Centre de Recherche en Informatique de Nancy
INRIA-Lorraine

Mémoire de D.E.A. INFORMATIQUE

Université de Nancy 1
U.F.R. S.T.M.I.A.
Département Informatique

Recherche de preuves en AF_2 par codage dans une logique d'ordre supérieur

Présenté le 7 septembre 1994 par

Manuel MUNIER

Composition du jury :

Monique	GRANDBASTIEN
Didier	GALMICHE
Dominique	MERY
Pierre	MARQUIS
Adam	CICHON

Table des matières

Introduction

Depuis plusieurs années de nombreux travaux ont été consacrés à l'étude de la déduction de programmes dans un cadre logique en utilisant des techniques basées sur la transformation de preuves [Sato 84] ou sur la démonstration de théorèmes [Manna 80]. Nous considèreront ici cette deuxième technique avec une approche particulière, la *programmation par preuves*, utilisant une logique (intuitionniste) constructive, ainsi que ses applications à la programmation [Galmiche 92a].

Différentes théories relatives à cette technique ont été développées, notamment la Théorie des Types de Martin-Löf ($MLTT$) [Martin-Löf 84], le Calcul des Constructions (CC) [Coquand 88, Paulin-Mohring 89] et l'Arithmétique Fonctionnelle du second ordre (AF_2) [Parigot 88, Krivine 90]. Programmer dans un tel cadre logique consiste à donner une spécification formelle (ou proposition dans le système), à en construire une preuve, puis à extraire automatiquement de cette preuve un code exécutable en utilisant, par exemple, l'isomorphisme *formulae as types* de Curry-Howard [Howard 80]. De ce fait, la notion de programmation se fonde sur la recherche de preuves en théorie des types [Galmiche 90, Helmink 90, Dowek 93, Paulin-Mohring 89, Nordström 90]. Certains résultats mathématiques nous permettent d'obtenir d'importantes propriétés telles que la correction et la terminaison des programmes ainsi synthétisés [Krivine 87].

Dans le cadre de ce rapport, nous allons étudier la recherche de preuves dans la théorie AF_2. Cette logique a déjà été l'objet de nombreux travaux, notamment en ce qui concerne l'études des liens entre preuves et programmes (les meilleures preuves ne donnent pas obligatoirement les meilleurs algorithmes [Galmiche 92b, Parigot 88]) et l'automatisation de cette recherche de preuves (notions de tactiques, de plans de preuves [Galmiche 92b, Galmiche 93] et de $\mathcal{R}$-preuves [Parigot 92a, Manoury 92, Parigot 92b]).

L'angle sous lequel nous allons aborder le problème de la recherche de preuves et de la synthèse de programmes en AF_2 est quelque peu particulier. Nous allons coder la logique AF_2 dans une autre logique d'ordre supérieur pour laquelle on sait définir des procédures de recherche de preuves qui reposent sur des classes de preuves complètes. Ceci est une approche complémentaire aux travaux cités ci-dessus. Elle doit permettre une meilleure compréhension de l'automatisation de

la recherche de preuves. Nous étudierons la recherche de preuves dans la logique intuitionniste d'ordre supérieur $\mathcal{I}$ pour laquelle ont été présentées dans [Miller 91] la notion de *preuve uniforme* (tout séquent ayant comme succédent une formule non atomique est la conclusion d'une règle d'introduction à droite) ainsi qu'une procédure de recherche *complète* (s'il existe une preuve, cette procédure la trouvera) mais *non-déterministe* associée. Notre objectif est donc de représenter la théorie AF_2 dans la logique $\mathcal{I}$ afin de pouvoir utiliser des procédures de recherche sur les formules d'AF_2 ainsi codées. En étudiant les résultats de cette recherche de preuves par codage dans une autre logique, on devrait pouvoir définir directement pour AF_2 une notion de preuve canonique, comme celles reposant sur les preuves uniformes de $\mathcal{I}$ ou bien d'autres [Nadathur 93], ainsi qu'une procédure de recherche de preuves (afin de pouvoir en extraire des programmes) correspondante.

Le premier chapitre sera consacré à la présentation de la théorie des types AF_2 [Parigot 88, Krivine 90]. On explicitera ensuite la notion de *programmation par preuves* dans ce cadre logique en insistant sur la représentation des types de données qui est essentielle pour la synthèse de programmes. Ceci sera illustré par un exemple simple que l'on retrouvera dans la suite du rapport pour mettre en évidence les différents points de notre travail.

Après une analyse de deux approches de la recherche de preuves, l'une reposant sur les notions de *tactiques* et de *plans de preuves* [Bundy 88, Bundy 91, Galmiche 92b, Galmiche 93], l'autre utilisant la notion de *preuves canoniques*, telles que les preuves uniformes [Miller 91] et les $\mathcal{R}$-preuves présentées dans [Parigot 92a, Parigot 92b, Manoury 92], nous proposerons une nouvelle approche qui sera étudiée au chapitre 3. Celle-ci consiste à coder AF_2 dans la logique $\mathcal{I}$ pour laquelle on dispose, en particulier, d'une procédure de recherche de preuves complète mais non déterministe basée sur la notion de preuves uniformes [Miller 91]. Nous pourrons ainsi étudier la recherche de preuves dans $\mathcal{I}$ sur les formules d'AF_2 codées.

Au second chapitre, nous présenterons la *logique intuitionniste d'ordre supérieur* $\mathcal{I}$, ainsi que le langage de programmation logique λProlog (version MALI [Brisset 93]) basé sur une restriction de $\mathcal{I}$: les formules de Harrop héréditaires d'ordre supérieur (*hh*) [Miller 91]. L'interprète de ce langage repose sur une implantation déterministe de la procédure de recherche de preuves uniformes. Nous pourrons ainsi mettre en œuvre notre codage sans devoir, entre autres, réécrire cette procédure.

Ayant étudié le codage du Calcul des Constructions dans $\mathcal{I}$ détaillé dans [Felty 93a], nous proposerons, dans le troisième chapitre, notre propre *codage* pour AF_2. Cette analyse sera faite en deux étapes : tout d'abord pour les termes et les jugements d'AF_2, puis pour les relations de convertibilité qui, comme nous

le verrons lors de la recherche de preuves, sont source de difficulté du fait de l'introduction d'un indéterminisme supplémentaire. Nous illustrerons ceci par le codage de l'exemple donné au chapitre 1. Nous détaillerons ensuite la *preuve de correction* de ce codage (si un jugement a une dérivation dans AF_2 alors son codage a une preuve dans $\mathcal{I}$), suivie d'une analyse du problème de la complétude.

A partir de là, nous pourrons aborder la *recherche de preuves* en analysant le comportement de la procédure de recherche donnée pour $\mathcal{I}$. Nous verrons alors que cette procédure, telle qu'elle est définie, ne permet pas de retrouver les preuves d'AF_2 utilisant la règle (app_i), et nous proposerons des modifications de cette procédure pour permettre l'automatisation de la recherche de preuves. La mise en œuvre de ce codage en λProlog (version MALI) nous conduira à analyser différents points, tels que la vérification de type.

Nous développerons ensuite notre exemple de façon à mettre en évidence les modifications nécessaires sur la procédure de recherche pour $\mathcal{I}$.

Dans le dernier chapitre, nous donnerons une représentation d'AF_2 sous forme de tactiques. Chacune d'entre elles correspondra à une règle d'inférence d'AF_2. En enchaînant ces tactiques, on peut ainsi construire une preuve d'une formule d'AF_2 donnée. De nombreux travaux ont été effectués afin d'automatiser cet enchaînement, notamment sur les notions de *tactiques* et de *plans de preuves* [Bundy 88, Bundy 91, Galmiche 92b, Galmiche 93], ainsi que sur celles de $\mathcal{R}$-*preuves* [Parigot 92a, Parigot 92b, Manoury 92]. L'objectif est de développer, au-dessus de cette notion de tactique, une procédure de recherche (sous la forme d'une méta-tactique) dont le rôle serait de définir l'ordre dans lequel seront appliquées les différentes tactiques, et qui serait basée sur les travaux effectués au chapitre précédent en définissant une classe de preuves canoniques similaire à celle des preuves uniformes et en tenant compte des modifications proposées pour la procédure de recherche de $\mathcal{I}$.

Dans le cadre de ce rapport, nous nous limiterons à la définition des *tactiques* de base et de quelques méta-tactiques élémentaires en nous inspirant des résultats présentés dans [Felty 93b, Barraband 93]. Nous proposerons ensuite une *procédure de recherche interactive* (l'utilisateur doit, à chaque étape, préciser la tactique à appliquer au but courant) pour laquelle l'indéterminisme au niveau de la règle (app_i) qui était source de problème au chapitre précédent est résolu en demandant les informations nécessaires à l'utilisateur.

La conclusion fera le bilan de cette étude et proposera quelques perspectives de travail, comme le développement d'une procédure de recherche pour AF_2 tenant compte des résultats établis au chapitre 3. Cette procédure pourrait, comme précisé au chapitre 4, être mise en œuvre en utilisant les notions de *tactique* et de *méta-tactique*.

Chapitre 1

Une théorie des types du second ordre : AF_2

Dans ce premier chapitre nous allons donc vous présenter le système logique AF_2 ainsi que le concept de programmation par preuves. En guise d'exemple, nous détaillerons la synthèse d'un algorithme calculant la somme de deux entiers naturels. Nous vous présenterons ensuite la notion de recherche de preuve en expliquant les différentes façons dont on peut aborder le problème.

1.1 Le système logique AF_2

AF_2 est un système de λ-calcul typé dont les types sont des formules de la logique du second ordre [Krivine 90]. De ce fait, AF_2 diffère du Calcul des Constructions [Coquand 88] de deux façons : d'une part, les types d'AF_2 ne peuvent dépendre de termes ; d'autre part, nous disposons dans AF_2 de la quantification au second ordre. Une autre caractéristique importante de la logique du second ordre en tant que langage de programmation, par rapport aux théories des types dans le style de Martin-Löf [Martin-Löf 84], est que la représentation des données dans le langage machine découle de la définition du type de données.

Une démonstration dans AF_2 d'une formule A est un arbre fini construit en utilisant les règles d'inférence d'AF_2 avec comme racine la formule A. Le système de déduction naturelle permet de démontrer une formule d'AF_2, modulo des hypothèses qui sont introduites puis déchargées au cours de la preuve. La structure de la démonstration est celle d'un λ-terme. Ce λ-terme est assimilé, d'après l'isomorphisme de Curry-Howard [Howard 80], à un programme alors que la formule dérivée est assimilée aux spécifications du programme [Parigot 88]. Par conséquent, AF_2 est vu comme un langage de programmation permettant à la fois la spécification, la synthèse, et la vérification des programmes [Parigot 92a, Parigot 92b].

Les termes de ce λ-calcul sont obtenus à partir des variables $x, y, z, ...$ par un nombre fini d'application des règles suivantes:

- Si t et u sont des termes alors $(t\ u)$ est un terme. Celui-ci représente l'application de la fonction t sur l'argument u.
- Si x est une variable et t un terme alors $\lambda x.t$ est un terme. Celui-ci représente la fonction qui attend un argument et renvoie le terme t dans lequel nous avons remplacé toutes les occurrences de x par l'argument.

La notion de calcul pour ces termes est la *réduction* et le λ-calcul peut être transformé en un langage de programmation en implantant cette réduction.

Les formules (ou types) de la logique du second ordre sont construites en utilisant le connecteur $\supset$, le quantificateur $\forall$, les variables d'individu $x, y, z, ...$, les variables de prédicat d'arité arbitraire $X, Y, Z, ...$, et des constantes de prédicat, de fonction et d'individu. Les termes d'individu sont construits à partir des constantes de fonctions et d'individus. Nous devons noter que dans la logique (intuitionniste) du second ordre, les symboles logiques $\bot, \vee, \wedge, \neg$ et $\exists$, ainsi que la relation d'identité $=$, peuvent être définis à partir de $\supset$ et $\forall$. Par exemple $A \wedge B$ est défini par la formule $\forall X.((A \supset (B \supset X)) \supset X)$.

Les jugements du langage sont de la forme $t : A$ avec les interprétations suivantes : «*t est un élément du type A*» ou «*t est une preuve de la formule A*». C'est le dernier cas qui est utilisé ici; les formules sont ensuite considérées comme des types et les démonstrations de ces formules comme des λ-termes. La construction des démonstrations est basée sur l'application des axiomes et des règles d'inférence. Les règles de déduction pour l'Arithmétique Fonctionnelle du second ordre se trouvent à la Figure 1 (A et B représentent des formules et Γ un ensemble de formules).

Le système est très simple : $\supset$ est le seul symbole logique ayant un contenu algorithmique. En effet, une preuve de $A \supset B$ est un algorithme qui transforme chaque preuve de A en une preuve de B. Les quantificateurs ont juste un rôle conceptuel permettant de décrire ce que le programme fait.

Du point de vue de la programmation, l'expression «*t est de type A*», qui lie un terme avec une formule, doit être lue comme «*le programme t réalise la spécification A*». Les propriétés les plus importantes des λ-termes obtenus à partir de preuves sont *la préservation de types* (le type est conservé par réduction) et *la terminaison* d'après le théorème de normalisation (les λ-termes obtenus à partir de preuves représentent des algorithmes qui terminent toujours). Ce système peut être considéré comme un langage de programmation nous permettant d'écrire des spécifications exactes et des programmes corrects. Le λ-calcul est considéré comme le code machine dans lequel les preuves sont compilées. Pour plus de précisions sur la sémantique de cette méthode de programmation, voir [Parigot 88, Krivine 87].

Règle de l'axiome ($axiom$)	$\Gamma, t : A \vdash t : A$	
Règle d'abstraction (abs_i)	$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B}$	
Règle d'application (app_i)	$\dfrac{\Gamma \vdash t : A \supset B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B}$	
Règle de généralisation (gen_1)	$\dfrac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall x.A}$	x non libre dans Γ
Règle de spécialisation (spe_1)	$\dfrac{\Gamma \vdash t : \forall x.A}{\Gamma \vdash t : [b/x]A}$	b est un individu
Règle de généralisation (gen_2)	$\dfrac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall X.A}$	X non libre dans Γ
Règle de spécialisation (spe_2)	$\dfrac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t : [B/X]A}$	B est une formule
Règle structurelle	$\dfrac{\Gamma \vdash t : A}{\Delta \vdash t : A}$	Δ est obtenu à partir de Γ par permutation, par affaiblissement ou par contraction

Figure 1 : Règles d'inférence en déduction naturelle d'AF_2

1.2 Preuves et programmation en AF_2

L'étude des relations entre la logique intuitionniste et l'informatique fait l'objet de nombreux travaux théoriques, et quelques systèmes ont été développés autour du paradigme «*programs as proofs*» dans ses différentes formulations [Coquand 88, Henson 89, Paulin-Mohring 89, Paulin-Mohring 93, Galmiche 90, Harper 93, Martin-Löf 84]. Nous allons détailler la notion de *programmation par preuves* dans le cadre logique défini précédemment, puis nous aborderons la représentation des types de données qui est essentielle pour les choix d'induction et donc la synthèse de programmes. Nous donnerons ensuite un exemple de preuve d'une spécification.

1.2.1 Programmer avec des preuves

En considérant ce cadre logique, la construction de programmes se divise en trois principales étapes :

1. Nous représentons les types de données par des formules. A partir de la définition au second ordre d'un type de données nous pouvons extraire une représentation de ces données en λ-calcul. Cette représentation est très importante pour la mise en œuvre. Par exemple, dans [Parigot 88] il est question de l'influence de cette représentation des données sur l'efficacité des programmes obtenus par dérivation.

   Afin d'illustrer nos propos, nous allons développer, au cours de ce rapport, l'exemple de l'addition sur les entiers naturels N. L'ensemble des entiers naturels peut être défini comme «*le plus petit ensemble contenant zéro et clos pour l'opération successeur*». Formellement, nous introduisons des constructeurs de type : une constante d'individu $\underline{0}$ (pour zéro) et une constante de fonction $\underline{s}$ (pour l'opération successeur), puis nous considérons la formule au second ordre Nx signifiant «*x est un entier naturel*» :

   $$Nx \equiv \forall X.\Big(\big(\forall y.(Xy \supset X(\underline{s}\ y))\big) \supset (X\underline{0} \supset Xx)\Big)$$

2. Nous exprimons les spécifications du programme par un ensemble d'équations définissant sémantiquement la fonction que nous voulons calculer. Par exemple, pour l'addition sur les entiers naturels, nous obtenons le système d'équations suivant :

   $$\begin{cases} plus(\underline{0}, x) & = & x \\ plus(x, \underline{s}(y)) & = & \underline{s}(plus(x, y)) \end{cases}$$

3. Le programme est obtenu en dérivant la proposition indiquant que la fonction a le type désiré. Pour notre exemple il s'agit de la proposition «*si x et y sont des entiers naturels, alors l'addition de x et de y, notée $plus(x, y)$, est un entier naturel*».

Ce que nous venons de présenter est une adaptation du schéma de construction général (spécification par des formules, construction d'une preuve puis extraction du programme de cette preuve) au cadre logique AF_2 [Parigot 88]. Il faut cependant signaler que l'efficacité d'un algorithme ne peut pas s'exprimer en λ-calcul et que les meilleures preuves ne donnent pas obligatoirement les meilleurs algorithmes [Galmiche 92b, Parigot 88].

1.2.2 Types de données

Nous avons trois façons de définir les types de données : de manière *itérative*, de manière *récursive*, et *à la Martin-Löf*. A titre d'exemple, nous allons donner

les trois représentations correspondantes du type entier naturel.

- **Définition itérative**

  Il s'agit de la définition donnée auparavant :

  $$Nx \equiv \forall X.\Big(\big(\forall y.(Xy \supset X(\underline{s}\ y))\big) \supset (X\underline{0} \supset Xx)\Big)$$

  Comme nous l'avons déjà signalé, une caractéristique importante de la logique du second ordre en tant que langage de programmation, par rapport aux théories des types dans le style de Martin-Löf [Martin-Löf 84], est que la représentation des données dans le langage machine découle de la définition du type de données.

  Par exemple, une représentation du constructeur $\underline{0}$ en λ-calcul est donnée par un terme du type $N\underline{0}$. Cela signifie qu'elle est obtenue par une dérivation formelle de $N\underline{0}$. Ainsi, la représentation du constructeur $\underline{0}$ est $0 = \lambda f.\lambda a.a$, ce qui est précisemment l'entier de Church 0.

  De la même manière, une représentation du constructeur $\underline{s}$ en λ-calcul est obtenue par une dérivation formelle de $\forall x.(Nx \supset N(\underline{s}\ x))$. Nous obtenons $s = \lambda\nu.\lambda f.\lambda a.(f\ (\nu\ f\ a))$, ce qui est précisemment le terme pour la fonction successeur sur les entiers de Church.

