Concepts mathematiques et informatiques formalises dans le calcul des constructions

T. Coquand, Gérard Huet

INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. : (1) 39 63 55 11

Rapports de Recherche

N°463

CONCEPTS MATHÉMATIQUES ET INFORMATIQUES FORMALISÉS DANS LE CALCUL DES CONSTRUCTIONS

Thierry COQUAND
Gérard HUET

Décembre 1985

RESUME

Nous présentons un essai de mécanisation de concepts mathématiques et informatiques dans le Calcul des Constructions. Tous les exemples ont été vérifiés par machine.

ABSTRACT

We present examples of mathematical and computer science concepts expressed in the Constructions Calculus. All these examples have been mechanically checked.

Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions

Thierry Coquand

Gérard Huet

Inria
Rocquencourt
France

Nous présentons un essai de mécanisation de concepts mathématiques et informatiques dans le Calcul des Constructions. Tous les exemples présentés ont été vérifiés par machine.[1]

1 Introduction

Le Calcul des Constructions est un langage logique dont le calcul de types implémente la déduction naturelle en calcul des prédicats d'ordre supérieur. Cette théorie s'appuie sur les travaux de De Bruijn [8,10,11], Girard [25,26] and Martin-Löf [45,49]. Le langage en a été présenté et motivé dans Coquand-Huet [17], et sa cohérence a été prouvée dans la thèse de Coquand [16]. Une version simplifiée, munie d'une sémantique détaillée, est présentée dans Coquand-Huet [19]. Une implémentation prototype a été développée à l'INRIA afin d'expérimenter avec le pouvoir expressif du système. De nombreux exemples ont été vérifiés mécaniquement sur cette implémentation [18,52].

Nous rappelons les règles du calcul dans une première section. Nous expliquons en détail les conventions d'écriture permises par le système. Le reste du papier est une session annotée d'un certain nombre d'exemples caractéristiques. Cette session présente l'ensemble des axiomes, définitions et théorèmes nécessaires à la compréhension des notions introduites, dans la tradition des Principia [76].

2 Le Calcul des Constructions.

2.1 Constructions: contextes, propositions et preuves.

Les Constructions sont des expressions bien typées d'un lambda-calcul typé dont les types sont des expresssions de même nature. Le langage de base s'appuie sur le formalisme Λ^{ω} de Nederpelt [53,54,21]. Nous avons quatre règles de formation:

$*$	*Univers*
$[x : M]N$	*Abstraction*
$(M\ N)$	*Application*
x	*Variable*

[1] Ce papier a été présenté au Colloque de Logique d'Orsay en Juillet 1985.

Dans la règle de formation pour l'abstraction, nous préférons la notation Automath $[x : M]N$ à la notation plus traditionnelle $\lambda x_M \cdot N$ pour deux raisons. Premièrement, le type M associé à la variable liée x peut être très compliqué, et la notation indicée deviendrait trop embrouillée, avec des indices de niveau arbitraire. Deuxièmement, cette opération d'abstraction sert à représenter des produits $\forall x \in M \cdot N$ aussi bien que des fonctions $\lambda x \in M \cdot N$. Le nom x est bien sur complètement arbitraire, et n'est utilisé qu'au niveau de l'interface d'entrée sortie. Dans la syntaxe abstraite, l'opérateur d'abstraction est binaire, avec deux composantes M et N. Les occurrences de la variable x apparaissant dans la syntaxe concrète du terme N sont remplacées par des indices de de Bruijn, qui reflètent la profondeur de la variable dans l'emboîtement des abstractions [9]. Ainsi la formule

$$[x : A]([y : B](x\ y)\ x)$$

est une représentation concrète de la structure abstraite

$$[A]([B](2\ 1)\ 1).$$

Comme il est d'usage en logique combinatoire, on écrit $(M\ N)$ pour l'application du terme N au terme M. On emploie aussi à l'occasion la représentation $\langle N \rangle M$, dans le style Automath.

Notre algèbre de termes est complétée par une constante $*$, qui joue le rôle de la sorte de tous les types (sans être elle-même un type). On peut voir également $*$ comme la sorte de toutes les propositions, suivant l'analogie entre types et propositions exprimée par l'isomorphisme de Curry-Howard. Dans les langages Automath, $*$ est noté τ.

Nous appelerons *construction* un terme bien construit, relativement à un algorithme de vérification de types que nous allons présenter. Cette vérification restreint les termes légaux suivant trois critères. Premièrement, les termes doivent être légaux du point de vue du scope des variables. Deuxièmement, les applications $(M\ N)$ ne sont légales que si le terme M a un type fonctionnel cohérent avec le type du terme N. Troisièmement, nous limitons notre calcul à trois couches de termes: les *contextes*, les *propositions* et les *preuves*. Les contextes sont simplement les termes construits par une suite d'abstractions à partir de $*$:

$$[x_1 : M_1][x_2 : M_2] \ldots [x_n : M_n]*$$

Intuitivement, les contextes sont des déclarations: on introduit les variables $x_1, \ldots, x_n$ avec leurs types. Les propositions sont les constructions dont le type est un contexte. Intuitivement, les propositions sont des formules logiques contenant éventuellement des variables libres. Par exemple, une proposition ayant pour type le contexte ci-dessus peut contenir des occurrences libres des variables $x_1, \ldots, x_n$. Une variable de proposition de ce type peut donc être considérée comme désignant une proposition n-aire. On appelle *énoncé* une proposition fermée, c'est à dire de type $*$.

Finalement, les preuves sont les constructions dont le type est une proposition. Intuitivement, une telle construction est une preuve de la proposition qui est son type. Une preuve fonctionnelle est une preuve dépendant d'hypothèses, vues comme ses paramètres. La proposition correspondante aura comme type ce contexte d'hypothèses. Cette vision fonctionnelle des preuves est conforme aux systèmes d'inférence de logique naturelle [61].

L'interprétation informatique des constructions suivant leurs trois niveaux est de considérer les contextes comme des déclarations, les propositions comme des spécifications, et les preuves comme des algorithmes réalisant la spécification qui est leur type. On peut d'ailleurs voir cette interprétation comme définissant une sémantique de la partie logique de notre formalisme, une proposition étant interprétée par l'ensemble de ses justifications, c'est à dire des algorithmes de ce type. Cette vision constructive de la sémantique est conforme à la logique intuitioniste [39]. Toutefois, l'identification d'une preuve avec une λ-expression permet d'éviter les codages par arithmétisation.

2.2 Typage

Nous n'avons pas la place ici de donner la théorie syntaxique complète de notre calcul. Nous renvoyons le lecteur intéressé à la thèse de Coquand [16]. Nous supposons connue l'opération de substitution $M[x/N]$ qui remplace les occurrences libres de la variable x dans le terme M par le terme N. Nous avons une règle de calcul unique, correspondant à la notion de β-réduction en λ-calcul. Cette règle remplace un sous-terme de la forme $([x : A]M\ N)$ par le terme $M[x/N]$. On peut montrer que cette règle définit une relation de calcul confluente et nœthérienne sur les termes typés. Tout terme bien construit M possède donc une forme irréductible unique, atteignable par une séquence arbitraire de calculs, que l'on appelle *forme canonique* de M, et que l'on désigne par $\downarrow(M)$.

Nous allons décrire l'algorithme de typage des constructions par un système d'inférence, dont les règles manipulent des *séquents* $\Gamma \vdash E$, où Γ est un contexte servant à déclarer les variables libres apparaissant dans l'expression E. Il est pratique de définir quelques notations permettant de manipuler les contextes comme des suites. Si Γ est un contexte et M un terme quelconque, on définit la concaténation $\Gamma; M$ de Γ et M récursivement. Si $\Gamma = *$, alors $\Gamma; M = M$. Si $\Gamma = [x : A]\Delta$, alors $\Gamma; M = [x : A]\ (\Delta; M)$. Lorsque Γ et Δ sont des contextes, on écrit $\Gamma \leq \Delta$ si et seulement s'il existe un contexte Θ tel que $\Delta = \Gamma; \Theta$. On dit que Γ est un *préfixe* de Δ. Finalement, si Γ est un contexte et M un terme quelconque, on écrit $\Gamma[x : M]$ pour $\Gamma; [x : M]*$. Cette notation permet d'écrire tout contexte non vide comme une suite $[x_1 : M_1]...[x_n : M_n]$ et de réserver $*$ pour le contexte vide.

Nous allons maintenant décrire précisément l'algorithme de typage. Les séquents sont de trois formes:

$$\Gamma \vdash * \qquad \textit{Contexte}$$

indique que Γ est un contexte bien construit.

$$\Gamma \vdash P : \Delta \qquad \textit{Proposition}$$

indique que P est une proposition bien construite dans le contexte Γ et de type Δ. Une telle construction entraîne que $\Gamma; \Delta$ est un contexte bien construit.

$$\Gamma \vdash M : P \qquad \textit{Preuve}$$

indique que le terme M est une preuve de la proposition P dans le contexte Γ. Une telle construction entraîne que P est bien construite dans le contexte bien construit Γ. Dans ce qui suit, les meta-variables Γ,Δ et Θ désignent des contextes, P et Q des propositions, M et N des termes non-contextes quelconques, T et U des contextes ou propositions quelconques, X et Y des termes quelconques.

