dans l'indentation de cet arbre.

Pour faire cette démonstration, nous n'avons utilisé que cinq règles écrites dans un format presque purement mathématique. Ces règles sont presque des définitions, leur validation sera facile et elles seront probablement réutilisables. Nous avons évité l'usage de règles trop spécifiques grâce à l'utilisation des fonctionnalités du prouveur interactif. C'est ce principe consistant à employer des règles simples par des fonctionnalités évoluées du prouveur qui permettent de garder minimales les règles ajoutées. Nous évitons ainsi de nous retrouver en train de reconstruire un prouveur automatique dédié à notre preuve à partir de règles manuelles complexes.

Une autre remarque importante est que nous n'avons pas fait toute la démonstration manuellement. Nous nous concentrons sur les parties délicates en déchargeant tous les buts faciles par la commande `pr`. Une telle démonstration est à la fois plus facile et plus sécuritaire qu'une démonstration manuelle, cette dernière étant souvent fastidieuse quand on s'interdit les raccourcis intuitifs inacceptables dans une preuve formelle.

Chapitre 9

Questions fréquemment posées

9.1 Pr peut nous engager dans une mauvaise voie pour la preuve

Il y a une différence fondamentale entre la commande `mp` et la commande `pr`.
Quelque soit la force de prouveur utilisée, la commande `mp` applique des règles de simplification du but et des hypothèses, ainsi que quelques mécanismes de résolution.
La commande `pr` est construite selon les mêmes principes, mais elle contient d'autres mécanismes. En particulier, des heuristiques de simplification des prédicats existentiels, de traitement des dernières égalités montées en hypothèses, et le déclenchement de preuves par cas. Ces ajouts rendent cette commande plus performante dans la plupart des cas.

Par exemple, si le but courant est de la forme

$x \in S$

et que l'hypothèse

$x \in A \cup a$

existe, alors la preuve courante est découpée en 2 cas :

$x = a \ \Rightarrow \ a \in S \ \wedge$
$x \in A \ \wedge \ not(x = a) \ \Rightarrow \ x \in S$

Autre exemple, si la formule $dom(A * B)$ apparaît dans le but courant P, alors 2 cas vont être générés :

$(B = \varnothing \ \Rightarrow \ [dom(A * B) = \varnothing]P) \ \wedge$
$(not(B = \varnothing) \ \Rightarrow \ [dom(A * B) = A]P)$

$[f = g]P$ signifie ici que toutes les occurrences de f dans P sont remplacées par g.

Toutefois, le déclenchement de preuves par cas peut s'avérer inutile et nécessiter de prou-

ver un lemme autant de fois qu'il y a de cas. En effet, ces preuves par cas se déclenchent selon certains critères locaux qui peuvent ne pas correspondre au type de preuve qu'il faudrait réaliser.

Par exemple, si le lemme se démontre par contradiction (plusieurs hypothèses sont contradictoires), le déclenchement d'une preuve par cas va multiplier inutilement le nombre de lemmes à démontrer par contradiction.

Il est donc conseillé de tenter en preuve interactive la commande `mp` avant la commande `pr`. Si `mp` échoue, on vérifie alors si `pr` permet de conclure ou de transformer le but en une forme plus facilement prouvable.

9.2 Utilisation d'un plan de preuve

La rédaction d'un plan de preuve, avant de commencer une démonstration interactive, est nécessaire si l'on veut que cette preuve soit la plus courte et la plus productive possible.

Il faut bien entendu s'être assuré au préalable que l'obligation de preuve que l'on veut démontrer est juste. Cela se fait par inspection visuelle du but et des hypothèses locales. Au cours de cette inspection, l'on doit être capable de déterminer les éléments qui permettent de démontrer cette obligation de preuve, c'est à dire quels sont les hypothèses nécessaires. Si besoin est, ces éléments seront couchés sur papier afin de pouvoir facilement les retrouver par la suite. Si l'obligation de preuve est complexe et est d'une lecture difficile, il est recommandé d'utiliser l'analyseur logique de formule afin d'avoir une vision plus synthétique de l'obligation de preuve. Si par contre l'obligation de preuve vous paraît fausse, il faut absolument exhiber un contre-exemple, c'est à dire une valuation particulière des variables définies en hypothèse qui permette de démontrer la contraposée du but courant.