  La propriété cruciale pour la programmation est que la représentation $(s^n\ 0)$ du nombre n, obtenue par une preuve de $N(\underline{s}^n\ \underline{0})$, est en fait l'unique terme du type $N(\underline{s}^n\ \underline{0})$ (modulo la réduction). Une preuve de ce résultat est donnée dans [Krivine 90].

- **Définition récursive**

  Nous introduisons un prédicat unaire N et deux constructeurs $\underline{0}$ (pour zéro) et $\underline{s}$ (pour l'opération successeur). Nous définissons l'interprétation intuitionniste $y \in Nx$ du type entier N comme la solution minimale K de l'équation

  $$Kx = \forall X.\Big(\big(\forall y.(Ky \supset (Xy \supset X(\underline{s}\ y)))\big) \supset (X\underline{0} \supset Xx)\Big)$$

  Une des motivations pour cette définition récursive est la suivante : l'équation $Nx = \forall X.((\forall y.(Ny \supset (Xy \supset X(\underline{s}\ y)))) \supset (X\underline{0} \supset Xx))$ est une formulation possible de l'induction où l'étape inductive n'est pas donnée pour un élément arbitraire, mais uniquement pour les entiers naturels.

  Une représentation 0 du constructeur $\underline{0}$ en λ-calcul est donnée par un terme du type $N\underline{0} = \forall X.((\forall y.(Ny \supset (Xy \supset X(\underline{s}\ y)))) \supset (X\underline{0} \supset X\underline{0}))$. Un tel terme est $\lambda f.\lambda a.a$

Une représentation s du constructeur $\underline{s}$ en λ-calcul est donnée par un terme du type $\forall x.(Nx \supset N(\underline{s}\ x))$. Nous obtenons $\lambda\nu.\lambda f.\lambda a.((f\ \nu)\ (\nu\ f\ a))$

Du fait que nous ayons un prédicat au lieu d'une formule, nous devons également définir l'interprétation classique Nx du type entier N. Nous prenons bien entendu «*le plus petit ensemble contenant* 0 *et clos par la fonction générée par s*». Nous avons ainsi

$$Nx \equiv \forall X.\Big(\big(\forall y.(Xy \supset X(\underline{s}\ y))\big) \supset (X\underline{0} \supset Xx)\Big)$$

– **Définition à la Martin-Löf**

Il est possible d'exprimer les types de données d'une autre manière en utilisant la Théorie des Types de Martin-Löf [Martin-Löf 84]. Chaque type est introduit par un système de règles d'inférence dont l'aspect constructif est exprimé par des opérations sur les objets preuve. Un des avantages de cette définition est qu'il n'est plus nécessaire de se poser la question : doit-on déplier tous les types immédiatement ou bien lorsque cela est nécessaire ? Ces règles sont de quatre types :

- Les règles de **formation** construisant le type voulu à partir de types déjà définis.
- Les règles d'**introduction** qui construisent des éléments du type. Par l'isomorphisme «*formulae as types*» de Curry-Howard ces règles introduisent des connecteurs logiques.
- Les règles d'**élimination** spécifiant des structures de contrôle (ie des schémas d'induction). Elles éliminent des connecteurs logiques.
- Les règles d'**égalité** qui donnent le résultat des expressions évaluées.

Voici le système de règles définissant le type entier noté N :

a) formation
$$\frac{}{N\ type}$$

b) introduction
$$\frac{}{0\ :\ N\underline{0}} \qquad \frac{n\ :\ Nx}{(s\ n)\ :\ N(\underline{s}\ x)}$$

c) élimination
$$\frac{n\ :\ Nx \qquad \alpha\ :\ C(\underline{0}) \qquad \begin{array}{c} u\ :\ C(y) \\ \vdots \\ (\beta\ u)\ :\ C(\underline{s}\ y) \end{array}}{rec\ n\ \alpha\ \beta\ :\ C(n)}$$

d) égalité
$$rec\ 0\ \alpha\ \beta \longrightarrow \alpha$$
$$rec\ (s\ n)\ \alpha\ \beta \longrightarrow \beta\ (rec\ n\ \alpha\ \beta)$$

Une différence essentielle de l'approche de Martin-Löf par rapport aux types itératifs ou récursifs définis précédemment est que la définition du type N donne l'implémentation de $\underline{0}$, $\underline{s}$ et rec.

La principale difficulté pour programmer avec des types de données récursifs exprimés de cette façon réside dans le fait que pour chaque type de données il est d'abord nécessaire de définir des opérateurs tels que rec. Cet inconvénient peut être contourné en utilisant une autre méthode de programmation basée sur le type de données universel U et un opérateur de point fixe [Parigot 92a].

Le choix de la représentation du type de données est essentiel pour la programmation par preuves du fait de son influence sur les λ-termes (ie les programmes) obtenus lors des dérivations. Les travaux présentés dans [Parigot 88, Parigot 92a, Galmiche 92b] concernant l'efficacité des programmes abordent d'ailleurs le problème au niveau de la définition des types de données.

1.2.3 Construction de preuves

Voici deux programmes représentant l'addition de deux entiers. La première utilise la définition itérative de N alors que la seconde utilise la définition de N dans le style de Martin-Löf. Selon la définition choisie nous obtiendront deux algorithmes différents, ce qui mettra en évidence l'influence de la représentation du type de données sur l'algorithme obtenu.

Il nous faut donner une dérivation de $\forall x.(Nx \supset \forall y.(Ny \supset Nplus(x,y)))$ dont la signification est : «*si x et y sont deux entiers naturels, alors l'addition de x et de y, notée $plus(x,y)$ est un entier naturel*». Le λ-terme construit durant cette dérivation sera la représentation en λ-calcul de la fonction $plus$.

a. **Version itérative**

La démonstration sera construite en déduction naturelle en utilisant les règles d'inférence présentées Figure 1 ainsi que les règles suivantes :

- La règle $(unfold)$ permettant de remplacer Nx par la formule au second ordre :

$$\frac{\Gamma \vdash t : Nx}{\Gamma \vdash t : \forall X.\Big(\big(\forall y.(Xy \supset X(\underline{s}\ y))\big) \supset (X\underline{0} \supset Xx)\Big)}$$

- La règle $(fold)$ effectuant l'opération inverse :

$$\frac{\Gamma \vdash t : \forall X.\Big(\big(\forall y.(Xy \supset X(\underline{s}\ y))\big) \supset (X\underline{0} \supset Xx)\Big)}{\Gamma \vdash t : Nx}$$

– Les règles (*rewrite*) permettant d'utiliser les égalités de la spécification.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1 \quad \Pi_2}{\Gamma \vdash tf : Xy \supset Xplus(x,y)}\ (app_i) \quad \dfrac{\dfrac{\Pi_3 \quad \Pi_4}{\Gamma \vdash uf : X\underline{0} \supset Xy}\ (app_i) \quad \Gamma \vdash a : X\underline{0}}{\Gamma \vdash ufa : Xy}\ (app_i)}{t : Nx, u : Ny, f : \forall z.(Xz \supset X(\underline{s}z)), a : X\underline{0} \vdash tf(ufa) : Xplus(x,y)}\ (app_i)}{t : Nx, u : Ny \vdash \lambda fa.tf(ufa) : (\forall z.(Xz \supset X(\underline{s}z))) \supset (X\underline{0} \supset Xplus(x,y))}\ (abs_i + abs_i)}{t : Nx, u : Ny \vdash \lambda fa.tf(ufa) : \forall X.((\forall z.(Xz \supset X(\underline{s}z))) \supset (X\underline{0} \supset Xplus(x,y)))}\ (gen_2)}{t : Nx, u : Ny \vdash \lambda fa.tf(ufa) : Nplus(x,y)}\ (fold)}{t : Nx \vdash \lambda ufa.tf(ufa) : \forall y.(Ny \supset Nplus(x,y))}\ (abs_i + gen_1)}{\vdash \lambda tufa.tf(ufa) : \forall x.(Nx \supset \forall y.(Ny \supset Nplus(x,y)))}\ (abs_i + gen_1)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{hypothèse dans } \Gamma}{\Gamma \vdash t : Nx}\ (axiom)}{\Gamma \vdash t : \forall Y.((\forall z.(Yz \supset Y(\underline{s}\ z))) \supset (Y\underline{0} \supset Yx))}\ (unfold)}{\Gamma \vdash t : (\forall z.(Xplus(z,y) \supset Xplus(\underline{s}\ z, y))) \supset (Xplus(\underline{0}, y) \supset Xplus(x,y))}\ (spe_2)}{\Gamma \vdash t : (\forall z.(Xplus(z,y) \supset X(\underline{s}\ plus(z,y)))) \supset (Xy \supset Xplus(x,y))}\ (rewrite)$$

preuve Π_1

$$\dfrac{\dfrac{\dfrac{\text{hypothèse dans } \Gamma}{\Gamma \vdash f : \forall z.(Xz \supset X(\underline{s}\ z))}\ (axiom)}{\Gamma \vdash f : Xplus(z', y) \supset X(\underline{s}\ plus(z', y))}\ (spe_1)}{\Gamma \vdash f : \forall z.(Xplus(z,y) \supset X(\underline{s}\ plus(z,y)))}\ (gen_1)$$

preuve Π_2

$$\dfrac{\dfrac{\dfrac{\text{hypothèse dans } \Gamma}{\Gamma \vdash u : Ny}\ (axiom)}{\Gamma \vdash u : \forall Y.((\forall z.(Yz \supset Y(\underline{s}\ z))) \supset (Y\underline{0} \supset Yy))}\ (unfold)}{\Gamma \vdash u : (\forall z.(Xz \supset X(\underline{s}\ z))) \supset (X\underline{0} \supset Xy)}\ (spe_2)$$

preuve Π_3

$$\dfrac{\text{hypothèse dans } \Gamma}{\Gamma \vdash f : \forall z.(Xz \supset X(\underline{s}\ z))}\ (axiom)$$

preuve Π_4

Le λ-terme $\lambda tufa.tf(ufa)$ construit lors de la dérivation représente l'algorithme de l'opération $plus$.

b. **Version récursive**

Le système de règles d'inférence utilisé comprend les règles de la Figure 1 ainsi que les règles de définition du type de données, notamment les règles d'introduction et d'élimination.

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{hypothèse dans } \Gamma}{\Gamma \vdash u : Nx}\ (axiom) \quad \dfrac{\dfrac{\text{hypothèse dans } \Gamma}{\Gamma \vdash v : Ny}\ (axiom)}{\Gamma \vdash v : Nplus(\underline{0}, y)}\ (rewrite) \quad \Pi_1}{u : Nx, v : Ny \vdash rec\ u\ v\ s : Nplus(x, y)}\ (elim.)}{u : Nx \vdash \lambda v.rec\ u\ v\ s : \forall y.(Ny \supset Nplus(x, y))}\ (abs_i + gen_1)}{\vdash \lambda uv.rec\ u\ v\ s : \forall x.(Nx \supset \forall y.(Ny \supset Nplus(x, y)))}\ (abs_i + gen_1)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{hypothèse}}{\Gamma, w : Nplus(z, y) \vdash w : Nplus(z, y)}\ (axiom)}{\Gamma, w : Nplus(z, y) \vdash (s\ w) : N(\underline{s}\ plus(z, y))}\ (intro.)}{\Gamma \vdash \lambda w.(s\ w) : Nplus(z, y) \supset N(\underline{s}\ plus(z, y))}\ (abs_i)}{\Gamma \vdash s : Nplus(z, y) \supset Nplus(\underline{s}\ z, y)}\ (rewrite)$$

preuve Π_1

Le λ-terme $\lambda uv.rec\ u\ v\ s$ construit lors de la dérivation représente l'algorithme de l'opération $plus$.

Nous pouvons remarquer que l'algorithme obtenu en utilisant la définition itérative est lui-même itératif, alors que celui obtenu avec la définition à la Martin-Löf est récursif. Comme nous l'avions déjà signalé, le choix de la représentation des types de données influe donc sur la structure du programme qui sera obtenu par dérivation. Il faut cependant savoir que les meilleures preuves ne donnent pas forcément les meilleurs algorithmes. Le problème de l'efficacité des programmes obtenus est traité dans [Galmiche 92b, Parigot 88]. Ici, nous ne nous intéresserons qu'á l'aspect automatique de la recherche de preuves.

1.3 Recherche de preuves

Nous venons de présenter la notion de programmation par preuves. L'idée essentielle est qu'un programme est extrait d'une preuve de sa spécification. Le point que nous allons maintenant aborder est la recherche de cette preuve. De nombreux travaux ont été effectués dans ce domaine afin d'automatiser cette recherche [Felty 93a, Galmiche 92b, Parigot 92a, Miller 91, Dowek 93].

1.3.1 Plans de preuves

Une première approche consiste à développer une procédure de recherche basée sur des heuristiques afin d'automatiser les preuves que nous faisons habituellement à la main. En utilisant la notion de *tactique* nous pouvons implanter les différentes règles d'inférence du cadre logique dans lequel nous travaillons. La procédure de recherche sera mise en œuvre par des *méta-tactiques* (appelées *tacticals* en anglais) qui conditionnent l'enchaînement des différentes tactiques.

L'intérêt d'une telle approche est que les preuves obtenues sont les mêmes que celles développées à la main. Elle est de plus applicable à tout type de données, qu'il soit itératif ou récursif. La notion de *plans de preuves* détaillée dans [Bundy 88, Bundy 91] et utilisée dans [Galmiche 92b, Galmiche 93] entre dans ce domaine de la recherche de preuves. Le principal inconvénient est que les procédures de recherche ainsi développées reposent sur des heuristiques, et de ce fait il est difficile d'obtenir une procédure qui soit générale et complète (ie s'il existe une preuve cette procédure la trouvera).

1.3.2 Preuves canoniques

Une seconde approche consiste à restreindre le domaine des preuves étudiées afin de disposer, sur ce nouveau domaine, d'une procédure de recherche complète. Parmi les travaux effectués nous pouvons citer les notions de *preuves uniformes* [Miller 91] et de $\mathcal{R}$-*preuves* [Parigot 92a, Parigot 92b, Manoury 92]. L'intérêt d'une telle approche réside dans le fait que l'on a une preuve formelle de

la complétude de la procédure de recherche pour les preuves du domaine ainsi défini.

1.3.3 Une troisième approche

Une troisième façon d'aborder le problème est de coder la logique dans laquelle nous voulons travailler dans une autre logique X pour laquelle nous disposons déjà d'une procédure de recherche. Nous pouvons ainsi utiliser cette procédure sur les formules codées de la logique initiale. Un exemple de cette approche est donné dans [Felty 93a] pour CC. Nous pouvons également inverser le problème en déduisant de la procédure de recherche existant pour X une procédure de recherche pour la logique intiale.

C'est cette dernière approche que nous allons développer pour AF_2 en nous basant sur les travaux présentés dans [Felty 93a] pour le Calcul des Constructions. En ce qui concerne la logique X, nous choisissons la logique intuitionniste d'ordre supérieur $\mathcal{I}$ pour laquelle a été définie la notion de *preuves uniformes* [Miller 91]. Celle-ci est en fait une extension des formules de Harrop héréditaires d'ordre supérieur (hh). Il existe également un langage de programmation logique, λProlog, basé sur hh. Nous pourrons ainsi vérifier sur machine notre codage sans pour autant devoir implanter la procédure de recherche.

Dans un premier temps, nous allons donc proposer un codage d'AF_2 dans la logique $\mathcal{I}$. Nous tenterons alors d'utiliser la procédure de recherche de preuves uniformes sur les formules d'AF_2 codées. En analysant les problèmes rencontrés nous proposerons des modifications de cette procédure. Par exemple, bien que le codage soit correct du point de vue théorique, les relations de convertibilité introduisent un indéterminisme supplémentaire. Une solution proposée dans [Felty 93a] consiste à n'utiliser que des termes en forme normale. De ce fait, les formules exprimant ces relations seraient remplacées par une procédure de normalisation. Cet indéterminisme serait ainsi supprimé.

Dans un deuxième temps, nous allons définir les tactiques représentant le système de règles d'inférence pour AF_2, comme dans la première approche de la recherche de preuves. Nous pourrons alors développer, à l'aide de méta-tactiques, une procédure de recherche pour AF_2 basée sur celle existant pour $\mathcal{I}$ et tenant compte des problèmes soulevés.

Dans le cadre de ce rapport nous n'étudierons que le codage d'AF_2 dans $\mathcal{I}$ et l'utilisation de la procédure de recherche de preuves uniformes, puis la représentation des règles d'AF_2 par des tactiques. Le développement d'une procédure de recherche pour AF_2 pourrait constituer une suite à ce travail.

Chapitre 2

Une logique intuitionniste d'ordre supérieur : $\mathcal{I}$

Dans ce chapitre nous allons présenter la logique $\mathcal{I}$ qui nous servira de méta-logique pour le codage d'AF_2. Après avoir donné une définition des formules de Harrop héréditaires d'ordre supérieur (hh), nous présenterons le langage λProlog basé sur cette logique. Nous insisterons sur certains choix d'implantation effectués sur la version que nous utiliserons (MALI Prolog, développé à l'IRISA Rennes [Brisset 93]). Nous mettrons ainsi en évidence une différence essentielle entre $\mathcal{I}$ et hh au niveau de l'opération BACKCHAIN néces-saire dans la procédure de recherche de preuves uniformes. Nous terminerons ce chapitre en donnant plusieurs exemples de programmes en λProlog.

2.1 Présentation de la logique $\mathcal{I}$

Comme nous l'avons déjà mentionné, cette logique possède la propriété de *preuve uniforme*. En se basant sur cette propriété nous décrirons la procédure de recherche non-déterministe proposée par Nadathur et Miller [Miller 91].

2.1.1 Le langage

Les types et les termes de $\mathcal{I}$ sont essentiellement ceux de la théorie des types simple [Church 40]. Nous supposons que nous avons au départ un ensemble de *types primitifs* qui contient au moins le symbole o, le type des propositions. Les *types fonctionnels* sont construits en utilisant le symbole binaire infixé $\rightarrow$: si τ et σ sont des types, alors $\tau \rightarrow \sigma$ est un type. Le constructeur de type $\rightarrow$ est associatif à droite. L'*ordre* d'un type primitif est 0 alors que l'ordre d'un type fonctionnel $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ (où $n \geq 0$ et τ_0 type primitif) est égal à $1 + max(order(\tau_1), \cdots, order(\tau_n))$.