Donnons tout d'abord les règles de construction de contextes:

$$\vdash * \qquad \textit{Contexte1}$$

$$\frac{\Gamma; \Delta \vdash *}{\Gamma[x : \Delta] \vdash *} \qquad \textit{Contexte2}$$

$$\frac{\Gamma; \Delta \vdash P : *}{\Gamma[x : \Delta; \downarrow(P)] \vdash *} \qquad \textit{Contexte3}$$

Un contexte ne peut donc déclarer que des prédicats et des hypothèses en forme canonique. Pour les termes non contextes, on donne des règles de typage, correspondant aux règles de formation de termes:

$$\frac{\Gamma \vdash *}{\Gamma \vdash x : T_x} \quad (\Gamma = \Gamma_1[x : T_x]\Gamma_2) \qquad \textit{Variable}$$

$$\frac{\Gamma[x:X] \vdash M : Y}{\Gamma \vdash [x:X]M : [x:X]Y} \qquad \textit{Abstraction}$$

$$\frac{\Gamma \vdash M : [x:P]X \quad \Gamma \vdash N : P}{\Gamma \vdash (M\ N) : \downarrow (X[x/N])} \qquad \textit{Application1}$$

$$\frac{\Gamma \vdash M : [x:\Delta]X \quad \Gamma \vdash P : \Theta}{\Gamma \vdash (M\ P) : \downarrow (X[x/P])}\ (\Delta \leq \Theta) \qquad \textit{Application2}$$

Le système présenté ci-dessus met systématiquement en forme canonique les expressions de type. Ceci n'est pas strictement nécessaire, mais simplifie la présentation. Quelques explications sont nécessaires. Tout d'abord, la règle *Application2* permet une coercion entre le contexte Θ et son préfixe Δ. Cette règle d'inclusion de types permet de diminuer à volonté la fonctionnalité d'une proposition en quantifiant universellement les hypothèses superflues. Donnons tout de suite un exemple. On se place dans le contexte vide. L'algorithme d'identité polymorphe est construit par la preuve

$$Id = [A:*][x:A]x,$$

de type la proposition

$$Un = [A:*][x:A]A.$$

La proposition Un peut se lire comme le schéma d'implication $A \to A$, lorsque A est une proposition quelconque. Mais elle peut aussi se lire comme l'énoncé $\forall A \cdot A \to A$ décrivant le cardinal 1. Il est donc légal de construire l'application $(Id\ Un)$, qui est l'algorithme d'identité spécialisé à la structure 1.

On peut remarquer également que les types sont toujours dans une forme où leur fonctionnalité est explicite. Dans le jargon du λ-calcul, on parle de forme η-saturée. Par exemple, le contexte $[x:[A:*]][y:x]$ n'est pas bien construit, car la fonctionnalité de y n'est pas apparente. Par contre, $[x:[A:*]][y:[A:*](x\ A)]$ est bien construit. On remarque que la règle de calcul préserve le type des constructions. La règle η n'est par contre pas valide à cause de l'exemple ci-dessus.

Si M est un terme bien construit dans le contexte Γ, on a $\Gamma \vdash M : T$ avec T unique, en forme canonique. On désigne T par $\tau_\Gamma(M)$, ou $\tau(M)$ losque le contexte Γ est clair.

Le Calcul des Constructions présenté ici est plus restreint que celui originellement proposé dans Coquand-Huet [17]. La restriction à trois niveaux est conforme avec la théorie présentée dans la thèse de Coquand [16] et dans Coquand-Huet [18]. En particulier, tout terme bien construit est fortement normalisable (toutes les séquences de calcul issues du terme terminent), ce qui justifie l'utilisation de formes canoniques pour les expressions de type. En tant que système logique le calcul est cohérent, dans la mesure où la proposition absurde $\nabla = [A:*]\ A$ n'a pas de preuve.

2.3 Abréviations

Nous utilisons plusieurs abréviations dans notre syntaxe concrète. Tout d'abord, on peut abrévier $[A:*]\ X$ en $!A \cdot X$. Par exemple, on écrit ∇ comme $!A \cdot A$. Cette abréviation s'itère, par exemple $!A, B, C \cdot X$. On peut interpréter le symbole "!" comme quantifiant sur toutes les propositions. (Et non seulement sur tous les énoncés, à cause de la règle d'inclusion des types expliquée ci-dessus: une variable de type $*$ peut être liée à une proposition ayant pour type un contexte arbitraire, considéré comme préfixe de quantification).

La deuxième abréviation consiste à autoriser l'expression $A \to B$ à la place de $[x:A]\ B$, lorsque x n'apparait pas libre dans B. Le terme $A \to B$, vu comme un type, est le type des fonctions de domaine A et de codomaine B. Un type *dépendent* $[x:A]\ (P\ x)$ est le type d'objets fonctionnels généralisés associant à la valeur X de leur domaine A une valeur de type $(P\ X)$. De

tels constructeurs de type existent par exemple dans la théorie intuitioniste des types de Martin-Löf [49]. Des constructeurs de type analogues existent déjà dans les langages de programmation usuels. Pensez à une procédure Algol admettant un paramètre entier n et un paramètre de tableau de dimension n.

Si l'on pense à $A \to B$ comme une proposition plutôt que comme un type, on peut interpréter la flèche $\to$ comme l'implication intuitioniste. Nos deux abréviations peuvent être vues comme particularisant, au niveau des propositions, les deux constructions de type du calcul de second ordre de Girard [25,26].

Quelques autres abréviations sont autorisées. Par exemple, la syntaxe du *let* de ISWIM [42] et ML [28] est autorisée, sous la forme de $[x = X]\ Y$. Ceci permet de simplifier des constructions complexes avec de multiples occurrences d'une sous-expression X en écrivant $[x = X]\ Y_x$ au lieu de la forme développée Y_X. Ceci présente deux avantages sur l'écriture sous forme de "redex" $([x : A]Y_x\ X)$; tout d'abord, les expressions sont plus lisibles. Ensuite, il n'est pas besoin de spécifier le type A, qui est remplacé implicitement par le type de X.

Les conventions usuelles de la logique combinatoire sont acceptées, et on peut écrire $(A\ B\ C)$ au lieu de $((A\ B)\ C)$. De même, $\to$ associe à droite. Ainsi, $A \to B \to C$ est une abréviation pour $[u : A][v : B]C$.

Nous avons décidé d'implémenter la version prototype du calcul des constructions dans le langage ML [28]. ML est utilisé également en tant que meta-langage du système, ce qui permet à l'utilisateur de macro-générer des constructions paramétriques compliquées. Dans l'implémentation, la syntaxe concrète des constructions est définie par une grammaire Yacc, dont les actions sémantiques engendrent des valeurs ML sous la forme d'arbres de syntaxe abstraite. Les expressions entre guillemets "..." sont ainsi analysées par Yacc. Un programme d'impression permet de restituer à l'utilisateur une forme concrète des constructions qu'il a fabriquées.

L'utilisateur peut manipuler une construction en cours de développement, en faisant naviguer l'interprète de ML à l'intérieur d'un contexte de constructions. Le paragraphe suivant présente les commandes dont il dispose.

2.4 Le système de théories

Nous avons enrichi le langage de base des constructions en autorisant des constantes, désignées par des identificateurs. Tout terme bien construit peut être nommé, et plus tard invoqué par ce nom. Ceci est cohérent avec la notation $[x = X]$ qui peut maintenant être vue comme la forme interne de déclaration d'une constante. On peut donc voir le contexte courant comme une suite de déclarations d'hypothèses, et une suite de déclarations de constantes correspondant à des constructions déjà vérifiées. On peut ajouter des hypothèses et des constantes, en naviguant vers l'intérieur de la construction en cours. On peut aussi décharger des hypothèses et des constantes en naviguant vers l'extérieur.

On peut distinguer les commandes de ce système rudimentaire de théories en deux catégories; les commandes élémentaires fabriquent une construction étape par étape; les commandes de haut niveau utilisent l'analyseur syntaxique pour compiler des suites de commandes élémentaires. Les commandes élémentaires construisent progressivement un terme courant. Ces commandes se comprennent par leur effet sur une petite machine, possédant un registre C (la construction courante), et une pile contenant l'environnement E, constitué du contexte courant d'hypothèses et de constantes. On empile également dans E les arguments déjà construits, en attendant qu'ils soient appliqués. L'environnement généralisé E est donc constitué de déclarations d'hypothèses $[x : M]$, de déclarations de constantes $[x = X]$, et d'arguments en attente $\langle M \rangle$.

- Commandes élémentaires.