Le plan de preuve peut contenir :
- le type démonstration à réaliser (preuve par cas, preuve par contradiction, décomposition du but). La détermination du type de démonstration est établi par l'utilisateur, en fonction du résultat de l'inspection visuelle de l'obligation de preuve, et de son expérience en matière de preuve interactive avec l'Atelier B.
- la liste ordonnée des hypothèses à ajouter. Cette liste peut être complétée avec les commandes interactives utilisées pour la démonstration de chaque hypothèse.
- l'état de la preuve courante, c'est à dire à quel endroit de l'arbre de preuve est on arrivé ?

Ce plan de preuve permet de :
- se localiser dans l'arbre de preuve : savoir ce que l'on est en train de démontrer.
- déterminer le chemin parcouru et le reste à faire.
- minimiser la taille de la démonstration interactive. Par exemple, il est préférable d'ajouter une hypothèse avant de lancer une preuve par cas plutôt que de l'ajouter successivement pour chacun de ces cas.

– réutiliser des portions de démonstrations d'une obligation de preuve à l'autre, si par exemple, les mêmes hypothèses doivent être ajoutées.

Il ne faut pas oublier de tenter un *TryEveryWhere* (commande `te`) dès que l'on pense que la démonstration que l'on vient de réaliser peut être appliquée avec succès à d'autres obligations de preuve de l'opération ou du composant.

9.3 Comment savoir s'il faut ajouter une règle manuelle

Il n'est pas toujours facile de savoir si, à un point donné de la démonstration, il est préférable d'ajouter une règle mathématique pour démontrer (ou aider à démontrer) le but courant, ou s'il est préférable de continuer à utiliser des commandes d'orientation de preuve, de simplification et de résolution du prouveur.

Plusieurs aspects sont à prendre en considération, et c'est à l'utilisateur, en fonction du contexte du développement B, qu'il reviendra de décider.

9.3.1 Validation de la règle

Une règle mathématique ajoutée par l'utilisateur va devoir être validée. Deux cas peuvent se produire :

– la règle est démontrée automatiquement par les outils de preuve de règles. Le travail de validation est réduit à sa plus simple expression. Dans ce cas, on peut ajouter sans soucis la règle au fichier Pmm et l'utiliser.
– la règle n'est pas démontrée automatiquement. Cela ne signifie pas que la règle est fausse. Simplement, le prouveur de prédicats et le prouveur arithmétique n'ont pas réussi à démontrer la justesse de cette règle. Cela peut être dû au fait que certains opérateurs B ne sont pas manipulés de manière efficace par le prouveur de prédicats (par exemple, closure), ou bien encore que les heuristiques utilisées pour la preuve ne sont pas suffisamment efficaces dans ce cas.
  La règle doit alors être démontrée manuellement. Des éléments de démonstration doivent être donnés par le concepteur de la règle afin de tracer le raisonnement ayant servi à sa création. Ces éléments de démonstration permettront à un relecteur de s'assurer de la justesse de la règle. On peut être plus exigeant et rédiger une démonstration mathématique complète de la justesse de la règle, en se reportant à l'axiomatique du B-Book.
  Dans tous les cas, une relecture par un tiers s'avère nécessaire, car ,selon notre expérience, il est très facile de créer une règle fausse sans que l'on s'en aperçoive.

9.3.2 Simplification des lemmes

Un but " trivial " peut ne pas être démontré par le prouveur de prédicats car les termes qu'il contient génèrent des prédicats complexes lors de la traduction du but en prédicats manipulables par le prouveur de prédicats. Dans ce cas, il suffit de remplacer chaque terme

complexe par une variable afin d'obtenir une règle plus facilement démontrable. Cette règle, une fois démontrée, est ajoutée au fichier Pmm, puis utilisée afin de démontrer le but. Par exemple, le but

$\{xx|xx \in INT \ \wedge \ xx\mathsf{mod}10 = 0\} \ \{xx|xx : \mathbb{Z} \ \wedge \ \exists yy.(yy \in \mathbb{Z} \ \wedge \ 10 * yy = xx)\} \ \wedge$
$\{xx|xx \in \mathbb{Z} \ \wedge \ \exists yy.(yy \in \mathbb{Z} \ \wedge \ 10 * yy = xx)\} \subseteq \{xx|xx \in \mathbb{Z} \ \wedge \ xx\mathsf{mod}10 = 0\}$
$\Rightarrow$
$\mathsf{min}(\{xx|xx \in \mathbb{Z} \ \wedge \ xx\mathsf{mod}10 = 0\}) = \mathsf{min}(\{xx|xx \in \mathbb{Z} \ \wedge \ \exists yy.(yy \in \mathbb{Z} \ \wedge \ 10 * yy = xx)\})$

n'est pas démontré par le prouveur de prédicats, ni par le prouveur.
Par contre, la règle