Pour chaque type τ nous supposons qu'il existe un ensemble dénombrable de constantes et de variables de ce type. Les λ-termes simplement typés sont construits de façon habituelle à partir de ces constantes et variables en utilisant l'application et l'abstraction. Nous supposons que le lecteur est familier avec les notions et propriétés usuelles de substitution et de α, β et η conversion pour le λ-calcul simplement typé. Vous trouverez de plus amples détails sur ces propriétés de base dans [Hindley 86]. Si x est une variable et si t est un terme, alors $[t/x]$ représente la substitution de t à toutes les occurrences libres de x en renommant systématiquement les variables liées afin d'éviter la capture de variable.

Une constante ou variable p de type $\tau_1 \to \cdots \to \tau_n \to o$ est appelée une constante ou variable de *prédicat*. Une *formule atomique* est un terme de type o de la forme $pt_1 \cdots t_n$ où p est une constante ou variable de prédicat. Le prédicat p est appelé la *tête* de cette formule atomique. Les connecteurs logiques sont définis en introduisant des constantes appropriées, comme dans [Church 40]. Les constantes $\wedge$ (conjonction) et $\supset$ (implication) sont toutes deux du type $o \to o \to o$, et $\forall_\tau$ (quantification universelle) est du type $(\tau \to o) \to o$ pour chaque type τ. L'expression $\forall_\tau(\lambda z.t)$ s'écrira tout simplement $\forall_\tau z.t$, ou $\forall z.t$ si le type τ peut être déduit du contexte. $\forall$ représentera donc le quantificateur universel polymorphique. Nous devons noter que dans la logique (intuitionniste) du second ordre, les symboles logiques $\bot, \vee, \wedge, \neg$ et $\exists$, ainsi que la relation d'identité $=$, peuvent être définis à partir de $\supset$ et $\forall$. Par exemple $A \wedge B$ est défini par la formule $\forall X.((A \supset (B \supset X)) \supset X)$.

2.1.2 Les formules de Harrop héréditaires d'ordre supérieur

Pour définir *hh* il nous faut introduire deux nouvelles classes de propositions appelées *formules de but* et *clauses définies* (ou simplement *clauses*). Soit $\mathcal{A}$ une variable syntaxique pour les formules atomiques, $\mathcal{G}$ une variable syntaxique pour les formules de but, et $\mathcal{D}$ une variable syntaxique pour les clauses définies. Ces deux classes de formules sont définies par la récursion mutuelle suivante :

$$\begin{array}{lll} \mathcal{D} & ::= & \mathcal{A} \mid \mathcal{A} \subset \mathcal{G} \mid \forall x.\mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \\ \mathcal{G} & ::= & \mathcal{A} \mid \mathcal{G} \wedge \mathcal{G} \mid \mathcal{G} \vee \mathcal{G} \mid \mathcal{D} \supset \mathcal{G} \mid \forall x.\mathcal{G} \mid \exists x.\mathcal{G} \\ \mathcal{A} & ::= & formules\ atomiques \end{array}$$

Nous pouvons remarquer que la forme générale d'une clause définie est soit $\forall x_1 \cdots \forall x_n.\mathcal{A}$ soit $\forall x_1 \cdots \forall x_n.(\mathcal{A} \subset \mathcal{G})$ où $n \geq 0$. La formule atomique $\mathcal{A}$ est appelée la *tête* de la clause, et dans le dernier cas, $\mathcal{G}$ est appelée le *corps* de la clause. Il y a encore une restriction sur les clauses définies : la tête d'une clause définie doit avoir une constante à sa tête, alors que la tête d'une formule de but atomique peut être soit une variable soit une constante. Un *programme logique* (ou simplement *programme*) est un ensemble fini de clauses définies.

2.1.3 Les preuves dans $\mathcal{I}$

La notion intuitionniste de preuve dans $\mathcal{I}$ peut être donnée en termes de preuves d'un calcul des séquents. Un *séquent* est une paire $\mathcal{P} \to B$, où $\mathcal{P}$ est un ensemble fini (éventuellement vide) de formules et où B est une formule. L'ensemble $\mathcal{P}$ est l'*antécédent* de ce séquent et B est son *succédent*. L'expression $B, \mathcal{P}$ représente l'ensemble $\mathcal{P} \cup \{B\}$. Cette notation est utilisée même si $B \in \mathcal{P}$. Les règles d'inférence pour les séquents sont données à la Figure 2. Il faut noter qu'une règle d'inférence est en fait un schéma valide pour toute substitution uniforme d'hypothèses pour $\mathcal{P}$, de formules pour A, B et C, de termes pour t, etc ... .

$$\frac{B, C, \mathcal{P} \to A}{B \wedge C, \mathcal{P} \to A}\ (\wedge L) \qquad \frac{\mathcal{P} \to B \quad C, \mathcal{P} \to A}{B \supset C, \mathcal{P} \to A}\ (\supset L) \qquad \frac{[t/x]B, \mathcal{P} \to A}{\forall_\tau x.B, \mathcal{P} \to A}\ (\forall L)\star$$

$$\frac{\mathcal{P} \to B \quad \mathcal{P} \to C}{\mathcal{P} \to B \wedge C}\ (\wedge R) \qquad \frac{B, \mathcal{P} \to C}{\mathcal{P} \to B \supset C}\ (\supset R) \qquad \frac{\mathcal{P} \to [c/x]B}{\mathcal{P} \to \forall_\tau x.B}\ (\forall R)\star\star$$

$$\frac{\mathcal{P} \to B}{\mathcal{P} \to B \vee C}\ (\vee R_1) \qquad \frac{\mathcal{P} \to C}{\mathcal{P} \to B \vee C}\ (\vee R_2) \qquad \frac{\mathcal{P} \to [t/x]B}{\mathcal{P} \to \exists x.B}\ (\exists R)\star\star\star$$

$\star$ t est un terme de type τ
$\star\star$ c est une constante de type τ qui n'est pas libre dans le séquent du bas
$\star\star\star$ t est un terme arbitraire

Figure 2 : Règles d'inférence à droite et à gauche pour $\mathcal{I}$

Une preuve du séquent $\mathcal{P} \to B$ est un arbre fini construit en utilisant ces règles d'inférence tel que la racine soit $\mathcal{P} \to B$ et que les feuilles soient des axiomes, c'est-à-dire des séquents $\mathcal{P}' \to B'$ tels que $B' \in \mathcal{P}'$. Les nœuds non-terminaux d'un tel arbre sont des instances des règles d'inférence de la Figure 2. Puisque nous n'avons pas de règle d'inférence pour la $\beta\eta$-conversion, nous supposerons, lors de la construction d'une preuve, que deux formules seront égales si elles sont $\beta\eta$-convertibles. Dans le séquent $\mathcal{P} \to B$ nous appelons *hypothèses* les formules de $\mathcal{P}$ et *but* la formule B. Si ce séquent a une preuve, nous écrivons $\mathcal{P} \vdash_{\mathcal{I}} B$ et nous disons que B est démontrable à partir des hypothèses $\mathcal{P}$.

Définition 1 *Soit $\mathcal{P}$ un ensemble fini de formules de $\mathcal{I}$. L'expression $|\mathcal{P}|$ représente le plus petit ensemble de paires $\langle \mathcal{G}, D \rangle$ d'ensembles finis de formules $\mathcal{G}$ et de formules D, tel que :*

- *Si $D \in \mathcal{G}$ alors $\langle \emptyset, D \rangle \in |\mathcal{P}|$.*
- *Si $\langle \mathcal{G}, D_1 \wedge D_2 \rangle \in |\mathcal{P}|$ alors $\langle \mathcal{G}, D_1 \rangle \in |\mathcal{P}|$ et $\langle \mathcal{G}, D_2 \rangle \in |\mathcal{P}|$.*

- *Si* $\langle \mathcal{G}, D_1 \vee D_2 \rangle \in |\mathcal{P}|$ *alors* $\langle \mathcal{G}, D_1 \rangle \in |\mathcal{P}|$ *ou* $\langle \mathcal{G}, D_2 \rangle \in |\mathcal{P}|$.
- *Si* $\langle \mathcal{G}, \forall_\tau x.D \rangle \in |\mathcal{P}|$ *alors* $\langle \mathcal{G}, [t/x]D \rangle \in |\mathcal{P}|$ *pour tout terme* t *de type* τ .
- *Si* $\langle \mathcal{G}, \exists_\tau x.D \rangle \in |\mathcal{P}|$ *alors il existe un terme* t *de type* τ *tel que* $\langle \mathcal{G}, [t/x]D \rangle \in |\mathcal{P}|$.
- *Si* $\langle \mathcal{G}, G \supset D \rangle \in |\mathcal{P}|$ *alors* $\langle \mathcal{G} \cup \{G\}, D \rangle \in |\mathcal{P}|$.

Ce système d'inférence a la propriété de *preuve uniforme* comme définie par Miller [Miller 91] et décrite par le théorème suivant :

Théorème 1 *Soit* $\mathcal{P}$ *un ensemble fini de formules et soit* B *une formule. Le séquent* $\mathcal{P} \rightarrow B$ *a une preuve si et seulement si il a une preuve dans laquelle chaque séquent contenant une formule non-atomique comme succédent est la conclusion d'une règle d'introduction à droite.*

En se basant sur cette propriété on peut définir une procédure de recherche non-déterministe pour les preuves de la logique $\mathcal{I}$ [Miller 91]. Cette procédure est décrite par les quatre opérations suivantes dans lesquelles G est le but que nous tentons de démontrer à partir de l'ensemble d'hypothèses $\mathcal{P}$:

- **AND :** Si G est $G_1 \wedge G_2$ alors essayer de démontrer que G_1 et G_2 se déduisent tous deux de $\mathcal{P}$.
- **OR :** Si G est $G_1 \vee G_2$ alors essayer de démontrer que G_1 ou G_2 se déduit de $\mathcal{P}$.
- **GENERIC :** Si G est $\forall_\tau x.G'$ alors choisir une nouvelle constante c de type τ et essayer de démontrer que $[c/x]G'$ se déduit de $\mathcal{P}$.
- **INSTANCE :** Si G est $\exists_\tau x.G'$ alors choisir un terme t de type τ et essayer de démontrer que $[c/x]G'$ se déduit de $\mathcal{P}$.
- **AUGMENT :** Si G est $D \supset G'$ alors essayer de démontrer que G' se déduit de $\mathcal{P} \cup \{D\}$.
- **BACKCHAIN :** Si G est atomique et s'il existe une paire $\langle \mathcal{G}, G \rangle \in |\mathcal{P}|$ alors essayer de démontrer que chacune des formules de $\mathcal{G}$ se déduit de $\mathcal{P}$. La preuve se termine si $\mathcal{G}$ est vide.

Nous pouvons remarquer que l'opération AUGMENT ajoute une clause au programme, alors que l'opération GENERIC ajoute un symbole à la signature. Remarquons également que nous n'avons pas inclus d'opération spécifique pour la conversion puisque nous considérons que les termes sont équivalents à la $\beta\eta$-conversion près. Cette procédure de recherche définit une structure assez rigide pour les méta-preuves démontrant qu'une formule de but donnée se déduit d'un programme donné. De telles preuves sont appelées des *preuves uniformes* [Miller 91].

2.2 Le langage λProlog

La mise en œuvre d'un interpréteur déterministe oblige à faire des choix qui ne sont pas spécifiés dans la description de l'interpréteur non-déterministe. Nous allons vous présenter les choix effectués pour l'implantation du langage λProlog (version MALI [Brisset 93]).

2.2.1 Mise en œuvre

L'ordre dans lequel sont exécutées les opérations AND et OR ainsi que l'ordre dans lequel sont testées les clauses définies pour l'opération BACKCHAIN sont déterminés de la même manière qu'en Prolog classique : les conjonctions et les disjonctions sont exécutées dans leur ordre d'apparition. En ce qui concerne l'opération BACKCHAIN, les clauses définies sont testées dans l'ordre où elles apparaissent dans le programme $\mathcal{P}$ en utilisant une stratégie de recherche en profondeur d'abord pour traiter les cas d'échec. Grâce à l'opération AUGMENT il est possible d'ajouter dynamiquement de nouvelles clauses. Cet ajout est effectué au début du programme.

L'opération INSTANCE est fortement indéterministe. Généralement, lorsque nous essayons de résoudre un but existentiel, nous ne possédons que très peu d'information sur la manière dont doit être choisi le λ-terme de la substitution. Plutôt que de "choisir" ce λ-terme, l'opération INSTANCE introduit une nouvelle variable logique comme terme de substitution. Cette variable sera instanciée utltérieurement par unification. De façon similaire, l'opération BACKCHAIN utilisera également des variables logiques : Nous choisissons une paire $\langle \mathcal{G}, A \rangle$ dans $|\mathcal{P}|$ telle que A soit atomique Nous remplaçons toutes les variables libres de A par des variables logiques puis nous essayons d'unifier A avec le but atomique courant G. Si l'unification réussit, nous appliquons la substitution résultante aux formules de $\mathcal{G}$ et essayons de prouver chacune d'entre elles.

Puisque la formule A de la paire choisie est atomique, elle est de la forme $t_1 \cdots t_n$. Dans hh, p doit être une constante, donc toute substitution de variables donnera comme résultat une formule atomique. Par contre, dans $\mathcal{I}$ si p est une variable et si A ne s'unifie pas avec le but, nous devons quand même tenir compte des substitutions qui transforment A en une formule non-atomique G, puis considérer l'ensemble $|G|$. Prenons par exemple le cas d'une substitution pour p transforme A en une implication $B \supset A'$ où A' est atomique. Nous devons alors regarder les paires de $|\{B \supset A'\}|$ dont le second élément est atomique. Dans notre cas, il n'y a que $\langle \{B\}, A' \rangle$. Nous essayons alors d'unifier A avec A'. Si cela réussit, B devient un sous-but supplémentaire à prouver avec les hypothèses de $\mathcal{G}$. Si l'unification échoue, nous devons continuer le processus.

Donc, dans $\mathcal{I}$, nous avons un indéterminisme supplémentaire dû au fait que

l'on doive "deviner" au moins la partie de la substitution qui détermine la structure logique de la formule sur laquelle nous utiliserons l'opération BACKCHAIN, et ce avant que l'unification ne soit utilisée. Au chapitre 3 nous proposerons une modification qui élimine cet indéterminisme mais qui donne une procédure de recherche incomplète.

2.2.2 La syntaxe

Les noms de variables commencent par une lettre majuscule, alors que ceux des constantes commencent par une minuscule. La λ-abstraction est repésentée par le symbole "\", ie $\lambda X \equiv X\backslash$. Les termes sont en fait considérés comme les représentants de classes d'équivalence de termes où la relation d'équivalence est la $\beta\eta$-conversion. Par exemple, les termes $X\backslash(f\ X)$, $Y\backslash(f\ Y)$, $(F\backslash Y\backslash(F\ Y))\ f$ et f représentent la même classe de termes.

Les symboles "," et ";" représentent les connecteurs $\wedge$ et $\vee$ respectivement et "," est prioritaire sur ";" . Le symbole ": $-$" signifie *impliqué par* alors que "=>" représente le connecteur $\supset$. Le premier symbole est le connecteur de plus haut niveau des clauses définies, comme en Prolog classique. Les implications dans les buts et dans les corps de clauses utilisent le symbole "=>".

Les variables libres d'une clause définie sont supposées être quantifiées universellement, alors que celles des buts sont supposées être quantifiées existentiellement. Les quantificateurs universel et existentiel, aussi bien dans les clauses définies que dans les buts, sont représentés par les constantes *pi* et *sigma* de type $(S \to o) \to o$ (où S est une variable de type), suivies d'une λ-abstraction. Par exemple, $\forall x.A \equiv pi\ X\backslash\ A$. De ce fait, les variables liées par $\forall$ ou $\exists$ sont représentées liées en λProlog.

La syntaxe complète du langage est donnée Figure 3.

Syntaxe logique

$$\begin{array}{l} \mathcal{D} ::= \mathcal{A} \mid \mathcal{A} \subset \mathcal{G} \mid \forall x.\mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \\ \mathcal{G} ::= \mathcal{A} \mid \mathcal{G} \wedge \mathcal{G} \mid \mathcal{G} \vee \mathcal{G} \mid \mathcal{D} \supset \mathcal{G} \mid \forall x.\mathcal{G} \mid \exists x.\mathcal{G} \\ \mathcal{A} ::= formules\ atomiques \end{array}$$

Syntaxe concrète (les quantificateurs universels dans $\mathcal{D}$ sont implicites)

$$\begin{array}{l} \mathcal{D} ::= \mathcal{A}. \mid \mathcal{A} :- \mathcal{G}. \mid \mathcal{D}\ \mathcal{D} \\ \mathcal{G} ::= \mathcal{A} \mid \mathcal{G}, \mathcal{G} \mid \mathcal{G}; \mathcal{G} \mid \mathcal{D} => \mathcal{G} \mid pi\ X\backslash\ \mathcal{G} \mid sigma\ X\backslash\ \mathcal{G} \end{array}$$

Figure 3 : Syntaxe des formules de Harrop héréditaires en λProlog

Les types primitifs sont introduits en utilisant une déclaration *kind* et les éléments de la signature sont introduits en utilisant une déclaration *type*. Par exemple, le type a et la l'élément de la signature $f : a \to a \to a$ seront déclarés de la façon suivante :

kind	a	*type*
type	f	$a \to a \to a$

Si les déclarations *kind* et *type* sont oubliées, elles seront alors inférées par l'interpréteur. Les déclarations de types peuvent contenir des variables de types (en lettres majuscules) puisque λProlog autorise le polymorphisme.

2.3 Programmation en λProlog

Voici deux exemples de programmes écrits en λProlog. Le premier définit les prédicats de manipulation de listes `append`, `reverse` et `member`. Le second, typique de la programmation en λProlog, est l'implantation d'un système de typage.

2.3.1 Manipulation de listes

```
append   [ ] Y Y.
append   [E|X] Y [E|Z]  :-  append X Y Z.

reverse  [ ] [ ].
reverse  [A|X] Y        :-  reverse X RX,
                            append RX [A] Y.
member   E [E| _ ].
member   E [ _ |L]      :-  member E L.
```

Comme nous pouvons le remarquer, ce programme est écrit de la même manière qu'en Prolog standard. Mais si nous désirons utiliser plusieurs sortes de liste (listes d'entiers, listes de listes, ...), il nous faut réécrire ces trois prédicats pour chaque type de liste.