  Raz: $C \leftarrow *$

  Ref *nom*: $C \leftarrow nom$

  Var *nom*: $E \leftarrow E\,[nom : C]$; $C \leftarrow *$

  Nomme *nom*: $E \leftarrow E\,[nom = C]$; $C \leftarrow *$

  Empile: $E \leftarrow E\,\langle C\rangle$; $C \leftarrow *$

  Decharge: Si $E = E'\,[x : M]$ alors $E \leftarrow E'$; $C \leftarrow [x : M]\,C$

  Oublie: Si $E = E'\,[x = M]$ alors $E \leftarrow E'$; $C \leftarrow [x = M]\,C$

  Appl: Si $E = E'\,\langle M\rangle$ alors $E \leftarrow E'$; $C \leftarrow (C\ M)$ [Si $(C\ M)$ est bien typé]

  Efface: Si $E = E'\,E_1$ alors $E \leftarrow E'$

- Commandes de haut niveau

  Abs *nom*: Suite de Decharge et de Oublie jusqu'à *nom* compris

  Soit *formule*: $C \leftarrow$ *formule* [Si bien construite]

  Decl *nom formule*: Soit *formule*; Var *nom*

  Const *nom formule*: Soit *formule*; Nomme *nom*

  Prouve *formule*: Vérifie que $\tau(C) = \downarrow(formule)$

  Lemme *nom formule*: Prouve *formule*; Nomme *nom*

  Preuve *formule1 formule2*: Soit *formule2*; Prouve *formule1*.

La description des commandes ci-dessus est volontairement simplifiée. Par exemple, nous ne détaillons pas la vérification de type effectuée lors de l'exécution de la commande Appl. En effet, l'explication du typage par les règles d'inférence n'est plus suffisante en présence de constantes, car on ne veut pas alors avoir tous les types en forme canonique, ce qui serait très coûteux. On ne discute pas ici non plus des problèmes d'allocation de mémoire pour les zônes E et C. Un papier séparé présentera en détail l'implémentation de la machine virtuelle des constructions.

2.5 Un petit exemple

Nous donnons ici l'exemple complètement détaillé d'une preuve effectuée pas à pas à l'aide des commandes élémentaires. Il s'agit de la définition de la connective logique $\wedge$, et de la preuve (sous forme de l'algorithme de première projection) de la proposition correspondant à la règle d'inférence usuelle "$\wedge$-elim-gauche", c'est à dire:

$$\forall A, B \cdot A \wedge B \rightarrow A.$$

Nous donnons cet exemple sous la forme suivante. La première colonne est l'instruction donnée par l'utilisateur. La deuxième colonne indique l'état de la machine après exécution de la commande, sous la forme $E \vdash C$.

Raz;	$\vdash *$
Var A;	$!A \vdash *$
Var B;	$!A, B \vdash *$
Var C;	$!A, B, C \vdash *$
Ref A;	$!A, B, C \vdash A$
Var x;	$!A, B, C\ [x : A] \vdash *$
Ref B;	$!A, B, C\ [x : A] \vdash B$
Var y;	$!A, B, C\ [x : A][y : B] \vdash *$
Ref C;	$!A, B, C\ [x : A][y : B] \vdash C$
Decharge;	$!A, B, C\ [x : A] \vdash [y : B]\ C$
Decharge;	$!A, B, C \vdash [x : A][y : B]\ C$
Var h;	$!A, B, C\ [h : [x : A][y : B]\ C] \vdash *$
Ref C;	$!A, B, C\ [h : [x : A][y : B]\ C] \vdash C$
Abs A;	$\vdash\ !A, B, C\ [h : [x : A][y : B]\ C]\ C$
Nomme $\wedge$;	$[\wedge = !A, B, C\ [h : [x : A][y : B]\ C]\ C] \vdash *$

Il faut comprendre la définition en donnant à la connective $\wedge$ un sens opératoire. L'idée est la suivante. Pour toutes propositions A et B, $A \wedge B$ est une méthode pour prouver toute proposition C, pourvu qu'on ait une méthode h qui à partir d'une preuve x de A et d'une preuve y de B, fournisse une preuve de C. Autrement dit, on a *défini* la connective $\wedge$ par sa *sémantique* (au sens intuitioniste). Donnons maintenant la preuve de $\forall A, B \cdot A \wedge B \rightarrow A$, dans un environnement qui contient la définition de $\wedge$.

Var A;	$\cdots[\wedge = \cdots]\cdots !A \vdash *$
Var B;	$!A, B \vdash *$
Ref B;	$!A, B \vdash B$
Empile;	$!A, B\langle B\rangle \vdash *$
Ref A;	$!A, B\langle B\rangle \vdash A$
Empile;	$!A, B\langle B, A\rangle \vdash *$
Ref $\wedge$;	$!A, B\langle B, A\rangle \vdash \wedge$
Appl;	$!A, B\langle B\rangle \vdash \langle A\rangle\wedge$
Appl;	$!A, B \vdash \langle B, A\rangle\wedge$.
Var h;	$!A, B\ [h : A \wedge B] \vdash *$
Ref A;	$!A, B\ [h : A \wedge B] \vdash A$
Var x;	$!A, B\ [h : A \wedge B][x : A] \vdash *$
Ref B;	$!A, B\ [h : A \wedge B][x : A] \vdash B$
Var y;	$!A, B\ [h : A \wedge B][x : A][y : B] \vdash *$
Ref x;	$!A, B\ [h : A \wedge B][x : A][y : B] \vdash x$
Decharge;	$!A, B\ [h : A \wedge B][x : A] \vdash [y : B]\ x$
Decharge;	$!A, B\ [h : A \wedge B] \vdash [x : A][y : B]\ x$
Empile;	$!A, B\ [h : A \wedge B]\langle[x : A][y : B]\ x\rangle \vdash *$
Ref A;	$!A, B\ [h : A \wedge B]\langle[x : A][y : B]\ x\rangle \vdash A$
Empile;	$!A, B\ [h : A \wedge B]\langle[x : A][y : B]\ x, A\rangle \vdash *$
Ref h;	$!A, B\ [h : A \wedge B]\langle[x : A][y : B]\ x, A\rangle \vdash h$
Appl;	$!A, B\ [h : A \wedge B]\langle[x : A][y : B]\ x\rangle \vdash \langle A\rangle h$
Appl;	$!A, B\ [h : A \wedge B] \vdash \langle[x : A][y : B]\ x, A\rangle h$
Prouve "A";	$\cdots \vdash \cdots$
Abs A;	$\cdots[\wedge = \cdots]\cdots \vdash !A, B\ [h : A \wedge B]\ (h\ A\ [x : A][y : B]\ x)$

On peut maintenant conclure la session ci-dessus en enregistrant le lemme prouvé, au moyen de la commande:

$$\textit{Lemme}\ \pi_1\ \text{“}!A, B \cdot A \wedge B \rightarrow A\text{”}.$$

La preuve de haut niveau correspondante peut se résumer à la construction de l'algorithme de première projection:

$$\textit{Const}\ \pi_1\ \text{“}[A : *][B : *][h : A \wedge B]\ (h\ A\ [x : A][y : B]\ x)\text{”};$$

et à la vérification:

$$\textit{Prouve}\ \text{“}!A, B \cdot A \wedge B \rightarrow A\text{”}$$

Remarques. La commande *Prouve* vérifie effectivement la validité d'une proposition, au sens intuitioniste de posséder une méthode de preuve, en tant qu'algorithme calculant sur des justifications. Nous avons réduit le mécanisme de vérification à un simple calcul de types dans le langage Λ, c'est à dire à la simple déduction naturelle. Nous n'avons pas besoin de postuler des connectives logiques, des axiomes ou règles d'inférence concernant l'usage de ces connectives. Notre système est donc fondamentalement différent de ceux qui postulent au départ des règles particulières, comme le PPλ de LCF [29] ou la théorie des types intuitioniste de Martin-Löf [49]. Le Calcul des Constructions est plus directement apparenté au langage Automath [8], et à la théorie des types de Martin-Löf de 1971 [45].

Le prix que nous avons à payer pour cette simplicité est qu'il nous faut développer un matériel préliminaire important avant d'avoir construit suffisamment de logique pour pouvoir faire des preuves de haut niveau. Les premiers exemples que nous présentons concernent ces préliminaires logiques. Ils ont une complexité intrinsèque, due à leur nature fondamentale. La situation est analogue à la compréhension d'un langage informatique, à partir de son implémentation en langage machine. Il est préférable d'admettre en premier lieu l'existence d'un micro-code et de comprendre les concepts de plus haut niveau en termes de ce micro-code, l'implémentation du micro-code en langage machine étant laissée au spécialiste.

Les exemples ci-dessous vont être présentés dans un symbolisme qui fait un compromis entre les commandes de haut niveau et les commandes de bas niveau, en autorisant l'utilisation de formules complexes, mais en rendant compte explicitement de la structure de l'environnement par l'utilisation de tabulations. La commande *Decl nom formule* se symbolise par *nom* : *formule*, avec introduction d'une tabulation. De même *Const nom formule* se symbolise par *nom* := *formule*. Les commandes d'abstraction *Decharge*, et *Abs* se symbolisent par le retour à la tabulation antérieure. La commande *Prouve formule* se symbolise par $\vdash$ *formule*, et la commande *Nomme nom* par =: *nom*. La commande *Soit formule* se symbolise simplement par l'écriture de *formule*.

2.6 Polymorphisme implicite et synthèse de types

Considérons la constante π_1 ci-dessus. Une construction typique utilisant π_1, dans le contexte d'une paire $[p : nat \wedge nat]$, est $(\pi_1\ nat\ nat\ p)$. Cette écriture est maladroite, car redondante: lier p à h devrait automatiquement lier l'argument de polymorphisme A à *nat* (resp. B à *nat*). On aimerait donc pouvoir écrire simplement $\pi_1(p)$, comme dans l'écriture usuelle. Il est clair que les arguments A et B peuvent être aisément synthétisés comme étant des composantes du type de p, que le système doit calculer de toute façon lors de la vérification des types. Cette notion de polymorphisme *implicite* existe déjà dans le langage ML [28].