$a \subseteq b \ \wedge$
$b \subseteq a$
$\Rightarrow$
$\mathsf{min}(a) = \mathsf{min}(b)$

est démontrée par les outils de preuve de règles.

```
PRI > vr(Back, (a<:b & b<:a => min(a)=min(b)))
The rule was proved
```

Elle est ajoutée au fichier Pmm, qui contient alors

```
THEORY MyRule IS
    a<:b &
    b<:a
    =>
    min(a)=min(b)
END
```

Elle est ensuite chargée en mémoire grâce à la commande `pc`.

```
PRI> pc
Loading theory MYRule
```

Elle peut enfin être appliquée :

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

9.4 Les différents niveaux des commandes interactives

On peut séparer les commandes d'orientation de la preuve en 2 catégories :
– commandes de haut niveau : `pp`, `pr`, `mp` qui correspondent à l'exécution d'un grand nombre de règles et de mécanismes. Ce sont des commandes résolutoires, c'est à dire que leur application est nécessaire pour démontrer n'importe quel but, même btrue.
– commandes de bas niveau : `dd`, `ah`, `ar`. ce sont des commandes qui s'exécutent en pas à pas. Il n'y a pas de résolution. Leur combinaison permet d'orienter la preuve.

Contrairement aux commandes de la catégorie ci-dessus, ces commandes nécessitent une bonne connaissance/expérience du prouveur interactif.

9.5 Utilisation de SearchRule

La commande SearchRule peut être utilisée dans 2 situations différentes :

– lorsque le prouveur automatique ne parvient pas à démontrer le but courant. La recherche des règles pouvant s'appliquer permet de déterminer quelle est la meilleure candidate et pour quelle raison cette règle ne s'applique pas. Par exemple, une hypothèse peut manquer. Il suffit dans ce cas d'ajouter cette hypothèse afin que cette règle puisse s'appliquer.
Il faut bien avoir à l'esprit que la commande SearchRule affiche toutes les règles susceptibles de s'appliquer et non pas celles qui s'appliquent.
– lors de l'ajout de règles manuelles. La commande SearchRule permet alors de vérifier que la règle qui est chargée en mémoire correspond bien à la règle attendue. Il ne faut pas oublier en effet que les règles contenues dans un fichier Pmm sont normalisée lors de leur chargement.

9.6 Ajout d'hypothèse fausse

Lors d'un ajout d'hypothèse, il faut prendre garde à ne pas ajouter une hypothèse fausse car bien entendu la démonstration de validité de cette hypothèse sera impossible et sera source de crises de nerfs de la part de l'utilisateur.

Comme on sait, si P est le but courant et que la commande `ah(H)` est appliquée alors le but devient

$$H \ \wedge \ (H \ \Rightarrow \ P).$$

L'erreur consiste à mal orthographier le nom de l'identificateur B, ou à inverser 2 noms de variables, ou mal parenthéser une expression : les risques d'erreur sont nombreux.

Le prouveur ne signale pas d'anomalie et l'utilisateur ne sera pas averti de son erreur. Seule une lecture attentive peut lui indiquer où se situe son erreur. Il peut arriver que depuis l'ajout d'hypothèse, plusieurs commandes de preuve ont été saisies, ce qui peut gêner la relecture et la détection de l'erreur.

Veuillez donc faire bien attention lorsque vous ajoutez une hypothèse.

9.7 Nombre de pas nécessaires à une preuve

Parfois si une preuve est trop longue, on se dit qu'on a fait une erreur et on recommence alors que c'est juste. Il faut savoir qu'en moyenne les preuves sont petites mais que certaines peuvent être beaucoup plus longues.

On peut considérer que 10 pas de preuve en moyenne pour les PO non démontrées automatiquement est une bonne métrique.

Toutefois, en fonction du type de modèle et la manière de modéliser, il est possible d'obtenir des PO complexes, nécessitant l'ajout de propriétés et/ou la réalisation de preuve par cas. Dans ce cas, il est bien évident que 10 pas ne constitue pas une bonne valeur puisque sous-estimée.