Une autre solution consiste à définir des prédicats polymorphiques, comme nous l'autorise λProlog. Les déclarations associées à ces trois prédicats seraient alors les suivantes :

```
type  append   (list A) -> (list A) -> (list A) -> o.
type  reverse  (list A) -> (list A) -> o.
type  member   A -> (list A) -> o.
```

Dans ces déclarations `A` représente une variable de type. Nous avons donc défini des schémas de prédicats dont des instances seront générées pour chaque substitution de types au variables de type. Le constructeur de type `list` et les constructeurs de termes `[ ]` et `'.'` sont habituellement prédéfinis. Nous donnons ici leurs déclarations à titre indicatif :

```
kind  list  type -> type.

type  [ ]   (list A).
type  '.'   A -> (list A) -> (list A).
```

2.3.2 Système de typage

Le système de règles de déduction pour la théorie des types simples est le suivant :

$$\frac{\Gamma \vdash t_1 : \alpha \to \beta \quad \Gamma \vdash t_2 : \alpha}{\Gamma \vdash (t_1\ t_2) : \beta}\ (\to E)$$

$$\frac{\Gamma, x : \alpha \vdash E : \beta}{\Gamma \vdash \lambda x.E : \alpha \to \beta}\ (\to I)$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (axiom)$$

Le programme λProlog mettant en œuvre cette logique est le suivant :

```
kind  lambda_term  type.
kind  simple_type  type.

type  application  lambda_term -> lambda_term -> lambda_term.
type  abstraction  (lambda_term -> lambda_term) -> lambda_term.
type  arrow        simple_type -> simple_type -> simple_type.
type  has_type     lambda_term -> simple_type -> o.

has_type (application T1 T2) Beta :-
   has_type T1 (arrow Alpha Beta),
   has_type T2 Alpha.

has_type (abstraction E) (arrow Alpha Beta) :-
   pi x\(has_type x Aplha => has_type (E x) Beta).
```

Un exemple de requête serait :

```
?- has_type (abstraction x\x) (arrow i i).
```

La seconde clause du prédicat `has_type` utilise presque toutes les caractéristiques de λProlog : la quantification universelle explicite dans les buts (`pi x\` signifie «*pour tout x*»), l'implication intuitionniste dans les buts (`=>` signifie «*implique*»), et les λ-termes (`E` est une λ-abstraction et `(E x)` représente l'application de `E` à `x`). Dans la requête, `x\x` correspond à «$\lambda x.x$».

2.4 Conclusion

Nous disposons donc maintenant d'une logique intuitionniste d'ordre supérieur ($\mathcal{I}$) pour laquelle est définie une procédure de recherche (complète mais non-déterministe) de preuves uniformes. Nous pouvons ainsi coder les jugements d'AF_2 dans $\mathcal{I}$ afin d'y appliquer cette procédure de recherche.

Pour mettre en œuvre ce codage nous disposons d'un langage de programmation logique (λProlog) basé sur *hh* qui est une restriction de $\mathcal{I}$. Nous avons vu que les formules de Harrop héréditaires (λProlog) étendent les clauses de Horn (Prolog classique) essentiellement de deux façons :

- La première extension autorise des expressions logiques plus riches dans les buts et les corps des clauses. L'implication et la quantification universelle ainsi que la disjonction et la quantification existentielle explicite sont rajoutées dans les buts et dans le corps des clauses, en plus de la conjonction d'atomes des clauses de Horn.

- La seconde extension des clauses de Horn rend le langage d'ordre supérieur en ce sens qu'il est possible de quantifier sur des symboles de prédicat et de fonction. De façon à instancier les variables de prédicat et de fonction avec des termes, les termes du premier ordre sont remplacés par des λ-termes simplement typés. L'application des λ-termes se fait au travers de la β-réduction alors que l'unification des λ-termes est réalisée par l'unification d'ordre supérieur.

Le chapitre suivant sera consacré au codage d'AF_2 dans la logique $\mathcal{I}$. Le codage proposé est basé sur celui du Calcul des Constructions détaillé dans [Felty 93a]. Dans ce même chapitre, nous étudierons ensuite le problème de la recherche de preuves sur les jugements d'AF_2 ainsi codés. L'analyse des problèmes posés pourrait alors devenir la première étape du développement d'une procédure de recherche pour AF_2.

Chapitre 3

Recherche de preuves en AF_2 par codage dans la logique $\mathcal{I}$

L'objectif de ce chapitre est de développer une nouvelle approche pour la recherche de preuves en se basant sur les travaux effectués pour CC dans [Felty 93a]. Celle-ci consiste à coder AF_2 dans une autre logique ($\mathcal{I}$) pour laquelle nous disposons d'une procédure de recherche. De cette façon, synthétiser des programmes en AF_2 revient à rechercher des preuves dans $\mathcal{I}$. Le *codage* que nous allons vous présenter se doit d'être correct, ie s'il existe une preuve de A en AF_2 alors il existe une preuve de $\langle\langle A \rangle\rangle$ dans $\mathcal{I}$, où $\langle\langle A \rangle\rangle$ est le codage de A. Nous discuterons ensuite de la complétude du codage (ie toute preuve de $\mathcal{I}$ est une preuve en AF_2), bien que celle-ci ne soit pas indispensable pour notre approche. Afin d'illustrer notre travail nous donnerons le codage concernant l'exemple du chapitre 1.

Dans la deuxième partie de ce chapitre, nous traiterons de la mise en œuvre de cette *recherche de preuves*. Nous verrons alors réapparaître le fait que hh est une restriction de $\mathcal{I}$: nous avons proposé un codage d'AF_2 dans $\mathcal{I}$, mais nous devons en fait travailler avec hh. Cela nous amènera à proposer quelques modifications de la procédure de recherche de preuves uniformes.

Bien qu'il ne soit pas possible, de cette manière, d'automatiser complètement la recherche de preuves, notre objectif aura été atteint. En effet, l'analyse des problèmes que nous aurons ainsi soulevé nous permettra d'envisager le développement d'une procédure de recherche, basée sur la notion de preuve uniforme, directement pour AF_2.

3.1 Codage d'AF_2 dans la logique $\mathcal{I}$

Nous allons procéder en deux parties. En premier lieu nous donnerons le codage des termes et des jugements d'AF_2. Puisque celui-ci utilise une représentation explicite de l'abstraction et de l'application (constantes abs_o, app_o, ...), nous devrons également traduire de manière explicite les relations de convertibilité.

Par exemple, la règle de conversion (β) au niveau des objets sera représentée par la formule suivante : $conv_o\ (app_o\ (abs_o\ B)\ A)\ (B\ A)$. Ceci sera l'objet de la deuxième partie.

3.1.1 Termes et jugements

Etant donnée une assertion $\Gamma \vdash \alpha$ le codage présenté ci-après fera correspondre à Γ un ensemble d'hypothèses et à α un but qui devra être prouvé à partir de ces hypothèses. Un tel codage opère sur des jugements. Il est également nécessaire de définir une traduction des objets et types d'AF_2 en λ-termes simplement typés. Ces deux codages sont interdépendants. En effet, coder un type de la forme $A \to B$ nécessite l'introduction d'une nouvelle variable, appelée f, de coder, en particulier, le jugement $fx\ :\ B$, puis de réaliser une abstraction sur f. Ceci nous donne : $\langle\langle A \to B \rangle\rangle = (\lambda f.(\forall x.([\![x : A]\!] \supset [\![fx : B]\!])))$. De la même façon, le codage d'un type de la forme $\forall x.A$ nécessite le codage du jugement $f\ :\ A$ où f est une nouvelle variable.

Nous supposons que les variables d'AF_2 sont divisées en six ensembles dénombrables : ν_o^1 et ν_o^2 pour les objets ; ν_t^1 et ν_t^2 pour les types ; ν_i^1 et ν_i^2 pour les individus. Par hypothèse, les variables libres de terme (respectivement de type, d'individu) devant être codées seront dans ν_o^1 (resp. ν_t^1, ν_i^1). Lorsque la traduction nécessitera de nouvelles variables de terme (resp. de type, d'individu), celles-ci seront prises dans ν_o^2 (resp. ν_t^2, ν_i^2).

Dans ce chapitre, puisque nous codons AF_2 dans $\mathcal{I}$, nous considèrerons $\mathcal{I}$ comme étant le méta-langage. Nous introduisons le méta-type *ob* qui sera le type des objets d'AF_2 une fois codés. Nous définissons une application bijective ρ_o des variables de $\nu_o^1 \cup \nu_o^2$ vers les méta-variables de type *ob*. Nous définissons de la même manière une bijection ρ_t des variables de $\nu_t^1 \cup \nu_t^2$ vers les méta-variables de type $ob \to o$, ainsi qu'une bijection ρ_i des variables de $\nu_i^1 \cup \nu_i^2$ vers les méta-variables de type i. Les types d'AF_2 seront traduits par des *prédicats sur des objets*. L'union de ces trois fonctions sera notée ρ, mais pour des raisons de lisibilité, ces traductions seront souvent implicites.

$$
\begin{array}{rl}
app_o = & ob \to ob \to ob \\
abs_o = & (ob \to ob) \to ob \\
\\
app_i = & i \to i \to i \\
abs_i = & (i \to i) \to i \\
\\
app_t = & (ob \to o) \to i \to (ob \to o)
\end{array}
$$

Figure 4 : Constantes pour le codage des termes d'AF_2

Nous introduisons deux constantes app_o et abs_o définissant l'application et l'abstraction pour les objets. Nous définissons de la même façon les constantes app_i et abs_i pour les individus. En ce qui concerne les types, nous n'introduisons que la constante app_t représentant l'application d'un individu à un type (ie l'application d'un paramètre à un prédicat). Ces constantes, ainsi que leurs types, sont données Figure 4.

Pour les objets	$\langle\langle t \rangle\rangle = \rho(t) = t$
	$\langle\langle tu \rangle\rangle = (app_o\ \langle\langle t \rangle\rangle\ \langle\langle u \rangle\rangle)$
	$\langle\langle \lambda x.t \rangle\rangle = (abs_o\ \lambda x.\langle\langle t \rangle\rangle)$
Pour les individus	$\langle\langle i \rangle\rangle = \rho(i) = i$
Pour les types	$\langle\langle A \rangle\rangle = \rho(A) = A$
	$\langle\langle P(i_1, \cdots, i_n) \rangle\rangle = (app_t\ \langle\langle P(i_1, \cdots, i_{n-1}) \rangle\rangle\ \langle\langle i_n \rangle\rangle)$
	$\langle\langle A \to B \rangle\rangle = (\lambda f.(\forall x.([\![x : A]\!] \supset [\![fx : B]\!])))$
	$\langle\langle \forall x.A \rangle\rangle = (\lambda f.(\forall x.[\![f : A]\!]))$
	$\langle\langle \forall X.A \rangle\rangle = (\lambda f.(\forall X.[\![f : A]\!]))$
Pour les jugements	$[\![t : A]\!] = (\langle\langle A \rangle\rangle\ \langle\langle t \rangle\rangle)$

Figure 5 : Codage des termes et des jugements d'AF_2

Nous pouvons maintenant définir le codage des termes et des jugements d'AF_2. Celui-ci est présenté Figure 5. Nous noterons $\langle\langle P \rangle\rangle$ le codage du terme P et $[\![t : A]\!]$ le codage du jugement $t\ :\ A$. Le codage des termes possède la propriété suivante.

Lemme 1 *Etant donnés P et Q des termes d'AF_2, x une variable, alors*

$$[\langle\langle Q \rangle\rangle / x]\ \langle\langle P \rangle\rangle\ =\ \langle\langle [Q/x]P \rangle\rangle$$

Preuve

- Cas où P et Q sont des objets :
  - $P = x$, alors
    $$\begin{aligned}[\langle\langle Q \rangle\rangle / x]\langle\langle P \rangle\rangle &= [\langle\langle Q \rangle\rangle / x]\ x \\ &= \langle\langle Q \rangle\rangle \\ &= \langle\langle [Q/x]\ x \rangle\rangle \\ &= \langle\langle [Q/x]\ P \rangle\rangle\end{aligned}$$

- $P = y \neq x$, alors
$$\begin{aligned}[\langle\langle Q\rangle\rangle/x]\langle\langle P\rangle\rangle &= [\langle\langle Q\rangle\rangle/x]\ y \\ &= y \\ &= \langle\langle [Q/x]\ y\rangle\rangle \\ &= \langle\langle [Q/x]\ P\rangle\rangle\end{aligned}$$

- $P = tu$, alors
$$\begin{aligned}[\langle\langle Q\rangle\rangle/x]\langle\langle P\rangle\rangle &= [\langle\langle Q\rangle\rangle/x]\ \langle\langle tu\rangle\rangle \\ &= [\langle\langle Q\rangle\rangle/x]\ (app_o\ \langle\langle t\rangle\rangle\ \langle\langle u\rangle\rangle) \\ &\Downarrow \text{ propriété de la substitution} \\ &= (app_o\ [\langle\langle Q\rangle\rangle/x]\ \langle\langle t\rangle\rangle \quad [\langle\langle Q\rangle\rangle/x]\ \langle\langle u\rangle\rangle) \\ &\Downarrow \text{ par hypothèse d'induction} \\ &= (app_o\ \langle\langle [Q/x]\ t\rangle\rangle \quad \langle\langle [Q/x]\ u\rangle\rangle) \\ &= \langle\langle [Q/x]\ t \quad [Q/x]\ u\rangle\rangle \\ &\Downarrow \text{ propriété de la substitution} \\ &= \langle\langle [Q/x]\ (tu)\rangle\rangle \\ &= \langle\langle [Q/x]\ P\rangle\rangle\end{aligned}$$

- $P = \lambda x.t$, alors
$$\begin{aligned}[\langle\langle Q\rangle\rangle/x]\langle\langle P\rangle\rangle &= [\langle\langle Q\rangle\rangle/x]\ \langle\langle \lambda x.t\rangle\rangle \\ &= [\langle\langle Q\rangle\rangle/x]\ (abs_o\ \lambda x.\langle\langle t\rangle\rangle) \\ &= (abs_o\ \lambda x.\langle\langle t\rangle\rangle) \\ &= \langle\langle \lambda x.t\rangle\rangle \\ &= \langle\langle [Q/x]\ (\lambda x.t)\rangle\rangle \\ &= \langle\langle [Q/x]\ P\rangle\rangle\end{aligned}$$

- $P = \lambda y.t$, alors
$$\begin{aligned}[\langle\langle Q\rangle\rangle/x]\langle\langle P\rangle\rangle &= [\langle\langle Q\rangle\rangle/x]\ \langle\langle \lambda y.t\rangle\rangle \\ &= [\langle\langle Q\rangle\rangle/x]\ (abs_o\ \lambda y.\langle\langle t\rangle\rangle) \\ &\Downarrow \text{ propriété de la substitution} \\ &= (abs_o\ \lambda y.([\langle\langle Q\rangle\rangle/x]\ \langle\langle t\rangle\rangle)) \\ &\Downarrow \text{ par hypothèse d'induction} \\ &= (abs_o\ \lambda y.\langle\langle [Q/x]\ t\rangle\rangle) \\ &= \langle\langle \lambda y.([Q/x]\ t)\rangle\rangle \\ &\Downarrow \text{ propriété de la substitution} \\ &= \langle\langle [Q/x]\ (\lambda y.t)\rangle\rangle \\ &= \langle\langle [Q/x]\ P\rangle\rangle\end{aligned}$$

- Le cas où P et Q sont des individus est trivial.

– Cas où P est un type :

  – $P = X$, alors
  $$\begin{aligned}[\langle\!\langle Q\rangle\!\rangle/X]\langle\!\langle P\rangle\!\rangle &= [\langle\!\langle Q\rangle\!\rangle/X]\ X \\ &= \langle\!\langle Q\rangle\!\rangle \\ &= \langle\!\langle [Q/X]\ X\rangle\!\rangle \\ &= \langle\!\langle [Q/X]\ P\rangle\!\rangle\end{aligned}$$

  – $P = A(x)$, alors
  $$\begin{aligned}[\langle\!\langle Q\rangle\!\rangle/x]\langle\!\langle P\rangle\!\rangle &= [\langle\!\langle Q\rangle\!\rangle/x]\ \langle\!\langle A(x)\rangle\!\rangle \\ &= [\langle\!\langle Q\rangle\!\rangle/x]\ (app_t\ A\ x) \\ &\Downarrow \text{ propriété de la substitution} \\ &= (app_t\ A\ \langle\!\langle Q\rangle\!\rangle) \\ &= \langle\!\langle A(Q)\rangle\!\rangle \\ &= \langle\!\langle [Q/x]\ A(x)\rangle\!\rangle \\ &= \langle\!\langle [Q/x]\ P\rangle\!\rangle\end{aligned}$$

  – $P = A \to B$, alors
  $$\begin{aligned}[\langle\!\langle Q\rangle\!\rangle/x]\langle\!\langle P\rangle\!\rangle &= [\langle\!\langle Q\rangle\!\rangle/x]\ \langle\!\langle A \to B\rangle\!\rangle \\ &= [\langle\!\langle Q\rangle\!\rangle/x]\ (\lambda f.(\forall y.([\![y : A]\!] \supset [\![fy : B]\!]))) \\ &\Downarrow \text{ propriété de la substitution} \\ &= (\lambda f.(\forall y.(([\langle\!\langle Q\rangle\!\rangle/x]\ [\![y : A]\!]) \supset ([\langle\!\langle Q\rangle\!\rangle/x]\ [\![fy : B]\!])))) \\ &= (\lambda f.(\forall y.([\![y : [Q/x]\ A]\!] \supset [\![fy : [Q/x]\ B]\!]))) \\ &= \langle\!\langle ([Q/x]\ A) \to ([Q/x]\ B)\rangle\!\rangle \\ &\Downarrow \text{ propriété de la substitution} \\ &= \langle\!\langle [Q/x]\ (A \to B)\rangle\!\rangle \\ &= \langle\!\langle [Q/x]\ P\rangle\!\rangle\end{aligned}$$

  – $P = \forall y.A$, alors
  $$\begin{aligned}[\langle\!\langle Q\rangle\!\rangle/x]\langle\!\langle P\rangle\!\rangle &= [\langle\!\langle Q\rangle\!\rangle/x]\ \langle\!\langle \forall y.A\rangle\!\rangle \\ &= [\langle\!\langle Q\rangle\!\rangle/x]\ (\lambda f.(\forall y.[\![f : A]\!])) \\ &\Downarrow \text{ propriété de la substitution} \\ &= (\lambda f.(\forall y.([\langle\!\langle Q\rangle\!\rangle/x]\ [\![f : A]\!]))) \\ &= (\lambda f.(\forall y.[\![f : [Q/x]\ A]\!])) \\ &= \langle\!\langle \forall y.([Q/x]\ A)\rangle\!\rangle \\ &\Downarrow \text{ propriété de la substitution} \\ &= \langle\!\langle [Q/x]\ (\forall y.A)\rangle\!\rangle \\ &= \langle\!\langle [Q/x]\ P\rangle\!\rangle\end{aligned}$$

  – Dans les autres cas, x n'a pas d'occurence libre dans P. Du fait qu'il n'y ait pas de substitution, ces cas sont triviaux.