Le cadre général dans lequel cette synthèse est possible est le suivant. Soit X un terme arbitraire en forme canonique. On peut l'écrire sous la forme :

$$X = [u_1 : T_1] \cdots [u_n : T_n](x\ X_1\ \cdots\ X_p)$$

où x est une variable, et les T_i et X_j sont de la même forme. Soit V un ensemble de variables. On appelle *occurrence rigide* de X relativement à V l'ensemble des positions dans X suivantes. D'abord, les occurrences rigides dans T_i relativement à $V \cup \{u_1, ..., u_{i-1}\}$, pour $i = 1, ..., n$. Ensuite, si $p = 0$, l'occurrence de x, et sinon, lorsque x appartient à $W = V \cup \{u_1, ..., u_n\}$, les occurrences rigides dans X_j relativement à W, pour $j = 1, ..., p$. Soit z une variable quelconque. On dit que X *détermine* z ssi z apparaît dans X à une occurrence rigide relativement à $\emptyset$. La notion d'occurrence rigide provient d'algorithmes de résolution d'équations en λ-calcul [32,35]. Intuitivement, les occurrences rigides relativement à V sont invariantes par substitution à des variables non dans V.

Nous allons maintenant décrire une généralisation de la commande *Const* en autorisant d'écrire *schema* := *formule*, où *schema* est une suite de chaînes de caractères et de déclarations $[x : X]$. Ceci permet d'introduire une constante avec sa représentation concrète sous forme "mixfixe". Les variables du schéma, qui doivent être toutes distinctes, sont automatiquement déclarées avant la construction de *formule* (de la gauche vers la droite), puis déchargées. Ainsi, la conjonction peut être déclarée comme opérateur infixe par:

$$[A : *] \wedge [B : *] \ := !C \cdot (A \to B \to C) \to C.$$

La commande *Const* généralisée permet d'une certaine manière de déclarer comme constructions des opérateurs de 1er ordre, munis d'une arité. On en profite pour munir l'opérateur d'une syntaxe concrète, mais cette commodité d'écriture n'est qu'un aspect superficiel du concept. Ce qui est plus important est que cette commande spécifie les arguments explicites associés à la constante, correspondant aux variables du schéma. Par exemple, dans la définition de $\wedge$ ci-dessus, les variables A et B sont explicites, et la conjonction est donc un opérateur binaire muni d'une syntaxe infixe. Les arguments supplémentaires sont optionnels. De même avec la construction :

A: $*$;
B: $*$;
$\pi_1([h : A \wedge B]) \ := \ (h\ A\ [x : A][y : B]\, x)$

Ici, la constante π_1 a un argument explicite h. Nous voulons maintenant abstraire π_1 en A et B, pour pouvoir utiliser π_1 comme une constante polymorphe dans le contexte vide. Dans ce cas on va pouvoir le faire sans rajouter explicitement des arguments de polymorphisme à π_1, car π_1 détermine A et B par son argument explicite h. Plus généralement, on pourra abstraire une variable u comme argument implicite d'une constante $C([x_1 : T_1], ..., [x_n : T_n]) \ := \ X$ ssi u est déterminée par un T_i.

Expliquons maintenant la situation générale. On suppose que l'on vient de déclarer la constante C ci-dessus, dans un contexte se terminant par la déclaration de variable u. L'environnement courant est donc de la forme :

$$\cdots [u : U][C = [x_1 : T_1], ..., [x_n : T_n]\ X] \vdash *$$

et on veut maintenant décharger u sans oublier C, grâce à la commande *Gen* que nous allons maintenant décrire. Tout d'abord, l'environnement devient :

$$\cdots [C = [u : U][x_1 : T_1], ..., [x_n : T_n]\ X] \vdash *$$

et maintenant il y a deux cas pour l'invocation de C. Si u est déterminée par l'un des T_i, alors on invoque C sans modification, l'instanciation de u étant fournie automatiquement par le vérificateur de types, qui va vérifier la concordance de U au type trouvé à l'occurrence rigide correspondante du type de l'argument i. Sinon, u devient un nouvel argument qui doit être explicitement appliqué à C.

Dans les exemples qui suivent, les variables sont toujours déchargées par la commande *Gen*, qui est simplement symbolisée par un retour de tabulation, avec indication entre crochets $\langle u \rangle$ de la variable supplémentaire introduite dans le deuxième cas ci-dessus. Remarquons que si C est déclarée avec deux variables explicites x et y, par exemple $C([x : T], [y : U]) := \cdots$, on peut obtenir la construction de C appliqué à un seul argument X par $[y : U]\, C(X, y)$, ce qui montre que notre facilité d'écriture ne restreint pas la généralité du formalisme.

3 Constructions Logiques

Nous allons construire progressivement les principaux concepts logiques, d'abord intuitionistes, puis classiques. Notre présentation suit [61,16].

3.1 Logique positive

Rappelons tout d'abord que le langage possède implicitement les concepts d'implication (intuitioniste) et de quantification universelle. En particulier, $A \to B$ n'est qu'une abréviation pour $[h : A]B$.

La proposition $!A{\cdot}A \to A$, c'est à dire la réflexivité de $\to$, se prouve simplement par l'algorithme d'identité polymorphe :

A: $*$
$\quad Id := [x : A]\, x$
$\quad \vdash A \to A \qquad \langle A \rangle.$

De même, la transitivité de $\to$ se prouve par l'algorithme de composition:

A: $*$
$\quad B$: $*$
$\quad\quad C$: $*$
$\quad\quad\quad [f : A \to B]\ ;\ [g : B \to C] := [x : A]\, (g\ (f\ x))$
$\quad\quad\quad \vdash (A \to B) \to (B \to C) \to (A \to C).$

. Notre facilité de polymorphisme implicite permet exactement les notations catégoriques : l'identité Id_A pour l'objet A est obtenue comme $(Id\ A)$ (qui peut aussi s'écrire $\langle A \rangle Id$), mais la composition des flèches $f : A \to B$ et $g : B \to C$ est notée simplement $f; g$, son domaine A et son co-domaine C étant implicites à partir de ceux de f et g.

Remarquons qu'il est possible d'interpréter les propositions comme des règles d'inférence, la relation de déduction $\vdash$ ayant les propriétés de l'implication intuitioniste. Par exemple, on peut lire la définition de la composition comme une règle de coupure:

$$\frac{A \to B \quad B \to C}{A \to C}$$

A titre d'exemple, donnons les combinateurs usuels [20] :

A: $*$
  B: $*$
    $K := [x : A][y : B]\, x$
    $\vdash A \to B \to A \qquad \langle A, B \rangle$
A: $*$
  B: $*$
    C: $*$
      $S := [f : A \to B \to C][g : A \to B][x : A]\, (f\ x\ (g\ x))$
      $\vdash (A \to B \to C) \to (A \to B) \to (A \to C) \qquad \langle A, B, C \rangle.$

Ces combinateurs, combinés par application, forment les preuves du calcul propositionnel minimal, ou logique implicationnelle positive. On remarque en effet que les types de K et S correspondent aux axiomes de Hilbert.

Nous définissons le produit, c'est à dire la conjonction, comme nous l'avons détaillé plus haut:
$[A : *] \wedge [B : *] := !C \cdot (A \to B \to C) \to C.$
On introduit une conjonction par l'algorithme de paire :

A: $*$
  B: $*$
    $< [x : A], [y : B] > := !C \cdot [h : A \to B \to C]\, (h\ x\ y)$
    $\vdash A \to B \to A \wedge B.$

Notez que la preuve a été faite dans le contexte $[A : *][B : *]$. L'algorithme de paire $<,>$ a été rendu polymorphe en déchargeant A et B.

Les preuves d'élimination de " $\wedge$ " à gauche et à droite correspondent respectivement aux algorithmes π_1 et π_2 de 1ère et 2ème projection.

On donne à titre d'exemple l'algorithme de Curryfication :

A: $*$
  B: $*$
    C: $*$
      $Curry\ [h : A \wedge B \to C] := [x : A][y : B]\, (h\ < x, y >).$

Remarquez la synthèse des arguments de polymorphisme dans l'application $< x, y >$ ci-dessus.

Nous laissons au lecteur la preuve de l'implication inverse. L'isomorphisme ainsi défini prouve le théorème de déduction.

L'équivalence intuitioniste s'écrit donc:
$[A : *] \leftrightarrow [B : *] := (A \to B) \wedge (B \to A).$

On construit de même la somme, ou disjonction intuitioniste:
$[A : *] \sqcup [B : *] := !C \cdot (A \to C) \to (B \to C) \to C.$

Cette définition suit ici encore la sémantique opératoire : une preuve de $A \sqcup B$ permet de prouver toute proposition C, à partir de méthodes h_1 et h_2 permettant de prouver C à partir de respectivement A et B.