Pour certains développements B, il arrive que des démonstrations aient plus de 500 pas, à cause de preuves par cas, contenant par exemple chacune de nombreuses commandes interactives presque identiques d'un cas sur l'autre (on peut bien sur s'inquiéter de la conservation de cette preuve lorsque les modèles évoluent).

Un moyen de réduire la taille d'une démonstration est de décomplexifier le modèle B. L'autre moyen consiste à ajouter une ou plusieurs règles utilisateur. Attention ! Elles doivent être facilement vérifiables et ne pas être trop spécifiques.

9.8 Règle qui ne s'applique pas

Lors de l'utilisation de règles, par l'intermédiaire de la commande `ar`, il faut se souvenir que si le but est de la forme $A \Rightarrow B$, A ne fait pas encore partie des hypothèses (il faut faire un `dd` pour cela).

Donc une règle de la forme

$$\mathsf{binhyp}(A) \Rightarrow B$$

ne peut pas s'appliquer tant que A n'est pas une hypothèse.

Attention aussi à éviter d'employer des règles bouclantes.

Par exemple, les règles :

$bcall1(BackwardRule(regle.1)) \wedge$
$\mathsf{binhyp}(H) \wedge$
$(H \Rightarrow B)$
$\Rightarrow$
B

$bcall1(BackwardRule(regle.2)) \wedge$
$\mathsf{binhyp}(H) \wedge$
$\mathsf{bgoal}(C \Rightarrow D) \wedge$
$(H \Rightarrow B) \wedge$
$\Rightarrow$
B

peuvent s'appliquer une infinité de fois.
Il est possible d'éviter ce phénomène :
– en appliquant les règles de manière unitaire : `ar(regle.1, Once)` au lieu de `ar(regle)`
– en gardant les règles. L'ajout de la garde `bnot(bsearch(H,C,R))` permet d'éviter un bouclage par la commande `ar`. Par contre, un appel à `pr(Tac(regle))` provoque un bouclage.
En règle générale, il faut éviter d'ajouter des hypothèses de cette manière. On préférera utiliser des règles forward.

9.9 Utilisation de pp(rp.0)

Le prouveur de prédicats, n'utilisant que le but courant sans aucune hypothèse (commande `pp(rp.0))` se révèle très efficace à l'utilisation, pour démontrer des lemmes intermédiaires. En effet, `pp` est particulièrement efficace quand le nombre d'hypothèses est peu élevé. On peut considérer que son efficacité décroît de façon exponentielle lorsque le nombre d'hypothèses croit.

Aussi, on se contente de sélectionner les hypothèses qui permettent de démontrer le but, par la commande `ah`, puis on applique la commande `pp(rp.0)`.

Le domaine d'utilisation de pp est :
– lemmes arithmétiques, (hors ** et modulos),
– propositions,
– ensembles,
– prédicats quantifiés.
Certains symboles sont moins bien manipulés par `pp` (au niveau de la preuve). Il s'agit entre autre de *closure*, $closure1$, $INTER$, $UNION$, inter, union .... Si votre but comporte un de ces symboles, il est préférable de tenter la simplification/démonstration du but par la combinaison règles manuelle/commande `pr`.

9.10 Pourquoi pr échoue, pp échoue et pp(rp.0) réussit dans certains cas

En preuve interactive, on dispose de 3 prouveurs qui ont chacun leur spécificité :
- le prouveur automatique (appelé par mp ou pr), qui applique des mécanismes généraux de simplification/résolution et les règles de la base de règles.
- le prouveur de prédicats, qui transforme le but à démontrer en prédicats, puis exécute quelques heuristiques chargées de démontrer le lemme par contradiction.
- le prouveur arithmétique, qui ne fonctionne qu'avec une inégalité en but. Il recherche alors, par combinaison linéaire des hypothèses arithmétiques, à démontrer le but par contradiction.

Ces 3 prouveurs ont des domaines qui se recouvrent partiellement. Il est bon de tester l'un ou l'autre des prouveurs sur un but pour voir s'il est effectivement démontré.

Les 2 derniers prouveurs sont uniquement des solveurs, en indiquant si le but est démontré ou non, sans transformation/simplification de celui-ci.

Le prouveur automatique, appliquant des règles d'équivalence, transforme (simplification, découpage) le but et peut le démontrer lorsqu'il se simplifie en btrue.