□

3.1.2 Convertibilité

Les relations de convertibilité pour les termes codés d'AF_2 peuvent être représentées par un ensemble de formules de $\mathcal{I}$. Nous introduisons les trois prédicats binaires $conv_o$, $conv_t$ et $conv_i$. Ceux-ci expriment la convertibilité au niveau des objets, au niveau des types et au niveau des individus.

$$\begin{aligned} conv_o &= ob \to ob \to o \\ conv_t &= (ob \to o) \to (ob \to o) \to o \\ conv_i &= i \to i \to o \end{aligned}$$

Les formules de $\mathcal{I}$ exprimant les relations de convertibilité données Figure 6. Sur cette figure ainsi que sur les suivantes nous avons supprimé les quantificateurs les plus externes. Nous supposons ainsi que les formules sont quantifiées universellement pour toutes leurs variables libres (écrites en majuscules). Pour les objets et les individus, la première formule code la règle (β). Pour les objets, les individus et les types nous avons ensuite, dans l'ordre, les formules repésentant la règle $(CONG)$, la réflexivité, la symétrie puis la transitivité. Le Lemme 2 exprime la correction de cette spécification.

Pour les objets	$conv_o\ (app_o\ (abs_o\ B)\ A)\ (B\ A)$
	$conv_o\ A\ C \wedge conv_o\ B\ D \supset conv_o\ (app_o\ A\ B)\ (app_o\ C\ D)$
	$conv_o\ A\ A$
	$conv_o\ B\ A \supset conv_o\ A\ B$
	$conv_o\ A\ C \wedge conv_o\ C\ B \supset conv_o\ A\ B$
Pour les individus	$conv_i\ (app_i\ (abs_i\ B)\ A)\ (B\ A)$
	$conv_i\ A\ C \wedge conv_i\ B\ D \supset conv_i\ (app_i\ A\ B)\ (app_i\ C\ D)$
	$conv_i\ A\ A$
	$conv_i\ B\ A \supset conv_i\ A\ B$
	$conv_i\ A\ C \wedge conv_i\ C\ B \supset conv_i\ A\ B$
Pour les types	$conv_t\ A\ B \wedge conv_i\ M\ N \supset conv_t\ (app_t\ A\ M)\ (app_t\ B\ N)$
	$conv_t\ A\ A$
	$conv_t\ B\ A \supset conv_t\ A\ B$
	$conv_t\ A\ C \wedge conv_t\ C\ B \supset conv_t\ A\ B$

Figure 6 : Relations de convertibilité pour AF_2

Lemme 2 *Soient A et B des types d'AF_2 et soit $\mathcal{P}$ l'ensemble des formules de la Figure 6. Alors $A =_\beta B$ si et seulement si $\mathcal{P} \to (conv_t\ \langle\!\langle A \rangle\!\rangle\ \langle\!\langle B \rangle\!\rangle)$ a une preuve dans $\mathcal{I}$.*

Preuve

La preuve de ce lemme se fait par induction sur la longueur d'une dérivation de $A =_\beta B$, ie on regarde cas par cas quelle a été la dernière règle appliquée. Du fait que chacune de ces règles est représentée par une formule de la Figure 6, cela ne pose pas de problème particulier.

L'induction est similaire pour les deux autres relations de convertibilité.

□

Nous ajoutons également à $\mathcal{P}$ les formules de la Figure 7. Celles-ci nous procurent la β-conversion pour les termes codés d'AF_2 pouvant apparaître soit dans les buts soit dans les hypothèses.

$$conv_t\ A\ B \wedge conv_o\ M\ N \wedge jud\ B\ N \supset jud\ A\ M$$

$$conv_t\ A\ B \wedge conv_o\ M\ N \wedge jud\ B\ N \wedge (jud\ A\ M \supset G) \supset G$$

Figure 7 : Formules pour la convertibilité d'AF_2

3.2 Un exemple

Afin d'illustrer ce codage, nous allons développer la traduction pour l'exemple de l'addition de deux entiers.

Nous rappelons que le programme sera extrait d'une dérivation de la formule $\forall x.(Nx \supset \forall y.(Ny \supset Nplus(x,y)))$, sachant que nous disposons du système d'équations suivant :

$$\begin{cases} plus(\underline{0}, x) & = & x \\ plus(x, \underline{s}(y)) & = & \underline{s}(plus(x,y)) \end{cases}$$

Pour la représentation du type entier, nous utiliserons la définition itérative, ie $Nx \equiv \forall X.\Big(\big(\forall y.(Xy \supset X(\underline{s}\ y))\big) \supset (X\underline{0} \supset Xx)\Big)$

3.2.1 Mise en œuvre du type de données

Nous introduisons deux constantes : `zero` pour $\underline{0}$ et `succ` pour $\underline{s}$. Nous avons ainsi :

$$\begin{cases} \langle\!\langle \underline{0} \rangle\!\rangle & = & \texttt{zero} \\ \langle\!\langle \underline{s} \rangle\!\rangle & = & \texttt{succ} \end{cases}$$

Le codage du type de données est alors défini par une relation de convertibilité entre $\langle\langle Nx \rangle\rangle$ et $\langle\langle \forall X.((\forall y.(Xy \supset X(\underline{s}\, y))) \supset (X\underline{0} \supset Xx))\rangle\rangle$. Ces deux termes sont considérés comme des types d'AF_2. Grâce à la symétrie de la convertibilité sur les types nous avons ainsi automatisé les opérations de *pliage* et de *dépliage* de la structure. Le résultat de ce codage est le suivant :

```
convt (appt nat N) (jud (f\ (pi x\ (jud (g\ (pi u\ ( (jud (h\ (pi
y\ (jud (i\ (pi v\ ( jud (appt x y) v => jud (appt x (appi succ y))
(appo i v)))) h ))) u) => (jud (j\ (pi w\ (jud (appt x zero) w => jud
(appt x N) (appo j w)))) (appo g u))))) f )))).
```

Remarque Si `A`, `B`, `M` et `N` sont des variables de λProlog, nous ne pouvons pas écrire le prédicat suivant :

```
A M :- convt A B , convo M N , B N.
```

C'est pour cette raison que nous avons du introduire un nouveau prédicat (`jud`) qui sera du type $(ob \to o) \to ob \to o$. La formule `(jud B t)` signifiera alors «*B est le type de t*». Nous pouvons ainsi réécrire le prédicat ci-dessus :

```
jud A M :- convt A B , convo M N , jud B N.
```

3.2.2 Codage de la spécification

De la même façon que pour le type entier, les équations seront représentées par des relations de convertibilité au niveau des individus. Ceci nous donne :

```
convi (appi (appi plus nzero) X) X.
convi (plus X (appi nsucc Y)) (appi nsucc (appi (appi plus X) Y)).
```

Finalement, la formule dont nous désirons trouver une dérivation sera codée de la manière suivante où `T` représente le λ-terme qui sera construit durant la preuve :

```
(jud (f\ (pi x\ (jud (g\ (pi u\ (jud (appt nat x) u => (jud (h\ (pi
y\ (jud (i\ (pi v\ (jud (appt nat y) v => jud (appt nat (appi (appi
plus x) y)) (appo i v)))) h))) (appo g u))))) f)))) T
```

3.3 Correction du codage

Dans cette partie nous allons démontrer que pour toute dérivation d'AF_2 il existe une preuve dans $\mathcal{I}$.

Définition 2 *Nous dirons que* Γ *est un pré-contexte si pour toute paire* $P : Q$ *de* Γ, $[\![P : Q]\!]$ *est bien formé. Nous noterons* $[\![\Gamma]\!]$ *l'ensemble contenant* $[\![P : Q]\!]$ *pour toute paire* $P : Q$ *de* Γ.

Lemme 3 *Soient* P, Q, P', Q' *et* R *des termes d'*AF_2 *tels que* $P =_{\beta\eta} P'$ *et* $Q =_{\beta\eta} Q'$. *Nous supposons que* $[\![P : Q]\!]$ *est bien formé. Soient* x *une variable et* Γ *un pré-contexte. Alors :*

(1) $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$ **ssi** $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P' \rangle\!\rangle)$

(2) $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![([R/x]\ P) : ([R/x]\ Q)]\!]$ **ssi** $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\langle\!\langle R \rangle\!\rangle / x]\ [\![P : Q]\!]$

(3) $\mathcal{P}, [\![\Gamma]\!], [\![x : Q]\!] \vdash_{\mathcal{I}} A$ **ssi** $\mathcal{P}, [\![\Gamma]\!], [\![x : Q']\!] \vdash_{\mathcal{I}} A$
pour toute formule A

(4) $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$ **ssi** $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q']\!]$

Preuve

- Pour prouver **(1)** nous avons par hypothèse que le jugement $[\![P : Q]\!]$ est bien formé. D'après le codage d'AF_2 proposé ci-dessus nous obtenons que $[\![P : Q]\!] = \langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P \rangle\!\rangle$. Nous avons donc que $[\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$ si et seulement si $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P \rangle\!\rangle)$.

  Or, par hypothèse, nous savons de plus que $P =_{\beta\eta} P'$ et que $Q =_{\beta\eta} Q$. En utilisant le Lemme 2 nous obtenons que $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (conv_o \ \langle\!\langle P \rangle\!\rangle \ \langle\!\langle P' \rangle\!\rangle)$ et que $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (conv_t \ \langle\!\langle Q \rangle\!\rangle \ \langle\!\langle Q \rangle\!\rangle)$. Etant donnée la symétrie du prédicat $conv_o$ nous obtenons également que $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (conv_o \ \langle\!\langle P' \rangle\!\rangle \ \langle\!\langle P \rangle\!\rangle)$.

  - ($\Rightarrow$) L'hypothèse est $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$, ie $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P \rangle\!\rangle)$.

    On utilise l'opération BACKCHAIN sur la formule suivante dont le schéma se trouve dans $\mathcal{P}$:

    $$conv_t \ \langle\!\langle Q \rangle\!\rangle \ \langle\!\langle Q \rangle\!\rangle \wedge conv_o \ \langle\!\langle P' \rangle\!\rangle \ \langle\!\langle P \rangle\!\rangle \wedge \langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P \rangle\!\rangle \supset \langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P' \rangle\!\rangle$$

    D'où le résultat : $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P' \rangle\!\rangle)$.

  - ($\Leftarrow$) L'hypothèse est $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P' : Q]\!]$, ie $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P' \rangle\!\rangle)$.

    On utilise l'opération BACKCHAIN sur la formule suivante dont le schéma se trouve dans $\mathcal{P}$:

    $$conv_t \ \langle\!\langle Q \rangle\!\rangle \ \langle\!\langle Q \rangle\!\rangle \wedge conv_o \ \langle\!\langle P \rangle\!\rangle \ \langle\!\langle P' \rangle\!\rangle \wedge \langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P' \rangle\!\rangle \supset \langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P \rangle\!\rangle$$

    D'où le résultat : $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\langle\!\langle Q \rangle\!\rangle \ \langle\!\langle P \rangle\!\rangle)$, ie $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$.

– Preuve de l'affirmation **(2)** :

– Dans le sens $\Rightarrow$:

$$\begin{aligned}
[\![([R/x]\ P) : ([R/x]\ Q)]\!] &= \langle\!\langle([R/x]\ Q)\rangle\!\rangle\ \langle\!\langle([R/x]\ P)\rangle\!\rangle \\
&\Downarrow \text{ d'après le Lemme 1} \\
&= \Big(\big[\langle\!\langle R\rangle\!\rangle/x\big]\ \langle\!\langle Q\rangle\!\rangle\Big)\ \Big(\big[\langle\!\langle R\rangle\!\rangle/x\big]\ \langle\!\langle P\rangle\!\rangle\Big) \\
&\Downarrow \text{ propriété de la substitution} \\
&= \big[\langle\!\langle R\rangle\!\rangle/x\big]\ \Big(\langle\!\langle Q\rangle\!\rangle\ \langle\!\langle P\rangle\!\rangle\Big) \\
&= \big[\langle\!\langle R\rangle\!\rangle/x\big]\ [\![P : Q]\!]
\end{aligned}$$

– Preuve identique dans le sens $\Leftarrow$.

Donc $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![([R/x]\ P) : ([R/x]\ Q)]\!]$ si et seulement si $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\langle\!\langle R\rangle\!\rangle/x]\ [\![P : Q]\!]$.

– Pour prouver **(3)** nous utilisons la formule ajoutée à $\mathcal{P}$ pour obtenir la convertibilté dans les buts et dans les hypothèses.

• Pour $\Rightarrow$ nous avons les hypoyhèses suivantes :

$$\begin{cases} \mathcal{P}, [\![\Gamma]\!], [\![x : Q]\!] \vdash_{\mathcal{I}} A \text{ , c'est-à-dire } \mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q\rangle\!\rangle\ x \vdash_{\mathcal{I}} A & (1) \\ Q =_{\beta\eta} Q', \text{ ie (d'après le Lemme 2) } \mathcal{P} \vdash_{\mathcal{I}} conv_t\ \langle\!\langle Q\rangle\!\rangle\ \langle\!\langle Q'\rangle\!\rangle & (2) \end{cases}$$

Posons $F = conv_t\ \langle\!\langle Q\rangle\!\rangle\ \langle\!\langle Q'\rangle\!\rangle \wedge conv_o\ x\ x \wedge \langle\!\langle Q'\rangle\!\rangle\ x \wedge \Big(\langle\!\langle Q\rangle\!\rangle\ x \supset A\Big)$. Nous obtenons alors la démonstration suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1\ \ \Pi_2\ \ \Pi_3\ \ \Pi_4}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q'\rangle\!\rangle\ x \vdash_{\mathcal{I}} F}\ (\wedge R) \quad \dfrac{A \vdash_{\mathcal{I}} A}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q'\rangle\!\rangle\ x, A \vdash_{\mathcal{I}} A}\ (aff.)}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q'\rangle\!\rangle\ x, F \supset A \vdash_{\mathcal{I}} A}\ (\supset L)}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q'\rangle\!\rangle\ x \vdash_{\mathcal{I}} A}\ (F \supset A \in \mathcal{P})}{\mathcal{P}, [\![\Gamma]\!], [\![x : Q']\!] \vdash_{\mathcal{I}} A}$$

$$\dfrac{\dfrac{\text{hypothèse (2)}}{\mathcal{P} \vdash_{\mathcal{I}} conv_t\ \langle\!\langle Q\rangle\!\rangle\ \langle\!\langle Q'\rangle\!\rangle}}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q'\rangle\!\rangle x \vdash_{\mathcal{I}} conv_t\ \langle\!\langle Q\rangle\!\rangle\ \langle\!\langle Q'\rangle\!\rangle}\ (aff.)$$

preuve Π_1

$$\dfrac{\dfrac{\mathcal{P} \vdash_{\mathcal{I}} conv_o\ x\ x}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q' \rangle\!\rangle x \vdash_{\mathcal{I}} conv_o\ x\ x}\ (aff.)}{}$$

preuve Π_2

$$\dfrac{\dfrac{\langle\!\langle Q' \rangle\!\rangle\ x \vdash_{\mathcal{I}} \langle\!\langle Q' \rangle\!\rangle\ x}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q' \rangle\!\rangle\ x \vdash_{\mathcal{I}} \langle\!\langle Q' \rangle\!\rangle\ x}\ (aff.)}{}$$

preuve Π_3

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{hypothèse (1)}}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q \rangle\!\rangle\ x \vdash_{\mathcal{I}} A}}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q' \rangle\!\rangle\ x, \langle\!\langle Q \rangle\!\rangle\ x \vdash_{\mathcal{I}} A}\ (aff.)}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q' \rangle\!\rangle\ x \vdash_{\mathcal{I}} \Big(\langle\!\langle Q \rangle\!\rangle\ x\Big) \supset A}\ (\supset R)}{}$$

preuve Π_4

• Pour $\Leftarrow$ il suffit de permuter Q et Q', le schéma de la preuve restant le même.

– Pour prouver **(4)** nous procédons par induction sur la structure de $[\![P : Q]\!]$. Nous présentons ici la preuve pour $\Rightarrow$, la preuve de $\Leftarrow$ étant identique en permutant Q et Q'.

Nous avons les hypothèses suivantes :

$$\begin{cases} \mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!], \text{ c'est-à-dire } \mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \langle\!\langle Q \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle & (1) \\ Q =_{\beta\eta} Q', \text{ ie (d'après le Lemme 2) } \mathcal{P} \vdash_{\mathcal{I}} conv_t\ \langle\!\langle Q \rangle\!\rangle\ \langle\!\langle Q' \rangle\!\rangle & (2) \end{cases}$$

En posant $F = conv_t\ \langle\!\langle Q \rangle\!\rangle\ \langle\!\langle Q' \rangle\!\rangle \wedge conv_o\ \langle\!\langle P \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle \wedge \langle\!\langle Q \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle$ nous avons alors la démonstration suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1\ \ \Pi_2\ \ \Pi_3}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} F}\ (\wedge R)\quad \dfrac{\langle\!\langle Q' \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle \vdash_{\mathcal{I}} \langle\!\langle Q' \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle}{\mathcal{P}, [\![\Gamma]\!], \langle\!\langle Q' \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle \vdash_{\mathcal{I}} \langle\!\langle Q' \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle}\ (aff.)}{\mathcal{P}, [\![\Gamma]\!], F \supset (\langle\!\langle Q' \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle) \vdash_{\mathcal{I}} \langle\!\langle Q' \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle}\ (\supset L)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \langle\!\langle Q' \rangle\!\rangle\ \langle\!\langle P \rangle\!\rangle}\ (\dagger)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q']\!]}$$

$$\dfrac{\dfrac{\dfrac{\text{hypothèse (2)}}{\mathcal{P} \vdash_{\mathcal{I}} conv_t\ \langle\langle Q \rangle\rangle\ \langle\langle Q' \rangle\rangle}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} conv_t\ \langle\langle Q \rangle\rangle\ \langle\langle Q' \rangle\rangle}\ (aff.)}{}$$

preuve Π_1

$$\dfrac{\dfrac{conv_o\ \langle\langle P \rangle\rangle\ \langle\langle P \rangle\rangle \in \mathcal{P}}{\mathcal{P} \vdash_{\mathcal{I}} conv_o\ \langle\langle P \rangle\rangle\ \langle\langle P \rangle\rangle}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} conv_o\ \langle\langle P \rangle\rangle\ \langle\langle P \rangle\rangle}\ (aff.)$$

preuve Π_2

$$\dfrac{\text{hypothèse (1)}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \langle\langle Q \rangle\rangle\ \langle\langle P \rangle\rangle}$$

preuve Π_3

(†) car $F \supset (\langle\langle Q' \rangle\rangle\ \langle\langle P \rangle\rangle) \in \mathcal{P}$

□

Théorème 2 (correction) *Soit Γ un contexte valide. Soient P et Q des termes d'AF_2. Si $\Gamma \vdash P : Q$ a une dérivation dans AF_2, alors $\mathcal{P}, [\![\Gamma]\!] \to [\![P : Q]\!]$ a une preuve dans $\mathcal{I}$.*

Preuve

Nous procédons par induction sur la hauteur d'une dérivation dans AF_2 de $\Gamma \vdash P : Q$.