L'élimination d'une somme est l'algorithme de construction par cas:

A: $*$;
B: $*$;
C: $*$;
Si $[c : A \sqcup B]$ *alors* $[x : A \to C]$ *sinon* $[y : B \to C]$:= $(c\ C\ x\ y)$.

Les algorithmes d'injection prouvent l'introduction gauche et droite de la somme. Nous laissons leur construction au lecteur.

3.2 Quantificateurs

La quantification universelle, ou produit généralisé, est implicite dans le langage, puisqu'on peut définir :

A: $*$
$\prod([P : A \to *])$:= $[x : A]\ (P\ x)$.

L'élimination de $\prod$, c'est à dire l'instanciation, est ici simplement l'application. Son introduction est simplement l'abstraction.

La quantification existentielle, ou somme généralisée, se construit comme suit. Dans le contexte $[A : *][P : A \to *]$, la proposition $\exists x \cdot (P\ x)$ permet de prouver tout énoncé B, à partir d'une preuve que B se déduit de $(P\ x)$, pour $x : A$ quelconque :

A: $*$
$\sum([P : A \to *])$:= $!B \cdot ([x : A]\ (P\ x) \to B) \to B$.

On introduit une quantification existentielle par la construction :

A: $*$
$\exists[P : A \to *]$:= $[x : A][h : (P\ x)]\ !B \cdot [p : [x : A]\ (P\ x) \to B]\ (p\ x\ h)$
$\vdash [P : A \to *][x : A]\ (P\ x) \to \sum(P)$.

Inversement, on peut projeter la somme sur un "témoin" qui vérifie le prédicat quantifié existentiellement : .

A: $*$
$\iota\ [P : A \to *]$:= $[p : \sum(P)]\ (p\ A\ [x : A]\ (P\ x) \to x)$
$\vdash [P : A \to *]\ \sum(P) \to A$.

Dans la pratique, on s'autorisera de *Skolemiser* la preuve p en une constante C utilisée comme abréviation pour $(\iota\ P\ p)$, ce qui rendra l'écriture plus conforme à la pratique mathématique.

Remarquez qu'il n'est pas possible ici de projeter p vers la preuve de $(P\ C)$ qu'elle encapsule. Notre somme est donc différente de celle axiomatisée par Martin-Löf [49]. Ceci est par contre conforme à l'interprétation d'un quantificateur existentiel comme type abstrait [44].

3.3 Logique classique

La contradiction, ou proposition absurde, permet de prouver toute proposition A par simple application:

∇ := $!A \cdot A$.

∇ n'a pas de preuve, et joue donc logiquement le rôle de la valeur de vérité *faux*. Nier une proposition revient à exprimer qu'elle entraîne ∇, d'où le concept de négation :
$\neg\,[A : *] \;:=\; A \to \nabla$.

La connective de Sheffer $A \mid B$ (lire "A contradictoire avec B") se définit par :
$[A : *] \mid [B : *] \;:=\; A \to B \to \nabla$.
Il est facile de montrer $!A, B \cdot (A \mid B) \leftrightarrow \neg(A \wedge B)$. Les autres connectives classiques s'expriment simplement en terme de $\mid$:
$[A : *] \Rightarrow [B : *] \;:=\; A \mid \neg B$
$[A : *] \vee [B : *] \;:=\; (\neg A) \mid (\neg B)$
$[A : *] \Leftrightarrow [B : *] \;:=\; (A \Rightarrow B) \wedge (B \Rightarrow A)$.

Appelons *fermeture classique* de la proposition A sa double négation :
$[[A : *]] \;:=\; \neg(\neg A)$.

Toute proposition nie sa négation :

A: $*$
  p: A
  q: $\neg A$
    $Neg\_Neg \;:=\; (q\ p) \qquad \langle A, p, q\rangle$
$\vdash !A \cdot A \to [A]$.

L'implication inverse n'est vraie que des propositions classiques :
Classique $[A : *] \;:=\; [A] \to A$.

On peut montrer que $\nabla, \neg, \mid$ produisent des propositions classiques, et donc aussi $\vee$ et $\Rightarrow$. Finalement, $\wedge$ préserve la propriété d'être classique, et $\Leftrightarrow$ produit donc également des propositions classiques.

En fait, un raisonnement classique consiste en général à montrer qu'un ensemble de propositions $\{A_1, ..., A_n\}$ est contradictoire. Les connectives $\nabla, \neg, \mid$ expriment cette notion pour $n = 0, 1, 2$ respectivement.

Remarquons qu'il est très simple de prouver le principe du tiers exclu :
$[A : *]\ (Id\ [A]) \vdash !A \cdot (\neg A) \vee A$.

4 Une théorie intuitioniste des ensembles

Dans cette section, nous montrons comment axiomatiser dans les Constructions les concepts usuels de théorie des ensembles. On se place dans un contexte global, dans lequel on a déclaré $[U : *]$. On peut penser à U comme étant l'*univers* du discours. Les *ensembles* seront assimilés à des prédicats sur U, c'est à dire à des propositions de type $U \to *$, que nous abrégeons en *Ens*. On appelera *famille* un ensemble d'ensembles, de type $Ens \to *$, abrégé en *Fam*. De même on appelle *relation* un ensemble de paires Currifiées, de type $U \to U \to *$, abrégé *Rel*. On se place donc dans le contexte:

U: $*$
  $Ens := U \to *$
  $Fam := Ens \to *$
  $Rel := U \to U \to *$

4.1 Ensembles, Familles, Relations

Les ensembles étant assimilés à leur prédicat caractéristique, l'appartenance se définit simplement comme étant l'application :
$[x : U] \in [P : Ens] \ := \ (P\ x)$.
Notons qu'il est essentiel de distinguer entre le signe " $\in$ " et le signe " : " de la relation de typage; la notion d'ensemble apparaît comme une notion dérivée (mais distincte) de celle de type. Cette distinction fondamentale remonte aux Principia [76].

On définit dans ce cadre les notions ensemblistes usuelles :
$[P : Ens] \subseteq [Q : Ens] \ := \ [x : U]\ x \in P \rightarrow x \in Q$
$[P : Ens] = [Q : Ens] \ := \ P \subseteq Q \wedge Q \subseteq P$
$\emptyset \ := \ [x : U]\ \nabla$
$[P : Ens] \cap [Q : Ens] \ := \ [x : U]\ x \in P \wedge x \in Q$
$[P : Ens] \amalg [Q : Ens] \ := \ [x : U]\ x \in P \sqcup x \in Q$
$[P : Ens] \cup [Q : Ens] \ := \ [x : U]\ x \in P \vee x \in Q$
$\sim [P : Ens] \ := \ [x : U]\ \neg\, x \in P$.

On remarque que les quantificateurs peuvent être interprétés comme des opérations sur les ensembles : $\prod(P)$ dit que P est universel, et $\sum(P)$ dit que P n'est pas vide.

Le type des ensembles est intrinsèquement plus riche que le type de l'univers. On peut exprimer cette forme du paradoxe de Russell (ou du théorème de Cantor) en montrant qu'il n'y a pas d'injection de type $Ens \rightarrow U$. Cette preuve se construit par contradiction :

F: $Ens \rightarrow U$
  G: $U \rightarrow Ens$
    H: $[P : Ens]\ (G\ (F\ P)) = P$
      $epimenide \ := \ [u : U]\ \neg u \in (G\ u)$
      $menteur \ := \ (F\ epimenide)$
      $paradoxe \ := \ menteur \in (G\ menteur)$
      $(\pi_1(H\ epimenide)\ menteur)$
      $\vdash paradoxe \rightarrow \neg paradoxe$
      $=: negatif$
      hyp: $paradoxe$
        $(negatif\ hyp\ hyp)$
      $\vdash \neg paradoxe$
      $=: contradiction$
      $(\pi_2(H\ epimenide)\ menteur)$
      $\vdash (\neg paradoxe) \rightarrow paradoxe$
      $=: positif$
      $(contradiction\ (positif\ contradiction))$
      $\vdash \nabla$.

L'intersection d'une famille d'ensembles se définit par :
$\cap[F : Fam] \ := \ [x : U][P : Set]\ (F\ P) \rightarrow x \in P$.

On peut alors vérifier que cet ensemble est effectivement le plus grand ensemble inclus dans tous les ensembles de la famille.

On peut définir sur les relations les notions analogues à $\subseteq$, $=$, $\emptyset$, $\cap$, $\cup$, $\sim$. Par convention, nous indexerons ces notions avec 2, comme par exemple $\cap_2$. De même, on peut considérer des familles de relations (de type $Fam_2 = Rel \to *$), et définir l'intersection ($\cap_2$) d'une telle famille.

Enfin, on peut introduire les notions usuelles associées aux relations :

$Reflexive\ [R : Rel] := [x : U]\ (R\ x\ x)$

$Symetrique\ [R : Rel] := [x : U][y : U]\ (R\ x\ y) \to (R\ y\ x)$

$Transitive\ [R : Rel] := [x : U][y : U][z : U]\ (R\ x\ y) \to (R\ y\ z) \to (R\ x\ z).$

4.2 Egalités intentionnelle et extensionnelle

L'égalité définie ci-dessus est l'égalité extensionnelle considéré traditionnellement en théorie des ensembles : deux ensembles sont égaux ssi ils ont les mêmes éléments. Il est possible également de définir une égalité intentionnelle, pour tout type. Suivant Leibniz, on dit que x et y sont identiques ssi ils admettent les mêmes propriétés (tout ensemble contenant x contient y) :

$$[x : U] \doteq [y : U] := [P : Ens]\ (x \in P) \to (y \in P).$$

La propriété de substitutivité est implicite dans la définition de $\doteq$. Montrons que $\doteq$ est bien une équivalence.