- La dernière règle appliquée dans la dérivation est la règle $(axiom)$:

  $P : Q \in \Gamma$, d'où $[\![P : Q]\!] \in [\![\Gamma]\!]$ par définition de $[\![\Gamma]\!]$.

  D'où $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$.

- La dernière règle appliquée dans la dérivation est la règle (abs_i) :

  $P : Q$ est donc de la forme $\lambda x.t : A \to B$.

  Nous avons l'hypothèse suivante : $\Gamma, x : A \vdash t : B$

Par hypothèse d'induction, celle-ci devient : $\mathcal{P}, [\![\Gamma]\!], [\![x : A]\!] \vdash_{\mathcal{I}} [\![t : B]\!]$

Posons $F = [\![P : Q]\!]$
$= [\![\lambda x.t : A \to B]\!]$
$= \langle\!\langle A \to B \rangle\!\rangle \ \langle\!\langle \lambda x.t \rangle\!\rangle$
$= (\lambda f.(\forall y.([\![y : A]\!] \supset [\![f y : B]\!]))) \ (abs_o \ \lambda x.\langle\!\langle t \rangle\!\rangle)$

Nous obtenons alors la démonstration suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{hypothèse}}{\mathcal{P}, [\![\Gamma]\!], [\![x : A]\!] \vdash_{\mathcal{I}} [\![t : B]\!]}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} ([\![x : A]\!] \supset [\![t : B]\!])} (\supset R)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\forall y.([\![y : A]\!] \supset [\![t : B]\!]))} (\forall L)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\forall y.([\![y : A]\!] \supset [\![(\lambda x.t) \ y : B]\!]))} (substitution)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\lambda f.(\forall y.([\![y : A]\!] \supset [\![f y : B]\!]))) \ \langle\!\langle \lambda x.t \rangle\!\rangle} (substitution)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} F}$$
$$\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$$

– La dernière règle appliquée dans la dérivation est la règle (app_i) :

$P : Q$ est donc de la forme $(tu) : B$.

Nous avons les hypothèses suivantes : $\begin{cases} \Gamma \vdash t : A \to B \\ \Gamma \vdash u : A \end{cases}$

Par hypothèse d'induction, celles-ci deviennent : $\begin{cases} \mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t : A \to B]\!] & (1) \\ \mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![u : A]\!] & (2) \end{cases}$

Nous obtenons alors la démonstration suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{hypothèse (1)}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t : A \to B]\!]}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \langle\!\langle A \to B \rangle\!\rangle \ \langle\!\langle t \rangle\!\rangle} (codage)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \lambda f.(\forall x.([\![x : A]\!] \supset [\![f x : B]\!])) \ \langle\!\langle t \rangle\!\rangle} (codage)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \forall x.([\![x : A]\!] \supset [\![t x : B]\!])} \qquad \dfrac{\text{hypothèse (2)}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![u : A]\!]}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t u : B]\!]} (\supset E)$$
$$\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$$

– La dernière règle appliquée dans la dérivation est la règle (gen_1) :

$P : Q$ est donc de la forme $t : \forall x.A$.

Nous avons l'hypothèse suivante : $\Gamma \vdash t : A$

Par hypothèse d'induction, celle-ci devient : $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t : A]\!]$

Nous obtenons alors la démonstration suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{hypothèse}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t : A]\!]}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \forall x.[\![t : A]\!]}\ (\forall R)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} (\lambda f.(\forall x.[\![f : A]\!]))\ \langle\!\langle t \rangle\!\rangle}\ (substitution)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \langle\!\langle \forall x.A \rangle\!\rangle\ \langle\!\langle t \rangle\!\rangle}\ (codage)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t : \forall x.A]\!]}\ (codage)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]}$$

– La dernière règle appliquée dans la dérivation est la règle (spe_1) :

$P : Q$ est donc de la forme $t : [b/x]A$.

Nous avons l'hypothèse suivante : $\Gamma \vdash t : \forall x.A$

Par hypothèse d'induction, celle-ci devient : $\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t : \forall x.A]\!]$

Nous obtenons alors la démonstration suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{hypothèse}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t : \forall x.A]\!]}}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \langle\!\langle \forall x.A \rangle\!\rangle\ \langle\!\langle t \rangle\!\rangle}\ (codage)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \lambda f.(\forall x.[\![f : A]\!])\ \langle\!\langle t \rangle\!\rangle}\ (codage)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} \forall x.[\![t : A]\!]}\ (\forall E)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\langle\!\langle b \rangle\!\rangle / x]\ [\![t : A]\!]}\ (Lemme\ 2.2)}{\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![t : [b/x]A]\!]}$$
$$\mathcal{P}, [\![\Gamma]\!] \vdash_{\mathcal{I}} [\![P : Q]\!]$$

– Lorsque la dernière règle appliquée dans la dérivation est la règle (gen_2) (respectivement (spe_2)), la preuve est identique à celle donnée pour la règle (gen_1) (respectivement (spe_1)).

□

Remarque Lorsque la dernière règle appliquée est (app_i), (spe_1) ou (spe_2) nous avons utilisé dans les démonstrations les règles $(\supset E)$ et $(\forall E)$. Or ces règles figurent dans le système NDI et non dans le système LJ qui a été utilisé pour définir la logique $\mathcal{I}$. Ces deux systèmes sont toutefois équivalents, ce qui signifie que ces deux règles seraient plutôt des lemmes dont la preuve dans LJ pourrait éventuellement nécessiter plusieurs lignes.

3.4 Complétude du codage

La complétude (ie toute preuve de $\mathcal{I}$ correspond à une dérivation en AF_2) ne peut pas être obtenue de manière aussi directe. En fait, il y a actuellement *trop* de preuves dans $\mathcal{I}$, et toutes ne correspondent pas à des dérivations d'AF_2. Les preuves supplémentaires sont le résultat des formules de la Figure 7, lesquelles permettent trop de liberté dans la conversion des termes.

Si nous nous restreignons à n'utiliser que des termes d'AF_2 réduits en forme normale nous obtenons alors la complétude. De façon plus précise, considérons la construction d'une preuve par séquents du codage d'une proposition dans le sens inverse. Pour toutes les règles d'inférence exceptée $(\forall L)$, si tous les termes d'AF_2 codés apparaissant dans les formules de la conclusion sont en forme normale, alors ceux apparaissant dans les hypothèses sont également en forme normale. Nous pouvons imposer un invariant de *mise en forme normale* en vérifiant, lors d'une application de $(\forall L)$, que tous les termes des hypothèses soient immédiatement mis en forme normale. Nous devons également enrichir la spécification de la conversion avec des formules indiquant si un terme est en forme normale.

Une solution consiste donc à modifier la définition du codage afin d'introduire des sous-buts de normalisation quand cela est nécessaire dans les buts et les hypothèses. Cette approche a été détaillée pour LF dans [Felty 88] et pour CC dans [Felty 93a].

3.5 Recherche de preuves

Nous rappelons que nous avons défini le codage d'AF_2 dans $\mathcal{I}$ afin de pouvoir utiliser la procédure de recherche de preuves uniformes de $\mathcal{I}$ sur les formules d'AF_2 ainsi codées. Cependant, l'opération BACKCHAIN nécessaire pour $\mathcal{I}$ est fortement indéterministe. Nous proposerons une modification de la procédure de recherche de $\mathcal{I}$ afin de résoudre ce problème.

Nous aborderons ensuite quelques considérations de mise en œuvre spécifiques à l'utilisation du langage λProlog. Nous verrons entre autres de quelle façon peuvent être représentés les types de données à la Martin-Löf.

3.5.1 La procédure de recherche pour $\mathcal{I}$

Informellement, la mise en œuvre de l'opération BACKCHAIN peut être décrite comme suit. Nous choisissons une paire $\langle \mathcal{G}, A \rangle$ dans $|\mathcal{P}|$ telle que A soit atomique Nous remplaçons toutes les variables libres de A par des variables logiques puis nous essayons d'unifier A avec le but atomique courant G. Si l'unification réussit, nous appliquons la substitution résultante aux formules de $\mathcal{G}$ et essayons de prouver chacune d'entre elles.

Puisque la formule A de la paire choisie est atomique, elle est de la forme $p\ t_1 \cdots t_n$. Dans hh, p doit être une constante, donc toute substitution de variables donnera comme résultat une formule atomique. Par contre, dans $\mathcal{I}$ si p est une variable et si A ne s'unifie pas avec le but, nous devons quand même tenir compte des substitutions qui transforment A en une formule non-atomique G, puis considérer l'ensemble $|G|$. Prenons par exemple le cas d'une substitution pour p transforme A en une implication $B \supset A'$ où A' est atomique. Nous devons alors regarder les paires de $|\{B \supset A'\}|$ dont le second élément est atomique. Dans notre cas, il n'y a que $\langle\{B\}, A'\rangle$. Nous essayons alors d'unifier A avec A'. Si cela réussit, B devient un sous-but supplémentaire à prouver avec les hypothèses de $\mathcal{G}$. Si l'unification échoue, nous devons continuer le processus.

Dans [Miller 91] il a été démontré que l'unification peut être utilisée pour mettre en œuvre l'opération BACKCHAIN pour hh et que cela est suffisant pour déterminer les substitutions. Cependant, du fait que dans $\mathcal{I}$ nous autorisions les variables en tête d'une sous-formule atomique, l'unification ne suffit plus car nous devons donc introduire une substitution pour p qui transforme A en une formule non-atomique, mais nous n'avons aucune indication sur ce que doit être cette substitution.

Vers une autre procédure La solution proposée par Amy Felty dans son codage de CC [Felty 93a] est de modifier l'interpréteur pour obtenir une procédure qui est incomplète mais qui se comporte comme la procédure de recherche *transitivement complète* donnée pour CC dans [Dowek 91]. De manière informelle, une procédure est transitivement complète si, lorsque nous essayons de prouver $\Gamma \vdash P : Q$, il est possible de démontrer une série de *lemmes* conduisant éventuellement au résultat désiré. Par exemple :

$$\begin{array}{ll}
\Gamma \vdash P_1 : Q_1 & \\
\Gamma, P_1 : Q_1 \vdash P_2 : Q_2 & \\
\vdots & \\
\Gamma, P_1 : Q_1, \cdots, P_n : Q_n \vdash P : Q & \text{où } n \geq 0
\end{array}$$

La modification à apporter à notre procédure peut être décrite simplement comme suit : ne pas restreindre l'application de l'opération BACKCHAIN aux formules atomiques, toujours tenter d'appliquer BACKCHAIN avant toute autre opération de recherche, et toujours utiliser l'unification pout tenter d'unifier le but courant avec la sous-formule de tête d'une hypothèse.

Prenons l'exemple suivant où nous voulons démontrer que $(t\ u)$ a pour type B sachant que u est de type A. Avant de pouvoir démontrer ceci, nous devons prouver le lemme suivant : t est de type $A \supset B$. En termes de recherche, cela signifie que nous devons trouver une preuve de $A \supset B$ avant d'obtenir une preuve de B. Cet exemple est exactement le cas de la règle d'inférence (app_i).

3.5.2 Mise en œuvre en λProlog

L'implantation en λProlog (version MALI) de ce codage nous conduit à analyser certains points propres au langage. La syntaxe de ce langage a déjà été donnée au chapitre 2. En outre, lorsque nous avons illustré notre codage par l'exemple de l'addition, nous avons introduit le prédicat `jud`.

Vérification de type

Nous définissons également le prédicat `typ` de type $(ob \to o) \to (ob \to o)$. La raison est la suivante : codons la formule $(\forall y.(I \supset I))$ où I est une variable libre du programme. Nous obtenons la formule suivante dans $\mathcal{I}$:

```
(jud (g\ (pi y\ (jud (f\ (pi x\ (jud I x => jud I
(appo f x)))) g))))
```

Mais lorsque l'interprète λProlog arrivera sur `(jud I x)` il ne déterminera pas le type de `I` et de `x`, mais le type de `(I x)`, à savoir $(ob \to o) \to ob$. Nous utilisons donc le prédicat `typ` pour signaler à l'interprèteur que `I` est un type. Nous traduirons donc la formule d'AF_2 initiale par :

```
(jud (g\ (pi y\ (jud (f\ (pi x\ (jud (typ I) x =>
jud (typ I) (appo f x)))) g))))
```

Dérivation automatique

En ce qui concerne la recherche automatique de preuves, celle-ci s'effectue en deux étapes et en *backtracking* :

- **1.** `jud A M :- A M .`

  Nous utilisons la recherche de preuves uniformes implantée dans λProlog.

- **2.** `jud A M :- convo A B , convt M N , jud B N .`

  Nous utilisons les relations de convertibilité. Cette étape est nécessaire du fait de la définition explicite de l'abstraction et de l'application.

Il faut cependant remarquer que la deuxième étape introduit un indéterminisme supplémentaire dans le choix des termes $\beta\eta$-équivalents. Une solution proposée dans [Felty 93a] consiste à remplacer la spécification de la convertibilité donnée Figure 6 par une procédure de normalisation. En effet, en ne travaillant qu'avec des termes en forme normale, cet indéterminisme devrait être supprimé.

Types de données à la Martin-Löf

Nous allons maintenant voir de quelle manière peuvent être représentés les types de données récursifs exprimés dans le style de Martin-Löf. Le point crucial est bien entendu l'implantation des règles d'induction.

Nous reprenons l'exemple du type entier dont le système de règles a été donné au chapitre 1. Nous introduisons quatre nouvelles constantes.

$$\texttt{nzero} = \langle\langle \underline{0} \rangle\rangle \qquad \texttt{tzero} = \langle\langle 0 \rangle\rangle$$
$$\texttt{nsucc} = \langle\langle \underline{s} \rangle\rangle \qquad \texttt{tsucc} = \langle\langle s \rangle\rangle$$

Nous pouvons maintenant mettre en œuvre les types de données récursifs à la Martin-Löf. Les règles d'introduction et d'élimination (ie d'induction) sont codées directement, moyennant les définitions de prédicats pour rechercher le paramètre n de l'induction et pour le remplacer par $\underline{0}$ ou par $(\underline{s}\ y)$ dans le cas des entiers. Nous obtenons ainsi le code λProlog suivant :

```
jud (appt nnat nzero) tzero.

jud (appt nnat (appi nsucc X)) (appo tsucc T) :- jud (appt nnat X) T.

jud (appt X I) (rec C A B) :-
      format "Induction sur les entiers avec " [ ], print (appt X I) ,nl, nl,
      trouve_param I Y,
      format " -- parametre trouve : " [ ], print Y, nl, nl,
      % ---
      subst_param I Y nzero I1,
      convi I1 I1bis,
      jud (appt X I1bis) A,
      format " -- 1ere etape terminee" [ ], nl, nl,
      % ---
      pi M\ ( (jud (appt nnat M) _U) => (subst_param I Y M I2, convi I2 I2bis,
         jud (appt X I2bis) _V) => (subst_param I Y (appi nsucc M) I3,
         convi I3 I3bis, print "2eme etape", nl, jud (appt X I3bis) B)),
      % ---
      jud (appt nnat Y) C.
```

En ce qui concerne les règles d'égalité, une solution est de les représenter par une relation de convertibilité au niveau des individus. Par exemple, la règle indiquant que la longueur d'une liste vide est égale à zéro (ie $long([\,]) = \underline{0}$) sera traduite par `convi (appi long vide) nzero`.

Cela pose cependant un problème. En effet, de par la symétrie de la relation $conv_i$, l'interpréteur remplace de temps en temps `nzero` par `(appi long vide)`

alors qu'il effectue une induction sur les entiers par exemple. Il est possible de résoudre ce problème en supprimant la symétrie de la relation $conv_i$. Nous n'avons plus alors des règles d'égalité mais des règles de réécriture.

3.5.3 Exemple d'application

Afin d'illustrer le fonctionnement de la procédure de recherche, nous allons détailler les premières étapes de son application à l'exemple de l'addition. Cet exemple nous permettra de mettre en évidence le problème lié à l'opération BACKCHAIN et d'appliquer la solution que nous avons proposée. Nous pourrons ainsi de définir la classe de preuves pour lesquelles la procédure de recherche de $\mathcal{I}$, dans sa version actuelle, diverge.

Nous partons de la formule suivante où `T` représente le λ-terme qui sera construit durant la preuve. Nous rappelons qu'il s'agit du codage de formule $\forall x.(Nx \supset \forall y.(Ny \supset Nplus(x,y)))$:

```
(jud (f\ (pi x\ (jud (g\ (pi u\ (jud (appt nat x) u => (jud (h\ (pi
y\ (jud (i\ (pi v\ (jud (appt nat y) v => jud (appt nat (appi (appi
plus x) y)) (appo i v)))) h))) (appo g u))))) f)))) T
```

En utilisant le prédicat `jud A M :- A M` nous obtenons :

```
(f\ (pi x\ (jud (g\ (pi u\ (jud (appt nat x) u => (jud (h\ (pi y\
(jud (i\ (pi v\ (jud (appt nat y) v => jud (appt nat (appi (appi plus
x) y)) (appo i v)))) h))) (appo g u))))) f))) T
```

Ceci est réduit en :

```
(pi x\ (jud (g\ (pi u\ (jud (appt nat x) u => (jud (h\ (pi y\ (jud
(i\ (pi v\ (jud (appt nat y) v => jud (appt nat (appi (appi plus x)
y)) (appo i v)))) h))) (appo g u))))) T))
```

En appliquant l'opération GENERIC, nous obtenons la formule suivante qui correspond à $Nx \supset \forall y.(Ny \supset Nplus(x,y))$:

```
(jud (g\ (pi u\ (jud (appt nat x) u => (jud (h\ (pi y\ (jud (i\ (pi
v\ (jud (appt nat y) v => jud (appt nat (appi (appi plus x) y)) (appo
i v)))) h))) (appo g u))))) T)
```

En utilisant le prédicat `jud A M :- A M` nous obtenons :

```
(g\ (pi u\ (jud (appt nat x) u => (jud (h\ (pi y\ (jud (i\ (pi v\
(jud (appt nat y) v => jud (appt nat (appi (appi plus x) y)) (appo i
v)))) h))) (appo g u))))) T
```

Ceci est réduit en :

```
(pi u\ (jud (appt nat x) u => (jud (h\ (pi y\ (jud (i\ (pi v\ (jud
(appt nat y) v => jud (appt nat (appi (appi plus x) y)) (appo i v))))
h))) (appo T u))))
```

En appliquant l'opération GENERIC, nous obtenons :

```
(jud (appt nat x) u => (jud (h\ (pi y\ (jud (i\ (pi v\ (jud (appt
nat y) v => jud (appt nat (appi (appi plus x) y)) (appo i v)))) h)))
(appo T u)))
```

En utilisant l'opération AUGMENT, nous ajoutons `(jud (appt nat x) u)` au début du programme et nous obtenons le nouveau but suivant qui correspond à $\forall y.(Ny \supset Nplus(x, y))$:

```
(jud (h\ (pi y\ (jud (i\ (pi v\ (jud (appt nat y) v => jud (appt
nat (appi (appi plus x) y)) (appo i v)))) h))) (appo T u))
```

En utilisant le prédicat `jud A M :- A M` nous obtenons :

```
(h\ (pi y\ (jud (i\ (pi v\ (jud (appt nat y) v => jud (appt nat (appi
(appi plus x) y)) (appo i v)))) h))) (appo T u)
```

Ceci est réduit en :

```
(pi y\ (jud (i\ (pi v\ (jud (appt nat y) v => jud (appt nat (appi
(appi plus x) y)) (appo i v)))) (appo T u)))
```

En appliquant l'opération GENERIC, nous obtenons la formule suivante qui correspond à $Ny \supset Nplus(x, y)$:

```
(jud (i\ (pi v\ (jud (appt nat y) v => jud (appt nat (appi (appi
plus x) y)) (appo i v)))) (appo T u))
```

En utilisant le prédicat `jud A M :- A M` nous obtenons :

```
(i\ (pi v\ (jud (appt nat y) v => jud (appt nat (appi (appi plus
x) y)) (appo i v)))) (appo T u)
```

Ceci est réduit en :

```
(pi v\ (jud (appt nat y) v => jud (appt nat (appi (appi plus x) y))
(appo (appo T u) v)))
```

En appliquant l'opération GENERIC, nous obtenons :

```
(jud (appt nat y) v => jud (appt nat (appi (appi plus x) y)) (appo
(appo T u) v))
```

En utilisant l'opération AUGMENT, nous ajoutons `(jud (appt nat y) v)` au début du programme et nous obtenons le nouveau but suivant qui correspond à $Nplus(x, y)$:

```
(jud (appt nat (appi (appi plus x) y))) (appo (appo T u) v)
```

Puisque nous utilisons la définition itérative du type entier, le prédicat `jud A M :- A M` échoue. Nous utilisons donc le prédicat `jud A M :- convo A B , convt M N , jud B N .` Ceci nous amène au prédicat `convt (appt nat N)` $\cdots$ qui déplie la structure du type entier. Nous obtenons ainsi le nouveau but ci-dessous qui correspond à $\forall X'.((\forall y'.(X'y' \supset X'(\underline{s}\ y'))) \supset (X'\underline{0} \supset X'plus(x, y)))$.

Si nous avions utilisé la représentation du type entier dans le style de Martin-Löf, après l'échec du prédicat `jud A M :- A M` nous aurions utilisé le prédicat codant la règle d'induction sur les entiers. Le paramètre d'induction aurait été `x`.

```
(jud (ff\ (pi xx\ (jud (gg\ (pi uu\ ( (jud (hh\ (pi yy\ (jud (ii\
(pi vv\ ( jud (appt xx yy) vv => jud (appt xx (appi succ yy)) (appo
ii vv)))) hh ))) uu) => (jud (jj\ (pi ww\ (jud (appt xx zero) ww => jud
(appt xx (appi (appi plus x) y)) (appo jj ww)))) (appo gg uu))))) ff
)))) (appo (appo T u) v)
```

En utilisant le prédicat `jud A M :- A M` nous obtenons :

```
(ff\ (pi xx\ (jud (gg\ (pi uu\ ( (jud (hh\ (pi yy\ (jud (ii\ (pi
vv\ ( jud (appt xx yy) vv => jud (appt xx (appi succ yy)) (appo ii vv))))
hh ))) uu) => (jud (jj\ (pi ww\ (jud (appt xx zero) ww => jud (appt
xx (appi (appi plus x) y)) (appo jj ww)))) (appo gg uu))))) ff ))) (appo
(appo T u) v)
```

Ceci est réduit en :

```
(pi xx\ (jud (gg\ (pi uu\ ( (jud (hh\ (pi yy\ (jud (ii\ (pi vv\ (
jud (appt xx yy) vv => jud (appt xx (appi succ yy)) (appo ii vv))))
hh ))) uu) => (jud (jj\ (pi ww\ (jud (appt xx zero) ww => jud (appt
xx (appi (appi plus x) y)) (appo jj ww)))) (appo gg uu))))) (appo (appo
T u) v) ))
```

En appliquant l'opération GENERIC, le prédicat `jud A M :- A M` , une β-réduction, puis une nouvelle fois l'opération GENERIC nous obtenons la formule suivante :

```
(jud (hh\ (pi yy\ (jud (ii\ (pi vv\ ( jud (appt xx yy) vv => jud
(appt xx (appi succ yy)) (appo ii vv)))) hh ))) uu) => (jud (jj\ (pi
ww\ (jud (appt xx zero) ww => jud (appt xx (appi (appi plus x) y)) (appo
jj ww)))) (appo (appo (appo T u) v) uu))
```

Nous utilisons alors l'opération AUGMENT afin d'ajouter au début du programme logique la formule de gauche de l'implication, puis nous continuons avec le but suivant qui correspond à $X'\underline{0} \supset X'plus(x, y)$:

```
(jud (jj\ (pi ww\ (jud (appt xx zero) ww => jud (appt xx (appi (appi
plus x) y)) (appo jj ww)))) (appo (appo (appo T u) v) uu))
```

En appliquant le prédicat `jud A M :- A M` , une β-réduction, puis l'opération GENERIC nous obtenons la formule suivante :

```
(jud (appt xx zero) ww => jud (appt xx (appi (appi plus x) y)) (appo
(appo (appo (appo T u) v) uu) ww))
```

En utilisant l'opération AUGMENT nous ajoutons au début du programme l'hypothèse `(jud (appt xx zero) ww)` . Nous poursuivons la dérivation avec le but suivant qui correspond à $X'plus(x, y)$:

```
jud (appt xx (appi (appi plus x) y)) (appo (appo (appo (appo T u)
v) uu) ww)
```

Arrivé à ce stade de la dérivation, il est nécessaire d'appliquer l'opération BACKCHAIN telle que nous l'avons modifiée afin de scinder la preuve en deux sous-preuves : l'une pour $X'y \supset X'plus(x, y)$, l'autre pour $X'y$, ce qui correspondrait aux deux sous-buts suivants :

```
(jud (fff (pi xxx (jud (appt xx y) xxx =>jud (appt xx (appi (appi
plus x) y)) (appo fff xxx))))) TT
```

```
(jud (appt xx y)) UU
```

avec `(appo TT UU)` $\equiv$ `(appo (appo (appo (appo T u) v) uu) ww)`

Comme nous avons pu le constater, la dérivation obtenue est la même que celle détaillée manuellement au chapitre 1. En outre, sur cet exemple relativement simple, nous avons été confronté au problème lié à la mise en œuvre de l'opération BACKCHAIN.

En fait, toutes les dérivations utilisant la règle (app_i) divergent, car au lieu de simuler l'application de cette règle par l'opération BACKCHAIN, l'interpréteur λProlog tente d'appliquer le prédicat `jud A M :- convo A B , convt`

`M N , jud B N .` Le *backtracking* et les relations de convertibilité vont alors construire des λ-termes équivalents de plus en plus gros sans jamais avancer dans la dérivation.

3.6 Conclusion

Nous venons donc d'étudier l'application de la procédure de recherche de preuves uniformes [Miller 91] aux formules d'AF_2 codées dans $\mathcal{I}$. Comme nous l'avons démontré, cette procédure de recherche, telle qu'elle est définie actuellement, n'est pas capable de trouver les preuves faisant appel à la règle (app_i). Pour que cela soit possible, il serait nécessaire de la modifier, notamment au niveau de l'opération BACKCHAIN. Une autre solution serait d'étudier la procédure de recherche de preuves pour *hh* proposée dans [Nadathur 93].

De ce fait, lorsque nous avons mis en œuvre ce codage en λProlog (version MALI) et que nous avons développé quelques exemples, nous nous sommes confrontés au probléme lié à l'opération BACKCHAIN, celle-ci ne pouvant être modifiée. Par conséquent, certains exemples fonctionnent, d'autres pas (en fait, ceux qui utilise la règle (app_i), donc la plupart). Toutefois, on voit qu'en résolvant le probléme au niveau de l'opération BACKCHAIN on pourrait développer automatiquement des preuves de spécifications en AF_2 (cf chapitre 1) et synthétiser des programmes.

Plutôt que de modifier l'opération BACKCHAIN, on pourrait développer une procédure de recherche pour AF_2 en se basant sur la notion de preuves uniformes et en utilisant les résultats de ce chapitre. C'est à cet effet qu'au chapitre suivant nous allons donner une représentation du système AF_2 sous forme de *tactiques*. Une procédure de recherche pourrait alors être définie au-dessus de cette notion de tactique, ie par une *méta-tactique*.

Chapitre 4

Définition de tactiques pour AF_2

Dans ce chapitre, nous allons donner une représentation d'AF_2 sous forme de tactiques. Chacune d'entre elles correspondra à une règle d'inférence d'AF_2. En enchaînant ces tactiques, nous pourrons ainsi construire une preuve d'une formule d'AF_2 donnée. De nombreux travaux ont été effectués afin d'automatiser cet enchaînement, notamment sur les notions de plans de preuves [Galmiche 92b] et de $\mathcal{R}$-preuves [Parigot 92a]. L'objectif est de développer une méta-tactique dont le rôle sera de définir l'ordre d'application des différentes tactiques.

En ce qui nous concerne, nous nous limiterons à la définition des tactiques de base et de quelques méta-tactiques élémentaires en nous inspirant des résultats présentés dans [Felty 93b, Barraband 93]. La première partie de ce chapitre sera consacrée à la définition des notions de tactique et de méta-tactique. Nous proposerons ensuite, dans la seconde partie, une représentation d'AF_2 sous forme de tactiques, puis nous la mettrons en pratique en utilisant le langage λProlog. Enfin, dans la troisième partie, nous proposerons une procédure de recherche (ie une méta-tactique) interactive afin de pouvoir vérifier les tactiques ainsi définies.

4.1 Notions de tactique et de méta-tactique

Une nouvelle approche pour la preuve de théorèmes a été introduite par Edinburgh : encapsuler la logique formelle dans un méta-langage de programma-tion, dans notre cas le langage λProlog. Les termes et les formules sont des valeurs de λProlog ayant une structure arborescente explicite et sont construits par des fonctions λProlog. Les théorèmes (ou jugements) ont un type abstrait *judgment*. Lors d'une preuve par chaînage avant, une règle d'inférence vérifie les prémisses reçus puis génère la conclusion.

Bien que les règles d'inférence soient des fonctions prenant des théorèmes (les hypothèses) et fournissant un théorème (la conclusion), il est possible d'établir une preuve guidée par le but. Nous utilisons alors des *tactiques* (en anglais des *tactics*) qui sont des fonctions réduisant un but en une liste de sous-buts. Nous

construisons ainsi un arbre ET de buts dont la racine est le but initial et dont les feuilles sont des axiomes.

Les théorèmes ne peuvent cependant être générés que par application de règles d'inférence. Chaque tactique doit donc renvoyer *une fonction de validation* qui construira le théorème lors de la remontée et fera également les vérifications d'usage sur les hypothèses (les théorèmes) qui lui auront été fournies par les sous-buts. Ces concepts sont détaillés dans [Milner 84]. Voici en exemple la tactique correspondant à la règle d'inférence (app_i) d'AF_2 (la syntaxe utilisée est celle de λProlog) :

```
app_i_tac  (Gamma    -->  (app_i T U) :: B)
           (andgoal  (Gamma -->  U :: A)
                     (Gamma -->  T :: (imp A B)) )  :- input A.
```

L'opérateur infixé `::` permet de construire un jugement. L'expression `x :: A` signifie «`x` est de type `A`». Le but courant est `(Gamma --> ... :: B)`, ie nous cherchons une preuve de `B` à partir de l'ensemble d'hypothèses `Gamma`. Cette tactique décomposera le but courant en deux sous-buts : `(Gamma --> U :: A)` et `(Gamma --> T :: (imp A B))`. L'opérateur `andgoal` indique que ces sous-buts devront être résolus tous les deux. Le prédicat `input A` permet à l'utilisateur d'indiquer la formule `A` qui sera utilisée par la tactique. La fonction de validation de cette tactique est `(app_i T U)`. La conclusion de cette tactique sera le théorème (ou jugement) `(app_i T U) :: B`.

Afin de pouvoir définir une procédure de recherche, même si celle-ci ne fait que demander à l'uitlisateur qu'elle est la prochaine tactique à utiliser, il est nécessaire de pouvoir enchaîner les tactiques. Pour cela nous utilisons des *méta-tactiques* (en anglais des *tacticals*). Voici quelques exemples de méta-tactiques :

- **Then :**

  Cette méta-tactique signifie «si la tactique `Tac1` réussit, alors appliquer la tactique `Tac2`». L'implantation en λProlog est la suivante :

  ```
  then Tac1 Tac2 InGoal OutGoal :-  Tac1 InGoal MidGoal,
                                    Tac2 MidGoal OutGoal.
  ```

- **Orelse :**

  Cette méta-tactique tente d'appliquer la tactique `Tac1`. Si celle-ci échoue, et seulement dans ce cas, elle essaye alors d'appliquer la tactique `Tac2`. L'implantation en λProlog est la suivante :

  ```
  orelse Tac1 Tac2 InGoal OutGoal :-  Tac1 InGoal OutGoal, !;
                                      Tac2 InGoal OutGoal.
  ```

– **Maptac :**

Cette méta-tactique permet d'appliquer une tactique donnée à une structure de but composée, ie une formule construite en utilisant des buts élémentaires (des jugements) et les opérateurs `andgoal`, `allgoal` et `allsgoal`. Le premier opérateur représente la conjonction de deux buts, alors que le deuxième (respectivement le troisième) représente le quantificateur universel du premier ordre (respectivement du second ordre) sur un but. L'implantation en λProlog est la suivante :

```
maptac Tac truegoal truegoal.

maptac Tac (andgoal InGoal1 InGoal2) OutGoal :-
      maptac Tac InGoal1 OutGoal1,
      maptac Tac InGoal2 OutGoal2,
      goalreduce (andgoal OutGoal1 OutGoal2) OutGoal.

maptac Tac (allgoal InGoal) OutGoal :-
      pi T (maptac Tac (InGoal T) (OutGoal1 T)),
      goalreduce (allgoal OutGoal1) OutGoal.

maptac Tac (allsgoal InGoal) OutGoal :-
      pi T (maptac Tac (InGoal T) (OutGoal1 T)),
      goalreduce (allsgoal OutGoal1) OutGoal.

maptac Tac InGoal OutGoal :- Tac InGoal OutGoal.
```

Le prédicat `goalreduce` permet d'éliminer les buts triviaux (`truegoal`) des expressions de buts plus complexes.

4.2 Définition des tactiques pour AF_2

Nous devons tout d'abord définir la représentation des termes, des formules (ou types) et des jugements d'AF_2 en λProlog. Nous expliquerons ensuite de quelle manière sont implantées les règles d'inférence, ou plus exactement les tactiques associées.

Nous supposons que les variables d'AF_2 sont divisées en six ensembles dénombrables : ν_o^1 et ν_o^2 pour les objets ; ν_t^1 et ν_t^2 pour les types ; ν_i^1 et ν_i^2 pour les individus. Par hypothèse, les variables libres de terme (respectivement de type, d'individu) devant être traduites seront dans ν_o^1 (resp. ν_t^1, ν_i^1). Lorsque la traduction nécessitera de nouvelles variables de terme (resp. de type, d'individu), celles-ci seront prises dans ν_o^2 (resp. ν_t^2, ν_i^2).

Dans ce chapitre, puisque nous représentons AF_2 dans les formules de Harrop héréditaires d'ordre supérieur (hh) qui sont à la base du langage λProlog, nous considèrerons hh comme étant le méta-langage. Nous introduisons le méta-type `proof_object` qui sera le type des objets d'AF_2 une fois traduits. Nous définissons une application bijective ρ_o des variables de $\nu_o^1 \cup \nu_o^2$ vers les méta-variables de type ob. Nous définissons de la même manière une bijection ρ_t des variables de $\nu_t^1 \cup \nu_t^2$ vers les méta-variables de type `bool`, ainsi qu'une bijection ρ_i des variables de $\nu_i^1 \cup \nu_i^2$ vers les méta-variables de type `i`. L'union de ces trois fonctions sera notée ρ, mais pour des raisons de lisibilité, ces traductions seront souvent implicites.

4.2.1 Représentation des types

Nous ne définissons que le codage des connecteurs logiques $\supset$ (implication), $\forall$ (quantificateur universel du premier ordre) et $\forall_S$ (quantificateur universel du second ordre). En effet, dans la logique (intuitionniste) du second ordre, les symboles logiques $\bot, \vee, \wedge, \neg$ et $\exists$, ainsi que la relation d'identité $=$, peuvent être définis à partir de $\supset$, $\forall$ et $\forall_S$. Par exemple $A \wedge B$ est défini par la formule $\forall X.((A \supset (B \supset X)) \supset X)$.

Nous introduisons une constante pour chacun des trois connecteurs. Le type de ces constantes est donné ci-dessous. `S` représente une variable de type.

```
imp      :  bool → bool → bool
forall   :  (i → bool) → bool
foralls  :  (S → bool) → bool
```

En ce qui concerne les prédicats sur les individus, nous procédons de la même manière que pour le démonstrateur automatique. Nous définissons explicitement l'application d'un individu à un prédicat grâce en introduisant la constante `app` de type `S → i → S`.

Nous pouvons maintenant définir la représentation $\langle\langle P \rangle\rangle$ d'une formule de type P d'AF_2. Celui-ci est donné Figure 8.

$$
\begin{aligned}
\langle\langle i \rangle\rangle &= \rho(i) = i \\
\langle\langle A \rangle\rangle &= \rho(A) = A \\
\langle\langle P(i_1, \cdots, i_n) \rangle\rangle &= (\texttt{app}\ \langle\langle P(i_1, \cdots, i_{n-1}) \rangle\rangle\ \langle\langle i_n \rangle\rangle) \\
\langle\langle A \to B \rangle\rangle &= (\texttt{imp}\ \langle\langle A \rangle\rangle\ \langle\langle B \rangle\rangle) \\
\langle\langle \forall x.A \rangle\rangle &= (\texttt{forall}\ x \backslash\ A) \\
\langle\langle \forall_S X.A \rangle\rangle &= (\texttt{foralls}\ X \backslash\ A)
\end{aligned}
$$

Figure 8 : Représentation des types (ou formules) d'AF_2

4.2.2 Représentation des objets

Nous n'allons pas en fait traduire les termes (ou objets) d'AF_2 comme nous les avons définis au chapitre 1. Nous utiliserons plutôt une extension de ces termes qui sera mieux adaptée à la représentation d'un *objet preuve*. En effet, si nous nous référons au système de règles d'inférence que nous avons donné au chapitre 1, lors de l'application de la règle (gen_1) par exemple, il n'en reste aucune trace dans le λ-terme représentant la preuve. De ce fait, nous enrichissons le langage des termes utilisé par AF_2 en introduisant les constantes suivantes :

```
abs_i  :  (proof_object → proof_object) → proof_object
app_i  :  proof_object → proof_object → proof_object
gen_1  :  (i → proof_object) → proof_object
spe_1  :  i → proof_object → proof_object
gen_2  :  (S → proof_object) → proof_object
spe_2  :  S → proof_object → proof_object
```

Les nouvelles règles d'inférence pour AF_2 utilisant ces notations sont présentées Figure 9. Ainsi, le λ-terme que nous obtiendront à la fin de la dérivation donnera une idée plus précise de la manière dont s'est déroulée la preuve.

Règle de l'axiome ($axiom$)	$\Gamma, t : A \vdash t : A$	
Règle d'abstraction (abs_i)	$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\texttt{abs\_i}\lambda x.t) : (\texttt{imp}\ A\ B)}$	
Règle d'application (app_i)	$\dfrac{\Gamma \vdash t : (\texttt{imp}\ A\ B) \quad \Gamma \vdash u : A}{\Gamma \vdash (\texttt{app\_i}\ t\ u) : B}$	
Règle de généralisation (gen_1)	$\dfrac{\Gamma \vdash t : A}{\Gamma \vdash (\texttt{gen\_1}\ t) : (\texttt{forall}\ x\backslash\ A)}$	x non libre dans Γ
Règle de spécialisation (spe_1)	$\dfrac{\Gamma \vdash t : (\texttt{forall}\ x\backslash\ A)}{\Gamma \vdash (\texttt{spe\_1}\ t) : [b/x]A}$	b est un individu
Règle de généralisation (gen_2)	$\dfrac{\Gamma \vdash t : A}{\Gamma \vdash (\texttt{gen\_2}\ t) : (\texttt{foralls}\ X\backslash\ A)}$	X non libre dans Γ
Règle de spécialisation (spe_2)	$\dfrac{\Gamma \vdash t : (\texttt{forall}\ X\backslash\ A)}{\Gamma \vdash (\texttt{spe\_2}\ t) : [B/X]A}$	B est une formule

Figure 9 : Nouvelles règles d'inférence en déduction naturelle d'AF_2

Remarque Si nous supprimons les constantes `gen_1` , `spe_1` , `gen_2` et `spe_2` nous retrouvons le langage des termes initial où `abs_i` et `app_i` représentent explicitement l'abstraction et l'application.

La représentation des termes et des types d'AF_2 proposée ci-dessus possède la propriété suivante :

Lemme 4 *Etant donnés P et Q des termes (respectivement des types) d'AF_2, x une variable, alors* $[\langle\langle Q \rangle\rangle / x]\ \langle\langle P \rangle\rangle\ =\ \langle\langle [Q/x]P \rangle\rangle$.

Preuve

La preuve se fait de façon évidente par induction sur la structure des termes (respectivement des types) et en utilisant les propriétés de la substitution.

□

4.2.3 Représentation des règles d'inférence

Celle-ci se fait de manière quasi-immédiate en utilisant le codage présenté ci-dessus et la notion de tactique développée en début de chapitre. Le prédicat `member` vérifie l'appartenance d'un élément à une liste. Le prédicat `nth_item` retourne le `N`^{eme} élément d'une liste. Le prédicat `nth_item_and_rest` fait de même mais retourne également la liste initiale privée de cet élément.

```
close_tac      (Gamma --> P :: A)
               truegoal
               :- member (P :: A) Gamma.

close_tacn N   (Gamma --> P :: A)
               truegoal
               :- nth_item N (P :: A) Gamma.

abs_i_tac      (Gamma --> (abs_i P) :: (imp A B))
               (allgoal PA ([(PA :: A) | Gamma] --> (P PA) :: B)).

app_i_tac      (Gamma --> (app_i T U) :: B)
               (andgoal (Gamma --> U :: A) (Gamma --> T :: (imp A B)))
               :- format "Entrez le terme sur lequel on applique la regle ...: " [ ],
               input A, nl.

app_i_tacn N   (Gamma --> PC :: C)
               (andgoal (Gamma1 --> PA :: A) ([(app_i P PA) :: B | Gamma1] --> PC :: C))
               :- nth_item_and_rest N (P :: (imp A B)) Gamma Gamma1.
```

```
gen_1_tac      (Gamma --> (gen_1 P) :: (forall A))
               (allgoal T (Gamma --> (P T) :: (A T))).

spe_1_tac N    (Gamma --> PC :: C)
               ([(spe_1 T P) :: (A T) | Gamma1] --> PC :: C)
               :- nth_item_and_rest N (P :: (forall A)) Gamma Gamma1.

gen_2_tac      (Gamma --> (gen_2 P) :: (foralls A))
               (allsgoal T (Gamma --> (P T) :: (A T))).

spe_2_tac N    (Gamma --> PC :: C)
               ([(spe_2 T P) :: (A T) | Gamma1] --> PC :: C)
               :- nth_item_and_rest N (P :: (foralls A)) Gamma Gamma1.
```

Remarque Le seul point qui pourrait surprendre est le cas des règles de spécialisation (spe_1) et (spe_2). En effet, plutôt que d'introduire un quantificateur universel sur la formule du but courant, nous en supprimons un dans une des hypothèses (l'hypothèse numéro N pour être précis). Opérationnellement, cela revient à la même chose. Ce petit stratagème nous permet cependant d'éviter de demander à l'utilisateur le terme sur lequel devra se faire la spécialisation, et nous évitera surtout de devoir remplacer toutes ses occurences libres dans la formule par la nouvelle variable liée par le quantificateur.

4.2.4 Types de données

La mise en œuvre des types de données définis de manière itérative ne nécessite que l'introdution de deux nouvelles tactiques : `fold` et `unfold`. Prenons par exemple le type Entier, cela nous donne :

Dépliage ($unfold$)
$$\frac{\Gamma \vdash t : Nx}{\Gamma \vdash (\texttt{unfold}\ t) : \forall X.\Big(\big(\forall y.(Xy \supset X(\underline{s}\ y))\big) \supset (X\underline{0} \supset Xx)\Big)}$$

Pliage ($fold$)
$$\frac{\Gamma \vdash t : \forall X.\Big(\big(\forall y.(Xy \supset X(\underline{s}\ y))\big) \supset (X\underline{0} \supset Xx)\Big)}{\Gamma \vdash (\texttt{fold}\ t) : Nx}$$

4.3 Une procédure de recherche interactive

Afin de pouvoir mettre en pratique ce travail, nous donnons maintenant une procédure de recherche interactive, ie elle demandera à chaque étape quelle est la tactique à appliquer au but courant. Les réponses de l'utilisateur pourront être :

- **backup :** Ceci permet de revenir à l'étape précédente.
- **quit :** Ceci permet d'arrêter la dérivation dans la branche actuelle. Si une autre branche était en suspend (cas de la règle (app_i) par exemple), la procédure continuera dans celle-ci.
- **do** $<$ *goal* $>$ **:** Accomplit le but spécifié avant de continuer la dérivation.
- $<$ *tactique* $>$ **:** Applique la tactique spécifiée au but courant.

Cette procédure de recherche pourrait être spécifiée en λProlog de la manière suivante :

```
inter (Gamma --> P :: A) NewGoal :-
   nl, format "---------------" [ ],
   nl,
   nl, format "Hypotheses :" [ ],
   nl, print_form_list Gamma 1,
   nl, format "But courant :" [ ],
   nl, format " " [ ], print A,
   nl,
   nl, format "Tactique a appliquer ...: " [ ],
   input Tac,
   ( (Tac = backup), !, fail;
     (Tac = quit) , (NewGoal = (Gamma --> P :: A));
     (Tac = (do G)), G, inter (Gamma --> P :: A) NewGoal;
     Tac (Gamma --> P :: A) MidGoal, maptac inter MidGoal NewGoal;
       inter (Gamma --> P :: A) NewGoal
   ).
```

Cette procédure de recherche, avec les tactiques de base associées, nous permet de construire, manuellement, des preuves de jugements dans AF_2. Nous pouvons remarquer que les tactiques `app_i_tac` et `app_i_tacn`, qui représentent l'application de la règle (app_i), demandent à l'utilisateur de préciser quelle est la formule A à utiliser (soit sous forme explicite, soit en indiquant son numéro s'il s'agit d'une hypothèse). De cette façon, le problème lié à l'opération BACKCHAIN rencontré au chapitre précédent est éliminé. En effet, celui-ci était dû au fait qu'il avait été nécessaire de restreindre l'opération BACKCHAIN pour obtenir une procédure

déterministe. Or, dans le cas présent, l'indéterminisme n'existe plus étant donné que c'est l'utilisateur qui contrôle le déroulement de la preuve. Il faut cependant préciser que nous serions de nouveau confronté à cet indéterminisme si nous tentions d'automatiser cette procédure sans tenir compte des travaux présentés au chapitre 3. Pour cette raison, seule une procédure de recherche interactive a été implantée en utilisant cette notion de tactique.

4.4 Conclusion

Ce que nous venons de vous présenter dans ce chapitre ne constitue donc que le premier pas dans le développement d'une procédure de recherche de preuves automatique pour AF_2. Nous rappelons que la procédure interactive donnée ci-dessus ne nous a servi qu'à vérifier les tactiques que nous avons définies. A long terme, celle-ci devrait être remplacée par une procédure automatique. Cette dernière pourrait être une procédure de recherche spécifique qui reposerait sur l'expérience du codage des preuves uniformes. Cela reviendrait donc á spécialiser la notion de preuve uniforme pour notre type d'application.

Conclusion

Nous venons donc d'aborder le problème de la recherche de preuves en AF_2 d'une manière indirecte. Nous nous sommes en effet placés dans une autre logique (hh) pour laquelle était définie une procédure de recherche de preuves dites uniformes [Miller 91] en proposant un codage d'AF_2 dans $\mathcal{I}$ qui est une extension de hh. L'utilisation de cette procédure sur les formules d'AF_2 codées a cependant mis en évidence un problème majeur. En effet, le codage proposé utilise la logique $\mathcal{I}$. Or l'opération BACKCHAIN nécessaire pour $\mathcal{I}$ est fortement indéterministe. Afin de résoudre ce problème nous avons présenté, dans ses grandes lignes, une modification de cette procédure de recherche.

Un autre problème, lié au codage, est apparu. Nous avons en effet défini un certain nombre de formules dans $\mathcal{I}$ nous permettant d'introduire des $\beta\eta$-conversions dans les buts et dans les hypothèses. Cela introduit cependant un indéterminisme supplémentaire dans le choix des termes $\beta\eta$-équivalents. Comme proposée pour CC dans [Felty 93a], une solution serait de remplacer les spécifications des relations de convertibilité par une procédure de normalisation. En travaillant ainsi avec des termes en forme normale, cet indéterminisme serait levé. Ceci impliquerait également une modification du codage afin d'introduire des sous-buts de normalisation.

Lors de la mise en œuvre du codage proposé en λProlog (version MALI), nous avons été confrontés au probléme lié à l'opération BACKCHAIN, celle-ci ne pouvant être modifiée. Par conséquent, certains exemples fonctionnent, d'autres pas (en fait, ceux qui utilise la règle (app_i), donc la plupart). Toutefois, on voit qu'en résolvant le probléme au niveau de l'opération BACKCHAIN on pourrait développer automatiquement des preuves de spécifications en AF_2 (cf chapitre 1) et synthétiser des programmes.

Plutôt que de modifier l'opération BACKCHAIN, on pourrait développer une procédure de recherche pour AF_2 en se basant sur la notion de preuves uniformes et en utilisant les résultats de ce chapitre. C'est à cet effet qu'au chapitre suivant nous allons donner une représentation du système AF_2 sous forme de *tactiques*. Une procédure de recherche pourrait alors être définie au-dessus de cette notion de tactique, ie par une *méta-tactique*. A cet effet, nous avons donné une représentation d'AF_2 sous forme de tactiques en λProlog. Chacune de ces tactiques de

base correspond à une règle d'inférence d'AF_2. Il est ainsi possible de définir une procédure de recherche sous la forme d'une méta-tactique qui contrôlerait l'ordre dans lequel seraient appliquées les différentes tactiques de base.

Il faut enfin préciser que ceci n'est qu'une approche possible du problème de la recherche de preuve en AF_2. La notion de $\mathcal{R}$-preuve [Parigot 92a, Parigot 92b, Manoury 92] vise également à automatiser cette recherche, de même que les travaux présentés dans [Nadathur 93]. D'autres travaux [Galmiche 92b, Parigot 88] ont pour but d'améliorer l'efficacité des programmes ainsi dérivés.

Bibliographie

[Barraband 93] C. Barraband. Synthèse de preuves et de programmes en λProlog. *Mémoire de DEA, Ecole Normale Supérieure Ecole Polytechnique, Universités de Paris VI, VII et XI*, Septembre 1993.

[Brisset 93] P. Brisset et O. Ridoux. The compilation of λProlog and its execution with MALI. Rapport, Rapport de recherche INRIA-Rennes 1831, Janvier 1993.

[Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. *9th International Conference on Automated Deduction, LNCS 310*, pages 111–120, 1988.

[Bundy 91] A. Bundy, F. Van Harmelen, J. Hesketh et A. Smaill. Experiments with proofs plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, 1991.

[Church 40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Coquand 88] T. Coquand et G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, Février/Mars 1988.

[Dowek 91] G. Dowek. *Démonstration Automatique dans le Calcul des Constructions*. Thèse de Doctorat, Université Paris 7, Décembre 1991.

[Dowek 93] G. Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):287–315, Juin 1993.

[Felty 88] A. Felty et D. Miller. Specifying theorem provers in a higher-order logic programming language. *9th International Conference on Automated Deduction, LNCS 310*, pages 61–80, Argonne, Illinois, USA, Mai 1988.

[Felty 93a] A. Felty. Encoding the calculus of constructions in a higher-order logic. *8th IEEE Symposium on Logic in Computer Science*, pages 233–244, Montreal, Canada, Juin 1993.

[Felty 93b] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11:43–81, 1993.

[Galmiche 90] D. Galmiche. Constructive system for automatic program synthesis. *Theoretical Computer Science*, 71(2):227–239, 1990.

[Galmiche 92a] D. Galmiche. Program development in constructive type theory. *Theoretical Computer Science*, 94:237–259, 1992.

[Galmiche 92b] D. Galmiche et O. Hermann. Automated proof and program development. C.M. Rattray & R.G. Clark, éditeur, *The Unified Computation Laboratory: Modelling, Specifications and Tools*, volume 35, pages 397–410. Oxford University Press, 1992.

[Galmiche 93] D. Galmiche et O. Hermann. SKIL : a system for programming with proofs. *LPAR'93, International Conference on Logic Programming and Automated Reasoning, LNAI 698*, pages 348–350, St. Petersbourg, Russie, Juillet 1993.

[Harper 93] R. Harper, F. Honsell et G. Plotkin. A framework for defining logics. *Journal of ACM*, 40(1):143–184, 1993.

[Helmink 90] L. Helmink. Resolution and type theory. *ESOP 90, LNCS 432*, pages 197–211, Copenhage, Danemark, Mai 1990.

[Henson 89] M.C. Henson. Program development in the constructive set theory tk. *Formal Aspects of Computing*, 1:173–192, 1989.

[Hindley 86] J.R. Hindley et J.P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

[Howard 80] W.A. Howard. The formulae-as-types notion of construction. *To H.B. Curry: essays in Combinatory Logic, λcalculus and formalism*, pages 479–490. Academic Press, 1980.

[Krivine 87] J.L. Krivine et M. Parigot. Programming with proofs. *6th Symposium on Computation Theory*, 1987. Wendisch-Rietz.

[Krivine 90] J.L. Krivine et M. Parigot. Programming with proofs. *J. Inf. Process. Cybern.*, 3:149–167, 1990.

[Manna 80] Z. Manna et R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[Manoury 92] P. Manoury et M. Simonot. *Des preuves de totalité de' fonctions comme synthèse de programmes.* Thèse de Doctorat, Equipe de Logique, Université de Paris VII, 1992.

[Martin-Löf 84] P. Martin-Löf. *Intuitionistic type theory.* Bibliopolis, 1984.

[Miller 91] D. Miller, G. Nadathur, F. Pfenning et A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Milner 84] R. Milner. The use of machines to assist in rigorous proof. *Philos. Trans. Roy. Soc. London*, pages 411–422, 1984.

[Nadathur 93] G. Nadathur. A proof procedure for the logic of hereditary harrop formulas. *Journal of Automated Reasoning*, 11:115–145, 1993.

[Nordström 90] B. Nordström, K. Petersson et Jan M. Smith. *Programming in MLTT: an introduction.* Oxford Science Publications, 1990.

[Parigot 88] M. Parigot. Programming with proofs : a second order type theory. *European Symposium on Programming 88, LNCS 300*, pages 145–159, Nancy, France, Mars 1988. Springer-Verlag.

[Parigot 92a] M. Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356, Mars 1992.

[Parigot 92b] M. Parigot, P. Manoury et M. Simonot. Propre : A programming language with proofs. *International Conference on Logic Programming and Automated Reasoning, LNAI 624*, pages 484–486, St. Petersbourg, Russie, Juillet 1992.

[Paulin-Mohring 89] C. Paulin-Mohring. Extracting F_w's programs from proofs in the calculus of constructions. *6th ACM Symposium on Principles of Programming Languages*, Austin, 1989.

[Paulin-Mohring 93] C. Paulin-Mohring et B. Werner. Synthesis of ML programs in the system coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[Sato 84] T. Sato et H. Tamaki. Transformational logic program synthesis. *International Conference on Fifth Generation Computer Systems, ICOT*, Tokyo, Japan, 1984.