La réflexivité de $\doteq$ se montre par l'algorithme d'identité :

$Refl_{\doteq} := [x : U][P : Ens]\ (Id\ (x \in P))$

$\vdash Reflexive\ \doteq.$

De même la transitivité de montre par composition :

$Trans_{\doteq} := [x : U][y : U][z : U][p : x \doteq y][q : y \doteq z][P : Ens]\ (p\ P); (q\ P)$

$\vdash Transitive\ \doteq.$

Par contre, la symétrie est moins triviale, compte tenu du fait que notre implication est intuitioniste :

x: U
  y: U
    p: $x \doteq y$
      P: Ens
        $Q = [z : U]\ (z \in P) \to (x \in P)$
        $(p\ Q\ (Id\ (x \in P)))$
        $\vdash (y \in P) \to (x \in P)$

$\vdash Symetrique\ \doteq.$

4.3 Opérations de fermeture

Nous formalisons la notion de fermeture d'une relation R par rapport à une propriété comme étant l'intersection de la famille des relations possédant la propriété et contenant R.

A titre d'exemple, nous définissons la fermeture transitive:

$[R : Rel]^+ := [u : U][v : U][S : Rel]\ (Transitive\ S) \to (R \subseteq_2 S) \to (S\ u\ v).$

Plus généralement, définissons:

$Fermeture\ [P : Fam_2][R : Rel] := [u : U][v : U][S : Rel]\ (P\ S) \to (R \subseteq_2 S) \to (S\ u\ v).$

Lorsque P est une conjonction, on Curryfie la définition, comme par exemple pour la fermeture transitive et réflexive :

$[R : Rel]^* := [u : U][v : U][S : Rel]\ (Transitive\ S) \to (Reflexive\ S) \to (R \subseteq_2 S) \to (S\ u\ v).$

On dit qu'une propriété P est "stable" ssi elle est stable par intersection :
$Stable\ [P : Fam_2]\ :=\ [Q : Fam_2]\ ([R : Rel]\ (Q\ R) \to (P\ R)) \to (P\ (\cap_2 Q))$
et on montre:
$(Stable\ P) \to (P\ (Fermeture\ P\ R))$.

Comme cas particulier, on peut montrer que la propriété transitive est stable, et en déduire que R^+ est transitive. De même, on montre par stabilité que R^+ contient R.

La prochaine section illustre les notions précédentes sur un exemple élémentaire en théorie des relations.

4.4 Lemme de Newman

Ce lemme, qui exprime que la confluence d'une relation peut être réduite à une vérification locale si cette relation est Nœthérienne, est fondamental pour l'étude des systèmes de simplification [55,40,36]. Il illustre bien l'utilisation de la logique d'ordre supérieur. Définissons tout d'abord les notions nécessaires à l'énoncé.

On se place dans le contexte d'une relation R fixée :
$R : Rel.$

Deux éléments sont dits *cohérents* s'ils admettent un majorant commun :
$Coherence\ [x : U][y : U]\ := !A.([z : U]\ (R^*\ x\ z) \to (R^*\ y\ z) \to A) \to A.$
Remarquons que cette définition n'est que la forme séquentialisée de la définition équivalente :

$$\sum (R^*\ x) \cap (R^*\ y).$$

La notion de *confluence* exprime une forme de déterminisme de R :
$Confluence\ :=\ [x : U][u : U][v : U]\ (R^*\ x\ u) \to (R^*\ x\ v) \to (Coherence\ u\ v).$

La confluence locale restreint cette propriété aux successeurs immédiats de x :
$Confluence\ locale\ :=\ [x : U][u : U][v : U]\ (R\ x\ u) \to (R\ x\ v) \to (Coherence\ u\ v).$

Une relation est *Nœthérienne* ssi elle n'admet pas de chaîne infinie. La formalisation de ce concept en tant que construction passe par son énoncé sous la forme du principe de récurrence Nœthérienne [13] :

$Noetherien\ :=\ [P : Ens]\ ([u : U]\ ([v : U](\ R\ u\ v) \to v \in P) \to u \in P) \to [u : U]\ u \in P.$

Le lemme de Newman s'énonce alors :
$Newman\ :=\ Noetherien \to Confluence\ locale \to Confluence.$

Il se prouve par récurrence Nœthérienne sur la propriété : $(Confluence\ x)$.

5 Constructions informatiques

Nous allons montrer dans ce chapitre que le Calcul des Constructions est bien adapté à formaliser certains concepts informatiques.

5.1 Algèbre universelle et types de données

Commençons par montrer comment formaliser les notions élémentaires d'Algèbre, et en particulier la notion d'algèbre libre sur une signature. On se place d'abord dans le cas homogène, c'est à dire dans le contexte :
$A : *$

Pour tout $n \geq 0$, on définit le *A-cardinal* $\overline{n}$ associé à n par récurrence :

$$\overline{0} = A$$

$$\overline{n+1} = \overline{n} \to A.$$

On définit maintenant la *fonctionnalité* $\varphi(\Sigma)$ associée à une signature Σ représentée sous la forme d'une liste d'opérateurs donnés avec leur arité par :

$$\varphi(\emptyset) = A$$

$$\varphi([F : n]\ \Sigma) = [F : \overline{n}]\ \varphi(\Sigma).$$

Dans la dernière clause, on aurait pu bien sûr écrire $\overline{n} \to \varphi(\Sigma)$. Toutes ces définitions sont aisément programmables dans le meta-langage des constructions.

On obtient maintenant l'*algèbre initiale* associée à la signature Σ en déchargeant A, c'est à dire en faisant abstraction sur le type support de l'algèbre:
$I(\Sigma) = !A \cdot \varphi(\Sigma)$.

Plaçons nous maintenant dans une Σ-algèbre arbitraire, c'est à dire dans le contexte Γ :

A: $*$
F_1: $\overline{n_1}$
...
F_s: $\overline{n_s}$

Si $M : I(\Sigma)$ est une construction arbitraire d'un élément de la Σ-algèbre initiale, on appelle *image* de M dans la Σ-algèbre Γ le terme $\widehat{M} = (M\ A\ F_1\ \cdots\ F_s)$. On remarque que ce terme est bien construit, et de type A. Cette notion d'image correspond, de manière classique, à prendre l'image de M par le Σ-morphisme unique de $I(\Sigma) \to \Gamma$. Par exemple, si on a $M_1 : I(\Sigma), M_2 : I(\Sigma), ..., M_{n_k} : I(\Sigma)$, alors $(F_k\ \widehat{M_1}\ \cdots\ \widehat{M_{n_k}}) : A$. On définit donc ainsi un F_k opérateur dans $I(\Sigma)$, que l'on appelle F_k-constructeur, et qui s'obtient en faisant abstraction de Γ, et d'une liste de n_k variables de type $I(\Sigma)$.

Donnons maintenant quelques exemples. Si $\Sigma = \emptyset$, on obtient $I(\Sigma) = \nabla$, l'algèbre vide. Si $\Sigma = [i : 0]$, on obtient $I(\Sigma) = Un := !A \cdot A \to A$, et le i-constructeur est $Id = [A : *][i : A]\ i$.

Avec $\Sigma = [v : 0][f : 0]$, on obtient $I(\Sigma) = Bool := !A \cdot A \to A \to A$, et les deux constructeurs sont les Booléens de Church [12] :
$Vrai := [A : *][v : A][f : A]\ v$
$Faux := [A : *][v : A][f : A]\ f$.

Lorsque $\Sigma = [s : 1][z : 0]$, on obtient $I(\Sigma) = Nat$, les entiers de Church :
$Nat := !A \cdot (A \to A) \to A \to A$
$S\ ([n : Nat]) := [A : *][s : A \to A][z : A]\ (s\ (n\ A\ s\ z))$
$0 := [A : *][s : A \to A][z : A]\ z$.

Lorsque $\Sigma = [c : 2][n : 0]$, on obtient $I(\Sigma) = Bin$, les arbres binaires :
$Bin := !A \cdot (A \to A \to A) \to A \to A$
$Cons\ [a_1 : Bin][a_2 : Bin] := [A : *][c : A \to A \to A][n : A]\ (c\ (a_1\ A\ c\ n)\ (a_2\ A\ c\ n))$
$Nul := [A : *][c : A \to A \to A][n : A]\ n$.

Il est facile de généraliser ces notions au cas non-homogène, en introduisant autant de sortes que nécessaire. Par exemple, la structure de liste s'axiomatise sur deux sortes A et B :
$Liste := !A, B \cdot (A \to B \to B) \to B \to B$.

Ici, l'opération d'ajouter un élément à une liste est doublement polymorphe, et nous pouvons aisément synthétiser ce polymorphisme :

A: $*$
B: $*$
$Ajout\ [x : A][L : (Liste\ A\ B)] := [J : A \to B \to B][V : B]\ (J\ x\ (L\ J\ V))$
$\vdash !A, B \cdot A \to (Liste\ A\ B) \to (Liste\ A\ B)$.

On pourrait également définir un opérateur unaire plus général *Ajouter* qui ajoute à des listes polymorphes :

A: $*$
$Ajouter\ [x : A] := [L : (Liste\ A)]\ !B.[J : A \to B \to B][V : B]\ (J\ x\ (L\ J\ V))$
$\vdash !A \cdot A \to (Liste\ A) \to (Liste\ A)$.

Remarquons que la liste vide est simplement de type *Liste* :
$Vide := [A : *][B : *][J : A \to B \to B][V : B]\ V$
$\vdash Liste$.

D'une manière générale, on peut aisément définir toutes les structures de données correspondant à des algèbres libres. On remarque que les propositions correspondantes sont exactement les types de Girard restreints au degré 2. Notre traitement est similaire à celui de Böhm et Berarducci [7].

Donnons quelques exemples de programmes sur les entiers. L'addition s'obtient par itération de successeur :
$Succ := [n : Nat]\ S(n)$
$[m : Nat] + [n : Nat] := (n\ Nat\ Succ\ m)$.

D'autres définitions sont possibles. La multiplication s'obtient par itération de l'addition :
$Add := [m : Nat][n : Nat]\ m + n$
$[m : Nat] * [n : Nat] := (n\ Nat\ (Add\ m)\ 0)$.

On peut également voir les entiers comme des itérateurs polymorphes. La multiplication de m et n aurait ainsi pu être définie comme la composition $m; n$.

L'exponentiation s'obtient par itération de la multiplication :
$Mult := [m : Nat][n : Nat]\ (m * n)$
$[m : Nat] \uparrow [n : Nat] := (n\ Nat\ (Mult\ m)\ S(0))$.

L'itération d'un entier sur un type fonctionnel peut produire des fonctions non primitives récursives, telle la fonction d'Ackermann :
$Ack\ [n : Nat] := (n\ (Nat \to Nat)\ ([f : Nat \to Nat][m : Nat]\ (m\ Nat\ f\ m))\ Succ)$.

5.2 Ordinaux

Tous les types d'algèbres ci-dessus sont d'une forme très simple, l'emboîtement des flèches étant limité. Très exactement, ils sont de degré 2, avec le degré δ défini par :

- $\delta(A) = 0$ $\quad$ *A variable*
- $\delta([F : T]\ X) = max\{1 + \delta(T), \delta(X)\}$.

Pour des types plus compliqués, on obtient des structures de données plus riches. Par exemple, les ordinaux peuvent être construits comme extension des entiers, en ajoutant une opération limite, qui associe une ordinal à toute suite d'ordinaux, représentée par une fonction de domaine Nat. On

définit donc :
$Ord := !A \cdot ((Nat \to A) \to A) \to (A \to A) \to A \to A$
$Lim\ [\sigma : Nat \to Ord] := [A : *][li : (Nat \to A) \to A][s : A \to A][z : A]\ (li\ [n : Nat](\sigma\ n\ A\ li\ s\ z))$
$Suc\ ([\alpha : Ord]) := [A : *][li : (Nat \to A) \to A][s : A \to A][z : A]\ (s\ (\alpha\ A\ s\ z))$
$Z := [A : *][li : (Nat \to A) \to A][s : A \to A][z : A]\ z.$

Il est facile de traduire un entier en l'ordinal correspondant, ce qui définit l'ensemble des ordinaux finis :
$Finis := [n : Nat]\ (n\ Ord\ ([\alpha : Ord]\ Suc(\alpha))\ Z).$

Remarquons l'instanciation de l'argument de polymorphisme de l'entier n en le type Ord. En effet, la signification de $*$ est de désigner un type (une proposition) arbitraire, et non pas seulement un type appartenant à une totalité circonscrite à la construction contenant cette occurrence de $*$. Autrement dit, nous utilisons de manière essentielle la non-prédicativité du calcul.

Le premier ordinal infini, ω, s'obtient comme limite des ordinaux finis :
$\omega := (Lim\ Finis).$

On programme sur les ordinaux de façon similaire aux entiers :
$[\alpha : Ord] \oplus [\beta : Ord] := (\beta\ Ord\ Lim\ ([\gamma : Ord]\ Suc(\gamma))\ \alpha)$
$[\alpha : Ord] \otimes [\beta : Ord] := (\beta\ Ord\ Lim\ ([\gamma : Ord]\ \alpha \oplus \gamma)\ Z)$
$[\alpha : Ord] \dagger [\beta : Ord] := (\beta\ Ord\ Lim\ ([\gamma : Ord]\ \alpha \otimes \gamma)\ Suc(Z)).$

On remarque que nos ordinaux sont en fait des *notations ordinales*, c'est à dire des ordinaux présentés avec des séquences fondamentales. En particulier, $1 \oplus \omega$ et ω sont des constructions distinctes.

On obtient ϵ_0 par itération de $\omega \dagger \omega \dagger \cdots$:
$\epsilon_0 := (\omega\ Ord\ Lim\ ([\gamma : Ord]\ \omega \dagger \gamma)\ Z).$

On peut maintenant utiliser les ordinaux pour décrire des hiérarchies fonctionnelles. Donnons quelques définitions préliminaires sur les fonctions entières :
$Incr := [f : Nat \to Nat][n : Nat]\ (S\ (f\ n))$
$Iter := [f : Nat \to Nat][n : Nat]\ (n\ Nat\ f\ n)$
$Diag := [\sigma : Nat \to Nat \to Nat][n : Nat]\ (\sigma\ n\ n).$

On définit maintenant la hiérarchie rapide de Schwichtenberg par:
$Rapide\ [\alpha : Ord] := (\alpha\ (Nat \to Nat)\ Diag\ Iter\ Succ)$
et la hiérarchie lente ne s'en distingue que par la fonctionnelle successeur :
$Lente\ [\alpha : Ord] := (\alpha\ (Nat \to Nat)\ Diag\ Incr\ Succ).$

Notons que $(Rapide\ \epsilon_0)$ est une fonction récursive totale, mais dont la terminaison est indépendante de l'arithmétique de Peano [22,41].

6 Conclusion

Le Calcul des Constructions est une formalisation de la déduction naturelle d'ordre supérieur, bien adapté au traitement par machine. Il fait la synthèse entre la théorie des types de Martin-Löf de 1971 et le système F de Girard, au sein d'une généralisation naturelle du langage A d'Automath. Nous avons tenté de montrer par des exemples que ce calcul est bien adapté à exprimer de manière

concise des notions mathématiques et informatiques.

Cette théorie a été développée en vue de son application à la conception d'environnements de programmation fondés sur des principes logiques permettant d'assurer le développement des programmes de manière compatible avec leur spécification. Le Calcul des Constructions permet de représenter de manière uniforme les types de données et des propositions logiques complexes. La cohérence des programmes avec leur spécifications peut être vérifiée mécaniquement. Nous proposons une méthodologie de développement de logiciel où le programmeur construit ses algorithmes avec l'assistance interactive du système de vérification de types. Dans certains cas, il sera même possible de synthétiser partiellement le programme.

De nombreux problèmes sont soulevés par cette approche. Par exemple, le formalisme est suffisamment puissant pour être considéré comme un langage de programmation de très haut niveau. Mais l'absence de récursion générale oblige à repenser l'activité de programmation. De nouvelles méthodologies, une nouvelle "discipline" de la programmation restent à définir. D'autre part, le langage machine suggéré par cette approche est le λ-calcul pur. Malgré la simplicité et l'uniformité de ce mécanisme de calcul, on ne connaît pas encore d'architecture de machine permettant son éxécution efficace. Il reste toutefois envisageable d'étendre le formalisme par l'ajout de constantes implémentées directement par les opérations d'une machine traditionnelle. Par exemple, on pourrait ajouter des entiers primitifs, et un opérateur de récursion. Le Calcul des Constructions devient alors la description d'un vérificateur de types extrêmement général pour les langages de programmation à venir [42].

References

[1] P. B. Andrews, Dale A. Miller, Eve Longini Cohen, Frank Pfenning "Automating higher-order logic." Dept of Math, University Carnegie-Mellon, (Jan. 1983).

[2] P. B. Andrews "Resolution in Type Theory." Journal of Symbolic Logic **36,3** (1971), 414–432.

[3] R. Backhouse "Algorithm development in Martin-Löf's type theory." University of Essex (July 1984).

[4] H. Barendregt "The Lambda-Calculus: Its Syntax and Semantics." North-Holland (1980).

[5] E. Bishop "Foundations of Constructive Analysis." (1967) McGraw-Hill, New-York.

[6] E. Bishop "Mathematics as a numerical language." Intuitionism and Proof Theory, Eds. J. Myhill, A.Kino and R.E.Vesley, North-Holland, Amsterdam, (1970) 53–71.

[7] C. Böhm, A. Berarducci "Automatic Synthesis of Typed Lambda–Programs on Term Algebras." Unpublished manuscript, (June 1984).

[8] N. G. de Bruijn "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29–61.

[9] N. G. de Bruijn "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem." Indag. Math. **34,5** (1972), 381–392.

[10] N. G. de Bruijn "Automath a language for mathematics." Les Presses de l'Université de Montréal, (1973).

[11] N. G. de Bruijn "A survey of the project Automath." (1980) in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[12] A. Church "The Calculi of Lambda-Conversion." Princeton U. Press, Princeton N.J. (1941).

[13] P. M. Cohn. "Universal Algebra." Reidel, 1965.

[14] R. L. Constable, J. L. Bates "Proofs as Programs." Dept. of Computer Science, Cornell University. (Feb. 1983).

[15] R. L. Constable, J. L. Bates "The Nearly Ultimate Pearl." Dept. of Computer Science, Cornell University. (Dec. 1983).

[16] Th. Coquand "Une théorie des constructions." Thèse de troisième cycle, Université Paris VII (Jan. 85).

[17] Th. Coquand, G. Huet "A Theory of Constructions." Preliminary version, presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis (June 84).

[18] Th. Coquand, G. Huet "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Austria (April 85).

[19] Th. Coquand, G. Huet "A Calculus of Constructions." Inria (Juin 85).

[20] H. B. Curry, R. Feys "Combinatory Logic Vol. I." North-Holland, Amsterdam (1958).

[21] D. Van Daalen "The language theory of Automath." Ph. D. Dissertation, Technological Univ. Eindhoven (1980).

[22] S. Fortune, D. Leivant, M. O'Donnell "The Expressiveness of Simple and Second-Order Type Structures." Journal of the Assoc. for Comp. Mach., **30,1**, (Jan. 1983) 151–185.

[23] G. Frege "Begriffschrift, a formula language, modeled upon that of arithmetic, for pure thought." (1879). Reprinted in From Frege to Gödel, J. van Heijenoort, Harvard University Press, 1967.

[24] G. Gentzen "The Collected Papers of Gerhard Gentzen." Ed. E. Szabo, North-Holland, Amsterdam (1969).

[25] J. Y. Girard "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. Proceedings of the Second Scandinavian Logic Symposium, Ed. J. E. Fenstad, North Holland (1970) 63–92.

[26] J. Y. Girard "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure." Thèse d'Etat, Université Paris VII (1972).

[27] K. Gödel "Uber eine bisher noch nicht benutze Erweitrung des finiten Standpunktes." Dialectica, **12** (1958).

[28] M. Gordon, R. Milner, C. Wadsworth "A Metalanguage for Interactive Proof in LCF." Internal Report CSR-16-77, Department of Computer Science, University of Edinburgh (Sept. 1977).

[29] M. J. Gordon, A. J. Milner, C. P. Wadsworth "Edinburgh LCF" Springer-Verlag LNCS **78** (1979).

[30] J. Herbrand "Recherches sur la théorie de la démonstration." Thèse, U. de Paris (1930). Ecrits logiques de Jacques Herbrand, PUF Paris (1968).

[31] W. A. Howard "The formulæ-as-types notion of construction." Unpublished manuscript (1969). Reprinted in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[32] G. Huet "A Unification Algorithm for Typed Lambda Calculus." Theoretical Computer Science, **1.1** (1975) 27–57.

[33] G. Huet "Constrained Resolution: a Complete Method for Type Theory." Ph.D. Thesis, Case Western Reserve University (1972).

[34] G. Huet "A Mechanization of Type Theory." Proceedings, 3rd IJCAI, Stanford (Aug. 1973).

[35] G. Huet "Résolution d'équations dans des langages d'ordre 1,2, ... ω." Thèse d'Etat, Université Paris VII (1976).

[36] G. Huet "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems." J. Assoc. Comp. Mach. **27,4** (1980) 797–821.

[37] L. S. Jutting "The language theory of Λ, a λ-calculus where terms are types." Unpublished manuscript (1984).

[38] L. S. Jutting "A translation of Landau's "Grundlagen" in AUTOMATH." Eindhoven University of Technology, Dept of Mathematics (Oct. 1976).

[39] S. C. Kleene "Introduction to Meta-mathematics." North Holland (1952).

[40] D. Knuth, P. Bendix "Simple word problems in universal algebras", dans : Computational Problems in Abstract Algebra, J. Leech Ed., Pergamon (1970) 263–297.

[41] G. Kreisel "On the interpretation of nonfinitist proofs, Part I, II." JSL **16** (1952, 1953).

[42] P. J. Landin "The next 700 programming languages." Comm. ACM **9** (1966) 157–166.

[43] D. Leivant "Polymorphic type inference." 10th ACM Conference on Principles of Programming Languages (1983).

[44] D. MacQueen, G. Plotkin, R. Sethi. "An ideal model for recursive polymorphic types." Proceedings, Principles of Programming Languages Symposium, Jan. 1984, 165–174.

[45] P. Martin-Löf "A theory of types." Unpublished manuscript (Oct. 1971).

[46] P. Martin-Löf "An intuitionistic Theory of Types: predicative part." Logic Colloquium 73, Eds. H. Rose and J. Sepherdson, North-Holland, (1974) 73–118.

[47] P. Martin-Löf "About models for intuitionistic type theories and the notion of definitional equality." Paper read at the Orléans Logic Conference (1972).

[48] P. Martin-Löf "Constructive Mathematics and Computer Programming." Logic, Methodology and Philosophy of Science VI, 153–175, (1980) North-Holland.

[49] P. Martin-Löf "Intuitionistic Type Theory." Studies in Proof Theory, Bibliopolis (1984).

[50] D. A. Miller "Proofs in Higher-order Logic." Ph. D. Dissertation, Carnegie-Mellon University (Aug. 1983).

[51] R. Milner "A Theory of Type Polymorphism in Programming." Journal of Computer and System Sciences **17** (1978) 348–375.

[52] C. Mohring "Exemples de Développement de Programmes dans la Théorie des Constructions." Rapport de DEA, Université d'Orsay (Sept 85).

[53] R. P. Nederpelt "Strong normalization in a typed λ calculus with λ structured types." Ph. D. Thesis, Eindhoven University of Technology (1973).

[54] R. P. Nederpelt "An approach to theorem proving on the basis of a typed λ-calculus." 5th Conference on Automated Deduction, Les Arcs, France. Springer-Verlag LNCS **87** (1980).

[55] M. H. A. Newman "On Theories with a Combinatorial Definition of "Equivalence"." Annals of Math. **43,2** (1942) 223–243.

[56] B. Nordström "Programming in Constructive Set Theory: Some Examples." Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Portmouth, New Hampshire (Oct. 1981) 141–154.

[57] B. Nordström, K. Petersson "Types and Specifications." Information Processing 83, Ed. R. Mason, North-Holland, (1983) 915–920.

[58] B. Nordström, J. Smith "Propositions and Specifications of Programs in Martin-Löf's Type Theory." BIT **24**, (1984) 288–301.

[59] T. Pietrzykowski, D. C. Jensen "A complete mechanization of ω-order type theory." Proceedings of ACM Annual Conference (1972).

[60] T. Pietrzykowski "A Complete Mechanization of Second-Order Type Theory." JACM **20** (1973) 333–364.

[61] D. Prawitz "Natural Deduction." Almqist and Wiskell, Stockolm (1965).

[62] D. Prawitz "Ideas and results in proof theory." Proceedings of the Second Scandinavian Logic Symposium (1971).

[63] J. C. Reynolds "Definitional Interpreters for Higher Order Programming Languages." Proc. ACM National Conference, Boston, (Aug. 72) 717–740.

[64] J. C. Reynolds "Towards a Theory of Type Structure." Programming Symposium, Paris. Springer Verlag LNCS **19** (1974) 408–425.

[65] J. C. Reynolds "Polymorphism is not set-theoretic." International Symposium on Semantics of Data Types, Sophia-Antipolis (June 1984).

[66] J. C. Reynolds "Three approaches to type structure." TAPSOFT Advanced Seminar on the Role of Semantics in Software Development, Berlin (March 1985).

[67] J. C. Reynolds "Types, abstraction, and parametric polymorphism." IFIP Congress'83, Paris (Sept. 1983).

[68] D. Scott "Constructive validity." Symposium on Automatic Demonstration, Springer-Verlag Lecture Notes in Mathematics, **125** (1970).

[69] J. R. Shoenfield "Mathematical Logic." Addison-Wesley (1967).

[70] J. Smith "Course-of-values recursion on lists in intuitionistic type theory." Unpublished notes, Göteborg University (Sept. 1981).

[71] J. Smith "The identification of propositions and types in Martin-Lof's type theory : a programming example." International Conference on Foundations of Computation Theory, Borgholm, Sweden, (Aug. 1983) Springer-Verlag LNCS **158**.

[72] S. Stenlund "Combinators λ-terms, and proof theory." Reidel (1972).

[73] W. Tait "A non constructive proof of Gentzen's Hauptsatz for second order predicate logic." Bull. Amer. Math. Soc. **72** (1966).

[74] M. Takahashi "A proof of cut-elimination theorem in simple type theory." J. Math. Soc. Japan **19** (1967).

[75] G. Takeuti "On a generalized logic calculus." Japan J. Math. **23** (1953).

[76] A. N. Whitehead, B. Russell "Principia Mathematica." Cambridge University Press (1903).

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique