Gilles Dowek

Théories des types

Septembre 1997

Gilles Dowek
INRIA-Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Gilles.Dowek@inria.fr
http://pauillac.inria.fr/~dowek

Le langage mathématique et les langages de programmation[1]

Récemment, la constitution d'une liste des langages de programmation a permis d'en dénombrer plus de deux mille. Plus de deux mille langages conçus en une quarantaine d'années (depuis l'apparition de Fortran, proposé en 1954 par John Backus et son équipe), cela fait, en moyenne, plus d'un langage par semaine. Bien sûr, tous ces langages ne peuvent pas être mis sur le même plan : certains sont morts depuis longtemps, d'autres sont réservés à des applications très pointues, néanmoins l'impression que donne l'informatique est souvent celle d'une tour de Babel peu accueillante.

Cette prolifération coûte cher à l'industrie du logiciel : elle rend difficile la conception de systèmes intégrant des modules déjà développés dans des langages différents, elle demande la conception d'interfaces, passerelles et autres traducteurs et fait de la maintenance de ces systèmes hybrides un véritable casse-tête. Pour mettre fin à cette cacophonie, le ministère de la défense américain a lancé en 1978 un concours afin de choisir un langage de programmation unique et de l'imposer comme langage de développement pour tous ses logiciels. Avec cette idée d'un *esperanto* de la programmation, le ministère de la défense américain renouait avec le rêve séculaire d'une langue artificielle, parfaite et universelle, langue que le théologien Raymond Lulle avait déjà tenté de construire au treizième siècle. Ce concours (remporté l'année suivante par une équipe dirigée par Jean Ichbiah) a permis la création du langage Ada, qui est aujourd'hui assez populaire, mais qui a échoué dans sa mission d'universalité.

Pourtant, limité au domaine restreint de la programmation, ce rêve d'un langage universel n'est peut-être pas tout à fait utopique. La grande majorité des musiciens utilise le même langage pour écrire ses partitions. De même, les mathématiciens de tous les pays utilisent un langage commun, ce qui n'empêche pas ce langage d'évoluer en fonction des besoins, ni chacun d'utiliser un style qui lui est propre. Si dans le domaine de la programmation nous n'avons pas encore atteint cette unité, c'est peut-être le signe que nous n'avons pas encore complètement compris ce que sont un programme ou un langage de programmation.

Qu'est-ce qu'un programme ?

Les manuels d'initiation à la programmation commencent souvent par expliquer qu'un ordinateur est une machine capable de traiter l'information de manières très diverses (contrairement à la machine de Pascal, tout juste bonne à effectuer immuablement des additions et des soustractions) mais qu'il faut d'abord expliquer à l'ordinateur ce qu'il doit faire et comment il doit le faire. Cette explication, qui s'appelle un *programme*, est exprimée dans un langage spécial, un *langage de programmation*. Par exemple, quand on demande à une banque un prêt pour acheter une voiture, celle-ci utilise un programme pour calculer le montant des mensualités de ce prêt en fonction de la somme empruntée, du nombre de mensualités et du taux d'intérêt.

[1] Ce chapitre introductif est issu d'un exposé au colloque *Voir, Entendre, Raisonner, Calculer*, Cité des sciences et de l'industrie, La Villette, Paris, 10-13 Juin 1997.

Ce programme utilise la formule bien connue

$$f(e,n,t) = \frac{e\ t}{1 - \frac{1}{(1+t)^n}}$$

où $f(e,n,t)$ est le montant des mensualités, e la somme empruntée, n le nombre de mensualités et t le taux d'intérêt mensuel.

Un programme est donc quelque chose qui permet d'associer une grandeur (la valeur de sortie) à une ou plusieurs grandeurs (les valeurs d'entrée). Un tel objet qui permet d'associer une grandeur à d'autres grandeurs est ce que les mathématiciens appellent une *fonction*. Le programme qu'utilise la banque n'est donc peut-être rien de plus que la fonction f qui associe le nombre $\frac{e\ t}{1-\frac{1}{(1+t)^n}}$ aux nombres e, n et t. De ce point de vue, un langage de programmation n'est rien de plus qu'un langage de définition de fonctions.

La nécessité même de concevoir des langages de programmation se trouve alors mise en question : s'il ne s'agit que de définir des fonctions pourquoi créer un nouveau langage, puisque le langage mathématique ordinaire convient très bien pour cela depuis plusieurs siècles ?

Les fonctions dans le langage mathématique

$$f = x \mapsto x \times x$$

C'est ainsi que les mathématiciens définissent la fonction qui à un nombre associe son carré : le terme $x \times x$ indique la valeur associée par la fonction à la valeur x. La valeur prise par la fonction f en 3, par exemple, est obtenue en remplaçant l'argument formel x par l'argument réel 3 dans ce terme, ce qui donne 3×3.

Cette notation existe presque littéralement dans la plupart des langages de programmation. Par exemple, dans le langage Pascal conçu en 1970 par Niklaus Wirth et son équipe, on définit cette même fonction par le texte

```
function f (x:integer) : integer;
begin
   f := x * x
end;
```

Hélas, les fonctions qu'on peut ainsi définir explicitement forment une toute petite partie des fonctions dont on a besoin en mathématiques ou en informatique. Par exemple, la fonction *puissance* qui associe le nombre x^n aux nombres x et n n'est pas définissable explicitement. Sur les bancs de l'école, nous avons appris que cette fonction avait la curieuse définition

$$puissance = x, n \mapsto \underbrace{x \times x \times ... \times x}_{n \text{ fois}}$$

Ces mêmes points de suspension ont été utilisés un peu plus tard, quand nous avons appris que la définition de la fonction *sinus* était

$$sinus = x \mapsto x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

Les points de suspension qui sont utilisés dans ces deux définitions n'ont pas tout à fait la même signification : dans celle de la fonction *sinus*, ils expriment une définition comme limite d'une série alors que dans celle de la fonction *puissance*, ils expriment une définition par récurrence. Une définition plus rigoureuse de la fonction *puissance* utilise donc explicitement la récurrence :

$$puissance(x, 0) = 1$$

$$puissance(x, n+1) = x \times puissance(x, n)$$

- On donne les propriétés que la fonction doit vérifier:

$$\forall x \ f(x) \geq 0$$

$$\forall x \ f(x) \times f(x) = x$$

- On montre qu'une fonction unique vérifie ces propriétés.
- On attribue le nom $\sqrt{\ }$ à la fonction.

La définition de la racine carrée

Autrement dit, la fonction *puissance* est définie comme l'unique fonction vérifiant ces deux équations.

Pourtant, il ne suffit pas de donner un système d'équations pour définir une fonction. Ainsi l'équation

$$f(x) = f(x)$$

n'est pas une définition correcte car elle est vérifiée par de nombreuses fonctions. De même l'équation

$$f(x) = 1 + f(x)$$

n'est pas non plus une définition correcte car elle n'est vérifiée par aucune fonction. La définition de la fonction *puissance* n'est donc correcte que parce qu'on peut démontrer qu'il existe une fonction unique vérifiant ces équations.

On comprend mieux alors le mécanisme véritablement utilisé pour définir des fonctions en mathématiques. On commence par donner un système d'équations, ou plus généralement une propriété, que doit vérifier la fonction, on montre ensuite qu'il existe une fonction unique vérifiant cette propriété, et on donne enfin un nom à cette fonction.

Pour attribuer un nom à la fonction on utilise en général une phrase de la forme "on appellera *puissance* l'unique fonction vérifiant les propriétés ..." Hormis ce nom, il n'y a pas d'expression désignant la fonction en question. D'un point de vue plus formel (qui est celui de ceux qui cherchent à concevoir des langages de programmation) on peut noter $[f \mid P(f)]$ l'unique objet vérifiant la propriété P de la même manière qu'on note $\{f \mid P(f)\}$ l'ensemble des objets vérifiant cette propriété. Cette notation qui permet d'exprimer un objet en en donnant une description est appelée *opérateur de descriptions*.

La fonction *puissance* se définit donc à présent ainsi

$$\mathit{puissance} = [f \mid \forall x \ f(x,0) = 1 \text{ et } \forall x \ \forall n \ f(x, n+1) = x \times f(x,n)]$$

De même que la grammaire du langage mathématique interdit de former l'expression $\frac{1}{0}$ qui n'a pas de sens, elle interdit l'utilisation de l'opérateur de descriptions avec une propriété qui n'est vérifiée par aucun objet. Il est, par exemple, impossible de former l'expression $[f \mid \forall x \ f(x) = 1 + f(x)]$. (En fait, il n'est pas essentiel que ces expressions dénuées de sens soient prohibées par la grammaire. On peut aussi adopter la convention selon laquelle l'expression $\frac{1}{0}$ est correcte et désigne un objet mathématique quelconque : l'ensemble vide, le nombre 0, ... ce qui est important est la propriété $x \times \frac{1}{x} = 1$ soit vraie uniquement quand x est différent de 0).

Quand plusieurs objets vérifient la propriété utilisée, différentes conventions peuvent être adoptées. L'utilisation de l'opérateur de descriptions est en général prohibée dans ce cas, mais une règle plus libérale autorise cette utilisation et permet, par exemple, la formation de l'expression $[x \mid x \times x = 9]$ qu'on ne lit plus dans ce cas "le nombre x tel que $x \times x = 9$" mais plutôt "un nombre x tel que $x \times x = 9$". L'opérateur de descriptions est alors appelé *opérateur de choix*, *opérateur ε de Hilbert* ou *opérateur τ de Bourbaki*. Le nombre

$[x \mid x \times x = 9]$ vaut ou bien 3 ou bien -3, sans qu'on sache s'il vaut 3 ou -3. Ainsi, on ne peut pas démontrer $[x \mid x \times x = 9] = 3$ ni $[x \mid x \times x = 9] = -3$ mais on peut démontrer $[x \mid x \times x = 9] \in \{3, -3\}$.

Les spécifications et les programmes

Quand un client commande un programme à un informaticien, il doit lui expliquer ce que doit faire ce programme. Cette explication s'appelle le *cahier des charges* ou encore la *spécification* du programme. Par exemple, quand un industriel commande à une société de service un programme d'inversion de matrices, la spécification tient en une ligne

$$f(M) \times M = I$$

alors que le programme peut être beaucoup plus long.

Si on adopte le langage mathématique comme langage de programmation, la construction d'un programme à partir d'une spécification P devient très facile : le "programme" $[f \mid P(f)]$ répond, par définition, à la spécification P. Par exemple, le "programme" $[f \mid \forall M\ f(M) \times M = I]$ est un programme d'inversion de matrices. De plus, ce programme est automatiquement un programme zéro-défaut, puisqu'il n'est qu'une reformulation de sa spécification. Programmer en langage mathématique est donc, entre autres, un moyen de concevoir des programmes zéro-défaut.

L'exécution des programmes

Quand un programme est terminé, il faut pouvoir l'exécuter. Ainsi, si on emprunte 30 000 F sur 24 mois au au taux mensuel de 0.64% (ce qui correspond à un taux annuel de 8%), le programme évoqué ci-dessus doit indiquer que les mensualités seront de 1353 F. En effet

$$f(30000, 24, 0.0064) = 1353$$

La valeur de sortie du programme est donc une expression E (dans cet exemple 1353) telle que la proposition

$$f(30000, 24, 0.0064) = E$$

soit vraie. Mais cela est-il suffisant ? La proposition

$$f(30000, 24, 0.0064) = 3 \times 11 \times 41$$

est également vraie, et

$$f(30000, 24, 0.0064) = f(30000, 24, 0.0064)$$

tout autant. Pourtant on serait bien en peine de dire si on accepte un prêt en sachant que le montant des mensualités est de $3 \times 11 \times 41$ F. Un abîme sépare l'expression 1353 (qui fournit une information utilisable) des expressions $3 \times 11 \times 41$ ou $f(30000, 24, 0.0064)$. Une expression, comme 1353, dans laquelle il n'y a plus rien à calculer est ce qu'en informatique on appelle une *valeur*. La valeur d'un nombre entier est, par exemple, sa représentation décimale. La valeur d'un nombre rationnel peut être son écriture sous forme d'une fraction irréductible, ou sous forme d'une description de son développement décimal périodique par une partie initiale et une période. Il semble qu'on ne puisse pas associer ainsi une expression unique aux nombres réels, aux ensembles ou aux suites de nombres entiers. En revanche, on sait le faire pour les suites et les ensembles finis. Un ensemble où tout élément à une valeur est appelé un *type de données*.

Les mathématiques entretiennent une position ambiguë par rapport à cette notion de valeur. D'un certain point de vue, quand on demande à un écolier d'effectuer l'addition $12 + 23$, on attend le résultat 35 et non $10+25$. Pourtant, le même écolier écrit $12+23 = 35$, en utilisant le signe "=" qui est parfaitement symétrique. Il y a en fait deux traditions qui coexistent dans les mathématiques. Du point de vue dénotationnel, les expressions $12 + 23$ et 35 désignent le même objet, ce qu'on exprime par la proposition $12 + 23 = 35$. Dans cette proposition les expressions $12 + 23$ et 35 jouent des rôles symétriques. En revanche, du point de vue opérationnel il y a une dissymétrie totale entre l'expression $12 + 23$ qui est une question et 35 qui est la réponse à cette question. Le drame de l'informatique est que le point de vue opérationnel, qui est celui des mathématiques de l'Antiquité centrées autour des méthodes opératoires, a peu à peu été étouffé, au cours de l'Histoire, par le point de vue dénotationnel. Le développement de l'informatique nous oblige aujourd'hui à renouer avec cette tradition opérationnelle.

Il nous faut donc concevoir un langage mathématique qui permette non seulement de démontrer que $12 + 23 = 35$ mais aussi de transformer l'expression $12 + 23$ en l'expression 35. La grammaire du langage mathématique ne doit pas uniquement donner des règles de raisonnement, mais aussi des règles de calcul, ces règles qui permettent de transformer une expression en une autre sont appelées *règles de réécriture*. La règle de réécriture la plus simple (bien qu'elle porte le nom barbare de *β-réduction*) est utilisée pour les fonctions définies explicitement. Elle consiste à remplacer les arguments formels par les arguments réels. Ainsi quand on applique la fonction $x, y \mapsto x$ aux nombres 3 et 4, on obtient l'expression $(x, y \mapsto x)(3, 4)$ qui se calcule en remplaçant x par 3 et y par 4 dans l'expression x, donnant le résultat 3. Le problème, qui fait que nous avons eu besoin d'inventer des langages de programmation, est que nous ne connaissons pas de règle similaire pour les fonctions définies implicitement à l'aide de l'opérateur de descriptions.

Quelques langages de programmation

Les langages de programmation traditionnels ne proposent donc pas directement d'opérateur de descriptions, mais ils proposent des constructions plus restreintes correspondant à des cas particuliers de l'utilisation de cet opérateur.

Quasiment tous les langages de programmation permettent de définir des fonctions par récurrence à l'aide de boucles. Ainsi, en Pascal, on peut définir la fonction *puissance* par la boucle :

```
r := 1; for i := 1 to n do r := x * r
```

L'exécution d'un tel programme demande de répéter n fois l'opération qui se trouve dans la boucle, c'est-à-dire à calculer successivement x^1, x^2, ..., x^n.

D'autres langages, comme le langage Lisp (conçu en 1962 par John Mc Carthy et son équipe) ou son successeur le langage ML (suggéré en 1966 par Peter Landin et conçu en 1978 par Robin Milner et son équipe) proposent l'utilisation de définitions récursives, c'est-à-dire feignant d'utiliser la fonction définie dans sa propre définition. Par exemple, en ML, on définit la fonction *puissance* ainsi :

```
let rec puissance = fun x n -> if n = 0 then 1 else x * (puissance x (n-1));;
```

Comme il est incorrect de définir un objet en utilisant l'objet défini lui-même, un tel programme doit se comprendre comme une définition implicite utilisant l'opérateur de descriptions :

$$puissance = [f \mid f = (x, n \mapsto \text{ if } n = 0 \text{ then } 1 \text{ else } x \times f(x, n-1))]$$

Définir ainsi des fonctions récursivement revient à utiliser l'opérateur de descriptions dans le cas particulier où les propriétés ont la forme $f = G(f)$. Exécuter un tel programme demande d'exécuter le corps de la définition $G(f)$ en appelant la fonction elle-même quand elle est utilisée dans sa propre définition.

Contrairement aux boucles, ce mécanisme permet de sortir des limites de l'utilisation de l'opérateur de descriptions autorisées par la grammaire du langage mathématique (qui impose qu'il existe un unique objet vérifiant la propriété). Ainsi les programmes ML

```
let rec f = fun x -> f x;;
```

```
let rec f = fun x -> 1 + (f x);;
```

correspondent aux définitions "illégales" $[f \mid f = x \mapsto f(x)]$ et $[f \mid f = x \mapsto 1 + f(x)]$. Cela se traduit par le fait que l'exécution de ces programmes mène à des calculs infinis (calculer $f(0)$ demande de calculer au préalable $f(0)$ qui demande de calculer ...). Pour donner un sens à ces définitions récursives incorrectes, Dana Scott a proposé en 1970 d'ajouter à l'espace des valeurs, une valeur supplémentaire "l'indéfini" et de voir les définitions récursives comme des définitions de fonctions sur cet espace de valeurs étendu, les deux définitions ci-dessus deviennent légales et correspondent à la même fonction constamment égale à l'indéfini. Autrement dit, l'exécution de ces deux programmes sur n'importe quelle entrée mène toujours à des calculs infinis.

À la différence du langage ML qui utilise l'opérateur de descriptions pour définir la fonction elle même, le langage Prolog (conçu en 1973 par Alain Colmerauer et son équipe) utilise l'opérateur de descriptions uniquement pour définir la valeur prise par la fonction, c'est-à-dire la valeur de sortie du programme. Par exemple la fonction *puissance* n'est plus définie par un terme de la forme $[f \mid P(f)]$ où P est une propriété caractéristique de cette fonction, mais par un terme de la forme $x, n \mapsto [y \mid Q(x,n,y)]$ où Q est une propriété caractéristique du nombre x^n. La propriété $Q(x,n,y)$ peut être par exemple :

$$\forall e\ (\forall a\ e(a,0,1) \text{ et } \forall a\ \forall p\ \forall r\ (e(a,p-1,r) \Rightarrow e(a,p,a\times r))) \Rightarrow e(x,n,y)$$

qui mène au programme Prolog

```
e(A,0,1).
e(A,P,S) :- Q is P - 1, e(A,Q,R), S is A * R.
```

L'exécution du programme demande de résoudre l'équation $Q(x,n,y)$ en y. Cette équation portant sur une valeur et non sur une fonction, on peut envisager sa résolution de manière assez systématique.

Plus généralement les langages de programmation par contraintes introduits par Joxan Jaffar et Jean-Louis Lassez en 1987 sont des extensions de Prolog utilisant des algorithmes spécialisés pour la résolution de ces équations.

Les fonctions non calculables

On peut donc voir les langages de programmation traditionnels comme des restrictions du langage mathématique. L'opérateur de descriptions n'est utilisé que dans certains cas particuliers, la restriction choisie distinguant un langage des autres. L'avantage de cette restriction est qu'elle permet d'exécuter les programmes. Son inconvénient est qu'elle limite l'expression des programmeurs. Un langage de programmation est toujours un compromis entre la puissance d'expression et la possibilité d'exécution.

La question qu'on peut alors se poser est celle de savoir s'il est possible de trouver des règles de calcul pour l'intégralité du langage mathématique, c'est-à-dire pour l'opérateur de descriptions dans toute sa généralité. La réponse, négative, à cette question est apportée par la théorie de la calculabilité : un résultat dû à Alan Turing et indépendamment à Alonzo Church et Stephen Kleene, en 1936, montre, en effet, qu'il existe des fonctions qui ne sont pas calculables. L'exemple le plus célèbre de fonction non calculable est le problème de l'arrêt : il est impossible de construire un programme qui prend en entrée un autre programme et indique s'il mène à des calculs finis ou infinis. On peut donc définir la fonction

$$h = [f \mid \text{pour tout } p \text{, si } p \text{ termine alors } f(p) = 0 \text{ sinon } f(p) = 1]$$

mais il n'y a pas de règle de calcul donnant systématiquement la valeur de $h(p)$.

Pour que cette définition soit correcte, il faut démontrer qu'une unique fonction vérifie la spécification. Cette démonstration utilise le fait que pour tout programme p, ou bien p termine ou bien p ne termine pas. Ce fait est lui-même établi par une règle de raisonnement, le tiers exclu, qui indique que pour toute proposition P on peut démontrer "P ou non P" sans avoir à démontrer "P" ni "non P".

Au début du vingtième siècle, une école de mathématiciens, appelés *intuitionnistes* ou *constructivistes*, a vivement critiqué cette règle de raisonnement. Comment peut-on prétendre savoir que la proposition "P

Un exemple de raisonnement rejeté par les constructivistes est le suivant. On veut montrer qu'il existe deux nombres irrationnels x et y tels que x^y soit rationnel. On raisonne ainsi : Le nombre $\sqrt{2}^{\sqrt{2}}$ est rationnel ou il est irrationnel.

- S'il est rationnel, on prend $x = y = \sqrt{2}$. Les nombres x et y sont irrationnels et x^y est rationnel par hypothèse.
- S'il est irrationnel, on prend $x = \sqrt{2}^{\sqrt{2}}$ et $y = \sqrt{2}$. Le nombre x est irrationnel par hypothèse, y est irrationnel et $x^y = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = 2$ qui est rationnel.

Ce raisonnement n'est pas valable pour les intuitionnistes car on ne peut pas supposer que le nombre $\sqrt{2}^{\sqrt{2}}$ est rationnel ou irrationnel sans démontrer d'abord ou bien qu'il est rationnel ou bien qu'il est irrationnel.

Un exemple de raisonnement non constructif

ou non P" est vraie, si on ne sait ni que "P" est vraie, ni que "non P" est vraie ? De même, comment peut-on prétendre avoir défini le nombre $h(p)$ si on ne sait même pas si ce nombre vaut 0 ou 1 ? Pour les intuitionnistes la définition de la fonction h ci-dessus est tout simplement incorrecte. Derrière Luitzen Egbertus Jan Brouwer, le chef de file de l'école intuitionniste, se sont ralliés des mathématiciens importants comme Hermann Weyl ou Andreï Kolmogorov.

Un théorème dû à Stephen Kleene montre que, quand on abandonne le tiers exclu, toutes les fonctions qu'on peut définir avec l'opérateur de descriptions sont calculables. Ce théorème prend naturellement des formes différentes en fonction de la théorie dans laquelle on se place pour démontrer l'existence et l'unicité de la fonction décrite. Autrement dit, le tiers exclu est indispensable pour montrer l'existence d'une fonction non calculable. Quand on abandonne le tiers exclu, il est donc possible de trouver des règles de calcul pour l'opérateur de descriptions dans toute sa généralité. Quand on définit une fonction $[f \mid P(f)]$ après avoir donné une démonstration constructive p de l'existence d'une fonction vérifiant la propriété P, on peut exécuter ce programme. Pour cela on applique à la démonstration p et à la valeur d'entrée une transformation appelée l'*élimination des coupures*. Le premier procédé d'élimination des coupures a été proposé en 1934 par Gerhard Gentzen pour les démonstrations exprimées dans l'arithmétique. Depuis, il a été généralisé à bien d'autres théories. En particulier, Jean-Yves Girard a proposé en 1971 un procédé d'élimination des coupures pour l'arithmétique d'ordre supérieur, qui est une variante de la théorie des ensembles.

On voit alors se dessiner une nouvelle méthodologie de la programmation : on commence par spécifier la fonction à programmer par une propriété P, on donne ensuite une démonstration constructive p de l'existence d'une fonction vérifiant cette propriété et on définit ensuite le programme $[f \mid P(f)]$. L'exécution de ce programme consiste à éliminer les coupures dans la démonstration p. Cette approche est à la base du langage AF2 suggéré par Daniel Leivant et conçu par Jean-Louis Krivine et Michel Parigot en 1987.

Les démonstrations comme objets

Si ce langage permet l'utilisation de l'opérateur de descriptions sous une forme très générale, il abandonne en revanche le principe selon lequel un programme est une fonction mathématique ordinaire : ce n'est pas la fonction $[f \mid P(f)]$, mais la démonstration p, qui est utilisée lors de l'exécution du programme.

Plus généralement, dans le langage mathématique traditionnel, le statut de la démonstration p par rapport à l'expression $[f \mid P(f)]$ n'est pas absolument clair. Si l'expression $[f \mid P(f)]$ est incorrecte quand il n'y a pas d'objet vérifiant la propriété P, quand faut-il démontrer l'existence d'un tel objet ? à la première occurrence de cette expression ? à chacune de ses occurrences ? pour démontrer que l'objet $[f \mid P(f)]$ vérifie la propriété P ? après avoir utilisé cette expression dans une démonstration ? La démonstration p fait elle partie de toutes les démonstrations qui utilisent une expression de la forme $[f \mid P(f)]$? Le langage mathématique traditionnel élude ces questions en utilisant, pour les définitions implicites, un processus en trois temps qui mêle expression de la propriété, démonstration de l'existence de l'objet et attribution d'un nom.

Pour résoudre, entre autres, ces questions, Per Martin-Löf a proposé en 1973, dans sa *Théorie des types intuitionniste* de donner aux démonstrations un statut d'objets mathématiques à part entière, comparable à celui des nombres ou des fonctions. La Théorie des types intuitionniste est une extension du langage mathématique habituel dans laquelle on peut parler non seulement du nombre 2 (par exemple, pour dire que c'est un nombre pair) mais aussi de la proposition "2 est un nombre pair" et de sa démonstration p.

Dans cette théorie et dans ses extensions, comme le *Calcul des constructions* proposé en 1985 par Thierry Coquand et Gérard Huet, une fonction n'est plus définie uniquement à partir d'une propriété qu'elle est censée vérifier, mais également à partir d'une démonstration de l'existence d'une fonction vérifiant cette propriété. On n'écrit donc plus $[f \mid P(f)]$, mais $[f \mid P(f), p]$. Ainsi le statut de la démonstration p est clarifié : c'est une composante de l'expression $[f \mid P(f), p]$. La grammaire du langage mathématique interdit alors la formation de l'expression $[f \mid P(f), p]$ quand p n'est pas une démonstration de l'existence d'un objet vérifiant la propriété P, mais cette propriété est alors décidable et ne demande donc pas à être argumentée. Dans ces théories, les démonstrations sont des fonctions définies explicitement, mais dans un langage étendu.

Dans ces théories, la fonction $[f \mid P(f), p]$, contenant la démonstration p, contient l'information de la démarche à suivre pour son exécution. On retrouve ainsi à la fois la possibilité de définir des programmes comme des fonctions ordinaires et celle d'exécuter ces programmes en éliminant les coupures dans la démonstration p.

Programmer dans ces langages consiste à prendre la spécification P donnée par le client et à lui rendre le programme $[f \mid P(f), p]$. Bien sûr, cela demande de construire la démonstration p de l'existence d'une fonction vérifiant la spécification, ce qui est un véritable travail de programmeur. Un avantage est que si cette démonstration est correcte (ce qui est une propriété décidable), le programme $[f \mid P(f), p]$ vérifie, par définition, cette spécification, c'est donc un programme zéro-défaut.

La gestion des ressources

Pour être réellement compétitive avec les approches plus traditionnelles, la programmation en langage mathématique devra sans doute encore résoudre le problème de la gestion des ressources. Dès l'apparition des premiers langages de programmation, les programmeurs ont eu le sentiment d'être dépossédés du contrôle des opérations effectuées dans leurs machines. Par exemple, en utilisant des noms symboliques pour désigner les variables du programme, ils ont eu le sentiment de perdre le contrôle de l'endroit où l'information était effectivement stockée dans la mémoire. Ce sentiment est d'autant plus vif que le langage utilisé est de haut niveau, mais il prend peut-être une forme nouvelle quand on programme en langage mathématique. En effet, pour trier par ordre alphabétique une liste de mots, par exemple la liste : *whisky*, *gin*, *vodka*, un programme permute, par exemple, les mots *whisky* et *gin* pour construire la liste *gin*, *whisky*, *vodka* puis *whisky* et *vodka* pour construire la liste *gin*, *vodka*, *whisky*. Si le programme garde en mémoire les trois listes :

whisky, *gin*, *vodka*
gin, *whisky*, *vodka*
gin, *vodka*, *whisky*

il est mal conçu. Tous les programmeurs savent que la première liste est devenue inutile dès que la deuxième est construite et donc qu'ils peuvent trier "en place" c'est-à-dire utiliser la même zone de la mémoire pour stocker les états successifs de la liste.

Si on se contente de démontrer l'existence d'une fonction associant une permutation triée à chaque liste, comment savoir si l'exécution de ce programme triera en place ou recopiera la liste à chaque étape ? Les mathématiques ne sont pas un langage de programmation très conscient de la gestion des ressources.

La programmation en langage mathématique pose de nouveaux défis aux théories de la compilation, de l'analyse statique et de la transformation de programme. Comment transformer un programme développé en langage mathématique en un programme en langage machine équivalent mais qui gère efficacement ses ressources et n'effectue pas de calculs inutiles ?

Un outil essentiel pour résoudre ces questions est sans doute la *logique linéaire*, développée par Jean-Yves Girard en 1987. Selon cette théorie, les mathématiques ne sont pas très conscientes de la gestion des ressources parce que "les hypothèses ne s'usent pas quand on s'en sert". Ainsi dans les logiques traditionnelles les propositions "P" et "P et P" sont équivalentes. En logique linéaire ces deux propositions ne sont plus équivalentes, supposer la première permet d'utiliser une unique fois l'hypothèse P dans une démonstration alors que supposer la seconde permet d'utiliser deux fois cette hypothèse. Dans la démonstration de l'existence d'une fonction de tri, une hypothèse utilisée une unique fois révèle qu'une fois la liste utilisée, elle peut être détruite, ce qui est exactement l'information nécessaire pour programmer le tri en place. La logique linéaire a déjà été utilisée pour gérer les ressources dans des compilateurs pour le langage ML. Ce type d'analyse est à étendre au cas de la programmation en langage mathématique.

Réactions sur le langage mathématique

La théorie des langages de programmation permet donc d'expliquer pourquoi le langage mathématique n'est pas complètement adapté à la programmation, en quoi les langages de programmation traditionnels sont des restrictions du langage mathématique, et comment il faut modifier ce dernier pour qu'il soit possible de l'utiliser pour programmer.

Depuis l'Antiquité, le langage des mathématiques a énormément évolué pour répondre aux besoins des mathématiciens qui ont varié au cours de l'histoire. Aujourd'hui, il semble que le développement de l'informatique suggère une nouvelle évolution du langage mathématique.

Naturellement, une suggestion d'évolution du langage mathématique connaît le succès uniquement quand elle dépasse le cadre étroit du problème qui l'a motivée et quand elle permet d'exprimer les mathématiques dans leur ensemble. Comme nous l'avons vu, par rapport à la théorie des ensembles qui est le langage des mathématiques du vingtième siècle, les réformes que les informaticiens suggèrent aujourd'hui concernent trois points :

- le retour à un point de vue plus opérationnel et la prise en compte de la notion de calcul,
- l'abandon du tiers exclu et la restriction aux mathématiques constructives,
- la prise en charge des démonstrations comme des objets mathématiques à part entière.

Il semble que le retour vers un point de vue plus opérationnel ne pose pas de véritable problème, que $12 + 23$ se calcule en 35 en plus du fait qu'il soit égal à 35 est une idée largement partagée. La prise en charge des démonstrations comme des objets mathématiques ne semble pas non plus poser de véritable problème. Au cours de leur histoire, les mathématiques ont toujours intégré les objets extramathématiques qu'elles utilisaient. Les fonctions qui étaient utilisées depuis l'Antiquité pour désigner des grandeurs (par exemple, dans l'expression "$sinus(0)$") sont devenues des objets mathématiques à part entière avec Newton et Leibniz, voire avec Euler, quand il est devenu possible d'exprimer des propriétés des fonctions elles-mêmes (et de dire que la fonction $sinus$ est, par exemple, continue, périodique, dérivable, etc.). De plus, il semble qu'avoir les démonstrations comme des objets à part entière résolve des problèmes hors du champ de la programmation, et en particulier que certaines formes de l'"axiome" du choix deviennent des théorèmes.

C'est, bien entendu, l'abandon du tiers exclu qui pose le plus de problèmes. Abandonner le tiers exclu revient à abandonner des résultats importants en analyse, par exemple le théorème selon lequel une fonction continue sur un intervalle fermé prend une valeur maximale. Plutôt que d'abandonner complètement les méthodes non constructives, il semble que le défi soit aujourd'hui davantage de concevoir un langage mathématique qui intègre en les distinguant clairement les arguments constructifs des arguments non constructifs.

Ces notes de cours sont consacrées à la Théorie des types de intuitionniste de Martin-Löf et au Calcul des constructions. Ces deux axiomatisations des mathématiques intègrent les démonstrations comme des objets et comprennent une notion de calcul, ce qui permet de les utiliser pour programmer, en particulier pour concevoir des programmes zero-defaut. Avant les chapitres consacrés à ces formalismes, la première partie est consacrée à la notion traditionnelle de démonstration et la deuxième aux axiomatisations traditionnelles des mathématiques (théorie des ensembles et théorie des types simples).

Bibliographie

[1] Thierry Coquand, Gérard Huet, *The calculus of constructions*, Information and Computation 76 (1988) p. 74-85.

[2] Jean-Yves Girard, Yves Lafont, Paul Taylor, *Proofs and types*, Cambridge University Press, Cambridge (1989).

[3] Jean-Louis Krivine, *Lambda calcul, types et modèles*, Masson, Paris (1990).

[4] Per Martin-Löf, *Constructive mathematics and computer programming*, Logic, Methodology and Philosophy of Science, VI, 1979, North-Holland (1982) p. 153-175.

[5] Per Martin-Löf, *Intuitionnistic type theory*, Bibliopolis, Napoli (1984).

Première partie

Les démonstrations

Chapitre 1

La logique du premier ordre

1.1 Les langages du premier ordre

Un langage logique permet de désigner des choses (la Lune, Pépin le bref, le nombre deux, l'ensemble des nombres pairs, ...) et d'exprimer des faits (la Lune est un satellite de la Terre, Pepin le bref est le père de Charlemagne, le nombre deux est pair, l'ensemble des nombres pairs est infini, ...). Une expression qui permet d'exprimer une chose est appelée un *terme*, une expression qui permet d'exprimer un fait une *proposition*.

Définition Un langage logique est constitué de variables, de symboles d'individu, de symboles de fonction et de symboles de prédicat. Les variables doivent être en nombre infini. À chaque symbole de fonction et de prédicat on associe son nombre d'arguments ou *arité*.

Exemple Le langage de l'arithmétique est constitué

- de variables x, x_0, x_1, x_2, ..., y, y_0, ...
- du symbole d'individu 0,
- des symboles de fonction S (*successeur*), $+$ et $\times$,
- du symbole de prédicat $=$.

L'arité du symbole S est 1, celle des symboles $+$, $\times$ et $=$ est 2.

Remarque Les symboles d'individu peuvent être vus comme des symboles de fonction d'arité nulle. Les symboles de prédicat d'arité nulle sont parfois appelés *symboles de proposition*. Un exemple de symbole de proposition en français est le symbole "pleut" dans la phrase "(il) pleut".

1.1.1 Les termes

Les termes sont formés par les deux règles suivantes :

- les variables sont des termes,
- les symboles d'individu sont des termes,
- si f est un symbole de fonction d'arité n et $t_1, ..., t_n$ sont des termes alors l'expression $f(t_1, ..., t_n)$ est un terme.

Exemple En arithmétique, les expressions suivantes sont des termes 0, $S(0)$ (plus informellement notée 1), $S(S(0))$, (plus informellement notée 2), $S(S(S(0)))$, (plus informellement notée 3), $+(0,0)$ (plus informellement notée $0+0$), x, $+(x,y)$, $+(\times(x,S(S(0))),S(y))$, ...

Remarque On a, bien souvent, d'une définition plus rigoureuse de la notion de terme. On doit alors définir les termes comme des suites finies de symboles pris dans l'ensemble constitué des variables, des symboles d'individu, de fonction et de prédicat et des *symboles impropres* "(", ")", ",". Une définition alternative consiste à les définir comme des arbres dont les nœuds internes sont labellés par des symboles de fonction et les feuilles par des variables et des symboles d'individu.

Définition Un terme, tel $+(\times(x,S(S(0))),S(y))$, qui contient des variables est dit *ouvert*, un terme qui n'en contient pas est dit *clos*.

1.1.2 Les propositions

Le moyen le plus simple de former une proposition est d'appliquer un symbole de prédicat à un certain nombre de termes, on peut, par exemple, former ainsi les propositions $=(+(S(0),S(0)),S(S(0)))$, c'est-à-dire $1+1=2$, ou $=(+(S(0),S(0)),S(S(S(0))))$, c'est-à-dire $1+1=3$. On a aussi besoin de former des propositions plus complexes, pour cela on utilise les connecteurs $\perp$ ("contradiction"), $\neg$ ("non"), $\wedge$ ("et"), $\vee$ ("ou"), $\Rightarrow$ ("implique"), $\Leftrightarrow$ ("si et seulement si"), et de quantificateurs $\forall$ ("pour tout") et $\exists$ ("il existe").

Les quantificateurs servent à exprimer que tous les objets du discours vérifient une propriété, ou qu'au moins un objet vérifie une propriété, mais sans spécifier lequel. Pour exprimer de telles propositions, les langues naturelles, comme le français, utilisent des pronoms indéfinis, par exemple "tous" et "quelques" : "tous les hommes sont mortels", "tous les nombres entiers sont supérieurs à 0", ... Formellement on écrirait cette dernière phrase

$$\text{tout} \geq 0$$

dans laquelle le symbole "tout" prend la place d'un terme comme argument du symbole de prédicat.

Hélas, ce mécanisme est ambigu quand plusieurs termes sont remplacés par de tels symboles. Par exemple la phrase "il y a un nombre entier supérieur à tout nombre entier" peut signifier ou bien que pour chaque nombre entier, il existe un nombre entier qui lui est supérieur (ce qui est vrai) ou bien qu'il existe un nombre qui est est supérieur à tous les nombres entiers (ce qui est faux).

On préfère donc mettre une variable comme argument du prédicat

$$x \geq 0$$

et indiquer ensuite la signification et la portée de cette variable par un quantificateur

$$\forall x\ (x \geq 0)$$

Ainsi, on distingue les propositions

$$\forall x\ \exists y\ (y \geq x)$$

qui est vraie et

$$\exists y\ \forall x\ (y \geq x)$$

qui est fausse.

Définition Les propositions sont formés par les règles suivantes :

- si P est un symbole de prédicat d'arité n et $t_1, ..., t_n$ sont des termes alors l'expression $P(t_1, ..., t_n)$ est une proposition,
- $\perp$ est une proposition, si A est une proposition alors $\neg A$ est une proposition et si A et B sont des propositions alors $A \wedge B$, $A \vee B$, $A \Rightarrow B$ et $A \Leftrightarrow B$ sont des propositions,
- si A est une proposition et x une variable alors $\forall x\ A$ et $\exists x\ A$ sont des propositions.

1.2 Les démonstrations

On veut maintenant se donner les outils qui permettent de démontrer des propositions.

1.2.1 Les axiomes

Une démonstration se construit à partir d'*axiomes* qui sont des propositions dont la vérité est admise sans argumentation. Un ensemble d'axiomes s'appelle une *théorie*.

Exemple La théorie de l'égalité est formée des axiomes suivants.
Principe d'identité:

$$\forall x\ (x = x)$$

Schéma d'axiome de Leibniz: pour chaque proposition A contenant une variable z l'axiome

$$\forall x\ \forall y\ ((x = y) \Rightarrow (A[z \leftarrow x] \Rightarrow A[z \leftarrow y]))$$

Définition La notation $A[x \leftarrow t]$ désigne la proposition A dans laquelle la variable x a été substituée par le terme t. Cette proposition se définit ainsi par récurrence sur la structure de A :

- $x[x \leftarrow t] = t$,
- si y est une variable distincte de x, alors $y[x \leftarrow t] = y$,
- $f(a_1, ..., a_n)[x \leftarrow t] = f(a_1[x \leftarrow t], ..., a_n[x \leftarrow t])$,
- $P(a_1, ..., a_n)[x \leftarrow t] = P(a_1[x \leftarrow t], ..., a_n[x \leftarrow t])$,
- $\bot[x \leftarrow t] = \bot$
- $(\neg A)[x \leftarrow t] = \neg A[x \leftarrow t]$
- $(A \wedge B)[x \leftarrow t] = A[x \leftarrow t] \wedge B[x \leftarrow t]$, $(A \vee B)[x \leftarrow t] = A[x \leftarrow t] \vee B[x \leftarrow t]$,
  $(A \Rightarrow B)[x \leftarrow t] = A[x \leftarrow t] \Rightarrow B[x \leftarrow t]$, $(A \Leftrightarrow B)[x \leftarrow t] = A[x \leftarrow t] \Leftrightarrow B[x \leftarrow t]$,
- $(\forall x\ A)[x \leftarrow t] = \forall x\ A$, $(\exists x\ A)[x \leftarrow t] = \exists y\ A$,
- si y est une variable distincte de x, $(\forall y\ A)[x \leftarrow t] = \forall y\ A[x \leftarrow t]$, $(\exists y\ A)[x \leftarrow t] = \exists y\ A[x \leftarrow t]$.
  Dans ce cas, il faut que y n'apparaisse pas dans t. Si c'est le cas, il est nécessaire de renommer la variable y dans $\forall y\ A$ ou $\exists y\ A$ avec une nouvelle variable.

Exemple L'arithmétique est la théorie obtenue en ajoutant à ces axiomes les axiomes de Peano :

$$\forall x\ \forall y\ (S(x) = S(y) \Rightarrow x = y)$$

$$\forall x\ \neg(0 = S(x))$$

pour chaque proposition A contenant une variable z l'axiome

$$(A[z \leftarrow 0] \wedge (\forall x\ (A[z \leftarrow x] \Rightarrow A[z \leftarrow S(x)]))) \Rightarrow \forall y\ A[z \leftarrow y]$$

$$\forall y\ (0 + y = y)$$

$$\forall x\ \forall y\ (S(x) + y = S(x + y))$$

$$\forall y\ (0 \times y = 0)$$

$$\forall x\ \forall y\ (S(x) \times y = (x \times y) + y)$$

D'autres exemples sont la géométrie élémentaire, la théorie des ensembles, ...

1.2.2 Les démonstrations au sens de Frege et Hilbert

Définition Axiomes logiques

Un *axiome logique* est une proposition de la forme ci-dessous, où A, B, C désignent des propositions quelconques et x une variable quelconque.

$$A \Rightarrow (B \Rightarrow A)$$

$$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$$

$$(\forall x\ (A \Rightarrow B)) \Rightarrow (A \Rightarrow \forall x\ B) \quad \text{(si } x \text{ n'apparaît pas dans } A\text{)}$$

$$(A \wedge B) \Rightarrow A$$

$$(A \wedge B) \Rightarrow B$$

$$A \Rightarrow B \Rightarrow (A \wedge B)$$

$$A \Rightarrow (A \vee B)$$

$$B \Rightarrow (A \vee B)$$

$$(A \vee B) \Rightarrow ((A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow C))$$

$$A \Rightarrow (\neg A \Rightarrow \bot)$$

$$(A \Rightarrow \bot) \Rightarrow \neg A$$

$$\bot \Rightarrow A$$

$$(A \Leftrightarrow B) \Rightarrow (A \Rightarrow B)$$

$$(A \Leftrightarrow B) \Rightarrow (B \Rightarrow A)$$

$$(A \Rightarrow B) \Rightarrow ((B \Rightarrow A) \Rightarrow (A \Leftrightarrow B))$$

$$\forall x\ A \Rightarrow A[x \leftarrow t]$$

$$A[x \leftarrow t] \Rightarrow \exists x\ A$$

$$\exists x\ A \Rightarrow ((\forall x\ (A \Rightarrow B)) \Rightarrow B) \quad \text{(si } x \text{ n'apparaît pas dans } B\text{)}$$

En logique classique, on ajoute l'axiome

$$A \vee \neg A$$

Définition (Démonstration au sens de Frege et Hilbert)

Soit Γ une théorie, c'est-à-dire un ensemble de propositions. Une *démonstration* dans Γ est une suite finie de propositions $P_1, ..., P_n$ telle que pour chaque i la proposition P_i est obtenue par l'une des *règles de déduction* ci-dessous.

- *Axiome* : P_i est un axiome, c'est-à-dire un axiome logique ou une proposition de Γ. On note cette règle

$$\frac{}{A} \text{ si } A \in \Gamma \text{ ou } A \text{ est un axiome logique}$$

- *Modus ponens* : $P_i = B$ et il y a deux entiers j et k strictement inférieurs à i et une proposition A tels que $P_j = A \Rightarrow B$ et $P_k = A$. On note cette règle

$$\frac{A \Rightarrow B \quad A}{B}$$

- *Généralisation* : $P_i = \forall x\ A$, il y a un entier j strictement inférieur à i tel que $P_j = A$ et x n'apparaît pas dans les axiomes de Γ. On note cette règle

$$\frac{A}{\forall x\ A} \text{ si } x \text{ n'apparaît pas dans } \Gamma$$

Définition Soit Γ une liste de propositions et A une proposition. Une *démonstration de A dans* Γ est une démonstration dans Γ, $P_1, ..., P_n$ dont la dernière proposition P_n est A.

Exemple Dans un langage dans lequel on a quatre symboles de proposition (symboles de prédicat d'arité nulle) P, Q, R, S, sous les axiomes $P \Rightarrow (R \Rightarrow S), Q \Rightarrow R, P, Q$, on a la démonstration suivante

$$\begin{array}{c} P \Rightarrow (R \Rightarrow S) \\ P \\ R \Rightarrow S \\ Q \Rightarrow R \\ Q \\ R \\ S \end{array}$$

1.2.3 Les suites et les arbres

Sous ces mêmes axiomes, on a aussi la démonstration

$$\begin{array}{c} Q \Rightarrow R \\ Q \\ R \\ P \Rightarrow (R \Rightarrow S) \\ P \\ R \Rightarrow S \\ S \end{array}$$

Ces deux démonstrations procèdent en démontrant S à partir de $R \Rightarrow S$ et R, pour démontrer $R \Rightarrow S$ elles utilisent les axiomes $P \Rightarrow (R \Rightarrow S)$ et P et pour démontrer R elles utilisent les axiomes $Q \Rightarrow R$ et Q. La seule différence tient dans le fait que la première démontre d'abord $R \Rightarrow S$ alors que la seconde démontre d'abord R. Comme l'ordre de démonstration des prémisses d'une règle est indifférent, on préfère souvent définir les démonstration comme des arbres.

Définition Une démonstration sous une liste d'axiomes Γ est un arbre dont chaque nœud est étiqueté par une proposition. La proposition étiquetant un nœud doit être produite par une règle de déduction à partir des propositions étiquettes de ses fils.

Une démonstration d'une proposition A sous une liste d'axiomes Γ est une démonstration sous les axiomes Γ dont la racine est étiquetée par la proposition A.

Exemple

$$\frac{\dfrac{P \Rightarrow (R \Rightarrow S) \quad P}{R \Rightarrow S} \qquad \dfrac{Q \Rightarrow R \quad Q}{R}}{S}$$

Proposition Une proposition est démontrable au sens de la première définition si et seulement si elle est démontrable au sens de la seconde.

1.2.4 Le lemme de déduction

Proposition Soit Γ une liste quelconque de propositions et A une proposition quelconque. La proposition $A \Rightarrow A$ est démontrable dans Γ.

Démonstration

$$\dfrac{\dfrac{(A \Rightarrow ((A \Rightarrow A) \Rightarrow A)) \Rightarrow ((A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A)) \quad A \Rightarrow ((A \Rightarrow A) \Rightarrow A)}{(A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A)} \qquad A \Rightarrow (A \Rightarrow A)}{A \Rightarrow A}$$

Proposition (Lemme de déduction)

Soit Γ une liste de propositions et A et B deux propositions, la proposition $A \Rightarrow B$ est démontrable dans Γ si et seulement si B est démontrable dans Γ, A.

Démonstration S'il y a une démonstration de π de $A \Rightarrow B$ dans Γ alors la démonstration

$$\dfrac{\dfrac{\pi}{A \Rightarrow B} \quad A}{B}$$

est une démonstration de B dans Γ, A.

Réciproquement on montre par récurrence sur la hauteur de l'arbre que s'il y a une démonstration de B dans Γ, A alors il y a une démonstration de $A \Rightarrow B$ dans Γ.

- Si la racine de l'arbre est étiquetée par un axiome logique ou une proposition de Γ alors B est un axiome et

  $$\dfrac{B \Rightarrow (A \Rightarrow B) \quad B}{A \Rightarrow B}$$

  est une démonstration de $A \Rightarrow B$.

- Si $B = A$, alors

  $$\dfrac{\dfrac{(A \Rightarrow ((A \Rightarrow A) \Rightarrow A)) \Rightarrow ((A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A)) \quad A \Rightarrow ((A \Rightarrow A) \Rightarrow A)}{(A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A)} \qquad A \Rightarrow (A \Rightarrow A)}{A \Rightarrow A}$$

  est une démonstration de $A \Rightarrow A$.

- Si B se déduit de $C \Rightarrow B$ et C par *modus ponens*, alors par hypothèse de récurrence, il existe des démonstrations π_1 et π_2 de $A \Rightarrow (C \Rightarrow B)$ et $A \Rightarrow C$ dans Γ. La démonstration

  $$\dfrac{\dfrac{(A \Rightarrow (C \Rightarrow B)) \Rightarrow ((A \Rightarrow C) \Rightarrow (A \Rightarrow B)) \quad \dfrac{\pi_1}{A \Rightarrow (C \Rightarrow B)}}{(A \Rightarrow C) \Rightarrow (A \Rightarrow B)} \qquad \dfrac{\pi_2}{A \Rightarrow C}}{A \Rightarrow B}$$

  est une démonstration de $A \Rightarrow B$ dans Γ.

- Si B se déduit de C par généralisation, alors $B = \forall x\ C$ et x n'apparaît pas dans Γ ni dans A. Par hypothèse de récurrence il y a une preuve π de la proposition $A \Rightarrow C$ dans Γ. La démonstration

  $$\dfrac{(\forall x\ (A \Rightarrow C)) \Rightarrow (A \Rightarrow \forall x\ C) \quad \dfrac{\dfrac{\pi}{A \Rightarrow C}}{\forall x\ (A \Rightarrow C)}}{A \Rightarrow \forall x\ C}$$

  est une démonstration de $A \Rightarrow B$ dans Γ.

Exemple Sous les axiomes $P, P \Rightarrow Q$, la démonstration suivante est une démonstration de Q :

$$\frac{P \Rightarrow Q \quad P}{Q}$$

On peut la transformer en une démonstration de $(P \Rightarrow Q) \Rightarrow Q$ sous l'axiome P. La démonstration obtenue est la suivante :

$$\dfrac{\dfrac{((P \Rightarrow Q) \Rightarrow (P \Rightarrow Q)) \Rightarrow (((P \Rightarrow Q) \Rightarrow P) \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)) \quad \dfrac{...}{(P \Rightarrow Q) \Rightarrow (P \Rightarrow Q)}}{((P \Rightarrow Q) \Rightarrow P) \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)} \quad \dfrac{P \Rightarrow ((P \Rightarrow Q) \Rightarrow P) \quad P}{(P \Rightarrow Q) \Rightarrow P}}{(P \Rightarrow Q) \Rightarrow Q}$$

1.3 Les langages du premier ordre multisortés

Dans certaine théories, on est amené à distinguer plusieurs sortes d'objets. Par exemple, si on a des symboles d'individu *français*, *anglais*, *France*, *Royaume-Uni*, ... et un prédicat binaire *langue*. On peut alors former les propositions

$$\textit{langue}(\textit{France},\textit{français})$$
$$\textit{langue}(\textit{Royaume-uni},\textit{anglais})$$

qui expriment que le français est une langue officielle de la France, l'anglais du Royaume-uni, ...

Étrangement, on peut aussi former la proposition

$$\textit{langue}(\textit{France},\textit{Royaume-uni})$$

qui exprime que le Royaume-uni est une langue officielle de la France, proposition qui bien plus que fausse semble dénuée de sens. Une variante de la logique du premier ordre, *la logique du premier ordre multisortée* permet alors de restreindre la grammaire du langage de manière à éviter ces propositions en distinguant plusieurs *sortes* d'objets, dans ce cas les pays et les langues.

À chaque symbole d'individu on associe une sorte, à chaque symbole de fonction d'arité n un *rang* $(s_1, ..., s_n, s_{n+1})$ où $s_1, ..., s_n$ sont les sortes de ses arguments et s_{n+1} celle de son résultat et à chaque symbole de prédicat d'arité n un *rang* $(s_1, ..., s_n)$ où $s_1, ..., s_n$ sont les sortes de ses arguments. On associe aussi une sorte à chaque variable de manière à ce qu'il y ait un nombre infini de variables de chaque sorte.

Les termes et les propositions sont alors définis de manière à n'appliquer les symboles de fonction et de prédicat qu'aux termes de la sorte correspondant à leur rang. Les axiomes logiques sont restreints de manière à ce qu'on ne puisse substituer une variable que par un terme de la même sorte.

Remarque Une théorie du premier ordre multisortée peut toujours se relativiser en une théorie du premier ordre ordinaire. À chaque symbole d'individu c on associe un symbole d'individu c', à chaque symbole de fonction f on associe un symbole de fonction f' de même arité, à chaque symbole de prédicat P on associe un symbole de prédicat P' de même arité. Pour chaque sorte s on introduit un symbole de prédicat $\mathcal{T}_s$ à un argument.

On traduit alors ainsi les termes et les propositions:

- $|c| = c'$,
- $|x| = x$,
- $|f(t_1, ..., t_n)| = f'(|t_1|, ..., |t_n|)$.
- $|P(t_1, ..., t_n)| = P'(|t_1|, ..., |t_n|)$,
- $|A \wedge B| = |A| \wedge |B|$, $|A \vee B| = |A| \vee |B|$, $|A \Rightarrow B| = |A| \Rightarrow |B|$, $|A \Leftrightarrow B| = |A| \Leftrightarrow |B|$.
- $|\forall x\ A| = \forall x\ (\mathcal{T}_s(x) \Rightarrow |A|)$, $|\exists x\ A| = \exists x\ (\mathcal{T}_s(x) \wedge |A|)$, où s est la sorte de la variable x.

On traduit une théorie en traduisant chacun des axiomes et en ajoutant pour chaque sorte s l'axiome

$$\exists x\ \mathcal{T}_s(x)$$

et pour chaque symbole de fonction f de rang $(s_1, ..., s_n, s_{n+1})$ l'axiome

$$\forall x_1\ ...\ \forall x_n\ ((\mathcal{T}_{s_1}(x_1) \wedge ... \wedge \mathcal{T}_{s_n}(x_n)) \Rightarrow (\mathcal{T}_{s_{n+1}}(f'(x_1, ..., x_n))))$$

On peut alors montrer que la proposition P est démontrable dans la théorie Γ si et seulement si la proposition P' est démontrable sous les axiomes Γ'.

Chapitre 2

La déduction naturelle

2.1 L'introduction d'hypothèses

L'introduction d'hypothèses paraît une étape naturelle dans l'expression d'une démonstration. Pour démontrer que $(n = 0) \Rightarrow (n+1 = 1)$, on veut pouvoir supposer que $n = 0$ puis démontrer que $n+1 = 1$. La notion de démonstration du chapitre précédent ne permet pas de faire cela directement. En revanche si on connaît une démonstration de $n + 1 = 1$ sous l'hypothèse $n = 0$, le lemme de déduction permet de transformer cette démonstration en une démonstration de $(n = 0) \Rightarrow (n+1 = 1)$, mais la démonstration obtenue est beaucoup plus longue que la démonstration initiale et très peu naturelle.

La *déduction naturelle* propose de prendre l'introduction d'une hypothèse comme l'une des règles de déduction. Si on prend cette règle, une étape de déduction peut modifier aussi bien la proposition P que la liste Γ. En déduction naturelle, une démonstration n'est plus un arbre de propositions, mais un arbre de couples (Γ, P) où Γ est une liste de propositions et P une proposition. Un tel couple se note $\Gamma \vdash P$ (lire "Gamma thèse P") et est appelé un séquent[1].

Définition Un *séquent* est une paire $\Gamma \vdash P$ où Γ est une liste de propositions et P une proposition.

Définition La *règle d'introduction d'hypothèses* est la règle permettant de déduire un séquent $\Gamma \vdash A \Rightarrow B$ du séquent $\Gamma, A \vdash B$. On la note

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

Remarque Quand on prend la règle d'introduction d'hypothèses comme une règle de déduction, les trois premiers schémas d'axiomes logiques sont inutiles. En effet, si Γ est une liste de proposition quelconque, on peut démontrer $\Gamma \vdash A \Rightarrow (B \Rightarrow A)$ ainsi

$$\frac{\dfrac{\Gamma, A, B \vdash A}{\Gamma, A \vdash B \Rightarrow A}}{\Gamma \vdash A \Rightarrow (B \Rightarrow A)}$$

[1] La notion de séquent est utilisée dans plusieurs formulations de la déduction, en particulier en déduction naturelle et en calcul des séquents. L'utilisation du terme "séquent" ne doit pas introduire de confusion : la déduction naturelle et le calcul des séquents sont deux formalismes très différents.

De même, le séquent $\Gamma \vdash (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$ se démontre ainsi

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta \vdash A \Rightarrow (B \Rightarrow C) \quad \Delta \vdash A}{\Delta \vdash B \Rightarrow C} \qquad \dfrac{\Delta \vdash A \Rightarrow B \quad \Delta \vdash A}{\Delta \vdash B}}{\Gamma, A \Rightarrow (B \Rightarrow C), A \Rightarrow B, A \vdash C}}{\Gamma, A \Rightarrow (B \Rightarrow C), A \Rightarrow B \vdash A \Rightarrow C}}{\Gamma, A \Rightarrow (B \Rightarrow C) \vdash (A \Rightarrow B) \Rightarrow (A \Rightarrow C)}}{\Gamma \vdash (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))}}{}$$

où $\Delta = \Gamma, A \Rightarrow (B \Rightarrow C), A \Rightarrow B, A$.

Enfin, si Γ est un contexte quelconque, x une variable n'apparaissant pas dans Γ et A et B deux propositions telles que x n'apparaît pas dans A on peut démontrer le séquent $\Gamma \vdash (\forall x\ (A \Rightarrow B)) \Rightarrow (A \Rightarrow \forall x\ B)$ ainsi

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \forall x\ (A \Rightarrow B), A \vdash \forall x\ (A \Rightarrow B) \quad \Gamma, \forall x\ (A \Rightarrow B), A \vdash (\forall x\ (A \Rightarrow B)) \Rightarrow (A \Rightarrow B)}{\Gamma, \forall x\ (A \Rightarrow B), A \vdash A \Rightarrow B} \quad \Gamma, \forall x\ (A \Rightarrow B), A \vdash A}{\Gamma, \forall x\ (A \Rightarrow B), A \vdash B}}{\Gamma, \forall x\ (A \Rightarrow B), A \vdash \forall x\ B}}{\Gamma, \forall x\ (A \Rightarrow B) \vdash A \Rightarrow \forall x\ B}}{\Gamma \vdash (\forall x\ (A \Rightarrow B)) \Rightarrow (A \Rightarrow \forall x\ B)}$$

2.2 Les axiomes logiques et les règles de déduction

Quand on a démontré les propositions A et B, on veut en déduire la proposition $A \wedge B$. Dans le système du chapitre précédent, on doit utiliser l'axiome logique $A \Rightarrow (B \Rightarrow (A \wedge B))$, et en déduire $B \Rightarrow (A \wedge B)$ et $A \wedge B$ avec le *modus ponens*.

Il est plus naturel de se donner directement la règle

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Dans le système du chapitre précédent, est-il équivalent de se donner l'axiome et la règle ? Chaque application de la règle peut clairement être simulée par une utilisation de l'axiome et deux applications du *modus ponens*. Mais qu'en est-il de la réciproque ?

Avec la règle ci-dessus on peut facilement démontrer le séquent $A, B \vdash A \wedge B$, mais pour en déduire $A \Rightarrow (B \Rightarrow (A \wedge B))$, il nous faut le lemme de déduction. Or, le lemme de déduction est-il toujours correct quand on ajoute la règle

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

? Pour traduire une démonstration dans laquelle on utilise cette règle, il faudrait pouvoir déduire $P \Rightarrow (A \wedge B)$ de $P \Rightarrow A$ et $P \Rightarrow B$. C'est-à-dire avoir un axiome similaire à celui qu'on cherche à remplacer. Donc, dans le système du chapitre précédent, on ne peut pas simplement remplacer l'axiome par la règle.

En déduction naturelle, en revanche, le lemme de déduction est donné par une règle, et il est facile de montrer que si on se donne la règle

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

on peut démontrer la proposition $A \Rightarrow (B \Rightarrow (A \wedge B))$ dans n'importe quel contexte Γ.

$$\dfrac{\dfrac{\dfrac{\Gamma, A, B \vdash A \quad \Gamma, A, B \vdash B}{\Gamma, A, B \vdash A \wedge B}}{\Gamma, A \vdash B \Rightarrow (A \wedge B)}}{\Gamma \vdash A \Rightarrow (B \Rightarrow (A \wedge B))}$$

On peut ainsi supprimer tous les axiomes logiques et les remplacer par des règles de déduction. Le système ainsi obtenu est appelé *la déduction naturelle*.

Donnons un autre exemple. L'axiome

$$(A \vee B) \Rightarrow ((A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow C))$$

peut être remplacé par la règle

$$\frac{\Gamma \vdash A \vee B \quad \Gamma \vdash A \Rightarrow C \quad \Gamma \vdash B \Rightarrow C}{\Gamma \vdash C}$$

Mais, comme il est équivalent de démontrer $\Gamma \vdash A \Rightarrow C$ ou $\Gamma, A \vdash C$, la règle peut encore se transformer en

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

Dans cette règle, le seul connecteur apparaissant explicitement est le connecteur $\vee$. De même, la plupart des règles la déduction naturelle concernent un seul connecteur. Ce qui permet de classer ces règles en fonction du connecteur qu'elles concernent.

Les règles concernant un même connecteur peuvent ensuite être classées en fonction de la position de ce connecteur : s'il apparaît dans la conclusion la règles est appelée *règle d'introduction*, s'il apparaît dans les prémisses la règle est appelée *règle d'élimination*. Par exemple le connecteur $\vee$ a deux règles d'introduction

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-intro}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-intro}$$

et une règle d'élimination

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\text{-élim}$$

Le *modus ponens*

$$\frac{A \Rightarrow B \quad A}{B}$$

devient une règle d'élimination de l'implication.

La règle de généralisation

$$\frac{A}{\forall x\ A}\ \text{si } x \text{ n'apparaît pas dans } \Gamma$$

devient une règle d'introduction du quantificateur universel.

La règle d'introduction d'une hypothèse

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

devient une règle d'introduction de l'implication.

2.3 Définition

Définition (Règles de la déduction naturelle)

$$\frac{}{\Gamma \vdash A}\ \text{axiome} \quad \text{si } A \in \Gamma$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow\text{-intro}$$

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow\text{-élim}$$

$$\frac{\Gamma, A \vdash B \quad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B} \Leftrightarrow\text{-intro}$$

$$\frac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Leftrightarrow\text{-élim}$$

$$\frac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash B}{\Gamma \vdash A} \Leftrightarrow\text{-élim}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{-intro}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{-élim}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{-élim}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee\text{-intro}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{-intro}$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee\text{-élim}$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg\text{-intro}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot} \neg\text{-élim}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot\text{-élim}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A} \forall\text{-intro} \quad \text{si } x \text{ n'apparaît pas dans } \Gamma$$

$$\frac{\Gamma \vdash \forall x\ A}{\Gamma \vdash A[x \leftarrow t]} \forall\text{-élim}$$

$$\frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\ A} \exists\text{-intro}$$

$$\frac{\Gamma \vdash \exists x\ A \quad \Gamma, A \vdash B}{\Gamma \vdash B} \exists\text{-élim} \quad \text{si } x \text{ n'apparaît pas dans } \Gamma \text{ ni dans } B$$

En logique classique, on ajoute la règle

$$\frac{}{\Gamma \vdash A \vee \neg A} \text{tiers exclu}$$

Proposition Si A est démontrable à partir des axiomes Γ dans le système du chapitre précédent, alors $\Gamma \vdash A$ est démontrable en déduction naturelle.

2.4 Une définition indépendante des connecteurs ?

Un seul connecteur par règle

Le déduction naturelle fait presque apparaître un seul connecteur par règle. Seules les règles de la négation et le tiers exclu dérogent à ce principe.

Les règles de la négation font intervenir explicitement la contradiction $\perp$ on peut remplacer ces règles par les règles suivantes :

$$\frac{\Gamma, A \vdash B \quad \Gamma, A \vdash \neg B}{\Gamma \vdash \neg A}\,\neg\text{-intro}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash B}\,\neg\text{-élim}$$

le seul inconvénient de ces règles étant d'avoir une occurrence de la négation à la fois dans les prémisses et la conclusion de la règle d'introduction de la négation.

Le tiers exclu peut, quant à lui, être remplacé par la règle

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\,\text{tiers exclu}$$

Il devient une règle supplémentaire d'élimination de la négation.

Les connecteurs apparaissant virtuellement dans les métavariables

Les règles de déduction expriment la signification des connecteurs. Pouvoir déduire B de $A \Rightarrow B$ et A fait partie de la signification de l'implication.

Le système de déduction naturelle ci-dessus, qui fait apparaître un seul connecteur par règle semble donner une définition indépendante de chacun des connecteurs. Cela ne signifie pas pour autant qu'une proposition n'utilisant qu'un ensemble restreint de connecteurs puisse toujours se prouver uniquement avec les règles concernant ces connecteurs. Par exemple la proposition

$$((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$$

(proposition de Peirce) ne peut pas être démontrée sans utiliser le tiers exclu, bien que la négation n'apparaisse pas dans la proposition elle-même. En effet, on commence par démontrer la proposition

$$\neg\neg(((P \Rightarrow Q) \Rightarrow P) \Rightarrow P)$$

avant d'en déduire par le tiers exclu

$$((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$$

Donc, même si elle n'apparaît pas explicitement dans la formulation de la règle *tiers exclu*, l'implication apparaît dans l'instance utilisée

$$\frac{\Gamma \vdash \neg\neg(((P \Rightarrow Q) \Rightarrow P) \Rightarrow P)}{\Gamma \vdash ((P \Rightarrow Q) \Rightarrow P) \Rightarrow P}$$

Et donc, le tiers exclu participe à la signification de l'implication.

Dans le calcul des séquents, nous verrons que si une proposition peut être démontrée, alors elle peut être démontrée en utilisant uniquement les règles concernant les connecteurs présents dans cette proposition.

2.5 Les coupures

2.5.1 La notion de coupure

Définition Une coupure est une démonstration dont la dernière règle est une règle d'élimination d'un symbole, dont la prémisse principale (c'est-à-dire la prémisse dans laquelle ce symbole apparaît) est démontrée par une règle d'introduction de ce symbole.

Exemple

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \quad \dfrac{\pi_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}}{\Gamma \vdash A}\ \wedge\text{-élim}$$

Dans cet exemple, il va de soi que, si notre but est de démontrer A, il est inutile de démontrer A et B pour en déduire $A \wedge B$ puis A.

On peut donc éliminer cette coupure et donner la preuve plus simple du même séquent

$$\dfrac{\pi_1}{\Gamma \vdash A}$$

De même, si on démontre une proposition A dans le cas général, pour en déduire $\forall x\ A$ puis $A[x \leftarrow t]$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash \forall x\ A}\ \forall\text{-intro}}{\Gamma \vdash A[x \leftarrow t]}\ \forall\text{-élim}$$

On peut éliminer cette coupure et transformer cette preuve en

$$\dfrac{\pi[x \leftarrow t]}{\Gamma \vdash A[x \leftarrow t]}$$

dans laquelle on démontre directement la proposition A dans le cas particulier où $x = t$.

Troisième exemple, si on démontre B sous les hypothèses Γ, A, pour en déduire $A \Rightarrow B$, et qu'on démontre A par ailleurs pour en déduire B

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-intro} \qquad \dfrac{\pi_2}{\Gamma \vdash A}}{\Gamma \vdash B}\ \Rightarrow\text{-élim}$$

On peut éliminer cette coupure et transformer cette preuve en

$$\dfrac{\pi}{\Gamma \vdash B}$$

où π est la preuve obtenue en supprimant dans π_1 les occurrences de A dans tous les contextes et en remplaçant les utilisations de la règle axiome $\Gamma, A, \Delta \vdash A$ par la preuve de $\Gamma, \Delta \vdash A$ obtenue en rajoutant les hypothèses Δ à chaque séquent de π_2.

Par exemple

$$\dfrac{\dfrac{A, A \Rightarrow B, B \vdash B}{A, A \Rightarrow B \vdash B \Rightarrow B}\ \Rightarrow\text{-intro} \qquad \dfrac{A, A \Rightarrow B \vdash A \quad A, A \Rightarrow B \vdash A \Rightarrow B}{A, A \Rightarrow B \vdash B}}{A, A \Rightarrow B \vdash B}\ \Rightarrow\text{-élim}$$

se transforme en

$$\frac{A, A \Rightarrow B \vdash A \quad A, A \Rightarrow B \vdash A \Rightarrow B}{A, A \Rightarrow B \vdash B}$$

On peut donner une transformation similaire pour chacun des connecteurs.

À chaque fois qu'une preuve contient un sous-arbre qui est une coupure, on peut éliminer cette coupure. La question est de savoir si, en itérant ce processus, on aboutit à une preuve sans coupures, ou s'il peut se poursuivre à l'infini. Le *théorème d'élimination des coupures* (voir chapite 6) montre précisément que ce processus termine toujours et donc que toute preuve peut se transformer en une preuve sans coupures.

2.5.2 Les démonstrations sans coupures

Proposition En logique intuitionniste, une preuve sans coupures et sans hypothèses se termine par une règle d'introduction.

Démonstration Par récurrence sur la taille de la preuve. Soit π une preuve sans coupures et sans hypothèses. La dernière règle de cette preuve ne peut pas être la règle d'axiome, car le contexte Γ est vide. Si c'est une règle d'élimination, la prémisse principale de cette règle a une preuve π' plus petite que π. Par hypothèse de récurrence π' se termine par une règle d'introduction. Cette règle d'introduction, forme une coupure avec la dernière règle de π. Cette dernière règle de π ne peut donc pas être une règle d'élimination. C'est donc une règle d'introduction.

Proposition Si une proposition de la forme $A \vee B$ peut être démontrée

- sans hypothèses,
- en logique intuitionniste

alors la proposition A ou la proposition B peut être démontrée.

Démonstration Si la proposition $A \vee B$ peut être démontrée, alors d'après le théorème d'élimination des coupures, elle a une preuve sans coupures. Cette preuve se termine par une règle d'introduction, cette règle ne peut être que $\vee$-intro et donc la proposition A ou la proposition B peut être démontrée.

Remarque Les deux hypothèses de cette proposition sont nécessaires. En effet avec l'hypothèse $A \vee B$ on peut montrer $A \vee B$ sans montrer A ni B. Avec le tiers exclu on peut montrer $A \vee \neg A$ sans montrer A ni $\neg A$.

Proposition Si une proposition de la forme $\exists x\ A$ peut être démontrée

- sans hypothèses,
- en logique intuitionniste

alors il y a un terme t tel que la proposition $A[x \leftarrow t]$ soit démontrable.

Démonstration Une preuve sans coupures de $\exists x\ A$ se termine nécessairement par la règle $\exists$-intro.

2.5.3 Les coupures avec des axiomes

Quand on démontre une proposition en utilisant des axiomes (par exemple dans l'arithmétique) on peut étendre la notion de coupure de manière à intégrer des coupures liées à ces axiomes. Par exemple, si on a démontré en arithmétique les propositions $P(0)$ et $\forall x\ (P(x) \Rightarrow (P(S(x))))$ on peut en déduire avec le schéma de récurrence la proposition $\forall x\ P(x)$, de quoi on peut déduire $P(100)$. Eliminer les coupures dans cette preuve revient à démontrer successivement $P(1)$, $P(2)$, $P(3)$, $P(4)$, $P(5)$, $P(6)$, $P(7)$, $P(8)$, $P(9)$, $P(10)$, $P(11)$, $P(12)$, $P(13)$, $P(14)$, $P(15)$, $P(16)$, $P(17)$, $P(18)$, $P(19)$, $P(20)$, $P(21)$, $P(22)$, $P(23)$, $P(24)$, $P(25)$, $P(26)$, $P(27)$, $P(28)$, $P(29)$, $P(30)$, $P(31)$, $P(32)$, $P(33)$, $P(34)$, $P(35)$, $P(36)$, $P(37)$, $P(38)$, $P(39)$, $P(40)$, $P(41)$, $P(42)$, $P(43)$, $P(44)$, $P(45)$, $P(46)$, $P(47)$, $P(48)$, $P(49)$, $P(50)$, $P(51)$, $P(52)$, $P(53)$, $P(54)$,

$P(55)$, $P(56)$, $P(57)$, $P(58)$, $P(59)$, $P(60)$, $P(61)$, $P(62)$, $P(63)$, $P(64)$, $P(65)$, $P(66)$, $P(67)$, $P(68)$, $P(69)$, $P(70)$, $P(71)$, $P(72)$, $P(73)$, $P(74)$, $P(75)$, $P(76)$, $P(77)$, $P(78)$, $P(79)$, $P(80)$, $P(81)$, $P(82)$, $P(83)$, $P(84)$, $P(85)$, $P(86)$, $P(87)$, $P(88)$, $P(89)$, $P(90)$, $P(91)$, $P(92)$, $P(93)$, $P(94)$, $P(95)$, $P(96)$, $P(97)$, $P(98)$, $P(99)$ et enfin $P(100)$.

Chapitre 3

Le calcul des séquents

Le calcul des séquents est une troisième formulation de la déduction (après les systèmes à la Frege-Hilbert et la déduction naturelle). Le calcul des séquents est un système moins intuitif que la déduction naturelle, mais il présente d'autres avantages : il est utile pour la recherche automatique de preuves, il est très régulier et l'idée qu'une hypothèse peut être utilisée zéro, une ou plusieurs fois y apparaît explicitement dans les règles de déduction.

3.1 Motivations

3.1.1 La recherche de démonstrations

Supposons que nous cherchions à démontrer le séquent $P, Q \vdash P \wedge Q$. Une preuve en déduction naturelle de ce séquent est obtenue en appliquant la règle d'introduction de la conjonction

$$\frac{P, Q \vdash P \quad P, Q \vdash Q}{P, Q \vdash P \wedge Q} \wedge\text{-intro}$$

Cette preuve est relativement facile à trouver : la proposition à démontrer est une conjonction, donc il se peut que la dernière règle de la preuve soit une introduction de la conjonction. Si on essaie cette règle, on se ramène aux séquents $P, Q \vdash P$ et $P, Q \vdash Q$ dont la conclusion figure dans la liste des hypothèses et qui sont donc démontrable par la règle *axiome*.

Supposons maintenant que nous cherchions à démontrer le séquent $P \wedge Q \vdash P$. Une preuve en déduction naturelle de ce séquent est obtenue en appliquant la règle d'élimination de la conjonction

$$\frac{P \wedge Q \vdash P \wedge Q}{P \wedge Q \vdash P} \wedge\text{-élim}$$

Cette preuve est beaucoup plus difficile à deviner. D'abord la forme de la proposition à démontrer P ne nous suggère en rien la règle d'élimination de la conjonction. Ensuite, même si on décide d'utiliser cette règle

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{-élim}$$

Le séquent $P \wedge Q \vdash P$ nous suggère de prendre $P \wedge Q$ pour Γ et P pour A, mais ne suggère rien pour B qui n'apparaît pas pas la conclusion de la règle.

Il y a ici une dissymétrie profonde entre les règles d'introduction

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow\text{-intro}$$

$$\frac{\Gamma, A \vdash B \quad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B} \Leftrightarrow\text{-intro}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{-intro}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee\text{-intro}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{-intro}$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg\text{-intro}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A} \forall\text{-intro} \quad \text{si } x \text{ n'apparaît pas dans } \Gamma$$

$$\frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\ A} \exists\text{-intro}$$

et les règles d'élimination

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow\text{-élim}$$

$$\frac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Leftrightarrow\text{-élim}$$

$$\frac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash B}{\Gamma \vdash A} \Leftrightarrow\text{-élim}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{-élim}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{-élim}$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee\text{-élim}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot} \neg\text{-élim}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot\text{-élim}$$

$$\frac{\Gamma \vdash \forall x\ A}{\Gamma \vdash A[x \leftarrow t]} \forall\text{-élim}$$

$$\frac{\Gamma \vdash \exists x\ A \quad \Gamma, A \vdash B}{\Gamma \vdash B} \exists\text{-élim} \quad \text{si } x \text{ n'apparaît pas dans } \Gamma \text{ ni dans } B$$

A chaque étape de la recherche, une règle unique d'introduction, celle du connecteur principal de la proposition à démontrer, peut être utilisée (sauf dans le cas de la disjonction qui a deux règles d'introduction) et il y a en général une seule possibilité pour instancier les métavariables de la règle (sauf dans le cas du quantificateur existentiel, car il faut choisir le terme t). Alors, que à chaque étape de la recherche, quasiment toutes les règles d'élimination sont utilisables et souvent de nombreuses manières.

3.1.2 Les règles gauches

Pour deviner que pour démontrer le séquent $P \wedge Q \vdash P$, il faut utiliser la règle d'élimination de la conjonction avec la proposition Q pour B, il faut utiliser des informations qui se trouvent dans les hypothèses du séquent.

L'idée du calcul des séquents est de conserver les règles d'introduction de la déduction naturelle et de remplacer les règles d'élimination par des règles d'introduction sur les hypothèses du séquent. Un exemple d'une telle règle est

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}\ \wedge\text{-gauche}$$

Par exemple, pour démontrer avec cette règle le séquent $P \wedge Q \vdash P$, il suffit d'appliquer cette règle pour obtenir le séquent $P, Q \vdash P$ dont la conclusion figure parmi les hypothèses.

Dans la preuve en déduction naturelle,

$$\frac{P \wedge Q \vdash P \wedge Q}{P \wedge Q \vdash P}\ \wedge\text{-élim}$$

la règle *axiome* est utilisée avec la proposition $P \wedge Q$. Alors que dans la preuve en calcul des séquents

$$\frac{P, Q \vdash P}{P \wedge Q \vdash P}\ \wedge\text{-gauche}$$

la règle *axiome* est utilisée avec la proposition P.

3.1.3 La règle de coupure

Dans la preuve en déduction naturelle

$$\frac{P \wedge Q \vdash P \wedge Q}{P \wedge Q \vdash P}\ \wedge\text{-élim}$$

dans laquelle l'utilisation de la règle d'élimination est immédiatement suivie d'une règle *axiome*, on sait que la proposition $P \wedge Q$ fait partie de la liste d'hypothèses du séquent, et donc qu'on peut, en calcul des séquents utiliser la règle gauche pour décomposer l'hypothèse $P \wedge Q$ en deux hypothèses P et Q.

En revanche si la proposition $P \wedge Q$ est démontrée par une preuve π quelconque

$$\frac{\dfrac{\pi}{\Gamma \vdash P \wedge Q}}{\Gamma \vdash P}\ \wedge\text{-élim}$$

il n'est plus aussi simple d'exprimer cette preuve en calcul des séquents. Pour cela on ajoute une nouvelle règle

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}$$

de manière à pouvoir par exemple justifier par la preuve π le fait d'ajouter la proposition $P \wedge Q$ parmi les hypothèses du séquent pour pouvoir appliquer la règle $\wedge$-gauche.

$$\frac{\dfrac{\pi}{\Gamma \vdash P \wedge Q} \quad \dfrac{\Gamma, P, Q \vdash P}{\Gamma, P \wedge Q \vdash P}\ \wedge\text{-gauche}}{\Gamma \vdash P}$$

Cette règle est appelée *règle de coupure*, par analogie avec la notion de coupure sur l'implication en déduction naturelle puisque dans un cas comme dans l'autre, une preuve qui comporte une coupure démontre une proposition A et indépendamment une proposition B sous l'hypothèse A pour en déduire B.

Malgré cette analogie, il faut remarquer qu'en calcul des séquents il y a une règle de coupure, alors qu'en déduction naturelle une coupure est simplement une succession d'une introduction et d'une élimination.

3.1.4 Les règles structurelles

En déduction naturelle, comme en calcul des séquents on peut remplacer la règle

$$\frac{}{\Gamma \vdash A}\text{axiome} \quad \text{si } A \in \Gamma$$

par une règle

$$\frac{}{A \vdash A}\text{axiome}$$

ainsi que deux règles permettant d'effacer et permuter des hypothèses d'un séquent

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}\text{affaiblissement}$$

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\text{permutation}$$

En calcul des séquents, on doit en outre ajouter une règle permettant de faire une copie d'une hypothèse avant de l'utiliser, pour pouvoir l'utiliser une seconde fois.

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\text{contraction}$$

3.2 Le calcul des séquents

3.2.1 Définition

$$\frac{}{A \vdash A}\text{axiome}$$

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\text{coupure}$$

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}\text{affaiblissement}$$

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\text{contraction}$$

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\text{permutation}$$

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C}\Rightarrow\text{-gauche}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\Rightarrow\text{-droite}$$

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Leftrightarrow B \vdash C}\Leftrightarrow\text{-gauche}$$

$$\frac{\Gamma \vdash B \quad \Gamma, A \vdash C}{\Gamma, A \Leftrightarrow B \vdash C}\Leftrightarrow\text{-gauche}$$

$$\frac{\Gamma, A \vdash B \quad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B}\Leftrightarrow\text{-droite}$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \wedge\text{-gauche}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{-droite}$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C} \vee\text{-gauche}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee\text{-droite}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{-droite}$$

$$\frac{\Gamma \vdash A}{\Gamma, \neg A \vdash B} \neg\text{-gauche}$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg\text{-droite}$$

$$\frac{}{\Gamma, \bot \vdash A} \bot\text{-gauche}$$

$$\frac{\Gamma, A[x \leftarrow t] \vdash B}{\Gamma, \forall x\ A \vdash B} \forall\text{-gauche}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A} \forall\text{-droite} \quad \text{si } x \text{ n'apparaît pas dans } \Gamma$$

$$\frac{\Gamma A \vdash B}{\Gamma, \exists x\ A \vdash B} \exists\text{-gauche} \quad \text{si } x \text{ n'apparaît pas dans } \Gamma \text{ ni dans } B$$

$$\frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\ A} \exists\text{-droite}$$

3.2.2 Le calcul des séquents classique

En déduction naturelle, le tiers exclu est une règle spéciale qui s'ajoute aux règles d'introduction de la disjonction $\vee$:

$$\frac{}{\Gamma \vdash A \vee \neg A}$$

on peut aussi le remplacer par la règle

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}$$

En calcul des séquents aussi, le tiers exclu peut se formuler comme une règle spéciale. Mais il y a une autre manière, bien plus élégante de l'exprimer.

Essayons de démontrer le séquent $\neg\neg(P \Rightarrow Q), P \vdash Q$. Si on applique la règle gauche de la négation on se ramène au séquent $P \vdash \neg(P \Rightarrow Q)$ qui n'est pas démontrable. L'idée est alors d'appliquer le tiers exclu pour se ramener à $\neg\neg(P \Rightarrow Q), P \vdash \neg\neg Q$ puis d'appliquer la règle droite de la négation de manière à obtenir $\neg\neg(P \Rightarrow Q), P, \neg Q \vdash \bot$ puis d'appliquer enfin la règle gauche de la négation comme ci-dessus. On obtient alors la démonstration suivante (qui se comprend mieux si on la lit de bas en haut, c'est-à-dire en partant de la conclusion).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}\,\text{axiome} \qquad \dfrac{}{P, Q \vdash Q}\,\text{axiome}}{P, P \Rightarrow Q \vdash Q}\,\Rightarrow\text{-gauche}}{P, \neg Q, P \Rightarrow Q \vdash \bot}\,\neg\text{-gauche}}{P, \neg Q \vdash \neg(P \Rightarrow Q)}\,\neg\text{-droite}}{\neg\neg(P \Rightarrow Q), P, \neg Q \vdash \bot}\,\neg\text{-gauche}}{\neg\neg(P \Rightarrow Q), P \vdash \neg\neg Q}\,\neg\text{-droite}}{\neg\neg(P \Rightarrow Q), P \vdash Q}\,\text{tiers exclu}$$

On peut comprendre l'utilisation du tiers exclu dans cette démonstration, comme un moyen d'éviter la destruction de la proposition Q lors de l'utilisation de la règle gauche de la négation, en stockant sa négation dans la liste d'hypothèses. Une idée alternative consiste à laisser la proposition Q dans la partie droite du séquent. Cela demande donc de considérer des séquents étendus dans lesquels il y a plusieurs hypothèses *et aussi plusieurs conclusions*. Ainsi la démonstration ci-dessus peut se réécrire

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P, Q}\,\text{axiome} \qquad \dfrac{}{P, Q \vdash Q}\,\text{axiome}}{P, P \Rightarrow Q \vdash Q}\,\Rightarrow\text{-gauche}}{P \vdash \neg(P \Rightarrow Q), Q}\,\neg\text{-droite}}{\neg\neg(P \Rightarrow Q), P \vdash Q}\,\neg\text{-gauche}$$

Le tiers exclu peut donc s'exprimer par cette possibilité d'avoir plusieurs conclusions dans un séquent (cette idée, habituelle en calcul des séquents, peut aussi être utilisée en déduction naturelle).

Il est également possible d'avoir zéro proposition en conclusion, dans ce cas il faut interpréter le séquent $\Gamma \vdash$ comme $\Gamma \vdash \bot$. De ce fait, on peut se passer du symbole $\bot$.

Le tiers exclu se démontre alors ainsi.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{\vdash A, \neg A}\,\neg\text{-droite}}{\vdash A, A \vee \neg A}\,\vee\text{-droite}}{\vdash A \vee \neg A, A}\,\text{permutation-droite}}{\vdash A \vee \neg A, A \vee \neg A}\,\vee\text{-droite}}{\vdash A \vee \neg A}\,\text{contraction-droite}$$

3.2.3 La logique linéaire

Le calcul des séquents est le seul formalisme dans lequel l'idée qu'une hypothèse puisse s'utiliser zéro, une ou plusieurs fois est explicitement exprimée par des règles de déduction (affaiblissement et contraction).

Cette permanence des hypothèses n'est pas vérifiée dans des situation dynamiques dans lesquelles les hypothèses s'usent quand on s'en sert. Par exemple des hypothèses "si j'ai deux francs, je peux acheter un café" et "si j'ai deux francs, je peux acheter un chocolat" on peut déduire la conséquence "paradoxale" "si j'ai deux francs, je peux acheter un café et je peux acheter un chocolat" en dupliquant l'hypothèse "j'ai deux francs" par la règle de contraction.

La *logique linéaire* [5] est la formulation de la déduction obtenue en supprimant les règles d'affaiblissement et de contraction dans le calcul des séquents classique.

En logique linéaire, il faut également être prudent avec les contractions implicites dans les règles de déduction. Par exemple, il importe de distinguer la règle droite de la conjonction

$$\dfrac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$$

où chacune des hypothèses de Γ est utilisée exactement une fois pour montrer A et exactement une fois pour montrer B, de la règle

$$\frac{\Gamma \vdash \Delta, A \quad \Gamma' \vdash \Delta', B}{\Gamma, \Gamma' \vdash \Delta, \Delta', A \wedge B}$$

dans laquelle chaque hypothèse de Γ, Γ' est utilisée exactement une fois ou bien pour montrer A ou bien pour montrer B.

Cela amène en fait à distinguer deux conjonctions différentes $\otimes$ et & dont les règles droites sont

$$\frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \otimes B}\ \otimes\text{-droite}$$

et

$$\frac{\Gamma \vdash \Delta, A \quad \Gamma' \vdash \Delta', B}{\Gamma, \Gamma' \vdash \Delta, \Delta', A \& B}\ \&\text{-droite}$$

En informatique, la logique linéaire permet de rendre compte de phénomènes dynamiques (affectation, interaction de processus, etc.). En logique, c'est un outil très puissant pour étudier les déductions intuitionnistes et classique.

3.3 Traductions

3.3.1 Du calcul des séquents en déduction naturelle

Proposition Tous les séquents d'une preuve en déduction naturelle de $\Gamma \vdash A$ ont la forme $\Gamma, \Delta \vdash B$.
Démonstration Par récurrence sur la structure de la preuve de $\Gamma \vdash A$.

Proposition Tout séquent $\Gamma \vdash A$ qui a une démonstration Π en calcul des séquents, a une démonstration en déduction naturelle.
Démonstration Par récurrence sur la structure de Π. Le séquent $\Gamma \vdash A$ étiquette la racine de Π. Cette racines a un certain nombres de fils $\pi_1, ..., \pi_n$, dont les racines sont étiquetées par des séquents $s_1, ..., s_n$, et le séquent $\Gamma \vdash A$ peut se déduire de $s_1, ..., s_n$ par une règle du calcul des séquents.

Par hypothèse de récurrence, les séquents s_n ont des démonstrations $\pi'_1, ..., \pi'_n$ en déduction naturelle. En fonction de la règle du calcul des séquents utilisée, on construit une preuve en déduction naturelle Π' du séquent $\Gamma \vdash A$.

- Une preuve de la forme

  $$\frac{}{A \vdash A}\ \text{axiome}$$

  se traduit en

  $$\frac{}{A \vdash A}\ \text{axiome}$$

- Une preuve de la forme

  $$\frac{\dfrac{\pi_1}{\Gamma \vdash A} \quad \dfrac{\pi_2}{\Gamma, A \vdash B}}{\Gamma \vdash B}\ \text{coupure}$$

  se traduit en

  $$\frac{\pi'_2 +}{\Gamma \vdash B}$$

où π'_2+ est obtenue en supprimant la proposition A dans les hypothèses de tous les séquents de π'_2 et en remplaçant tous les axiomes $\Gamma, \Delta \vdash A$ par

$$\frac{\pi'_1+}{\Gamma, \Delta \vdash A}$$

où π'_1+ est obtenue en ajoutant Δ aux hypothèses de tous les séquents de π'_1.

- Une preuve de la forme

$$\frac{\frac{\pi}{\Gamma \vdash A}}{\Gamma, B \vdash A} \text{ affaiblissement}$$

se traduit en

$$\frac{\pi'+}{\Gamma, B \vdash A}$$

où $\pi'+$ est obtenue en ajoutant la proposition B dans les hypothèses de tous les séquents de π'.

- Une preuve de la forme

$$\frac{\frac{\pi}{\Gamma, B, B \vdash A}}{\Gamma, B \vdash A} \text{ contraction}$$

se traduit en

$$\frac{\pi'+}{\Gamma, B \vdash A}$$

où $\pi'+$ est obtenue en supprimant l'une des occurrences de la proposition B dans les hypothèses de tous les séquents de π'.

- Une preuve de la forme

$$\frac{\frac{\pi}{\Gamma, B, A, \Delta \vdash C}}{\Gamma, A, B, \Delta \vdash C} \text{ permutation}$$

se traduit en

$$\frac{\pi'+}{\Gamma, A, B, \Delta \vdash A}$$

où $\pi'+$ est obtenue en permutant les occurrences de A et B dans les hypothèses de tous les séquents de π'.

- Une preuve de la forme

$$\frac{\frac{\pi_1}{\Gamma \vdash A} \quad \frac{\pi_2}{\Gamma, B \vdash C}}{\Gamma, A \Rightarrow B \vdash C} \Rightarrow \text{-gauche}$$

se traduit en

$$\frac{\pi'_2+}{\Gamma, A \Rightarrow B \vdash C}$$

où π'_2+ est obtenue en supprimant dans les hypothèses de tous les séquents de π'_2 la proposition B, en y ajoutant $A \Rightarrow B$ et en remplaçant tous les axiomes de la forme $\Gamma, A \Rightarrow B, \Delta \vdash B$ par

$$\frac{\Gamma, A \Rightarrow B, \Delta \vdash A \Rightarrow B \quad \frac{\pi'_1+}{\Gamma, A \Rightarrow B, \Delta \vdash A}}{\Gamma, A \Rightarrow B, \Delta \vdash B} \Rightarrow \text{-élim}$$

où π'_1+ est obtenu en ajoutant $A \Rightarrow B, \Delta$ aux hypothèses de tous les séquents de π'_1.

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \quad \dfrac{\pi_2}{\Gamma, B \vdash C}}{\Gamma, A \Leftrightarrow B \vdash C} \Leftrightarrow \text{-gauche}$$

se traduit en

$$\dfrac{\pi_2'+}{\Gamma, A \Leftrightarrow B \vdash C}$$

où $\pi_2'+$ est obtenue en supprimant dans les hypothèses de tous les séquents de π_2' la proposition B, en y ajoutant $A \Leftrightarrow B$ et en remplaçant tous les axiomes de la forme $\Gamma, A \Leftrightarrow B, \Delta \vdash B$ par

$$\dfrac{\Gamma, A \Leftrightarrow B, \Delta \vdash A \Leftrightarrow B \quad \dfrac{\pi_1'+}{\Gamma, A \Leftrightarrow B, \Delta \vdash A}}{\Gamma, A \Leftrightarrow B, \Delta \vdash B} \Leftrightarrow \text{-élim}$$

où $\pi_1'+$ est obtenu en ajoutant $A \Leftrightarrow B, \Delta$ aux hypothèses de tous les séquents de π_1'.

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash B} \quad \dfrac{\pi_2}{\Gamma, A \vdash C}}{\Gamma, A \Leftrightarrow B \vdash C} \Leftrightarrow \text{-gauche}$$

se traduit en

$$\dfrac{\pi_2'+}{\Gamma, A \Leftrightarrow B \vdash C}$$

où $\pi_2'+$ est obtenue en supprimant dans les hypothèses de tous les séquents de π_2' la proposition A, en y ajoutant $A \Leftrightarrow B$ et en remplaçant tous les axiomes de la forme $\Gamma, A \Leftrightarrow B, \Delta \vdash A$ par

$$\dfrac{\Gamma, A \Leftrightarrow B, \Delta \vdash A \Leftrightarrow B \quad \dfrac{\pi_1'+}{\Gamma, A \Leftrightarrow B, \Delta \vdash B}}{\Gamma, A \Leftrightarrow B, \Delta \vdash A} \Leftrightarrow \text{-élim}$$

où $\pi_1'+$ est obtenu en ajoutant $A \Leftrightarrow B, \Delta$ aux hypothèses de tous les séquents de π_1'.

- Une preuve de la forme

$$\dfrac{\dfrac{\pi}{\Gamma, A, B \vdash C}}{\Gamma, A \wedge B \vdash C} \wedge\text{-gauche}$$

se traduit en

$$\dfrac{\pi'+}{\Gamma, A \wedge B \vdash C}$$

où $\pi'+$ est obtenue en supprimant dans les hypothèses de tous les séquents de π' les propositions A et B, en y ajoutant $A \wedge B$ et en remplaçant tous les axiomes de la forme $\Delta \vdash A$ par

$$\dfrac{\Delta \vdash A \wedge B}{\Delta \vdash A} \wedge\text{-élim}$$

et tous les axiomes de la forme $\Delta \vdash B$ par

$$\dfrac{\Delta \vdash A \wedge B}{\Delta \vdash B} \wedge\text{-élim}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma, A \vdash C} \quad \dfrac{\pi_2}{\Gamma, B \vdash C}}{\Gamma, A \vee B \vdash C} \vee\text{-gauche}$$

se traduit en

$$\dfrac{\Gamma, A \vee B \vdash A \vee B \quad \dfrac{\pi_1'+}{\Gamma, A \vee B, A \vdash C} \quad \dfrac{\pi_2'+}{\Gamma, A \vee B, B \vdash C}}{\Gamma, A \vee B \vdash C} \vee\text{-élim}$$

où $\pi_1'+$ est obtenue en ajoutant $A \vee B$ dans les hypothèses de tous les séquents de π_1' et $\pi_2'+$ est obtenue en ajoutant $A \vee B$ dans les hypothèses de tous les séquents de π_2'.

- Une preuve de la forme

$$\dfrac{\dfrac{\pi}{\Gamma \vdash A}}{\Gamma, \neg A \vdash B} \neg\text{-gauche}$$

se traduit en

$$\dfrac{\dfrac{\Gamma, \neg A \vdash \neg A \quad \dfrac{\pi'+}{\Gamma, \neg A \vdash A}}{\Gamma, \neg A \vdash \bot} \neg\text{-élim}}{\Gamma, \neg A \vdash B} \bot\text{-élim}$$

où $\pi'+$ est obtenue en ajoutant $\neg A$ dans les hypothèses de tous les séquents de π'.

- Une preuve de la forme

$$\dfrac{}{\Gamma, \bot \vdash A} \bot\text{-gauche}$$

se traduit en

$$\dfrac{\Gamma, \bot \vdash \bot}{\Gamma, \bot \vdash A} \bot\text{-élim}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi}{\Gamma, A[x \leftarrow t] \vdash B}}{\Gamma, \forall x\ A \vdash B} \forall\text{-gauche}$$

se traduit en

$$\dfrac{\pi'+}{\Gamma, \forall x\ A \vdash B}$$

où $\pi'+$ est obtenue en supprimant dans les hypothèses de tous les séquents de π la proposition $A[x \leftarrow t]$, en y ajoutant $\forall x\ A$ et en remplaçant tous les axiomes de la forme $\Delta \vdash A[x \leftarrow t]$ par

$$\dfrac{\Delta \vdash \forall x\ A}{\Delta \vdash A[x \leftarrow t]} \forall\text{-élim}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi}{\Gamma, A \vdash B}}{\Gamma, \exists x\ A \vdash B} \exists\text{-gauche}$$

se traduit en

$$\dfrac{\Gamma, \exists x\ A \vdash \exists x\ A \quad \dfrac{\pi'+}{\Gamma, \exists x\ A, A \vdash B}}{\Gamma, \exists x\ A \vdash B} \exists\text{-gauche}$$

où $\pi'+$ est obtenue en ajoutant dans les hypothèses de tous les séquents de π' la proposition $\exists x\ A$.

- Les règles droites sont traduites par les règles d'introduction.

Proposition Une preuve dans le calcul des séquents qui n'utilise pas la règle de coupure est traduite en déduction naturelle en une preuve sans coupures.
Démonstration Par récurrence sur la structure de la preuve.

Exercice Schroeder-Heister [7] propose (entre autres) de remplacer les règles de déduction naturelle $\wedge$-élim et $\forall$-élim par les règles suivantes :

$$\frac{\Gamma \vdash A \wedge B \quad \Gamma, A, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash \forall x\ A \quad \Gamma, A[x \leftarrow t] \vdash B}{\Gamma \vdash B}$$

Montrer que le système obtenu est équivalent à la déduction naturelle. En quoi cela simplifie-t-il la traduction du calcul des séquents en déduction naturelle ?

Kleene [6] propose de modifier les règles gauches du calcul des séquents en gardant dans les prémisses une copie de la proposition de la conclusion décomposée. Par exemple

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \wedge\text{-gauche}$$

est remplacée par

$$\frac{\Gamma, A \wedge B, A, B \vdash C}{\Gamma, A \wedge B \vdash C}$$

Montrer que le système obtenu est équivalent au calcul des séquents. En quoi cela simplifie-t-il la traduction du calcul des séquents en déduction naturelle ?

Quelles sont les seules règles donnant lieu à un traitement global de la preuve quand on traduit le calcul des séquents modifié par Kleene dans la déduction naturelle modifiée par Schroeder-Heister ?

3.3.2 De la déduction naturelle en calcul des séquents

Proposition Si A appartient à la liste Γ alors $\Gamma \vdash A$ est démontrable en calcul des séquents.
Démonstration Si Γ a la forme $B_1, ..., B_p, A, C_1, ..., C_q$

On construit la preuve

$$\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{...}\text{affaiblissement}}{A, B_1, ..., B_p, C_1, ..., C_q \vdash A}\text{affaiblissement}}{...}\text{permutation}}{B_1, ..., B_p, A, C_1, ..., C_q \vdash A}\text{permutation}$$

Proposition Tout séquent démontrable en déduction naturelle est démontrable en calcul des séquents.
Démonstration

- Une preuve de la forme

$$\frac{}{\Gamma \vdash A}\text{axiome}$$

  se traduit en

$$\frac{...}{\Gamma \vdash A}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash A \Rightarrow B} \quad \dfrac{\pi_2}{\Gamma \vdash A}}{\Gamma \vdash B} \Rightarrow \text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi_1'}{\Gamma \vdash A \Rightarrow B} \quad \dfrac{\dfrac{\pi_2'}{\Gamma \vdash A} \quad \dfrac{...}{\Gamma, B \vdash B}}{\Gamma, A \Rightarrow B \vdash B} \Rightarrow \text{-gauche}}{\Gamma \vdash B} \text{coupure}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash A \Leftrightarrow B} \quad \dfrac{\pi_2}{\Gamma \vdash A}}{\Gamma \vdash B} \Leftrightarrow \text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi_1'}{\Gamma \vdash A \Leftrightarrow B} \quad \dfrac{\dfrac{\pi_2'}{\Gamma \vdash A} \quad \dfrac{...}{\Gamma, B \vdash B}}{\Gamma, A \Leftrightarrow B \vdash B} \Leftrightarrow \text{-gauche}}{\Gamma \vdash B} \text{coupure}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash A \Leftrightarrow B} \quad \dfrac{\pi_2}{\Gamma \vdash B}}{\Gamma \vdash A} \Leftrightarrow \text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi_1'}{\Gamma \vdash A \Leftrightarrow B} \quad \dfrac{\dfrac{\pi_2'}{\Gamma \vdash B} \quad \dfrac{...}{\Gamma, A \vdash A}}{\Gamma, A \Leftrightarrow B \vdash A} \Leftrightarrow \text{-gauche}}{\Gamma \vdash A} \text{coupure}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A} \wedge\text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi'}{\Gamma \vdash A \wedge B} \quad \dfrac{\dfrac{...}{\Gamma, A, B \vdash A}}{\Gamma, A \wedge B \vdash A} \wedge\text{-gauche}}{\Gamma \vdash A} \text{coupure}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash A \vee B} \quad \dfrac{\pi_2}{\Gamma, A \vdash C} \quad \dfrac{\pi_3}{\Gamma, B \vdash C}}{\Gamma \vdash C} \vee\text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi_1'}{\Gamma \vdash A \vee B} \quad \dfrac{\dfrac{\pi_2'}{\Gamma, A \vdash C} \quad \dfrac{\pi_3'}{\Gamma, B \vdash C}}{\Gamma, A \vee B \vdash C} \vee\text{-gauche}}{\Gamma \vdash C} \text{coupure}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \quad \dfrac{\pi_2}{\Gamma \vdash \neg A}}{\Gamma \vdash \bot} \neg\text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi_2'}{\Gamma \vdash \neg A} \quad \dfrac{\dfrac{\pi_1'}{\Gamma \vdash A}}{\Gamma, \neg A \vdash \bot} \neg\text{-gauche}}{\Gamma \vdash \bot} \text{coupure}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi}{\Gamma \vdash \bot}}{\Gamma \vdash A} \bot\text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi'}{\Gamma \vdash \bot} \quad \dfrac{}{\Gamma, \bot \vdash A} \bot\text{-gauche}}{\Gamma \vdash A} \text{coupure}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi}{\Gamma \vdash \forall x\ A}}{\Gamma \vdash A[x \leftarrow t]} \forall\text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi'}{\Gamma \vdash \forall x\ A} \quad \dfrac{\dfrac{\dots}{\Gamma, A[x \leftarrow t] \vdash A[x \leftarrow t]}}{\Gamma, \forall x\ A \vdash A[x \leftarrow t]} \forall\text{-gauche}}{\Gamma \vdash A[x \leftarrow t]} \text{coupure}$$

- Une preuve de la forme

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash \exists x\ A} \quad \dfrac{\pi_2}{\Gamma, A \vdash B}}{\Gamma \vdash B} \exists\text{-élim}$$

se traduit en

$$\dfrac{\dfrac{\pi_1'}{\Gamma \vdash \exists x\ A} \quad \dfrac{\dfrac{\pi_2'}{\Gamma, A \vdash B}}{\Gamma, \exists x\ A \vdash B} \exists\text{-gauche}}{\Gamma \vdash B} \text{coupure}$$

- Les règles d'introduction se traduisent par les règles droites.

3.3.3 Traduction des preuves sans coupures

Nous voulons montrer le *théorème d'élimination des coupures pour le calcul des séquents* qui exprime que tout séquent qui a une preuve en calcul des séquents a aussi une preuve qui n'utilise pas la règle de coupure (autrement dit la règle de coupure est redondante).

Nous pouvons, ou bien montrer ce résultat directement, c'est-à-dire donner une méthode pour transformer toute preuve du calcul des séquents en une preuve sans coupures, ou bien le déduire du théorème d'élimination des coupures pour la déduction naturelle, en traduisant une preuve π du calcul des séquents en une preuve π' en déduction naturelle, puis en transformant cette preuve en une preuve en déduction naturelle π'' sans coupures, puis en traduisant cette preuve en une preuve en calcul des séquents π''' sans coupures.

Pour cela il nous faut une traduction des preuves sans coupures de la déduction naturelle vers les preuves sans coupures du calcul des séquents.

La traduction ci-dessus ne vérifie pas cette propriété, par exemple la preuve

$$\dfrac{P, P \Rightarrow Q \vdash P \Rightarrow Q \quad P, P \Rightarrow Q \vdash P}{P, P \Rightarrow Q \vdash Q} \Rightarrow \text{-élim}$$

se traduit en

$$\dfrac{\dfrac{P \Rightarrow Q \vdash P \Rightarrow Q}{P, P \Rightarrow Q \vdash P \Rightarrow Q} \quad \dfrac{\dfrac{P \vdash P}{P, P \Rightarrow Q \vdash P} \quad \dfrac{Q \vdash Q}{P, P \Rightarrow Q, Q \vdash Q}}{P, P \Rightarrow Q, P \Rightarrow Q \vdash Q} \Rightarrow \text{-gauche}}{P, P \Rightarrow Q \vdash Q} \text{coupure}$$

Nous cherchons donc une autre traduction des preuves de la déduction naturelle vers le calcul des séquents. Nous donnons ici une telle traduction uniquement dans le cas particulier, où le seul connecteur est l'implication.

Proposition Une preuve Π en déduction naturelle sans coupure et dont la dernière n'est pas $\Rightarrow$-intro a la forme suivante

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow B \quad \dfrac{\pi_1}{\Gamma \vdash A_1}}{\cdots} \Rightarrow \text{-élim}}{\Gamma \vdash A_{n-1} \Rightarrow A_n \Rightarrow B} \Rightarrow \text{-élim} \quad \dfrac{\pi_{n-1}}{\Gamma \vdash A_{n-1}}}{\Gamma \vdash A_n \Rightarrow B} \Rightarrow \text{-élim} \quad \dfrac{\pi_n}{\Gamma \vdash A_n}}{\Gamma \vdash B} \Rightarrow \text{-élim}$$

Démonstration Par récurrence sur la structure de Π. Si la dernière règle est la règle axiome alors la preuve a la forme ci-dessus pour $n = 0$. Si la dernière règle est $\Rightarrow$-élim alors la preuve Π a la forme

$$\dfrac{\dfrac{\Pi'}{\Gamma \vdash A \Rightarrow B} \quad \dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash B} \Rightarrow \text{-élim}$$

Comme la preuve Π est sans coupures, la dernière règle de Π' ne peut être $\Rightarrow$-intro, donc par hypothèse de récurrence, Π' a la forme ci-dessus et Π a donc également la forme ci-dessus.

Proposition Toute preuve sans coupure en déduction naturelle peut se traduire en une preuve sans coupures en calcul des séquents.

- Si la dernière règle est la règle $\Rightarrow$-intro, la preuve a la forme

$$\dfrac{\dfrac{\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B} \Rightarrow\text{-intro}$$

  on la traduit en la preuve

$$\dfrac{\dfrac{\pi'}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B} \Rightarrow\text{-droite}$$

- si dernière règle est axiome ou $\Rightarrow$-élim, alors la preuve a la forme

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow B \quad \dfrac{\pi_1}{\Gamma \vdash A_1}}{\cdots} \Rightarrow \text{-élim}}{\Gamma \vdash A_{n-1} \Rightarrow A_n \Rightarrow B} \Rightarrow \text{-élim} \quad \dfrac{\pi_{n-1}}{\Gamma \vdash A_{n-1}}}{\Gamma \vdash A_n \Rightarrow B} \Rightarrow \text{-élim} \quad \dfrac{\pi_n}{\Gamma \vdash A_n}}{\Gamma \vdash B} \Rightarrow \text{-élim}$$

La proposition $A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow B$ est donc un élément de Γ. On construit la preuve suivante.

$$\dfrac{\dfrac{\dfrac{\pi'_1}{\Gamma \vdash A_1} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\pi'_{n-1}}{\Gamma \vdash A_{n-1}} \quad \dfrac{\dfrac{\pi'_n}{\Gamma \vdash A_n} \quad \dfrac{...}{\Gamma, B \vdash B}}{\Gamma, A_n \Rightarrow B \vdash B} \Rightarrow\text{-gauche}}{\Gamma, A_{n-1} \Rightarrow A_n \Rightarrow B \vdash B} \Rightarrow\text{-gauche}}{...} \Rightarrow\text{-gauche}}{\Gamma, A_2 \Rightarrow ... \Rightarrow A_n \Rightarrow B \vdash B} \Rightarrow\text{-gauche}}{\Gamma, A_1 \Rightarrow ... \Rightarrow A_n \Rightarrow B \vdash B} \Rightarrow\text{-gauche}}{\Gamma \vdash B} \text{contraction+permutations}$$

Corollaire Pour le langage restreint à l'implication, la règle de coupure est redondante en calcul des séquents.

Pour en savoir plus sur la traduction des preuves sans coupures du calcul des séquents en des preuves sans coupures de la déduction naturelle voir [1, 5].

Bibliographie

[1] V. Danos, J.-B. Joinet, H. Schellinx, LKQ and LKT: sequent calculi for second order logic based upon dual linear decompositions of classical implication. *Workshop on Linear Logic*, J.-Y. Girard, Y. Lafont, L. Régnier (Ed.), Cornell (1993).

[2] J. Gallier, Constructive logics. Part 1: A tutorial on proof systems and typed λ-calculi, *DEC-PRL research report*, 8, (1991).

[3] J.-Y. Girard, Linear Logic. *Theoretical Computer Science*, 50 (1987) pp. 1-102.

[4] J.-Y. Girard, Y. Lafont, P. Taylor, Proofs and Types, *Cambridge University Press* (1989).

[5] H. Herbelin, Séquents qu'on calcule, Thèse de Doctorat, Université de Paris 7 (1995).

[6] S. Kleene, Introduction to metamathematics, North-Holland (1952)

[7] P. Schroeder-Heister, A natural extension of natural deduction, *Journal of Symbolic Logic*, 49, 4, (1984) pp. 1284-1300.

Deuxième partie

La théorie des ensembles et la théorie des types simples

Chapitre 4

Le lambda-calcul simplement typé

Une théorie du premier ordre spécifie deux langages : d'une part un langage de termes qui désignent des choses (par exemple, en arithmétique, le terme $S(S(0))$), d'autre part un langage de propositions qui expriment des faits (en arithmétique, la proposition $S(0) + S(0) = S(S(0))$). En arithmétique du premier ordre, les termes désignent des nombres entiers. Il est impossible d'exprimer l'objet : *fonction qui à x associe* $3 \times x$. De même, il est impossible d'exprimer le fait : *Il existe une fonction bijective des entiers dans les multiples de* 3. Un langage dans lequel on peut exprimer de tels objets fonctionnels et de tels faits est appelé *langage d'ordre supérieur* (ce qui ne l'empèche pas d'être par ailleurs un langage du premier ordre, les ensembles et les fonctions pouvant êtres vus comme des objets de base d'un univers plus vaste).

Ce chapitre est consacré à un langage de termes d'ordre supérieur : le λ-calcul simplement typé. Le langage des propositions d'ordre supérieur correspondant est l'objet du chapitre suivant.

Notations informelles

En mathématiques informelles on désigne une fonction par un terme du premier ordre qui comporte une variable, par exemple $3 \times x$. Cette notation comporte une ambiguïté car elle ne distingue pas *le résultat de la fonction* de *la fonction* elle-même. Ainsi, quand on dit que $3 \times x$ est multiple de 3, c'est du résultat de la fonction dont on parle, alors que quand on dit que $3 \times x$ est croissante c'est de la fonction elle-même. Cette ambiguïté est parfois partiellement levée dans des notations telles que

$$\frac{d(2 \times x + y)}{dx} \quad \text{ou} \quad \int_a^b (2 \times x + y)dx$$

Néanmoins, même dans les mathématiques informelles, on sent parfois le besoin de distinguer le résultat et la fonction et on note la fonction

$$x \mapsto 3 \times x$$

Pourtant on est habituellement réticent à utiliser une telle notation ailleurs que dans une définition. Ainsi, on écrit

$$f = (x \mapsto 3 \times x)$$

puis

$$(f\ 4) = 12$$

$$\int_0^1 f = \frac{3}{2}$$

mais pas

$$((x \mapsto 3 \times x)\ 4) = 12$$

$$\int_0^1 (x \mapsto 3 \times x) = \frac{3}{2}$$

Le λ-calcul est la systématisation de cette notation. En λ-calcul on écrit aussi bien

$$(f\ 4)$$

$$\int_0^1 f$$

que

$$((x \mapsto 3 \times x)\ 4)$$

$$\int_0^1 (x \mapsto 3 \times x)$$

Par ailleurs (bien que cela soit assez anecdotique), on note en λ-calcul les fonctions $\lambda x\ t$ et non $x \mapsto t$.

Les types

Si on étend les langages du premier ordre par un opérateur d'abstraction fonctionnelle (λ), on obtient le λ-calcul pur dont les termes sont formés par trois règles

- les variables et les symboles d'individu et de fonction sont des termes,
- si t et u sont des termes, alors $(t\ u)$ est un terme,
- si t est un terme, alors $\lambda x\ t$ est un terme.

Dans les langages du premier ordre, les symboles de fonctions (d'arité non nulle) ne sont pas des termes : S n'est pas un terme, seul $S(0)$ est un terme. En λ-calcul les symboles de fonction sont des termes (qui expriment des fonctions) ainsi S est un terme, 0 est un terme et $(S\ 0)$ est formé par la règle d'application. De même, $+$ est un terme, $(+\ 0)$ est un terme et $(+\ 0\ 0)$ est un terme.

Ce langage (le λ-calcul pur) permet de former des termes dénués de sens tels que $(S\ 0\ 0)$, $(S\ S)$, $(S\ \lambda x\ x)$ ou $(\lambda x\ (x\ x)\ \lambda x\ (x\ x))$. Ces termes posent plusieurs problèmes :

- il ne correspondent pas à des objets mathématiques dont nous avons une compréhension intuitive,
- quand nous ajouterons un langage pour former des propositions, ils seront responsables de paradoxes,
- certains d'entre eux sont non normalisables, c'est-à-dire que le simple fait de remplacer les arguments formels par les arguments réels dans une expression peut conduire à des calculs infini.

Dans les langages du premier ordre, le terme $(S\ 0\ 0)$ ne peut pas être formé car le symbole S prend un seul argument, le terme $(S\ S)$ ne peut pas être formé car le symbole S n'est pas un terme, quant aux termes $(S\ \lambda x\ x)$ et $(\lambda x\ (x\ x)\ \lambda x\ (x\ x))$, ils ne peuvent pas être formés car il n'y a pas d'opérateur λ.

En λ-calcul, il faut donc également restreindre les règles de construction des termes de façon à prohiber la formation de ces termes. Une notion d'arité n'est plus suffisante, puisque le terme $(S\ 0)$ est bien formé, mais pas le terme $(S\ \lambda x\ x)$, il faut donc associer à chaque terme une information exprimant, non seulement le nombre de ses arguments, mais également leur *degré de fonctionnalité*, c'est-à-dire le nombre et (récursivement) le degré de fonctionnalité de leurs arguments. Une telle information est appelée un *type*.

4.1 Le λ-calcul simplement typé

4.1.1 Le λ-calcul simplement typé à un seul type de base

Définition L'ensemble des *types* est inductivement défini par

- ι est un type,
- si A et B sont des types, alors $(A \to B)$ est un type.

Notation On note $A_1 \to A_2 \to ... \to A_n \to B$ le type $(A_1 \to (A_2 ... \to (A_n \to B)...))$.

Proposition Tout type a la forme $A_1 \to A_2 \to ... \to A_n \to \iota$
Démonstration Par récurrence sur la structure des types.

Définition Une *variable typée* est un couple (x, A) où x est une variable et A un type. Un *contexte* est une liste de variables typées, telle que chaque variable apparaisse au plus une fois dans cette liste.

Définition L'ensemble des *termes* est inductivement défini par

- les variables sont des termes,
- si t et u sont des termes, alors $(t\ u)$ est un terme,
- si t est un terme, x une variable et A un type, alors $\lambda x : A\ t$ est un terme.

Notation On note $(t\ u_1\ ...\ u_n)$ le terme $(...((t\ u_1)\ u_2)\ ...\ u_n)$.

Définition Soit Γ un contexte, t un terme et A un type. L'ensemble des triplets (Γ, t, A) *bien typés* est défini inductivement par

- si Γ est un contexte et $(x, A) \in \Gamma$ alors (Γ, x, A) est bien typé,
- si $(\Gamma, t, A \to B)$ et (Γ, u, A) sont bien typés alors $(\Gamma, (t\ u), B)$ est bien typé,
- si $(\Gamma[x : A], t, B)$ est bien typé alors $(\Gamma, \lambda x : A\ t, A \to B)$ est bien typé.

Notation Soit Γ un contexte, t un terme et A un type. Si le triplet (Γ, t, A) est bien typé, on dit que *t a le type A dans* Γ, et on écrit $\Gamma \vdash t : A$. Soit Γ un contexte et t un terme, s'il existe un type A tel que $\Gamma \vdash t : A$ le terme t est dit *bien typé* dans Γ.

Exemple Soit le contexte $0 : \iota,\ S : \iota \to \iota$, les termes $(S\ 0)$, $\lambda f : \iota \to \iota\ (f\ 0)$ sont bien typés. Les termes $(S\ 0\ 0)$, $(S\ S)$, $(S\ \lambda x : \iota\ x)$ et $(\lambda x : \iota\ (x\ x)\ \lambda x : \iota\ (x\ x))$ ne sont pas bien typés.

Proposition Soit Γ un contexte et t un terme, il existe au plus un type A tel que $\Gamma \vdash t : A$. Si $\Gamma \vdash t : A$, A est donc *le* type de t dans Γ.

Proposition Il existe un algorithme qui prend en argument un contexte Γ et un terme t et qui décide si t est typable dans Γ et qui si t est typable, retourne le type de t dans Γ.

4.1.2 Le λ-calcul simplement typé avec plusieurs types de base

Dans la définition ci-dessus, le type d'un terme indique uniquement le degré de fonctionnalité d'un terme, c'est-à-dire le nombre et le degré de fonctionnalité de ses arguments. Il est également possible de distinguer plusieurs sortes d'objets de base (les entiers, les booléens, les chaînes de caractères, etc.), c'est-à-dire de se donner un ensemble de *types de base* et définir ainsi l'ensemble des types.

Définition L'ensemble des *types* est inductivement défini par

- les types de base sont des types,
- si A et B sont des types, alors $(A \rightarrow B)$ est un type.

Proposition Tout type a la forme $A_1 \rightarrow A_2 \rightarrow ... \rightarrow A_n \rightarrow B$ où B est un type de base.

4.1.3 Le λ-calcul selon Church et selon Curry

La présentation du λ-calcul que nous avons adoptée est traditionnellement appelée *présentation de Church*. Une présentation alternative, *de Curry*, consiste à ne pas noter les types de variables liées dans les termes. Ainsi les termes du λ-calcul de Curry sont des termes purs. Par exemple dans la présentation de Church on a

$$\vdash \lambda x : \iota \ x : \iota \rightarrow \iota$$

$$\vdash \lambda x : (\iota \rightarrow \iota) \ x : (\iota \rightarrow \iota) \rightarrow (\iota \rightarrow \iota)$$

alors que dans la présentation de Curry on a

$$\vdash \lambda x \ x : \iota \rightarrow \iota$$

$$\vdash \lambda x \ x : (\iota \rightarrow \iota) \rightarrow (\iota \rightarrow \iota)$$

Dans la présentation de Curry la décidabilité de la typabilité est un peu plus difficile à établir, et un terme typable n'a plus un type unique (mais il y a un résultat plus faible, de description des types d'un terme par un *type principal*).

Dans le reste de ce chapitre tous les termes considérés sont supposés bien typés.

4.2 La réduction, l'équivalence

4.2.1 L'α-équivalence

Définition Deux termes sont dits α-équivalents s'ils ne diffèrent que par renommages des variables liées n'introduisant pas de captures. Par exemple les termes $\lambda x : \iota \ x$ et $\lambda y : \iota \ y$ sont α-équivalents. En revanche les termes $\lambda x : \iota \ \lambda y : \iota \ (f \ x \ y)$ et $\lambda x : \iota \ \lambda x : \iota \ (f \ x \ x)$ ne sont pas α-équivalents puisque le renommage de la variable y en x introduit la capture de l'occurrence de x qui réfère maintenant au second λ et plus au premier.

En λ-calcul on raisonne toujours modulo α-équivalence, c'est-à-dire qu'on ne distingue pas deux termes α-équivalents (ou encore, que l'on raisonne sur les classes d'α-équivalence).

4.2.2 La substitution

Soit t et a deux termes et x une variable. On note $t[x \leftarrow a]$ le terme obtenu en substituant la variable x par le terme a dans t, c'est-à-dire le terme défini par récurrence sur la structure de t par

- si t est la variable x alors $t[x \leftarrow a] = a$,
- si t est une variable y distincte de x alors $t[x \leftarrow a] = y$,
- si $t = (u \ v)$ alors $t[x \leftarrow a] = (u[x \leftarrow a] \ v[x \leftarrow a])$,
- si $t = \lambda x : A \ u$ alors $t[x \leftarrow a] = t$,
- si $t = \lambda y : A \ u$ et $y \neq x$ alors $t[x \leftarrow a] = \lambda y : A \ u[x \leftarrow a]$.

Dans cette dernière règle il faut que y n'apparaisse pas dans a. Si c'est le cas, il est nécessaire de renommer la variable y dans t avec une nouvelle variable.

4.2.3 La β-réduction, la β-équivalence

L'équivalence de deux termes modulo le remplacement des arguments formels par les arguments réels, c'est-à-dire modulo le remplacement des sous-termes de la forme $((\lambda x : A\ t)\ u)$ par le terme $t[x \leftarrow u]$ est appelée β-équivalence.

Définition Un *β-radical* est un terme de la forme $((\lambda x : A\ t)\ u)$.

Définition La relation entre termes $t\ \rhd u$ (*t se β-réduit sur une étape en u*) est la plus petite relation telle que

- $((\lambda x : A\ t)\ u)\ \rhd t[x \leftarrow u]$,
- si $t\ \rhd u$ alors $(t\ v)\ \rhd (u\ v)$,
- si $t\ \rhd u$ alors $(v\ t)\ \rhd (v\ u)$,
- si $t\ \rhd u$ alors $\lambda x : A\ t\ \rhd \lambda x : A\ u$.

La relation $t\ \rhd^* u$ (*t se β-réduit sur u*) est définie comme la fermeture reflexive-transitive de la relation $\rhd$, c'est-à-dire comme la plus petite relation telle que

- si $t\ \rhd u$ alors $t\ \rhd^* u$,
- $t\ \rhd^* t$,
- si $t\ \rhd^* u$ et $u\ \rhd^* v$ alors $t\ \rhd^* v$.

La relation $t \equiv u$ (*t est β-équivalent à u*) est définie comme la fermeture reflexive-symétrique-transitive de la relation $\rhd$, c'est-à-dire comme la plus petite relation telle que

- si $t\ \rhd u$ alors $t \equiv u$,
- $t \equiv t$,
- si $t \equiv u$ alors $u \equiv t$,
- si $t \equiv u$ et $u \equiv v$ alors $t \equiv v$.

4.2.4 La $\beta\eta$-réduction, la $\beta\eta$-équivalence

Outre le remplacement des arguments formels par les arguments réels, une autre transformation comparable des termes peut être incorporée à l'équivalence. C'est la transformation de $\lambda x : A\ (t\ x)$ en t quand x n'apparaît pas dans t. Cette transformation permet par exemple d'identifier les termes S et $\lambda x : \iota\ (S\ x)$.

Définition Un *η-radical* est un terme de la forme $\lambda x : A\ (t\ x)$, où la variable x n'apparaît pas dans le terme t.

Définition La relation entre termes $t\ \rhd_{\beta\eta} u$ (*t se $\beta\eta$-réduit sur une étape en u*) est la plus petite relation telle que

- $((\lambda x : A\ t)\ u)\ \rhd_{\beta\eta} t[x \leftarrow u]$,
- $\lambda x : A\ (t\ x)\ \rhd_{\beta\eta} t$ si x n'apparaît pas dans t,
- si $t\ \rhd_{\beta\eta} u$ alors $(t\ v)\ \rhd_{\beta\eta} (u\ v)$,
- si $t\ \rhd_{\beta\eta} u$ alors $(v\ t)\ \rhd_{\beta\eta} (v\ u)$,

- si $t \rhd_{\beta\eta} u$ alors $\lambda x : A\ t \rhd_{\beta\eta} \lambda x : A\ u$.

On définit de même les relations $\rhd^*_{\beta\eta}$ et $\equiv_{\beta\eta}$ comme les fermetures reflexive-transitive et reflexive-symétrique-transitive de cette relation.

Définition On note également $t \rhd_\eta u$ (*t se η-réduit sur une étape en u*) la plus petite relation telle que

- $\lambda x : A\ (t\ x) \rhd_\eta t$ si x n'apparaît pas dans t,
- si $t \rhd_\eta u$ alors $(t\ v) \rhd_\eta (u\ v)$,
- si $t \rhd_\eta u$ alors $(v\ t) \rhd_\eta (v\ u)$,
- si $t \rhd_\eta u$ alors $\lambda x : A\ t \rhd_\eta \lambda x : A\ u$.

Proposition (Préservation du type) Si $\Gamma \vdash t : T$ and $t \rhd t'$ then $\Gamma \vdash t' : T$

4.3 La confluence

Définition Un λ-calcul est dit *confluent* si chaque fois qu'un terme t se réduit sur deux termes u_1 et u_2, il existe un terme v, tel que u_1 et u_2 se réduisent sur v.

4.3.1 La confluence en λ-calcul selon Curry

En λ-calcul selon Curry les termes sont des termes purs, la confluence de la β-réduction et de le $\beta\eta$-réduction est donc conséquence de la confluence de ces relations dans le λ-calcul pur [5] et de la préservation du type.

4.3.2 La confluence en λ-calcul selon Church

En λ-calcul selon Church, les démonstrations de confluence doivent être adaptées. La démonstration de confluence du λ-calcul pur s'adapte facilement pour le β-réduction qui est confluente sur tous les termes (bien typés ou non) du λ-calcul simplement typé. En revanche la $\beta\eta$-réduction, n'est pas confluente sur tous les termes.

Exemple (Nederpelt) Le terme $\lambda x : A\ (\lambda y : B\ y\ x)$ se β-réduit en $\lambda x : A\ x$ et se η-réduit en $\lambda y : B\ y$ et il n'existe pas de terme u tel que $\lambda x : A\ x \rhd^*_{\beta\eta} u$ et $\lambda y : B\ y \rhd^*_{\beta\eta} u$.

Il faut adapter la démonstration de confluence du λ-calcul pur en considérant uniquement les termes bien typés [2].

4.4 La normalisation

L'objectif de cette section est de montrer que quand on remplace dans un terme bien typé les arguments formels par les arguments réels on arrive après un nombre fini d'étapes à un terme normal, c'est-à-dire un terme qui ne peut plus être réduit.

4.4.1 Les suites de réductions

Définition Soit t un terme, un *radical* de t est un sous-terme de t qui est un radical. Un terme sans radical est dit *normal*.

Proposition Un terme normal a la forme $t = \lambda x_1 : A_1\ ...\ \lambda x_n : A_n\ (x\ u_1\ ...\ u_p)$ où x est une variable.

Démonstration Le terme t a la forme $\lambda x_1 : A_1 \ ... \ \lambda x_n : A_n \ t'$ où t' n'est pas une abstraction. Le terme t' a la forme $(t'' \ u_1 \ ... \ u_p)$ où t'' n'est pas une application. Le terme t'' n'est pas une application, ce n'est pas une abstraction (si $p \neq 0$ parce que t est normal, si $p = 0$ parce que t' n'est pas une abstraction), c'est donc une variable.

Définition Une *suite de réductions* issue d'un terme t est une suite (finie ou infinie) $t_0, t_1, t_2, ...$ de termes telle que $t_0 = t$ et pour tout i tel que t_i et t_{i+1} soient définis, $t_i \ \rhd t_{i+1}$.

Définition Un terme t est dit *faiblement normalisable* s'il existe un terme normal u tel que $t \ \rhd^* u$.

Définition Un terme est dit *fortement normalisable* si toute suite de réductions issue de t est finie.

Proposition Tout terme fortement normalisable est faiblement normalisable.
Démonstration Soit t un terme fortement normalisable et soit Φ une fonction qui associe à chaque terme non normal u un terme $(\Phi \ u)$ tel que $u \ \rhd (\Phi \ u)$ (par exemple le terme obtenu en réduisant le radical le plus à gauche dans u). La suite $(\Phi^i \ t)$ est une suite de réductions issue de t. Comme le terme t est fortement normalisable, cette suite est finie. Soit n le plus grand entier tel que $(\Phi^n \ t)$ soit défini. Le terme $(\Phi^n \ t)$ est normal et $t \ \rhd^* (\Phi^n \ t)$. Le terme t est donc faiblement normalisable.

Proposition Dans le λ-calcul pur il existe un terme faiblement mais non fortement normalisable.
Démonstration Le terme $(\lambda x \ y \ (\lambda z \ (z \ z) \ \lambda z \ (z \ z)))$ est faiblement normalisable, puisqu'il se réduit sur y. En revanche, il n'est pas fortement normalisable puisqu'il se réduit sur lui-même.

Remarque Montrer qu'un terme est faiblement normalisable revient à montrer qu'il existe une manière de réduire ce terme qui donne une forme normale (dans ce cas, d'après le théorème de standardisation, en réduisant toujours le radical le plus à gauche on obtient une telle forme normale [4]). En revanche montrer qu'un terme est fortement normalisable revient à montrer que quelque soit la manière dont le terme est réduit, la suite de réductions aboutit toujours à une forme normale.

4.4.2 La normalisation forte

Définition L'ensemble des termes *réductibles* de type T est défini par récurrence sur la structure de T.

- Si T est atomique alors t est réductible si et seulement s'il est fortement normalisable,
- si $T = A \to B$ alors t est réductible si et seulement si pour tout terme réductible u de type A, $(t \ u)$ est réductible.

Proposition (1) Les termes réductibles sont fortement normalisables. (2) Les variables sont des termes réductibles.
Démonstration On montre par récurrence sur la structure de T que
(1) les termes réductibles de type T sont fortement normalisables,
(2) les variables de type T sont réductibles.

- Si T est un type atomique alors (1) les termes réductibles de type T sont par définition les termes fortement normalisables et (2) les variables de types T sont fortement normalisables donc réductibles.
- Si $T = A_1 \to ... \to A_n \to B$ (B atomique) alors (1) soit t un terme réductible de type T et $x_1, ..., x_n$ des variables de type $A_1, ..., A_n$. Par hypothèse de récurrence les variables $x_1, ..., x_n$ sont des termes réductibles, le terme $(t \ x_1 \ ... \ x_n)$ est donc réductible. Le terme $(t \ x_1 \ ... \ x_n)$ est réductible et son type est atomique, il est donc fortement normalisable. Le terme t est donc également fortement normalisable car si la suite $t_0, t_1, t_2, ...$ est une suite de réductions issue de t, la suite $(t_0 \ x_1 \ ... \ x_n), (t_1 \ x_1 \ ... \ x_n), (t_2 \ x_1 \ ... \ x_n), ...$ est une suite de réductions issue de $(t \ x_1 \ ... \ x_n)$ et cette suite est donc finie. (2) Soit x une variable de

type T et u_1, ..., u_n des termes réductibles de type A_1, ..., A_n. Par hypothèse de récurrence, les termes u_1, ..., u_n sont fortement normalisables. Toute suite de réduction issue du terme $(x\ u_1\ ...\ u_n)$ réduit des radicaux dans les termes u_1, ..., u_n, elle est donc finie. Le terme $(x\ u_1\ ...\ u_n)$ est donc fortement normalisable et de type atomique, il est donc réductible. La variable x est donc un terme réductible.

Proposition Tout terme est réductible.
Démonstration On démontre par récurrence sur la structure de t que si t est un terme quelconque et u_1, ..., u_n des termes réductibles alors le terme $t[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]$ est réductible.

- Si t est une variable alors $t[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]$ est ou bien une variable ou bien l'un des u_i. C'est donc un terme réductible.

- Si $t = (u\ v)$, alors on a $t[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n] = (u[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]\ v[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n])$.

  Les termes $u[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]$ et $v[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]$ sont réductibles par hypothèse de récurrence. Le terme $t[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]$ est donc réductible.

- Si $t = \lambda y : A\ u$, on a $t[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n] = \lambda y : A\ u[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]$. Notons u' le terme $u[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]$.

  Soit $B_1 \rightarrow ... \rightarrow B_n \rightarrow C$ (C atomique) le type de u et soient v, w_1, ..., w_n des termes réductibles de type A, B_1, ... B_n. Considérons une suite de réductions $t_0, t_1, t_2, ...$ issue du terme $(\lambda y : A\ u'\ v\ w_1\ ...\ w_n)$ et montrons que cette suite est finie. Dans la suite $t_0, t_1, t_2, ...$, ou bien le radical de tête n'est jamais réduit, ou bien il est réduit à l'étape k.

  Si le radical de tête n'est jamais réduit, le terme u' est réductible par hypothèse de récurrence et les termes v, w_1, ..., w_n sont réductibles par hypothèse. Ces termes sont donc fortement normalisables. Comme la suite $t_0, t_1, t_2, ...$ ne réduit des radicaux que dans ces termes, elle est donc finie.

  Si le radical de tête est réduit à l'étape k, le terme t_k a la forme $(u'^*[y \leftarrow v^*]\ w_1^*\ ...\ w_n^*)$ où u'^*, v^*, w_1^*, ..., w_n^* sont des réduits de u', v, w_1, ..., w_n. Le terme $(u'^*[y \leftarrow v^*]\ w_1^*\ ...\ w_n^*)$ est un réduit du terme $(u'[y \leftarrow v]\ w_1\ ...\ w_n)$. Le terme $u'[y \leftarrow v] = u[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n, y \leftarrow v]$ est réductible par hypothèse de récurrence et les termes w_1, ..., w_n sont réductibles donc le terme $(u'[y \leftarrow v]\ w_1\ ...\ w_n)$ est réductible. Ce terme est réductible et son type est atomique, il est donc fortement normalisable, le terme $t_k = (u'^*[y \leftarrow v^*]\ w_1^*\ ...\ w_n^*)$ également et la suite $t_0, t_1, t_2, ...$ est donc finie.

  Le terme $(\lambda y : A\ u'\ v\ w_1\ ...\ w_n)$ est fortement normalisable. Ce terme est fortement normalisable et son type est atomique, il est donc réductible. Le terme $t[x_1 \leftarrow u_1, ..., x_n \leftarrow u_n] = \lambda y : A\ u'$ est donc réductible.

Corollaire (Tait) Tout terme est fortement normalisable.

4.4.3 La normalisation forte de la $\beta\eta$-réduction

Proposition Tout terme est fortement normalisable pour la $\beta\eta$-réduction.
Démonstration La démonstration est très similaire à celle de la normalisation forte de la β-réduction. La seule différence est que dans la suite de réductions issue du terme $(\lambda y : A\ u'\ v\ w_1\ ...\ w_n)$ il se peut que le radical de tête soit réduit par la règle η et non β.

Dans ce cas, le terme $(\lambda y : A\ u'\ v\ w_1\ ...\ w_n)$ se réduit en $k-1$ étapes en un terme $(\lambda y : A\ u'^*\ v^*\ w_1^*\ ...\ w_n^*)$ où u'^*, v^*, w_1^*, ..., w_n^* sont des réduits de u', v, w_1, ..., w_n tels que le terme u'^* ait la forme $(u''\ y)$ avec y qui n'apparaît pas dans u''. Le terme $(\lambda y : A\ u'^*\ v^*\ w_1^*\ ...\ w_n^*) = (\lambda y : A\ (u''\ y)\ v^*\ w_1^*\ ...\ w_n^*)$ se réduit alors à l'étape k en le terme $(u''\ v^*\ w_1^*\ ...\ w_n^*)$. Le terme $(u''\ v^*)$ est un réduit de $u'[y \leftarrow v]$ et donc le terme $(u''\ v^*\ w_1^*\ ...\ w_n^*)$ un réduit de $(u'[y \leftarrow v]\ w_1\ ...\ w_n)$. Comme ci-dessus on conclut en remarquant que ce terme est fortement normalisable et que la suite $t_0, t_1, t_2, ...$ est donc finie.

4.4.4 La normalisation faible

En corollaire du théorème de normalisation forte, on peut déduire que tout terme est faiblement normalisable. Ce résultat admet également une démonstration directe, nettement plus simple que celle du théorème de normalisation forte.

Définition L'ordre lexicographique sur les couples d'entiers est défini par $(a, b) < (a', b')$ si et seulement si $a < a'$ ou ($a = a'$ et $b < b'$).

Proposition L'ordre lexicographique sur l'ensemble des couples d'entiers est bien fondé, c'est-à-dire que toute suite décroissante est finie.
Démonstration On montre que pour tout entier a, pour tout entier b, toute suite s décroissante telle que $s_0 = (a, b)$ est finie. Par récurrence sur a.

- Cas de base : on montre que pour tout entier b, toute suite s décroissante s telle que $s_0 = (0, b)$ est finie. Par récurrence sur b.
  - Cas de base : on montre que toute suite s décroissante telle que $s_0 = (0, 0)$ est finie. C'est trivial car s a un élément.
  - Récurrence : on suppose que (1) toute suite décroissante s telle que $s_0 = (0, b')$, $b' < b$ est finie et on montre que toute suite décroissante s telle que $s_0 = (0, b)$ est finie. Ou bien la suite s a un élément auquel cas elle est finie, ou bien s_1 a la forme $(0, b')$ avec $b' < b$. Par l'hypothèse (1), la suite $s_1, s_2, ...$ est finie donc la suite $s_0, s_1, s_2, ...$ aussi.
- Récurrence : on suppose que (2) pour tout entier b, toute suite s décroissante telle que $s_0 = (a', b)$, $a' < a$ est finie et on montre que quelque soit l'entier b, toute suite décroissante s telle que $s_0 = (a, b)$ est finie. Par récurrence sur b.
  - Cas de base : on montre que toute suite décroissante s telle que $s_0 = (a, 0)$ est finie. Ou bien la suite s a un élément auquel cas elle est finie, ou bien $s_1 = (a', b')$ avec $a' < a$. Par l'hypothèse (2), la suite $s_1, s_2, ...$ est finie donc la suite $s_0, s_1, s_2, ...$ aussi.
  - Récurrence : on suppose que (3) toute suite décroissante s telle que $s_0 = (a, b')$, $b' < b$ est finie et on montre que toute suite décroissante s telle que $s_0 = (a, b)$ est finie. Ou bien la suite s a un élément auquel cas elle est finie. Ou bien $s_1 = (a', b')$ avec $a' < a$ ou ($a' = a$ et $b' < b$) dans le premier cas on applique l'hypothèse (2) et dans le second l'hypothèse (3). Dans les deux cas, on obtient que la suite $s_1, s_2, ...$ est finie donc que la suite $s_0, s_1, s_2, ...$ aussi.

Définition La *taille d'un type* T est le nombre d'occurrences du symbole $\rightarrow$ dans T. Elle est définie par récurrence sur la structure du type par

- $|T| = 0$ si T est atomique,
- $|A \rightarrow B| = 1 + |A| + |B|$

La *taille d'un radical* $(\lambda x : A\ u\ v)$ est la taille du type $A \rightarrow B$ du terme $\lambda x : A\ u$.

Définition Soit t un terme. Un radical de t de la forme $(\lambda x : A\ u\ v)$ de taille k est un *radical interne de taille maximale* si tout les radicaux de t ont une taille inférieure ou égale à k tous les radicaux de v ont une taille strictement inférieure à k.

Proposition Soit t un terme non normal, il existe dans t un radical interne de taille maximale.
Démonstration Par récurrence sur la structure de t.

Définition Soit Φ la fonction qui associe à chaque terme t non normal le terme $(\Phi\ t)$ obtenu en réduisant dans t un radical parmi les radicaux internes de taille maximale (par exemple, le plus à gauche).

Proposition Soit t un terme non normal, k la taille de ses radicaux de taille maximale et n le nombre de radicaux de taille k dans t. Soit k' la taille des radicaux de taille maximale dans $(\Phi\ t)$ et n' le nombre de radicaux de taille k' dans $(\Phi\ t)$. On a $(k', n') < (k, n)$.
Démonstration Soit $(\lambda x : A\ u\ v)$ le radical réduit dans t pour donner le terme $(\Phi\ t)$, soit $A \to B$ le type du terme $\lambda x : A\ u$. Les radicaux de $(\Phi\ t)$ sont de trois sortes.

- Les résidus [4] de radicaux de t qui ne sont pas des radicaux de v. Ces radicaux ont exactement un résidu.
- Les résidus de radicaux de t qui sont des radicaux de v. Tous ces radicaux ont une taille strictement inférieure à k, car le radical $(\lambda x : A\ u\ v)$ est un radical interne de taille maximale.
- Les radicaux créés par la réduction, ces radicaux sont à leur tour de deux sortes :
  - ceux créés (vers le bas) par la substitution de x en v si v est une abstraction, l'ordre de ces radicaux est $|A| < k = |A \to B|$,
  - ceux créés (vers le haut) par la transformation de $(\lambda x : A\ u\ v)$ en $u[x \leftarrow v]$ si ce terme est une abstraction. Ces radicaux ont alors la taille $|B| < k = |A \to B|$.

Les radicaux des deux dernières catégories sont tous de taille strictement inférieure à k. Ceux de la première catégorie qui sont de taille k étaient déjà des radicaux de t et l'un des radicaux de t de taille k (le radical réduit) n'a pas de résidu. Le nombre de radicaux de t de taille k dans $(\Phi\ t)$ est donc strictement inférieur au nombre de radicaux de taille k dans t. Si le nombre de radicaux de taille k dans $(\Phi\ t)$ est nul on a $k' < k$, sinon on a $k = k'$ et $n < n'$.

Proposition Tout terme est faiblement normalisable.
Démonstration La suite $(\Phi^i\ t)$ est finie. En effet, la suite (k_i, n_i) où k_i est la taille des radicaux de taille maximale dans $(\Phi^i\ t)$ et n_i le nombre de radicaux de taille k_i dans $(\Phi^i\ t)$ est strictement décroissante elle est donc finie car l'ordre lexicographique est bien fondé sur les couples d'entiers.

4.4.5 La normalisation faible de la $\beta\eta$-réduction

Proposition Tout terme est faiblement normalisable pour la $\beta\eta$-réduction.
Démonstration Ici encore, ce résultat peut se déduire de la normalisation forte, mais il admet aussi une démonstration plus simple.

Soit t un terme. D'après le théorème de normalisation faible de la β-réduction il existe un terme β-normal u tel que $t \rhd^* u$. On considère ensuite la suite de réductions $u_0, u_1, u_2, ...$ issue de u en réduisant un η-radical à chaque étape (par exemple le plus à gauche). Cette suite est finie parce que le nombre d'occurrences de variables dans u_{i+1} est strictement inférieur au nombre d'occurrences de variables dans u_i. Soit n le plus grand entier tel que u_n soit défini.

Pour montrer que le terme u_n est $\beta\eta$-normal, on montre que si v est un terme β-normal, et v' un terme obtenu en réduisant dans v un η-radical alors v' est également β-normal. Soit $\lambda x : A\ (w\ x)$ le η-radical de v réduit en w. Ce sous terme de v n'est pas membre gauche d'une application dans v car v est β-normal. Le terme w n'est donc pas membre gauche d'une application dans v' et de ce fait la réduction de $\lambda x : A\ (w\ x)$ en w ne crée pas de β-radical.

4.4.6 Applications de la normalisation et de la confluence

Proposition Tout terme se réduit sur un terme normal unique.
Démonstration Soit t un terme. D'après le théorème de normalisation, il existe un terme normal u tel que $t \rhd^* u$. D'après le théorème de confluence, ce terme est unique. En effet, supposons que $t \rhd^* u_1$ et $t \rhd^* u_2$, alors il existe un terme u tel que $u_1 \rhd^* u$ et $u_2 \rhd^* u$. Comme les termes u_1 et u_2 sont normaux, $u_1 = u = u_2$.

Définition Soit t un terme. L'unique terme normal u tel que $t \triangleright^* u$ est appelé *la forme normale de* t.

Proposition Deux termes t et u sont équivalents s'ils ont la même forme normale.
Démonstration Par récurrence sur la longueur d'une dérivation de $t \equiv u$.

Proposition Il existe un algorithme qui prend en argument un terme t et calcule sa forme normale.
Démonstration Il existe un algorithme Φ qui prend en argument un terme non normal t et retourne un terme $(\Phi\ t)$ tel que $t \triangleright (\Phi\ t)$ (par exemple un algorithme qui réduit le radical le plus à gauche dans t). La forme normale d'un terme t peut se calculer en itérant l'algorithme Φ, jusqu'à obtenir un terme normal. D'après le théorème de normalisation, cette itération termine toujours.

Proposition L'équivalence entre deux termes est décidable
Démonstration Soient t et u deux termes, pour décider de l'équivalence de t et u on calcule les formes normales de ces deux termes. Les termes t et u sont équivalents si et seulement si leurs formes normales sont égales (modulo α-conversion).

4.4.7 La forme η-longue

S'il est clair que les termes S et $\lambda x : \iota\ (S\ x)$ doivent être considérés comme équivalents, il est moins clair que le terme S doive-être la forme normale de ces deux termes. En effet, ces termes exprimant une fonction (de type $\iota \to \iota$), il est plus naturel de choisir comme forme normale le terme $\lambda x : \iota\ (S\ x)$ qui est une abstraction.

Définition Un terme t de type $A_1 \to ... \to A_n \to B$ (B atomique) est dit *β-normal η-long* s'il est de la forme $\lambda x_1 : A_1\ ...\ \lambda x_n : A_n\ (x\ u_1\ ...\ u_p)$ et les termes $u_1\ ...\ u_p$ sont eux-mêmes β-normaux η-longs.

Définition Soit t un terme β-normal on définit sa *forme β-normale η-longue t'*.

On commence par définir la forme β-normale η-longue d'une variable $x : A_1 \to ... \to A_n \to B$ (B atomique) par récurrence sur la structure de son type comme le terme

$$x' = \lambda y_1 : A_1\ ...\ \lambda y_n : A_n\ (x\ y_1'\ ...\ y_n')$$

où y_i' est la forme β-normale η-longue de la variable y_i de type A_i. On définit ensuite la forme β-normale η-longue d'un terme β-normal $t = \lambda x_1 : A_1\ ...\ \lambda x_p : A_p\ (x\ u_1\ ...u_k)$ de type $A_1 \to ... \to A_n \to B$ (B atomique) par récurrence sur la structure de ce terme comme le terme

$$t' = \lambda x_1 : A_1\ ...\ \lambda x_p : A_p\ \lambda x_{p+1} : A_{p+1}\ ...\ \lambda x_n : A_n\ (x\ u_1'\ ...\ u_k'\ x_{p+1}'\ ...x_n')$$

où u_i' est la forme β-normale η-longue du terme u_i et x_i' la forme β-normale η-longue de la variable x_i.

Proposition Tout terme est $\beta\eta$-équivalent à un terme β-normal η-long unique.
Démonstration

- Existence. Tout terme est équivalent à la forme β-normale η-longue de sa forme β-normale.
- Unicité. Commençons par montrer que si t et u sont deux termes β-normaux et $t \triangleright_\eta u$ alors t et u ont même forme β-normale η-longue. Par récurrence sur la structure de t. Soit $t = \lambda x_1 : A_1\ ...\ \lambda x_n : A_n\ (x\ a_1\ ...\ a_p)$. Si $a_p = x_n$ et $u = \lambda x_1 : A_1\ ...\ \lambda x_{n-1} : A_{n-1}\ (x\ a_1\ ...\ a_{p-1})$ alors $t' = u'$. Sinon u a la forme $\lambda x_1 : A_1\ ...\ \lambda x_n : A_n\ (x\ a_1\ ...\ a_{k-1}\ b\ a_{k+1}\ a_p)$ et $a_k \triangleright_\eta b$. Par hypothèse de récurrence $a_k' = b'$ et donc $t' = u'$.

  Montrons maintenant qu'un terme t, β-normal a la même forme β-normale η-longue que sa forme $\beta\eta$-normale. Comme le terme t est β-normal et que la réduction d'un η-radical dans un terme β-normal n'introduit pas de β-radicaux, les radicaux réduits dans t sont tous des η-radicaux. D'après le résultat ci-dessus, les forme β-normale η-longue de t et de sa forme $\beta\eta$-normale sont identiques.

  Un terme β-normal η-long étant sa propre forme β-normale η-longue, il est la forme β-normale η-longue de sa forme $\beta\eta$-normale.

Montrons maintenant que si t est $\beta\eta$-équivalent à des termes β-normaux η-longs u_1 et u_2 alors $u_1 = u_2$. Les termes u_1 et u_2 sont $\beta\eta$-équivalents ils ont donc même forme $\beta\eta$-normale v. Soit v' la forme β-normale η-longue du terme v. On a $u_1 = v' = u_2$.

Définition La forme β-normale η-longue d'un terme quelconque t est la forme β-normale η-longue de sa forme $\beta\eta$-normale (ou de façon équivalente de sa forme β-normale).

Proposition Deux termes sont $\beta\eta$-équivalents s'ils ont la même forme β-normale η-longue.

Remarque Nous avons vu au paragraphe précédent que pour décider la $\beta\eta$-équivalence de deux terme t et u on pouvait comparer leur formes $\beta\eta$-normale. D'après la proposition ci-dessus il est également possible de comparer leur formes β-normale η-longue. Le calcul de la forme β-normale η-longue d'un terme est plus simple que celui de sa forme $\beta\eta$-normale qui peut demander de tester à plusieurs reprises si un terme de la forme $\lambda x\ (t\ x)$ est un η-radical ou non, c'est-à-dire si la variable x a ou non une occurrence dans le terme t.

4.5 L'expressivité calculatoire

Quand on dit que deux expressions mathématiques désignent le même objet on peut distinguer plusieurs manières d'argumenter cette égalité.

- Les deux expressions sont identiques, par exemple $0 = 0$,
- Les deux expressions sont équivalentes modulo remplacement des arguments formels par les arguments re'els, par exemple
$$\lambda x : \iota\ ((\lambda y : \iota\ y)\ x) = \lambda x : \iota\ x$$
- Un simple calcul suffit pour montrer l'égalité, mais ce calcul n'est pas nécessairement limité au remplacement des arguments formels par les arguments réels, par exemple
$$(\times\ (S(S(S(S(S(S\ 0))))))\ (S(S\ 0))) = (\times\ (S(S(S(S\ 0))))\ (S(S(S\ 0))))$$
- Un réel raisonnement est demandé pour montrer l'égalité en question. Par exemple
$$f = \lambda x : \iota\ 0$$
où f est la fonction telle que $(f\ n) = 0$ si n est la somme de quatre carrés et $(f\ n) = 1$ sinon.

Dire que les trois premières égalités, à la différence de la quatrième, peuvent être établies par un simple calcul signifie qu'il existe des termes particuliers (*normaux*) et un algorithme qui transforme chaque terme en un terme normal (*sa forme normale*) tel que si la proposition $t = u$ est démontrable alors les termes t et u ont la même forme normale.

Les égalités démontrées en utilisant les axiomes de l'arithmétique, sans le schéma de récurrence, peuvent être établies par un simple calcul, car la forme normale d'un terme de l'arithmétique peut se calculer par les règles
$$(+\ n\ 0)\ \rhd n$$
$$(+\ n\ (S\ m))\ \rhd (S\ (+\ n\ m))$$
$$(\times\ n\ 0)\ \rhd 0$$
$$(\times\ n\ (S\ m))\ \rhd (+\ (\times\ n\ m)\ n)$$
$$(S\ n)\ \rhd (S\ n') \text{ si } n\ \rhd n'$$
qui définissent une relation de réduction fortement normalisable et confluente. Les égalités démontrées en utilisant la β-conversion (ou la $\beta\eta$-conversion) peuvent être établie par un simple calcul car la β-réduction (et

la $\beta\eta$-réduction) sont fortement normalisables et confluentes. En revanche les égalités de fonctions démontrées en utilisant l'axiome d'extensionnalité (comme l'égalité $f = \lambda x : \iota\ 0$ ci-dessus) ne peuvent pas, en général, être établie par un simple calcul.

En ce qui concerne cette notion de calcul, deux traditions s'opposent.

- Selon la première tradition la β-réduction (le remplacement des arguments formels par les arguments réels) n'est pas la seule manière calculatoire d'établir une égalité, il faut donc élargir les règles de réduction par de nouvelles règles.
- Selon la seconde tradition, toute transformation peut (et doit) se ramener à la β-réduction par un codage approprié.

Dans un cas comme dans l'autre, il importe dans l'étude d'un système de λ-calcul de caractériser les fonctions que l'on peut calculer par la β-réduction.

4.5.1 La β-expressivité et la calculabilité

Définition Soit un système de λ-calcul tel que la réduction soit confluente et les entiers normaux. Une fonction f de $\mathcal{N}^n$ dans $\mathcal{N}$ est dite β-exprimable s'il existe un terme t tel que pour tout n-uplet d'entiers $(a_1, ..., a_n)$, le terme $(t\ a_1\ ...\ a_n)$ soit bien formé et se β-réduise sur $(f\ a_1\ ...\ a_n)$.

Proposition Dans un système de λ-calcul tel que la réduction soit confluente et les entiers normaux, toute fonction β-exprimable est semi-calculable.

Corollaire Toute fonction β-exprimable dans le λ-calcul pur est semi-calculable.

Théorème Toute fonction semi-calculable est β-exprimable dans le λ-calcul pur.

Proposition Dans un système de λ-calcul tel que la réduction soit confluente et *normalisable* et les entiers normaux, toute fonction β-exprimable est calculable.

Proposition Il n'existe pas de système de λ-calcul tel que la réduction soit confluente et normalisable et les entiers normaux tel que toute fonction calculable soit β-exprimable.
Démonstration L'idée de la démonstration est une variation sur l'argument diagonal de Cantor. On suppose qu'il existe tel système de λ-calcul. A chaque terme représentant une fonction calculable à un argument on associe un entier (son code de Gödel). Connaissant un terme t exprimant une fonction f et un entier n on peut réduire le terme $(t\ n)$ de façon à calculer $(f\ n)$, donc la fonction φ à deux arguments telle que

- $(\varphi\ a\ b) = (f\ b)$ si a est le code d'un terme exprimant la fonction f,
- $(\varphi\ a\ b) = 0$ sinon

est calculable. (Cette fonction qui prend en argument, un programme a et un entier b et retourne le résultat du programme a appliqué à l'entrée b est un interpréteur). On considère ensuite la fonction calculable $(g\ a) = 1 + (\varphi\ a\ a)$. Cette fonction est exprimée par un terme t, soit n le code de t. On a $(\varphi\ n\ n) = (g\ n) = 1 + (\varphi\ n\ n)$ ce qui est contradictoire.

4.5.2 Les fonctions exprimables sur les entiers de Peano

Proposition Les fonctions β-exprimables dans le λ-calcul simplement typé sont les fonctions constantes et les fonctions ajoutant une constante à l'un de leurs arguments.
Démonstration Les fonction constantes sont exprimées par les termes de la forme

$$\lambda x_1 : \iota\ ...\ \lambda x_n : \iota\ (S\ (...(S\ 0)...))$$

et les fonctions qui ajoutent une constante à l'un de leurs arguments par les termes de la forme

$$\lambda x_1 : \iota \ ... \ \lambda x_n : \iota \ (S \ (...(S \ x_i)...))$$

Réciproquement, soit f une fonction β-exprimée par le terme t. Cette fonction est également exprimée par la forme normale de t, on peut donc sans perte de généralité supposer t normal. On montre par récurrence sur la taille de t que f est une fonction constante ou une fonction qui ajoute une constante à l'un de ses arguments. Soit $t = \lambda x_1 : \iota \ ... \ \lambda x_p : \iota \ (x \ u_1 \ ... \ u_k)$.

- Si $x = x_i$ alors $k = 0$ et $t = \lambda x_1 : \iota ... \lambda x_n : \iota \ x_i$, cette fonction est la fonction qui ajoute 0 à son $i^{\text{ème}}$ argument.

- Si $x = 0$ alors $k = 0$ et $t = \lambda x_1 : \iota ... \lambda x_n : \iota \ 0$, cette fonction est la fonction constante égale à zéro.

- Si $x = S$ alors $k = 0$ ou $k = 1$

  - si $k = 1$ alors $t = \lambda x_1 : \iota \ ... \ \lambda x_n : \iota \ (S \ u)$, par hypothèse de récurrence, le terme $\lambda x_1 : \iota \ ... \ \lambda x_n : \iota \ u$ exprime une fonction constante égale à r ou une fonction qui ajoute r à son $i^{\text{ème}}$ argument. Dans le premier cas, la fonction f est la fonction constante égale à $r + 1$, dans le second c'est la fonction qui ajoute $r + 1$ à son $i^{\text{ème}}$ argument.
  - Si $k = 0$ alors $t = \lambda x_1 : \iota \ ... \ \lambda x_n : \iota \ S$ et f est la fonction qui ajoute 1 à son $(n+1)^{\text{ème}}$ argument.

4.5.3 Les fonctions exprimables sur les entiers de Church

Une des raisons pour lesquelles l'ensemble des fonctions β-exprimables en λ-calcul simplement typé est si pauvre est l'impossibilité d'exprimer des fonctions définies par récurrence sur l'un de leurs arguments. Pour pallier ce manque, une idée est de représenter les entiers non plus axiomatiquement avec des symboles 0 et S mais comme des itérateurs : les entiers de Church.

Définition Les entiers de Church

L'entier n est représenté comme le terme $\lambda x : \iota \ \lambda f : \iota \to \iota \ (f \ (f \ ...(f \ x)...))$ avec n occurrences du symbole f. Les entiers de Church sont de type $\iota \to (\iota \to \iota) \to \iota$.

Remarque Si g est une fonction de type $\iota \to \iota$ alors $\lambda x : \iota \ (n \ x \ g)$ est la fonction g itérée n fois.

Définition L'ensemble des *polynômes* est le plus petit ensemble de fonctions de $\mathcal{N}^n$ dans $\mathcal{N}$ clos par composition et contenant la fonction nulle, la fonction successeur, les projections, l'addition et la multiplication.

L'ensemble des *polynômes étendus* est le plus petit ensemble de fonctions de $\mathcal{N}^n$ dans $\mathcal{N}$ clos par composition et contenant la fonction nulle, la fonction successeur, les projections, l'addition, la multiplication, *la fonction caractéristique de* $\{0\}$ *et la fonction caractéristique de* $\mathcal{N} - \{0\}$

Proposition Soit Γ le contexte $x_1 : \iota \to (\iota \to \iota) \to \iota, \ ..., \ x_n : \iota \to (\iota \to \iota) \to \iota, \ z_1 : \iota, \ ..., \ z_r : \iota, \ f : \iota \to \iota$ et t un terme normal tel que $\Gamma \vdash t : \iota$.

Il existe polynôme P et un entier $i \in \{1, ..., r\}$ tels que quelque soient $a_1, ..., a_n$ des entiers de Church non nuls, le terme $t[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ ait la forme normale $(f \ (...(f \ z_i)...))$ avec $(P \ a_1 \ ... \ a_n)$ symboles f.
Démonstration Par récurrence sur la taille de t. Le terme t est de type ι donc $t = (x \ u_1 \ ... \ u_k)$.

- Si $x = z_j$, on pose $(P \ a_1 \ ... \ a_n) = 0$ et $i = j$.

- Si $x = f$, $t = (f \ u)$, il existe un polynôme Q et un indice j tels que quelque soient $a_1, ..., a_n$ des entiers de Church non nuls, le terme $u[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ ait pour forme normale $(f \ (...(f \ z_j)...))$ avec $(Q \ a_1 \ ... \ a_n)$ symboles f. On pose $(P \ a_1 \ ... \ a_n) = 1 + (Q \ a_1 \ ... \ a_n)$ et $i = j$.

- Si $x = x_m$ alors $t = (x_m\ u_1\ (\lambda z_{r+1} : \iota\ u_2))$ ou $t = (x_m\ u_1\ f)$.

  Par hypothèse de récurrence, il existe un polynôme Q et un entier j tel que le terme $u_1[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ ait pour forme normale $(f\ (...(f\ z_j)...))$ avec $(Q\ a_1\ ...\ a_n)$ symboles f.

  Si t a la forme $(x_m\ u_1\ (\lambda z_{r+1} : \iota\ u_2))$ alors il existe un polynôme Q' et un entier j' tels que le terme $u_2[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ ait pour forme normale $(f\ (...(f\ z_{j'})...))$ avec $(Q'\ a_1\ ...\ a_n)$ symboles f. Si t a la forme $(x_m\ u_1\ f)$ on pose $(Q'\ a_1\ ...\ a_n) = 1$ et $j' = r + 1$.

  Si $j' = r + 1$ alors on pose $(P\ a_1\ ...\ a_n) = (Q\ a_1\ ...\ a_n) + a_m(Q'\ a_1\ ...\ a_n)$ et $i = j$ sinon on pose $(P\ a_1\ ...\ a_n) = (Q'\ a_1\ ...\ a_n)$ et $i = j'$. Comme $a_m \neq 0$ le terme $t[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ a pour forme normale $(f\ (...(f\ z_i)...))$ avec $(P\ a_1\ ...\ a_n)$ symboles f.

Corollaire Soit Γ le contexte $x_1 : \iota \to (\iota \to \iota) \to \iota,\ ...,\ x_n : \iota \to (\iota \to \iota) \to \iota,\ z : \iota,\ f : \iota \to \iota$ et t un terme normal tel que $\Gamma \vdash t : \iota$. Il existe un polynôme P tel que quelque soient $a_1, ..., a_n$ des entiers de Church non nuls, le terme $t[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ ait la forme normale $(f\ (...(f\ z)...))$ avec $(P\ a_1\ ...\ a_n)$ symboles f.

Proposition Soit Γ le contexte $x_1 : \iota \to (\iota \to \iota) \to \iota,\ ...,\ x_n : \iota \to (\iota \to \iota) \to \iota,\ z : \iota,\ f : \iota \to \iota$ et t un terme normal tel que $\Gamma \vdash t : \iota$. Il existe polynôme étendu P tel que quelque soient $a_1, ..., a_n$ des entiers de Church (nuls ou non), le terme $t[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ ait pour forme normale $(f\ (...(f\ z)...))$ avec $(P\ a_1\ ...\ a_n)$ symboles f.
Démonstration Pour chaque partie A de $\{1, ..., n\}$ il existe un polynôme P_A tels que quelque soient $a_1, ..., a_n$ des entiers de Church tels que $a_i = 0$ si et seulement si $i \in A$ le terme $t[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ ait pour forme normale $(f\ (...(f\ z)...))$ avec $(P\ a_1\ ...\ a_n)$ symboles f. Soit χ_A le polynôme étendu tel que

- $(\chi_A\ a_1\ ...\ a_n) = 1$ si ($a_i = 0$ si et seulement si $i \in A$)
- $(\chi_A\ a_1\ ...\ a_n) = 0$ sinon.

On pose
$$(P\ a_1\ ...\ a_n) = \sum_A (\chi_A\ a_1\ ...\ a_n)(P_A\ a_1\ ...\ a_n)$$

Théorème (Schwichtenberg) Les fonctions β-exprimables dans le λ-calcul simplement typé sur les entiers de Church sont les polynômes étendus.
Démonstration La fonction nulle est exprimable par le terme
$$\lambda x_1 : \iota \to (\iota \to \iota) \to \iota\ ...\ \lambda x_n : \iota \to (\iota \to \iota) \to \iota\ \lambda z : \iota\ \lambda f : \iota \to \iota\ z$$
la fonction successeur par le terme
$$\lambda x : \iota \to (\iota \to \iota) \to \iota\ \ \lambda z : \iota\ \lambda f : \iota \to \iota\ (f\ (x\ z\ f))$$
les projections par les termes
$$\lambda x_1 : \iota \to (\iota \to \iota) \to \iota\ ...\ \lambda x_n : \iota \to (\iota \to \iota) \to \iota\ x_i$$
l'addition par
$$\lambda x : \iota \to (\iota \to \iota) \to \iota\ \lambda y : \iota \to (\iota \to \iota) \to \iota\ \lambda z : \iota\ \lambda f : \iota \to \iota\ (x\ (y\ z\ f)\ f)$$
la multiplication par
$$\lambda x : \iota \to (\iota \to \iota) \to \iota\ \lambda y : \iota \to (\iota \to \iota) \to \iota\ \lambda z : \iota\ \lambda f : \iota \to \iota\ (x\ z\ \lambda w : \iota\ (y\ w\ f))$$
la fonction caractéristique de $\{0\}$ par
$$\lambda x : \iota \to (\iota \to \iota) \to \iota\ \ \lambda z : \iota\ \lambda f : \iota \to \iota\ (x\ (f\ z)\ \lambda w : \iota\ z)$$

la fonction caractéristique de $\mathcal{N} - \{0\}$ par

$$\lambda x : \iota \to (\iota \to \iota) \to \iota \;\; \lambda z : \iota \; \lambda f : \iota \to \iota \; (x \; z \; \lambda w : \iota \; (f \; z))$$

Les polynômes étendus sont donc β-exprimables dans le λ-calcul simplement typé. Réciproquement, soit F une fonction β-exprimée par un terme t. La forme normale du terme t est ou bien

$$\lambda x_1 : \iota \to (\iota \to \iota) \to \iota \; ... \; \lambda x_n : \iota \to (\iota \to \iota) \to \iota \; x_i$$

ou bien

$$\lambda x_1 : \iota \to (\iota \to \iota) \to \iota \; ... \; \lambda x_n : \iota \to (\iota \to \iota) \to \iota \; \lambda z : \iota \; (x_i \; z)$$

ou bien

$$t = \lambda x_1 : \iota \to (\iota \to \iota) \to \iota \; ... \; \lambda x_n : \iota \to (\iota \to \iota) \to \iota \; \lambda z : \iota \; \lambda f : \iota \to \iota \; u$$

Dans les deux premiers cas, la fonction F est une projection, c'est donc un polynôme étendu. Dans le dernier cas il existe un polynôme étendu P tel que quelque soient $a_1, ..., a_n$ des entiers de Church, le terme $u[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$ ait pour forme normale $(f \; (...(f \; z)...))$ avec $(P \; a_1 \; ... \; a_n)$ symboles f. La forme normale du terme $(t \; a_1 \; ... \; a_n)$ est donc l'entier de Church $(P \; a_1 \; ... \; a_n)$ et la fonction F est le polynôme étendu P.

Remarque L'ensemble des fonctions *primitives récursives* est le plus petit ensemble contenant la fonction nulle, la fonction successeur, les projections et clos par composition et définition par récurrence. Toutes les fonctions primitives récursives ne sont pas β-exprimables dans le λ-calcul simplement typé (par exemple la fonction $n \mapsto 2^n$ est primitive récursive mais ce n'est pas un polynôme étendu).

Bien que les entiers de Church soient des itérateurs, l'ensemble des fonctions β-exprimables dans le λ-calcul simplement typé n'est pas clos par définition par récurrence. En effet, il est possible en λ-calcul simplement typé d'itérer une fonction seulement si cette fonction a le type $\iota \to \iota$. En particulier il est impossible d'itérer une fonction comme $n \mapsto 2 \times n$ qui associe un entier à chaque entier.

Bibliographie

[1] H. Barendregt, Lambda Calculi with Types, Handbook of Logic in Computer Science, (S. Abramsky, D.M. Gabbay, T.S.E. Maibaum Ed.), *Oxford University Press* (1992).

[2] H. Geuvers, The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi, *Logic in Computer Science* (1992) pp. 453-460.

[3] J.Y. Girard, Y. Lafont, P. Taylor, Proofs and Types, *Cambridge University Press* (1989).

[4] G. Huet, Initiation au λ-calcul, *Notes de cours de DEA* (1988-1993).

[5] J.R. Hindley, J.P. Seldin, Introduction to combinators and λ-calculus, *Cambridge University Press* (1986).

[6] J.L. Krivine, Lambda-calul, types et modèles, *Masson* (1990).

Chapitre 5

La théorie des types simples

Au chapitre précédent, nous avons vu un langages de termes d'ordre supérieur : le λ-calcul simplement typé. Ce chapitre est consacré au langage des propositions correspondant : la *théorie des types simples* aussi appelée *théorie des types de Church* ou *logique d'ordre supérieur*.

5.1 La théorie naïve des ensembles

5.1.1 Les fonctions et les ensembles

L'arithmétique du premier ordre comporte des symboles qui expriment des fonctions (les symboles d'individu de fonction 0, S, $+$, $\times$) et des symboles qui expriment des ensembles (le symbole de prédicat $=$). Mais il n'est pas possible de construire des expressions qui expriment des fonctions ou des ensembles qui ne sont pas donnés comme symboles de base. L'idée de la *théorie naïve des ensembles* est

- d'étendre le langage des termes du premier ordre pour pouvoir exprimer des fonctions et des ensembles (prédicats),
- d'étendre le langage des propositions de façon à pouvoir quantifier sur les fonctions et les ensembles.

Informellement, on exprime une fonction par un terme du premier ordre qui contient une variable (exemple : $2 \times x$). Nous avons vu que pour lever l'ambiguïté entre fonction et valeur prise par la fonction, il est préférable de noter la fonction $\lambda x\ (2 \times x)$ (ou $x \mapsto (2 \times x)$). De même, on exprime un ensemble (prédicat) par une proposition qui contient une variable libre. Par exemple, l'ensemble des nombres pairs par la proposition $\exists y\ (x = 2 \times y)$. Ici encore, pour lever l'ambiguïté entre ensemble et appartenance d'un objet à l'ensemble, on note l'ensemble $\lambda x\ \exists y\ (x = 2 \times y)$ (ou $\{x \mid \exists y\ (x = 2 \times y)\}$). Quand on applique le prédicat (l'ensemble) $pair = \lambda x\ \exists y\ (x = 2 \times y)$ au nombre 0 par exemple, on obtient la proposition $(pair\ 0) = (\lambda x\ \exists y\ (x = 2 \times y)\ 0)$ qui se β-réduit en la proposition $\exists y\ (0 = 2 \times y)$.

Aux axiomes habituels, par exemple de l'arithmétique, on ajoute les *axiomes de conversion* :

$$t = u$$

si t et u sont deux termes tels que $t \rhd u$ et

$$P \Leftrightarrow Q$$

si P et Q sont des propositions telles que $P \rhd Q$.

5.1.2 Le λ-calcul et le schéma de compréhension

Dans la définition d'une théorie mathématique, il y a toujours une alternative : ou bien on donne une notation explicite pour les objets de la théorie et des axiomes qui expriment leur propriétés, ou bien on donne des axiomes exprimant l'existence d'objets qui vérifient ces propriétés.

Ainsi, on peut définir la théorie des groupes en donnant un symbole de fonction $+$ d'arité 2 pour la loi, un symbole d'individu e pour l'élément neutre et un symbole de fonction I d'arité 1 pour l'opposé et donner les axiomes suivants :

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

$$\forall x\ ((x + e) = x\ \wedge\ (e + x) = x)$$

$$\forall x\ ((x + (I\ x)) = e\ \wedge\ ((I\ x) + x) = e)$$

Ou bien on peut se donner uniquement le symbole de fonction $+$ et les axiomes

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

$$\exists e\ (\forall x\ ((x + e) = x\ \wedge\ (e + x) = x)\ \wedge\ \forall x\ \exists y\ ((x + y) = e\ \wedge\ (y + x) = e))$$

Quand une théorie est exprimée par des axiomes d'existence, il est toujours possible de la reformuler en explicitant son langage (par skolémisation). En revanche, la transformation inverse n'est pas toujours évidente. Dans cet exemple, il n'est pas évident d'exprimer la théorie des groupes en supprimant le symbole $+$.

Le λ-calcul est une formulation explicite de la théorie des fonctions et des ensembles puisqu'on donne une notation pour ces objets. Une formulation alternative consiste à ne pas donner de notation pour les fonctions et à donner des axiomes d'existence (*axiomes de compréhension*). Pour chaque terme t du langage on donne un axiome

$$\exists f\ \forall x_1\ ...\ \forall x_n\ ((f\ x_1\ ...\ x_n) = t)$$

De même pour chaque proposition P on donne un axiome

$$\exists E\ \forall x_1\ ...\ \forall x_n\ ((E\ x_1\ ...\ x_n) \Leftrightarrow P)$$

Ainsi on ne dispose plus de la notation $\lambda x\ \exists y\ (x = 2 \times y)$ (ou $\{x \mid \exists y\ (x = 2 \times y)\}$) pour l'ensemble des entiers pairs, mais on a un axiome

$$\exists E\ \forall x\ ((E\ x) \Leftrightarrow \exists y\ (x = 2 \times y))$$

Quand on skolemise cette théorie on obtient un langage pour les fonctions et les ensembles qui ressemble au λ-calcul, mais qui présente, malgré tout, certaines différences. Ce langage est appelé langage des combinateurs (voir, par exemple, [2]).

5.1.3 L'extensionnalité

Outre les axiomes habituels de l'égalité

$$\forall x\ (x = x)$$

$$\forall x\ \forall y\ ((x = y) \Rightarrow (P[z \leftarrow x] \Leftrightarrow P[z \leftarrow y]))$$

il faut poser des axiomes qui permettent de démontrer l'égalité de deux fonctions ou de deux ensembles : les *axiomes d'extensionnalité*.

$$\forall f\ \forall g\ ((\forall x\ (f\ x) = (g\ x)) \Rightarrow f = g)$$

$$\forall E\ \forall F\ ((E \Leftrightarrow F) \Rightarrow E = F)$$

5.1.4 Le paradoxe de Russell

De nombreux systèmes voisins de cette théorie ont été proposée dans l'histoire des mathématiques (la théorie des ensembles de Cantor (1872), la *begriffschrift* de Frege (1879), le système de Church basé sur le λ-calcul pur (1932) peuvent être considérés comme tels). Malheureusement, tous ces langages sont contradictoires. Une contradiction est donnée par le paradoxe de Russell.

Soit $R = \lambda x \neg (x\ x)$ (que l'on peut également noter $R = \{x \mid \neg (x\ x)\}$) l'ensemble des ensembles qui ne se s'appartiennent pas eux-mêmes. Par définition l'ensemble R s'appartient si et seulement si il ne s'appartient pas, ce qui est contradictoire. Plus précisément la proposition "l'ensemble R s'appartient" s'écrit $A = (R\ R) = (\lambda x \neg (x\ x)\ \lambda x \neg (x\ x))$. Cette proposition se réduit sur $\neg (\lambda x \neg (x\ x)\ \lambda x \neg (x\ x))$ c'est-à-dire sur $\neg A$. On a donc un axiome $A \Leftrightarrow \neg A$ ("L'ensemble R s'appartient si et seulement si il ne s'appartient pas"). De cet axiome on peut déduire $\neg A$ donc A donc $\perp$.

De façon similaire, si on a formulé la théorie naïve des ensembles avec un schéma d'axiome de compréhension on peut montrer que l'axiome

$$\exists R\ \forall x\ ((R\ x) \Leftrightarrow \neg (x\ x))$$

est contradictoire.

Historiquement, le premier paradoxe a été découvert en 1897 par Burali-Forti, il a ensuite été simplifié par Russell en 1902. Par la suite, Rosser et Kleene, puis Curry, ont montré que ce paradoxe pouvait également s'exprimer dans un système de Church basé sur le λ-calcul pur. Le terme A qui se réduit sur $\neg A$ puis sur $\neg \neg A$ etc. est le premier exemple de terme non normalisable trouvé dans le λ-calcul pur. Plus généralement si t est un terme quelconque le terme

$$u = (\lambda x\ (t\ (x\ x))\ \lambda x\ (t\ (x\ x)))$$

se réduit sur $(t\ u)$ et est donc un point fixe de t. Un point fixe de l'identité, qui se réduit sur lui-même, est obtenu en prenant $t = \lambda x\ x$. On retombe ainsi sur le terme $(\lambda x\ (x\ x)\ \lambda x\ (x\ x))$.

5.1.5 La théorie des ensembles et la théorie des types

En théorie naïve des ensemble

- tout prédicat est un objet,
- tout prédicat peut s'appliquer à tout objet.

La conjonction de ces deux phénomènes permet de construire une proposition contradictoire : le paradoxe de Russell. Pour éviter ce paradoxe deux voies sont possibles : abandonner le premier principe ou abandonner le second. Ces deux voies mènent à deux langages différents : la théorie des ensembles et la théorie des types.

5.2 La théorie des ensembles

5.2.1 La formation des termes

L'idée de la *théorie des ensembles* de Zermelo (1908) (et de sa variante la *théorie des ensembles de Zermelo-Fraenkel*) est d'abandonner le principe selon lequel tout prédicat est un objet. Quand P est une proposition, il n'est pas toujours possible de former l'ensemble $\lambda x\ P$ (encore noté $\{x \mid P\}$). Autrement dit, il n'est pas toujours possible de former un ensemble "en compréhension" c'est-à-dire en donnant une propriété caractéristique de ses éléments. On ne peut former un tel ensemble que dans les cas particuliers suivants.

- Si A et B sont des termes alors on peut former le terme $\{x \mid ((x = A) \vee (x = B))\}$ appelé la paire A, B.
- Si A est un terme alors on peut former le terme $\{x \mid \exists y\ ((A\ y) \wedge (y\ x))\}$ appelé l'union des éléments de A.

- Si A est un terme alors on peut former le terme $\{x \mid \forall y\ (x\ y) \Rightarrow (A\ y)\}$ appelé l'ensemble des parties de A.

- Si A est un terme et P une proposition quelconque, alors on peut former le terme $\{x \mid (A\ x) \wedge P\}$ appelé le sous-ensemble de A des éléments qui vérifient P.

Ainsi le terme $\{x \mid \neg\ (x\ x)\}$ n'est pas bien formé.

La question de savoir si un ensemble appartient à un autre ensemble est en revanche toujours pertinente. Par exemple l'ensemble vide ne s'appartient pas car on peut démontrer la proposition (bien formée) $\neg(\{x \mid (A\ x) \wedge \bot\}\ \{x \mid (A\ x) \wedge \bot\})$ (car cette proposition se réduit sur $\neg((A\ \{x \mid (A\ x) \wedge \bot\}) \wedge \bot)$ qui est démontrable).

5.2.2 La restriction du schéma de compréhension

Dans le cas où on ne donne pas de notation explicite pour les ensembles, mais des axiomes d'existence, on obtient une formulation alternative qui comprend les axiomes suivants.

- $\forall A\ \forall B\ \exists C\ \forall x\ ((C\ x) \Leftrightarrow ((x = A) \vee (x = B)))$ (axiome de la paire),

- $\forall A\ \exists C\ \forall x\ ((C\ x) \Leftrightarrow (\exists y\ ((A\ y) \wedge (y\ x))))$ (axiome de la réunion),

- $\forall A\ \exists C\ \forall x\ ((C\ x) \Leftrightarrow (\forall y\ ((x\ y) \Rightarrow (A\ y))))$ (axiome des parties),

- $\forall y_1\ ...\ \forall y_n\ \forall A\ \exists C\ \forall x\ ((C\ x) \Leftrightarrow ((A\ x) \wedge P))$ (sous-ensemble (axiome de compréhension restreint)) où $y_1, ..., y_n$ sont les variables libres de P distinctes de x.

5.2.3 Expression au premier ordre

On peut exprimer la théorie des ensembles comme une théorie de premier ordre. Si A et B sont deux termes, on ne peut pas noter $(A\ B)$ la proposition "B appartient à A") mais on doit introduire un nouveau symbole de prédicat $\in$ et noter cette proposition $B \in A$.

Les axiomes ci-dessus se traduisent ainsi.

- $\forall A\ \forall B\ \exists C\ \forall x\ (x \in C \Leftrightarrow ((x = A) \vee (x = B)))$ (axiome de la paire),

- $\forall A\ \exists C\ \forall x\ (x \in C \Leftrightarrow (\exists y\ (y \in A \wedge x \in y)))$ (axiome de la réunion),

- $\forall A\ \exists C\ \forall x\ (x \in C \Leftrightarrow (\forall y\ (y \in x \Rightarrow x \in A)))$ (axiome des parties),

- $\forall y_1\ ...\ \forall y_n\ \forall A\ \exists C\ \forall x\ (x \in C \Leftrightarrow ((x \in A) \wedge P))$ (sous-ensemble (axiome de compréhension restreint)) où $y_1, ..., y_n$ sont les variables libres de P distinctes de x.

5.2.4 Un peu de mathématiques

Les fonctions

Nous nous sommes concentrés sur la construction des ensembles (prédicats) négligeant la construction des fonctions. Cela est du au fait que, traditionnellement, en théorie des ensembles on code les fonctions comme des relations, c'est-à-dire des prédicats à plusieurs arguments ou encore des ensembles de couples. Et il n'y a donc pas de termes de fonction à proprement parler, ni de schéma de compréhension pour les fonctions. Mais, *a priori*, rien n'interdit de concevoir une théorie similaire dans laquelle les fonctions également sont des objets primitifs.

Les entiers

Traditionnellement, en théorie des ensembles, il n'y a qu'un seul objet de base (l'ensemble vide). Les entiers ne sont donc pas des objets de base et il faut les construire. Une possibilité est de se donner un axiome d'existence d'un ensemble B infini, et de construire les entiers comme les cardinaux finis de B, c'est-à-dire comme des objets de l'ensemble des parties de l'ensemble des parties de B.

Une autre possibilité est de construire les entiers de façon à ce que l'entier n soit l'ensemble des entiers strictement inférieur à n. On peut montrer ainsi l'existence de l'objet 0, 1, 2, etc. En revanche il est nécessaire de poser un axiome d'existence de l'ensemble des entiers.

De manière générale, il est toujours nécessaire de poser l'existence d'un ensemble infini (qu'il s'agisse d'un ensemble infini B quelconque ou de l'ensemble des entiers) car en l'absence d'un tel axiome, il y a des modèles de la théorie des ensembles dans lesquels tout ensemble est fini.

5.3 La théorie des types

5.3.1 Les symboles de fonctions et de prédicats

L'idée de la *théorie des types* de Whitehead et Russell (1910) (puis simplifiée par Ramsey, Chwistek et Church) est d'abandonner le principe selon lequel tout prédicat ou toute fonction peut s'appliquer à tout objet. On associe ainsi un type à chaque expression du langage. Les termes du langage *et les propositions* sont des λ-termes simplement typés d'un λ-calcul à deux types de base : ι et o. Intuitivement ι est le type des objets de base et o le type des propositions. Les ensembles sont exprimés comme des fonctions à valeur dans le type des propositions, par exemple, l'ensemble des nombres pairs comme la fonction qui associe la proposition "n est pair" au nombre n. On donne donc le type $\iota \to \ldots \to \iota \to \iota$ aux symboles de fonction d'arité n et le type $\iota \to \ldots \to \iota \to o$ aux symboles de prédicat d'arité n. Par exemple, on donne les types suivants aux symboles de l'arithmétique.

$$0 : \iota$$

$$S : \iota \to \iota$$

$$+ : \iota \to \iota \to \iota$$

$$\times : \iota \to \iota \to \iota$$

$$=: \iota \to \iota \to o$$

5.3.2 Les connecteurs et les quantificateurs

Donner un type adéquat aux symboles de prédicat permet de donner le type o aux proposition atomiques. Pour donner également ce type aux propositions formée par un connecteur, on donne les types suivants aux symboles $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$.

$$\wedge\ : o \to o \to o$$

$$\vee\ : o \to o \to o$$

$$\Rightarrow : o \to o \to o$$

$$\Leftrightarrow : o \to o \to o$$

$$\neg : o \to o$$

Ainsi si P et Q sont deux termes de type o (deux propositions) le terme $(\wedge\ P\ Q)$ (aussi noté $P \wedge Q$) est de type o.

Le cas des quantificateurs, est un peu plus complexe puisque dans l'expression $\forall\, x\ (x\ =\ x)$ il semble que le quantificateur $\forall$ soit une fonction qui prenne en argument une variable de type ι et un terme de type o. Il est impossible d'exprimer dans le type du symbole $\forall$ le fait que son premier argument doit être une variable.

Par ailleurs, la variable x est liée dans $\forall\ x\ (x\ =\ x)$ donc cette proposition ne peut pas se construire par simple application d'un symbole $\forall$ à deux termes x et $(x\ =\ x)$. Une possibilité est de considérer le symbole $\forall$ comme un symbole primitif du λ-calcul (au même titre que λ) et d'ajouter une règle de formation, de façon à ce que si P est un terme de type o et x une variable de type ι, alors $\forall x\ P$ est un terme de type o. Une autre possibilité est de considérer la proposition $\forall x\ P$ comme une abréviation pour $\forall\ \lambda x : \iota\ P$. Ainsi on donne le type

$$\forall : (\iota \to o) \to o$$

et la variable x est bien liée dans $\forall x\ P$. De même on donne le type

$$\exists : (\iota \to o) \to o$$

Ces quantificateurs s'appliquent donc à des objets de type $\iota \to o$ c'est-à-dire à des ensembles (prédicats) d'objets de base. Intuitivement $\forall\ E$ signifie que E est l'ensemble de tous les objets de type ι et $\exists\ E$ signifie que E contient au moins un élément.

5.3.3 L'égalité et la quantifications sur les objets fonctionnels

Dans le langage défini ci-dessus on ne peut exprimer l'égalité de deux objets que quand ces objets sont de type ι, de même on ne peut quantifier que sur des variables de type ι. Pour exprimer des égalités d'objets de tous les types et quantifier sur les objets de tous les types, on introduit un nombre infini des symboles $=$, $\forall$, $\exists$, paramétrés par leur type.

$$=_A : A \to A \to o$$

$$\forall_A : (A \to o) \to o$$

$$\exists_A : (A \to o) \to o$$

La possibilité de former un terme de type o en quantifiant sur une variable de type o (ou une variable de type A telle que o ait une occurrence dans o) permet de former des propositions exprimant qu'une propriété s'applique à une classe d'objets comprenant cette proposition elle-même. Par exemple, la proposition

$$\forall_o p\ (p \Rightarrow p)$$

exprime le fait que toute proposition s'implique elle-même. En particulier, elle exprime le fait qu'elle s'implique elle-même. Un formalisme logique permettant l'expression d'une telle proposition partiellement auto-référente est dit *imprédicatif*.

5.3.4 Définition

Les termes de la théorie des types sont des λ-termes simplement typés, à deux types de base ι et o, formés sur un contexte comprenant une infinité de variables de chaque type et des variables distinguées

$$=_A : A \to A \to o$$

$$\wedge : o \to o \to o$$

$$\vee : o \to o \to o$$

$$\Rightarrow : o \to o \to o$$

$$\Leftrightarrow : o \to o \to o$$

$$\neg : o \to o$$

$$\forall_A : (A \to o) \to o$$

$$\exists_A : (A \to o) \to o$$

On pose ensuite les axiomes suivants.
Égalité :

$$\forall_A x\ (x =_A x)$$

$$\forall_A x\ \forall_A y\ ((x =_A y) \Rightarrow (P[z \leftarrow x] \Leftrightarrow P[z \leftarrow y]))$$

Extensionnalité :

$$\forall_{A \to B} f\ \forall_{A \to B} g\ (\forall_A x\ (f\ x) =_B (g\ x)) \Rightarrow f =_{A \to B} g$$

$$\forall_o E\ \forall_o F\ (E \Leftrightarrow F) \Rightarrow E =_o F$$

Conversion :
Si t et u sont deux termes de type A et $t\ \triangleright u$

$$t =_A u$$

Si P et Q sont deux termes de type o et $P\ \triangleright Q$

$$P \Leftrightarrow Q$$

On note $\Gamma \vdash P$ le jugement *la proposition P est démontrable sous les hypothèses* Γ. Ce jugement est défini par les règles de déduction naturelle.

$$\frac{}{\Gamma \vdash P}\text{axiome}\quad \text{si } P \in \Gamma$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \Rightarrow\text{-intro}$$

$$\frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q} \Rightarrow\text{-élim}$$

$$\frac{\Gamma, P \vdash Q \quad \Gamma, Q \vdash P}{\Gamma \vdash P \Leftrightarrow Q} \Leftrightarrow\text{-intro}$$

$$\frac{\Gamma \vdash P \Leftrightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q} \Leftrightarrow\text{-élim}$$

$$\frac{\Gamma \vdash P \Leftrightarrow Q \quad \Gamma \vdash Q}{\Gamma \vdash P} \Leftrightarrow\text{-élim}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \wedge\text{-intro}$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \wedge\text{-élim}$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \wedge\text{-élim}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \vee\text{-intro}$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \vee\text{-intro}$$

$$\frac{\Gamma \vdash P \vee Q \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R} \vee\text{-élim}$$

$$\frac{\Gamma, P \vdash \bot}{\Gamma \vdash \neg P} \neg\text{-intro}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash \neg P}{\Gamma \vdash \bot}\ \neg\text{-élim}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \bot\text{-élim}$$

$$\frac{\Gamma \vdash (E\ x)}{\Gamma \vdash \forall_A E}\ \forall\text{-intro} \quad x \text{ de type } A, \text{ n'apparaît pas dans } \Gamma$$

$$\frac{\Gamma \vdash \forall_A E}{\Gamma \vdash (E\ t)}\ \forall\text{-élim} \quad t \text{ de type } A$$

$$\frac{\Gamma \vdash (E\ t)}{\Gamma \vdash \exists_A E}\ \exists\text{-intro} \quad t \text{ de type } A$$

$$\frac{\Gamma \vdash \exists_A E \quad \Gamma, (E\ x) \vdash Q}{\Gamma \vdash Q}\ \exists\text{-élim} \quad x \text{ de type } A, \text{ n'app. pas dans } Q, \Gamma$$

$$\frac{}{\Gamma \vdash P \vee \neg P}\ \text{tiers exclu}$$

On peut également considérer des variantes intuitionnistes (en supprimant le tiers exclu) ou intentionnelles (en supprimant l'extensionnalité).

5.3.5 L'égalité et la conversion

Dans le système présenté ci-dessus, si P et Q sont deux propositions convertibles (par exemple $P = (0 =_\iota 0)$ et $Q = ((\lambda x : \iota\ x\ 0) =_\iota 0)$, et que l'on a une démonstration de P on peut construire une démonstration de Q). On commence par utiliser l'axiome de convertibilité pour obtenir une démonstration de $P \Leftrightarrow Q$, puis on utilise la règle d'élimination de l'équivalence logique ($\Leftrightarrow$-élim) pour obtenir une démonstration de Q. La démonstration de P et celle de Q ne sont pas, au sens strict, identiques puisque celle de Q comporte deux étapes de plus.

Il est raisonnable de vouloir éliminer ces arguments de convertibilité des démonstrations. En effet la réduction étant fortement normalisable et confluente, la convertibilité entre deux termes est décidable. Le fait que deux propositions soient convertibles peut s'établir par un *calcul* et ne demande pas de *raisonnement*. De ce fait on peut préférer un système logique où les démonstrations ne contiennent pas ces arguments calculatoires. Dans un tel système, les démonstrations sont plus succinctes (car elles ne contiennent pas l'information de convertibilité), elles sont un peu plus difficiles à vérifier (puisque pour vérifier une telle démonstration, il faut reconstruire cette information de convertibilité) mais la vérification reste décidable (puisque la convertibilité entre deux termes est décidable).

On obtient un tel système en dérivant non plus des termes du λ-calcul mais des classes d'équivalences de termes modulo la relation de convertibilité. Dans un tel système, les (classes des) propositions P et Q ci-dessus sont identiques et toute démonstration de l'une est une démonstration de l'autre. Les axiomes de conversion sont inutiles et peuvent être supprimés.

5.3.6 Le λ-calcul et le schéma de compréhension

Comme en théorie naïve des ensembles et en théorie des ensembles on peut choisir d'avoir un langage explicite pour les fonctions et les ensembles ou bien d'avoir des axiomes d'existence. On peut donc supprimer le symbole λ du langage ainsi que les axiomes de conversion et se donner les axiomes de compréhension typés

$$\exists_{A_1 \to \ldots \to A_n \to B} f\ \forall_{A_1} x_1\ \ldots\ \forall_{A_n} x_n (f\ x_1\ \ldots\ x_n) =_B t$$

$$\exists_{A_1 \to \ldots A_n \to o} E\ \forall_{A_1} x_1\ \ldots\ \forall_{A_n} x_n\ (E\ x_1\ \ldots\ x_n) \Leftrightarrow P$$

5.3.7 Expression au premier ordre

On peut exprimer la théorie des types comme une théorie du premier ordre multisortée. Les sortes sont tous les types ι, o, $\iota \to \iota$, ... Si t et u sont deux termes de types respectifs $A \to B$ et A, on ne peut pas noter l'application de t à u $(t\ u)$ mais on doit introduire un symbole de fonction $\alpha_{A,B}$ et noter ce terme $\alpha_{A,B}(t,u)$. De même, on doit distinguer les propositions des termes de type o (un terme ne pouvant pas être aussi une proposition dans une théorie du premier ordre). Si t est un terme de type o, on ne peut pas noter la proposition correspondante t mais on doit introduire un symbole de prédicat ε et noter la proposition $\varepsilon(t)$.

Les axiomes ci-dessus se reformulent

$$\exists_{A_1 \to \ldots \to A_n \to B} f\ \forall_{A_1} x_1\ \ldots\ \forall_{A_n} x_n \varepsilon(\alpha(\ldots\alpha(f, x_1), \ldots x_n) =_B t)$$

$$\exists_{A_1 \to \ldots A_n \to o} E\ \forall_{A_1} x_1\ \ldots\ \forall_{A_n} x_n\ \varepsilon(\alpha(\ldots, \alpha(E, x_1), \ldots x_n)) \Leftrightarrow P$$

Remark (Quine) Cette théorie du premier ordre multisortée peut se relativiser en une théorie du premier ordre ordinaire. Sous cette forme la théorie des types apparaît, elle aussi, comme une théorie dans laquelle le schéma de compréhension a été restreint. Il est restreint aux termes et aux propositions qui sont la relativisation d'un terme ou d'une proposition du langage multisorté.

5.3.8 L'élimination des coupures

Si on identifie les propositions équivalentes, la notion de coupure dans la théorie des types est similaire est la notion ordinaire de coupure : une règle d'introduction suivie d'une règle d'élimination. Le théorème d'élimination des coupures est, cela dit, plus difficile, car en éliminant une coupure, on peut on peut substituer une variable de proposition ou de prédicat en introduisant de nouveaux connecteurs et quantificateurs (voir chapitre 8).

Si on garde les axiomes de conversion comme des axiomes, on est amené à définir une notion de coupure associée à ces axiomes.

5.3.9 Un peu de mathématiques

Les connecteurs et les quantificateurs

La théorie des types minimale est la théorie obtenue en restreignant la théorie des types au connecteur $\Rightarrow$ et au quantificateur $\forall$. Dans cette théorie, les autres connecteurs et quantificateurs peuvent être définis.

$$\wedge = \lambda A \lambda B\ \forall C\ ((A \Rightarrow B \Rightarrow C) \Rightarrow C)$$

$$\vee = \lambda A \lambda B\ \forall C\ ((A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C)$$

$$\bot = \forall C\ C$$

$$\neg = \lambda A\ (A \Rightarrow \bot)$$

$$\exists_T = \lambda P\ \forall C : o\ ((\forall x\ (P\ x) \Rightarrow C) \Rightarrow C)$$

Les règles de déduction sur ces connecteurs et quantificateurs peuvent alors se déduire des règles des règles sur l'implication et le quantificateur universel. Par exemple, si π est une démonstration de $A \wedge B$, c'est-à-dire de $\Gamma \vdash \forall C\ ((A \Rightarrow B \Rightarrow C) \Rightarrow C)$ on peut construire une démonstration de $\Gamma \vdash A$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash \forall C\ ((A \Rightarrow B \Rightarrow C) \Rightarrow C)}}{\Gamma \vdash (A \Rightarrow B \Rightarrow A) \Rightarrow A} \qquad \dfrac{\Gamma, A, B \vdash A}{\Gamma \vdash A \Rightarrow B \Rightarrow A}}{\Gamma \vdash A}$$

L'égalité

L'égalité également peut se définir en théorie des types.

$$=_T = \lambda a \ \lambda b \ \lambda p : T \to o \ ((p \ a) \Rightarrow (p \ b))$$

On peut alors démontrer les deux axiomes de l'égalité :

$$\forall x \ (x = x)$$

$$\forall x \ \forall y \ (x = y \Rightarrow (\forall A \ ((A \ x) \Rightarrow (A \ y))))$$

Les entiers

Les entiers de Peano En théorie des types les entiers peuvent être définis comme des objets de base. Une première possibilité consiste à considérer le type ι comme le type des entiers, on se donne des symboles $0 : \iota, S : \iota \to \iota$ et on ajoute les axiomes de Peano aux axiomes ci-dessus.

$$\forall x \ \forall y \ ((S \ x) = (S \ y) \Rightarrow x = y)$$

$$\forall x \ \neg(0 = (S \ x))$$

$$\forall E \ (((E \ 0) \wedge (\forall x \ ((E \ x) \Rightarrow (E \ (S \ x))))) \Rightarrow \forall n \ (E \ n))$$

Une seconde possibilité consiste à poser que les entiers sont de type ι, mais que ce type peut également contenir d'autres objets.

On commence par poser des axiomes exprimant que le type ι est infini, c'est-à-dire qu'il existe une injection non surjective de ι dans ι. On appelle S cette injection, et 0 un élément de ι qui n'est pas dans l'image de S. On se donne donc des symboles $0 : \iota, S : \iota \to \iota$ et les axiomes

$$\forall x \ \forall y \ ((S \ x) = (S \ y) \Rightarrow x = y)$$

$$\forall x \ \neg(0 = (S \ x))$$

puis on définit ensuite l'ensemble des entiers comme le plus petit ensemble qui contient 0 et qui est clos par S, c'est-à-dire comme l'intersection de tous les ensembles qui contiennent 0 et qui sont clos par S. N est alors l'ensemble des objets n de type ι qui appartiennent à tous les ensembles X contenant 0 et clos par S.

$$N = \lambda n \ (\forall X \ ((X \ 0) \wedge \forall x \ ((X \ x) \Rightarrow (X \ (S \ x)))) \Rightarrow (X \ n))$$

On peut alors démontrer le principe de récurrence,

$$\forall E \ (((E \ 0) \wedge (\forall x \ ((E \ x) \Rightarrow (E \ (S \ x))))) \Rightarrow \forall n \ ((N \ n) \Rightarrow (E \ n)))$$

En effet donnons nous un ensemble E qui vérifie $(E \ 0) \wedge \forall x \ ((E \ x) \Rightarrow (E \ (S \ x)))$ et un objet n qui vérifie $(N \ n)$. On a $(N \ n)$ c'est-à-dire

$$\forall X \ (((X \ 0) \wedge \forall y \ ((X \ y) \Rightarrow (X \ (S \ y)))) \Rightarrow (X \ n))$$

on en déduit

$$((E \ 0) \wedge \forall y \ ((E \ y) \Rightarrow (E \ (S \ y)))) \Rightarrow (E \ n)$$

et donc $(E \ n)$.

Les entiers de Cantor Il est également possible de poser un axiome d'infinité des objets de type ι et de construire les entiers comme des cardinaux finis. On commence par définir la relation d'équipotence sur les ensembles d'objets de bases (type $\iota \to o$), on définit ensuite les cardinaux comme les classes d'équivalence (de type $(\iota \to o) \to o$) de cette relation. On définit le cardinal 0 et la fonction successeur, et on prouve les propositions

$$\forall x\ \forall y\ ((S\ x) = (S\ y) \Rightarrow x = y)$$

$$\forall x\ \neg(0 = (S\ x))$$

On définit ensuite l'ensemble des entiers comme le plus petit ensemble contenant 0 et clos par S et on démontre l'axiome de récurrence, comme ci-dessus.

Les entiers de Church Il est enfin possible de définir les entiers comme des itérateurs (entiers de Church), c'est-à-dire comme des objets de type $\iota \to (\iota \to \iota) \to \iota$. Dans ce cas, il est également nécessaire de poser l'existence d'un ensemble infini d'objets de type ι pour démontrer que 0 n'est pas un successeur et que la fonction successeur est injective. Le principe de récurrence, quant à lui, reste un axiome.

De manière générale, il est toujours nécessaire de poser un axiome d'infinité car en l'absence d'un tel axiome il y a des modèles de la théorie des types dans lesquels tout ensemble est fini.

Le calcul et le raisonnement

Si on code les entiers comme des entiers de Church, on peut définir l'addition comme le terme

$$+ = \lambda n : \iota \to (\iota \to \iota) \to \iota\ \lambda p : \iota \to (\iota \to \iota) \to \iota\ \ \lambda x : \iota\ \lambda f : \iota \to \iota\ (n\ (p\ x\ f)\ f)$$

Dans ce cas les termes $2 + 2$ et 4 sont convertibles. De même les propositions $2 + 2 = 4$ et $4 = 4$ sont convertibles. Si on se place dans le système du paragraphe 3.5, toute démonstration de $4 = 4$ est une démonstration de $2 + 2 = 4$. Autrement dit, pour démontrer la proposition $2 + 2 = 4$, il suffit d'utiliser le premier axiome de l'égalité $\forall x\ (x = x)$, et de substituer x par 4.

En revanche, si on définit les entiers axiomatiquement comme des objets de base à l'aide d'un symbole 0 et S et que l'on a des axiomes pour l'addition et la multiplication, pour démontrer $2 + 2 = 4$, il faut utiliser à deux reprises l'axiome $\forall x\ \forall y\ (x + (S\ y)) = (S\ (x + y))$.

Dans le premier des systèmes, la proposition $2 + 2 = 4$ a une démonstration plus succincte que dans le second. Il est donc possible dans ce système de supprimer des démonstration certains arguments calculatoires. Le fait que l'argument qui permet d'établir la proposition $2 + 2 = 4$ puisse être supprimé des démonstrations est bien entendu lié au fait que la fonction $+$ est β-exprimable. De manière générale, plus un système logique est expressif du point de vue calculatoire, plus il est possible de supprimer des arguments calculatoires des démonstrations.

Supprimer les arguments calculatoires des démonstrations les rend plus succinctes ce qui est important par exemple en démonstration automatique. La théorie des types simples a une expressivité calculatoire relativement faible (puisque les polynômes étendus forment une petite partie des fonctions calculables). Cette remarque motive l'étude de systèmes logiques plus expressifs du point de vue calculatoire.

5.4 L'existence de fonctions

5.4.1 L'utilisation de l'axiome du choix

Au premier ordre, pour démontrer une proposition de la forme $\exists x\ P$ la méthode standard consiste à donner un témoin, c'est-à-dire un entier vérifiant la propriété en question puis à utiliser la règle d'introduction du quantificateur existentiel pour conclure. Dans certains cas, il est nécessaire de donner plusieurs témoins sous certaines hypothèses et de recoller les morceaux de démonstration avec le tiers exclu (c'est ce qu'indique le théorème de Herbrand). Dans la théorie des types simples, il n'est pas toujours possible de procéder ainsi, car de nombreuses fonctions dont on veut parler ne peuvent pas être directement exprimée par un λ-terme.

Supposons que nous avons un symbole 0 de type ι et un symbole S de type $\iota \to \iota$, et les axiomes de Peano. Il n'existe pas de λ-terme qui exprime la fonction f telle que

$$(f\ x) = 0 \text{ si } x = 0$$

$$(f\ x) = 1 \text{ sinon}$$

On ne voit pas, de ce fait, comment montrer la proposition

$$\exists f\ \forall x\ (x = 0 \Rightarrow (f\ x) = 0) \wedge (\neg(x = 0) \Rightarrow (f\ x) = 1))$$

En revanche il est simple de montrer (par exemple, par récurrence sur x) que pour tout x, il existe un y tel que $y = 0$ si x est nul et $y = 1$ sinon.

$$\forall x\ \exists y\ (x = 0 \Rightarrow y = 0) \wedge (\neg x = 0 \Rightarrow y = 1))$$

Pour conclure l'existence de la fonction, il faut utiliser une forme de l'axiome du choix

$$\forall P\ (\forall x\ \exists y\ (P\ x\ y)) \Rightarrow \exists f\ \forall x\ (P\ x\ (f\ x))$$

5.4.2 L'existence de l'addition

Ce mécanisme permet de montrer l'existence de n'importe quelle fonction définie par récurrence. A titre d'exemple, montrons l'existence de l'addition, c'est-à-dire qu'il existe une fonction + telle que

$$\forall y\ (+\ 0\ y) = y$$

$$\forall x\ \forall y\ (+\ (S\ x)\ y) = (S\ (+\ x\ y))$$

L'addition comme relation

Commençons par montrer que l'addition peut *être représentée* en théorie des types, c'est-à-dire qu'il existe un prédicat $Plus$ de type $\iota \to \iota \to \iota \to o$ tel que la proposition $(Plus\ a\ b\ c)$ est démontrable si et seulement si $a + b = c$. Cette représentation est l'analogue de la représentation des fonctions partielles semi-calculables dans l'arithmétique de Peano, que l'on utilise dans la démonstration des théorèmes de Church et Gödel. On peut de même montrer que toutes les fonctions partielles semi-calculables sont représentables dans la théorie des types. L'addition est représentée par le prédicat

$$Plus = \lambda x\ \lambda y\ \lambda z\ \forall w\ (\forall n\ (w\ 0\ n\ n) \wedge \forall n\ \forall p\ \forall q\ (w\ n\ p\ q) \Rightarrow (w\ (S\ n)\ p\ (S\ q))) \Rightarrow (w\ x\ y\ z)$$

Intuitivement, les nombres a, b, c sont reliés par le prédicat $Plus$ s'ils sont reliés par tous les prédicats w tels que

$$\forall n\ (w\ 0\ n\ n)$$

$$\forall n\ \forall p\ \forall\ q\ (w\ n\ p\ q) \Rightarrow (w\ (S\ n)\ p\ (S\ q))$$

Il est facile de démontrer les propositions

$$\forall x\ (Plus\ 0\ x\ x)$$

$$\forall x\ \forall y\ \forall z\ (Plus\ x\ y\ z) \Rightarrow (Plus\ (S\ x)\ y\ (S\ z))$$

On démontre ensuite que cette relation (prédicat) est fonctionnelle, c'est-à-dire que pour tout x et y, il existe un z unique tel que $(Plus\ x\ y\ z)$.

$$\forall x\ \forall y\ \exists z\ ((Plus\ x\ y\ z) \wedge \forall k\ ((Plus\ x\ y\ k) \Rightarrow k = z))$$

Cette proposition se démontre par récurrence sur x. On doit donc démontrer les deux propositions

$$\forall y\ \exists z\ ((Plus\ 0\ y\ z) \wedge \forall k\ ((Plus\ 0\ y\ k) \Rightarrow k = z))$$

et

$$(\forall y\ \exists z\ ((Plus\ x\ y\ z) \wedge \forall k\ ((Plus\ x\ y\ k) \Rightarrow k = z))) \Rightarrow$$
$$(\forall y\ \exists z\ ((Plus\ (S\ x)\ y\ z) \wedge \forall k\ ((Plus\ (S\ x)\ y\ k) \Rightarrow k = z)))$$

Pour démontrer la première de ces propositions on démontre

$$((Plus\ 0\ y\ y) \wedge \forall k\ ((Plus\ 0\ y\ k) \Rightarrow k = y))$$

La proposition $(Plus\ 0\ y\ y)$ se démontre en utilisant le théorème $\forall x\ (Plus\ 0\ x\ x)$. Pour démontrer $\forall k\ ((Plus\ 0\ y\ k) \Rightarrow k = y)$ on montre $k = y$ sous l'hypothèse $(Plus\ 0\ y\ k)$ c'est-à-dire

$$\forall w\ (\forall n\ (w\ 0\ n\ n) \wedge \forall n\ \forall p\ \forall q\ (w\ n\ p\ q) \Rightarrow (w\ (S\ n)\ p\ (S\ q))) \Rightarrow (w\ 0\ y\ k)$$

Dans cette hypothèse on instancie la variable w par le prédicat

$$W = \lambda a\ \lambda b\ \lambda c\ (a = 0 \Rightarrow b = c)$$

et on conclut sans peine.

Pour démontrer la seconde de ces propositions on montre

$$(((Plus\ (S\ x)\ y\ (S\ z)) \wedge \forall k\ ((Plus\ (S\ x)\ y\ k) \Rightarrow k = (S\ z))))$$

sous l'hypothèse

$$(((Plus\ x\ y\ z) \wedge \forall k\ ((Plus\ x\ y\ k) \Rightarrow k = z)))$$

La proposition $(Plus\ (S\ x)\ y\ (S\ z))$ se démontre en utilisant le théorème $\forall x\ \forall y\ \forall z\ (Plus\ x\ y\ z) \Rightarrow (Plus\ (S\ x)\ y\ (S\ z))$. Pour démontrer la proposition $\forall k\ ((Plus\ (S\ x)\ y\ k) \Rightarrow k = (S\ z))$, on montre la proposition $k = (S\ z)$ sous l'hypothèse $(Plus\ (S\ x)\ y\ k)$ c'est-à-dire

$$\forall w\ (\forall n\ (w\ 0\ n\ n) \wedge \forall n\ \forall p\ \forall q\ (w\ n\ p\ q) \Rightarrow (w\ (S\ n)\ p\ (S\ q))) \Rightarrow (w\ (S\ x)\ y\ k)$$

Dans cette hypothèse on instancie la variable w par le prédicat

$$W' = \lambda a\ \lambda b\ \lambda c\ (Plus\ a\ b\ c) \wedge ((a = (S\ x) \wedge b = y) \Rightarrow (c = (S\ z)))$$

et on conclut sans peine.

En corollaire on montre facilement

$$\forall y\ \forall z\ ((Plus\ 0\ y\ z) \Rightarrow y = z)$$

$$\forall x\ \forall y\ \forall z\ ((Plus\ (S\ x)\ y\ z) \Rightarrow \exists u\ (z = (S\ u) \wedge (Plus\ x\ y\ u)))$$

L'addition comme fonction

En utilisant l'axiome du choix, on déduit de la proposition $\forall x\ \forall y\ \exists z\ (Plus\ x\ y\ z)$ la proposition

$$\exists p\ \forall x\ \forall y\ (Plus\ x\ y\ (p\ x\ y))$$

En utilisant les deux corollaires ci-dessus, on déduit

$$\exists p\ (\forall y\ y = (p\ 0\ y)) \wedge \forall x\ \forall y\ (p\ (S\ x)\ y) = (S\ (p\ x\ y))$$

5.4.3 Les fonctions dont on peut prouver l'existence

Représentation des fonctions

Définition Une fonction est *représentable* dans une théorie s'il existe un prédicat P tel que $(P\ a\ b)$ soit démontrable si et seulement si $b = (f\ a)$.

Proposition (Représentation des fonctions semi-calculables) Toute fonction f semi-calculable, est représentable en théorie des types intuitionniste (resp. classique) étendue par les axiomes de Peano.

Proposition (Semi-calculabilité des fonctions représentables)

Soit f une fonction partielle représentable en théorie des types intuitionniste (resp. classique) avec les axiomes de Peano. Alors, la fonction f est semi-calculable.

Démonstration Un entier a donné, on énumère tous les couples $< b, p >$ où b est un entier et p une chaîne de caractère (ou un entier codant cette chaîne de caractères) jusqu'à trouver un couple $< b, p >$ tel que p soit une démonstration de $(P\ a\ b)$ (cette recherche mène à des calculs infinis quand il n'existe pas de b tel que $b = (f\ a)$).

La logique intuitionniste

Définition Soit f une fonction semi-calculable et P un prédicat représentant cette fonction. On dit qu'on peut *prouver l'existence de f* en théorie des types intuitionniste si la proposition $\forall x\ \exists y\ (P\ x\ y)$ est démontrable (ou, ce qui revient au même en utilisant l'axiome du choix, si la proposition $\exists f\ \forall x\ (P\ x\ (f\ x))$ est démontrable).

Proposition En théorie des types, on peut prouver l'existence de toutes les fonctions définies par récurrence (comme l'addition).

Idée de la démonstration On procède comme pour l'addition.

Proposition (Kleene) Soit P un prédicat quelconque si la proposition $\forall x\ \exists y\ (P\ x\ y)$ est démontrable en logique intuitionniste alors il existe une fonction calculable f telle que pour tout a et tout b, $b = (f\ a)$ si et seulement si la proposition $(P\ a\ b)$ est démontrable.

Idée de la démonstration Ce théorème s'appuie sur le théorème d'élimination des coupures dans la théorie des types avec les axiomes de Peano (voir chapitre 8). Supposons que nous ayons une démonstration intuitionniste de $\forall x\ \exists y\ (P\ x\ y)$. Soit un entier a, en utilisant la règle $\forall$-élim on construit une démonstration intuitionniste de la proposition $\exists y\ (P\ a\ y)$. On élimine les coupures dans cette démonstration. On obtient une démonstration sans coupures de $\exists y\ (P\ a\ y)$, la dernière règle de cette démonstration ne peut être que $\exists$-intro. Ce qui nous donne un témoin b tel que $(P\ a\ b)$ soit démontrable.

Corollaire Soit f une fonction représentée par un prédicat P en logique intuitionniste. Si la proposition $\forall x\ \exists y\ (P\ x\ y)$ est démontrable en logique intuitionniste alors la fonction f est calculable.

Remarque Si f est une fonction dont on peut prouver l'existence en théorie des types, alors l'élimination des coupures dans la démonstration de $\forall x\ \exists y\ (P\ x\ y)$ spécialisée à a est un algorithme de calcul de $(f\ a)$.

Proposition Il existe une fonction calculable f telle que la proposition $\forall x\ \exists y\ (P\ x\ y)$ ne soit pas démontrable.

Idée de la démonstration Il n'y a pas de langage qui permet d'exprimer exactement les fonctions calculables.

Remarque D'après les propositions ci-dessus, les fonctions dont on peut prouver l'existence en théorie des types intuitionniste avec les axiomes de Peano sont donc un sur-ensemble des fonctions primitives récursives (définissables par récurrence) mais un sous-ensemble strict des fonctions calculables.

La logique classique

Proposition Il existe une fonction semi-calculable mais non calculable, représentée par un prédicat P tel qu'on puisse montrer l'existence de cette fonction en théorie des types classique.
Démonstration Soit *norm* le prédicat représentant la fonction (calculable) qui à a associe 0 si a est le code de Gödel d'un terme normal et 0 sinon. Soit *beta* le prédicat représentant la fonction (calculable) qui a et b associe 0 si a et b sont les codes de Gödel de deux termes tels que le premier se β-réduise en une étape sur le second. Soit $Norm$ le prédicat

$$Norm = \lambda a\ \exists u\ \exists n\ ((u\ 0) = a \wedge (norm\ (u\ n)\ 0) \wedge \forall p\ (p < n) \Rightarrow (beta\ (u\ p)\ (u\ (p+1))\ 0))$$

et

$$P = \lambda a\ \lambda b\ ((Norm\ a) \Rightarrow b = 0) \wedge (\neg(Norm\ a) \Rightarrow b = 1)$$

Soit g la fonction partielle semi-calculable représentée par P. La fonction g est telle que

- $(g\ a) = 0$ si a est le code de Gödel d'un terme t qui est normalisable (dans ce cas la proposition $(Norm\ a)$ peut se démontrer en théorie des types),
- $(g\ a) = 1$ si a est le code de Gödel d'un terme t qui est non normalisable *et tel que la proposition* $\neg(Norm\ a)$ *peut se démontrer en théorie des types*,
- $(g\ a)$ n'est pas défini sinon.

Comme le problème de l'arrêt est indécidable, la fonction g n'est pas calculable, pourtant la proposition

$$\forall x\ \exists y\ (((Norm\ x) \Rightarrow y = 0) \wedge (\neg(Norm\ x) \Rightarrow y = 1))$$

est démontrable en logique classique (en utilisant le tiers exclu $(Norm\ a) \vee \neg(Norm\ a)$).

5.4.4 L'axiome de descriptions

Il peut sembler étonnant d'avoir besoin de l'axiome du choix pour montrer l'existence d'une fonction aussi simple que l'addition. En fait une forme très faible de cet axiome *l'axiome de descriptions* est suffisante :

$$\forall x\ \exists_1 y\ (P\ x\ y) \Rightarrow \exists_1 f\ (P\ x\ (f\ x))$$

où $\exists_1 x\ (P\ x)$ signifie $\exists x\ ((P\ x) \wedge (\forall y\ (P\ y) \Rightarrow y = x))$.

5.4.5 L'opérateur de choix, l'opérateur de description

Pour ces objets construits avec l'axiome du choix ou l'axiome de descriptions, on peut, une fois de plus, ou bien donner un axiome d'existence ou donner une notation explicite et un axiome pour exprimer les propriétés des objets. On peut donc se donner un opérateur de choix C et un axiome

$$\forall E\ (\exists x\ (E\ x)) \Rightarrow (E\ (C\ E))$$

ou un opérateur de description D et un axiome

$$\forall E\ (\exists_1 x\ (E\ x)) \Rightarrow (E\ (D\ E))$$

Ainsi l'addition a une notation qui est

$$p = \lambda x\ \lambda y\ (D\ \lambda z\ (Plus\ x\ y\ z))$$

Bibliographie

[1] P.B. Andrews, An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, *Academic Press*, Orlando (1986).

[2] G. Dowek, Lambda-calculus, combinators and the comprehension scheme, *Typed Lambda Calculi and Applications* (1995) pp. 154-170. *Rapport de Recherche* 2565, INRIA (1995).

[3] J.L. Krivine, Théorie Axiomatique des Ensembles, *Presses Universitaires de France*, Paris (1969).

Troisième partie

Les démonstrations objets de la théorie

Chapitre 6

Les types dépendants - L'isomorphisme de Curry-De Bruijn-Howard

La sémantique de Heyting et Kolmogorov propose d'exprimer les démonstrations des théorèmes comme des objets fonctionnels :

- une démonstration de $A \Rightarrow B$ comme une fonction qui associe, à toute démonstration de A, une démonstration de B,
- une démonstration de $\forall x\ A$ comme une fonction qui associe, à tout objet t, une démonstration de $A[x \leftarrow t]$,
- une démonstration de $A \wedge B$ comme un couple formé d'une démonstration de A et d'une démonstration de B,
- *etc.*

Les démonstrations des axiomes sont des objets donnés *a priori*. Par exemple, quand on pose un axiome $A \Rightarrow B$, on se donne une fonction f qui associe une démonstration de B à toute démonstration de A. La nature des démonstrations des propositions atomiques importe peu, seul importe le fait que si a est une démonstration de A alors $(f\ a)$ est une démonstration de B. De manière plus générale, la nature des démonstrations importe moins que la relation qui existe entre les démonstrations des différentes propositions.

Comme ces démonstrations sont des fonctions, on les exprime par des λ-termes.

6.1 L'isomorphisme de Curry

On se place dans un fragment de la logique du premier ordre : la *logique propositionnelle minimale*. Ce formalisme est obtenu en considérant des propositions formées à partir des propositions atomiques avec l'implication.

Pour démontrer des propositions, on se donne trois règles de déduction naturelle

$$\frac{P \in \Gamma}{\Gamma \vdash P}\text{hypothèse}$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \Rightarrow\text{-intro}$$

$$\frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q} \Rightarrow\text{-élim}$$

On représente les démonstrations par des λ-termes simplement typés, on introduit donc pour chaque proposition atomique un type de base $(\Phi\ A)$: le type de ses éventuelles démonstrations. Une démonstration de $A \Rightarrow B$ est un terme fonctionnel associant une démonstration de B à chaque démonstration de A, son type est $A' \to B'$ où A' est le type des démonstrations de A et B' le type des démonstrations de B. On étend donc la fonction Φ en une fonction associant à toutes les propositions le type de leurs éventuelles démonstrations par

$$(\Phi\ (A \Rightarrow B)) = (\Phi\ A) \to (\Phi\ B)$$

La fonction Φ est une bijection de l'ensemble des propositions vers l'ensemble des types.

Proposition Soient des propositions $A_1, ..., A_n, A$. Si il existe une démonstration en déduction naturelle du jugement $A_1, ..., A_n \vdash A$ alors il existe un λ-terme simplement typé de type $(\Phi\ A)$ dans le contexte $x_1 : (\Phi\ A_1), ..., x_n : (\Phi\ A_n)$.
Démonstration Soit π une démonstration du jugement $A_1, ..., A_n \vdash A$. On construit, par récurrence sur la structure de π, un terme de type $(\Phi\ A)$ dans le contexte $x_1 : (\Phi\ A_1), ..., x_n : (\Phi\ A_n)$.

- Si la dernière règle de la démonstration est la règle *hypothèse*, alors la proposition A est l'une des propositions A_i. Le terme x_i a le type $(\Phi\ A)$ dans le contexte $x_1 : (\Phi\ A_1), ..., x_n : (\Phi\ A_n)$.

- Si la dernière règle de la démonstration est la règle $\Rightarrow$-intro, alors la proposition A a la forme $B \Rightarrow C$ et il existe une démonstration du jugement $A_1, ..., A_n, B \vdash C$. Par hypothèse de récurrence, il existe un t de type $(\Phi\ C)$ dans le contexte $x_1 : (\Phi\ A_1), ..., x_n : (\Phi\ A_n), x_{n+1} : (\Phi\ B)$. Le terme $\lambda x_{n+1} : (\Phi\ B)\ t$ a le type $(\Phi\ A)$ dans le contexte $x_1 : (\Phi\ A_1), ..., x_n : (\Phi\ A_n)$.

- Si la dernière règle de la démonstration est la règle $\Rightarrow$-élim, alors il existe une proposition B et deux démonstrations de $B \Rightarrow A$ et B. Par hypothèse de récurrence, il existe deux termes t et u de types respectifs $(\Phi\ B) \to (\Phi\ A)$ et $(\Phi\ B)$ dans le contexte $x_1 : (\Phi\ A_1), ..., x_n : (\Phi\ A_n)$. Le terme $(t\ u)$ a le type $(\Phi\ A)$ dans le contexte $x_1 : (\Phi\ A_1), ..., x_n : (\Phi\ A_n)$.

On note $(\varphi\ \pi)$ le terme ainsi construit.

Exemple La proposition $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$ est démontrable sans axiomes. Le terme

$$t = \lambda x : (\Phi\ A) \to (\Phi\ B) \to (\Phi\ C)\ \lambda y : (\Phi\ A) \to (\Phi\ B)\ \lambda z : (\Phi\ A)\ ((x\ z)\ (y\ z))$$

est de type $(\Phi\ ((A \to B \to C) \to (A \to B) \to A \to C))$ dans le contexte vide.

Proposition La fonction φ est une bijection de l'ensemble des démonstrations dans l'ensemble des termes.
Démonstration Cette fonction est trivialement injective. Pour montrer qu'elle est surjective on se donne un terme t de type A et on construit par récurrence sur la structure de t une démonstration de $(\Phi^{-1}\ A)$.

Les fonctions Φ et φ définissent donc un isomorphisme (l'isomorphisme de Curry) entre la *logique propositionnelle minimale* et le λ-calcul simplement typé, puisqu'elles sont bijectives et que π est une démonstration de A sous les hypothèses $A_1, ..., A_n$ si et seulement si $(\varphi\ \pi)$ est un terme de type $(\Phi\ A)$ dans le contexte $x_1 : (\Phi\ A_1), ..., x_n : (\Phi\ A_n)$.

Comme toujours on utilise les mêmes notations pour les objets isomorphes. On dira donc de façon équivalente que

$$\lambda x : A \to B \to C\ \lambda y : A \to B\ \lambda z : A\ ((x\ z)\ (y\ z))$$

est un terme de type $(A \to B \to C) \to (A \to B) \to A \to C$ ou que c'est une démonstration de la proposition $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$.

6.2 La logique minimale

Le λ-calcul simplement typé permet donc d'exprimer les démonstrations de la logique propositionnelle minimale, il est malheureusement trop peu expressif pour permettre de représenter toutes les démonstrations de la théorie des types simples, ni même de la logique du premier ordre. Nous allons donc considérer une extensions du λ-calcul simplement typé qui permet de représenter les démonstrations de la logique du premier ordre utilisant le quantificateur universel : le *λ-calcul avec types dépendants* ($\lambda\Pi$-calcul). Le $\lambda\Pi$-calcul peut être considérés indépendamment de l'interprétation fonctionnelle des démonstrations. Nous allons donc dans un premier temps présenter ce calcul, puis dans un second temps, étendre l'isomorphisme de Curry.

6.2.1 Les types dépendants

Un exemple de type dépendant est celui du type des tableaux d'entiers. Dans la plupart des langages de programmation, la taille d'un tableau est une information exprimée dans son type. Ainsi il n'y a pas un type *tab* mais une famille de types (*tab* 0), (*tab* 1), (*tab* 2), ... pour les tableaux de taille 0, 1, 2, ... Considérons la fonction f qui à un entier n associe le tableau de taille n qui ne contient de des 0 :

$$(f\ 0) = [\,]$$

$$(f\ 1) = [0]$$

$$(f\ 2) = [0, 0]$$

$$(f\ 3) = [0, 0, 0]$$

$$etc.$$

L'argument de cette fonction a toujours le type *nat* (type des entiers), en revanche le type du résultat de la fonction n'est pas toujours le même, ce type est $(tab\ n)$ où n est la valeur de l'argument de la fonction. Pour donner un type à cette fonction, on aimerait écrire $nat \to (tab\ n)$ mais cette notation n'exprime pas le fait que le n qui apparaît dans cette expression réfère à l'argument de la fonction. En plus de l'indication que l'argument de la fonction est de type *nat*, il faut indiquer le nom sous lequel cet argument sera désigné dans le reste du type. On note ce type

$$\Pi n : nat\ (tab\ n)$$

La notation $A \to B$ devient un cas particulier de la notation $\Pi x : A\ B$ où le nom x de l'argument de la fonction n'est pas utilisé dans B et, de ce fait, ne demande pas à être indiqué.

6.2.2 Les types en tant que termes

Le langage des types du $\lambda\Pi$-calcul est plus complexe que celui du λ-calcul simplement typé. En effet, les expressions $(tab\ 0)$, $(tab\ 1)$, $\Pi n : nat\ (tab\ n)$ sont des types mais pas les expressions $(tab\ 0\ 0)$ (parce que la famille de types *tab* prend un seul argument) ou $(tab\ true)$ (parce que l'argument du symbole *tab* doit être un entier et non un booléen). De même, le contexte $x : (tab\ n)$ n'est pas bien formé puisque la variable n n'est déclarée nulle part.

On sent qu'il faut des règles de formation des types et des contextes similaires à celle des termes. On considère donc que les types comme des termes particuliers, et on leur donne le type de base : $Type$. On considère des jugements de la forme $\Gamma \vdash t : T$ qui signifient que t a le type T dans le contexte Γ. Un cas particulier de ce jugement est $\Gamma \vdash t : Type$ qui exprime le fait que t est un type. On considère également une autre forme de jugement : Γ *bien formé* qui exprime le fait que Γ est un contexte bien formé.

Avant d'exprimer ainsi le $\lambda\Pi$-calcul, redonnons la définition du λ-calcul simplement typé dans ce langage. Le contexte vide est bien formé :

$$\overline{[\,]\ \text{bien formé}}$$

Déclaration d'une variable de type :

$$\frac{\Gamma \text{ bien formé}}{\Gamma, A : Type \text{ bien formé}}$$

Les variables de type sont des types :

$$\frac{\Gamma \text{ bien formé} \quad A : Type \in \Gamma}{\Gamma \vdash A : Type}$$

Formation des types fonctionnels :

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : Type}{\Gamma \vdash A \to B : Type}$$

Déclaration d'une variable :

$$\frac{\Gamma \vdash A : Type}{\Gamma, x : A \text{ bien formé}}$$

Les variables sont des termes :

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

Application :

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash t' : A}{\Gamma \vdash (t\ t') : B}$$

Abstraction :

$$\frac{\Gamma \vdash A : Type \quad \Gamma, x : A \vdash B : Type \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A\ t : A \to B}$$

6.2.3 Le $\lambda\Pi$-calcul

Pour que les termes $(tab\ 0)$, $(tab\ 1)$, ... soient des termes de type $Type$, il faut que la variable tab ait le type $nat \to Type$. Comme le terme $nat \to Type$ est le type de tab on aimerait qu'il soit lui même de type $Type$. De même comme le terme $Type$ est le type de $(tab\ 0)$ on aimerait qu'il soit lui même de type $Type$. Hélas, si on pose $Type : Type$ il est possible de typer des termes non normalisables (et une théorie fondée sur un tel calcul serait incohérente) comme le montre le paradoxe de Girard. On est donc obligé d'introduire un nouveau type de base $Kind$ pour les types de familles de types, c'est-à-dire les types des termes contenant une occurrence du symbole $Type$. Il y a donc quatre catégories de termes

- $Kind$,
- les genres : $Type$, $nat \to Type$, ... dont le type est $Kind$
- les types (et les familles de types) : nat, $(tab\ 0)$, tab, ... dont le type est un genre,
- les objets : 0, [0], ... dont le type est un type.

Ces termes sont formés ainsi.

- $Kind$ est le seul terme de sa catégorie.
- Les genres sont $Type$ et les genres formés par produit (par exemple : $nat \to Type$, c'est-à-dire $\Pi x : nat\ Type$).
- Les types et les familles de types sont les variables de type et de famille de types (par exemple : nat, tab) et les types et familles de types formés par application (par exemple : $(tab\ 0)$), abstraction (par exemple : $\lambda n : nat\ (tab\ (S\ n))$) et par produit (par exemple : $\Pi n : nat\ (tab\ n)$ et $nat \to nat$, c'est-à-dire $\Pi x : nat\ nat$).

- Les objets sont les variables d'objet (par exemple : 0), les objets formés par application (par exemple : $(S\ 0)$) et par abstraction (par exemple : $\lambda x : nat\ x$).

Définition $\lambda\Pi$-calcul
Le contexte vide est bien formé :

$$\overline{[\,]\ \text{bien formé}}$$

Déclaration d'une variable (de type ou de famille de types) :

$$\frac{\Gamma \vdash T : Kind}{\Gamma, x : T \text{ bien formé}}$$

Déclaration d'une variable (d'objet) :

$$\frac{\Gamma \vdash T : Type}{\Gamma, x : T \text{ bien formé}}$$

Type est un genre :

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash Type : Kind}$$

Les variables sont des termes :

$$\frac{\Gamma \text{ bien formé} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

Produit (pour les genres)

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Kind}{\Gamma \vdash \Pi x : T\ T' : Kind}$$

Produit (pour les types)

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Type}{\Gamma \vdash \Pi x : T\ T' : Type}$$

Application :

$$\frac{\Gamma \vdash t : \Pi x : T\ T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

Abstraction (pour les familles de types) :

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Kind \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T\ t : \Pi x : T\ T'}$$

Abstraction (pour les objets) :

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Type \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T\ t : \Pi x : T\ T'}$$

Conversion :

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : Type \quad \Gamma \vdash T' : Type \quad T \equiv T'}{\Gamma \vdash t : T'}$$

La dernière règle (conversion) est utile par exemple dans le cas suivant. On forme une famille de types $tab' = \lambda n : nat\ (tab\ (S\ n))$ telle que $(tab'\ n)$ soit le type des tableaux de $n+1$ éléments. En appliquant tab' à 0 on obtient $(\lambda n : nat\ (tab\ (S\ n))\ 0)$ qui est convertible avec $(tab\ (S\ 0))$ mais qui *n'est pas* le terme $(tab\ (S\ 0))$. Ainsi la règle de conversion est nécessaire pour que le terme [0] qui a le type $(tab\ (S\ 0))$ ait aussi le type $(tab'\ 0)$.

La règle de formation des genres par produit permet de former le genre $nat \to Type$ mais pas le genre $Type \to Type$. Il est donc possible de considérer des tableaux paramétrés par le nombre de leurs éléments mais non par le types de leurs éléments (tableaux d'entiers, tableaux de booléens, *etc.*). D'autres extensions du λ-calcul simplement typé (types polymorphes, constructeurs de types, *etc.*) permettent de telles constructions.

Définition Un terme t de type T dans un contexte Γ

- est un *objet* si $\Gamma \vdash T : Type$,
- est un *type* si $T = Type$,
- est une *famille de types* si $\Gamma \vdash T : Kind$,
- est un *genre* si $T = Kind$.

Proposition Tout terme bien typé est un objet, une famille de types ou un genre. Les types sont des familles de types.

6.2.4 Les termes purs typables

On cherche à caractériser l'ensemble des termes purs que l'on peut typer dans le $\lambda\Pi$-calcul.

Proposition Soit Γ un contexte et t et T deux termes tels que $\Gamma \vdash t : T$. Si t est un objet (c'est-à-dire si $\Gamma \vdash T : Type$) alors t est une variable, une application $t = (u\ v)$ ou une abstraction $t = \lambda x : T\ u$. Si le terme t est une application, alors les termes u et v sont des objets, si le terme t est une abstraction, alors le terme u est un objet.
Démonstration Par récurrence sur la longueur de la dérivation du jugement $\Gamma \vdash t : T$.

Définition Soit t un terme qui est un objet, le contenu de t est le λ-terme pur $|t|$ défini par récurrence sur la structure de t par :

- $|x| = x$ si x est une variable,
- $|(u\ v)| = (|u|\ |v|)$,
- $|\lambda x : T\ u| = \lambda x\ |u|$.

Définition Un terme pur u est dit typable dans le $\lambda\Pi$-calcul s'il existe un contexte Γ et des termes t et T tels que $\Gamma \vdash t : T$, t soit un objet dans Γ et $u = |t|$.

Exemple Le terme $\lambda x\ x$ est typable car c'est le contenu du terme $\lambda x : \iota\ x$ qui a le type $\iota \to \iota$ dans le contexte $\iota : Type$

Proposition Les termes purs typables dans le $\lambda\Pi$-calcul sont exactement les termes purs typables dans le λ-calcul simplement typé.
Démonstration On définit une fonctions β qui associe à chaque terme du $\lambda\Pi$-calcul un type du λ-calcul simplement typé.

$$(\beta\ Kind) = \iota$$

$$(\beta\ Type) = \iota$$

$$(\beta\ x) = \iota$$

$$(\beta\ (\Pi x : A\ B)) = (\beta\ A) \to (\beta\ B)$$

$$(\beta\ (\lambda x : A\ t)) = (\beta\ t)$$

$$(\beta\ (t\ u)) = (\beta\ t)$$

On définit de même une fonction α qui associe à chaque objet du $\lambda\Pi$-calcul un terme du λ-calcul simplement typé.

$$(\alpha\ x) = x$$

$$(\alpha\ \lambda x : A\ t) = \lambda x : (\beta\ A)\ (\alpha\ t)$$

$$(\alpha\ (u\ v)) = ((\alpha\ u)\ (\alpha\ v))$$

Soit $\Gamma = x_1 : A_1, ..., x_n : A_n$ un contexte, t et T deux termes tels que $\Gamma \vdash t : T$. Soit $t' = (\alpha\ t)$ et $T' = (\beta\ T)$. Soit

$$\Gamma' = \iota : Type, x_1 : (\beta\ A_1), ..., x_n : (\beta\ A_n)$$

On a $\Gamma' \vdash t' : T'$ dans le λ-calcul simplement typé (par récurrence sur la structure de la dérivation de $\Gamma \vdash t : T$). Enfin si t est un objet alors $|t| = |(\alpha\ t)|$, donc si un λ-terme pur u est typable dans le $\lambda\Pi$-calcul, soit Γ, t et T tels que $\Gamma \vdash t : T$ dans le $\lambda\Pi$-calcul et $|t| = u$. Soient Γ', t', T' comme ci-dessus, on a $\Gamma' \vdash t' : T'$ dans le λ-calcul simplement typé et $|t'| = u$, le terme u est donc typable dans le λ-calcul simplement typé.

Exemple Le terme $\lambda n\ \lambda x\ (cons\ n\ 0\ x)$ est typable dans le $\lambda\Pi$-calcul. En effet, soit

$$\Gamma = nat : Type, 0 : nat, S : nat \to nat, tab : nat \to Type, nil : (tab\ 0),$$
$$cons : \Pi n : nat\ (nat \to (tab\ n) \to (tab\ (S\ n)))$$

on a

$$\Gamma \vdash \lambda n : nat\ \lambda x : (tab\ n)\ (cons\ n\ 0\ x) : \Pi n : nat\ ((tab\ n) \to (tab\ (S\ n)))$$

En traduisant ce jugement dans le λ-calcul simplement typé, on obtient

$$\Gamma' = \iota : Type, nat : \iota, 0 : \iota, S : \iota \to \iota, tab : \iota \to \iota, nil : \iota, cons : \iota \to \iota \to \iota \to \iota$$

et

$$\Gamma' \vdash \lambda n : \iota\ \lambda x : \iota\ (cons\ n\ 0\ x) : \iota \to \iota \to \iota$$

Le terme $\lambda n\ \lambda x\ (cons\ n\ 0\ x)$ est donc également typable dans le λ-calcul simplement typé.

6.2.5 La normalisation

Comme les termes purs typables dans le $\lambda\Pi$-calcul sont également typables dans le λ-calcul simplement typé, ils sont fortement normalisables. Mais la normalisation des termes purs, contenus termes typés, ne suffit pas pour établir la normalisation des termes typés eux-mêmes. Par exemple, le terme $\lambda x : (tab\ (\lambda y : nat\ y\ 0))\ x$ n'est pas normal puisqu'il se réduit en $\lambda x : (tab\ 0)\ x$, bien que son contenu $\lambda x\ x$ soit normal. On modifie donc la fonction α de façon à prendre également en compte le type de la variable liée. De plus on veut montrer que si $\Gamma \vdash t : T$ alors t est fortement normalisable que t soit un objet, une famille de types ou un genre. On étend donc la fonction α à tous les termes du calcul.

Définition On définit la fonctions α' qui associe à chaque terme du $\lambda\Pi$-calcul un terme du λ-calcul simplement typé.

$$(\alpha'\ x) = x$$

$$(\alpha'\ \lambda x : A\ t) = (\lambda y : \iota\ \lambda x : (\beta\ A)\ (\alpha'\ t)\ (\alpha'\ A))$$

$$(\alpha'\ (u\ v)) = ((\alpha'\ u)\ (\alpha'\ v))$$

$$(\alpha'\ Kind) = a$$

$$(\alpha'\ Type) = a$$

$$(\alpha'\ (\Pi x : A\ B)) = (\pi_{(\beta\ A)}\ (\alpha'\ A)\ \lambda x : (\beta\ A)\ (\alpha'\ B))$$

Proposition Soit $\Gamma = x_1 : A_1, ..., x_n : A_n$ un contexte, t et T deux termes tels que $\Gamma \vdash t : T$. Soit $t' = (\alpha'\ t)$ et $T' = (\beta\ T)$. Soit

$$\Gamma' = \iota : Type, a : \iota, \pi_{T_1} : \iota \to (T_1 \to \iota) \to \iota, ..., \pi_{T_k} : \iota \to (T_k \to \iota) \to \iota, x_1 : (\beta\ A_1), ..., x_n : (\beta\ A_n)$$

où π_{T_1}, ..., π_{T_k} sont les symboles de la forme π_T ayant une occurrence dans t'. On a $\Gamma' \vdash t' : T'$ dans le λ-calcul simplement typé.

Proposition Soit t et t' deux termes du $\lambda\Pi$-calcul si t se β-réduit en une étape en t' alors $(\alpha'\ t)$ se β-réduit en au moins une étape en $(\alpha'\ t')$.

Proposition Tout terme typé dans le $\lambda\Pi$-calcul est fortement normalisable.
Démonstration Soit $t_1, t_2, ...$ une suite de réductions issue de t. La suite $(\alpha'\ t_1), ..., (\alpha'\ t_2), ...$ est une suite de réductions issue de $(\alpha'\ t)$. D'après le théorème de normalisation forte dans le λ-calcul simplement typé, cette suite est finie. Il en est de même de la suite $t_1, t_2, ...$

6.2.6 La confluence

La confluence de la β-réduction dans le $\lambda\Pi$-calcul se montre par les méthodes habituelles. En revanche la confluence de la $\beta\eta$-réduction demande une démonstration relativement technique que nous ne détaillerons pas ici.

6.2.7 La décidabilité de la typabilité

Selon Church

Dans le λ-calcul simplement typé, il existe un algorithme simple qui prend en argument un contexte Γ et un terme t et ou bien retourne le type T de t dans Γ, ou bien répond que ce terme n'a pas de type dans Γ. Cet algorithme utilise le fait que, pour chaque terme, une seule règle de typage peut s'appliquer et que chaque règle ne demande de calculer que le type de sous-termes de t. Cet algorithme s'exprime donc ainsi.

- Si t est une variable, t peut être typé uniquement par la règle de typage des variables, on recherche donc cette variable dans Γ. Si on la trouve on retourne le type associé à cette variable. Sinon le terme n'est pas typable.

- Si $t = (u\ v)$, alors t peut être typé uniquement par la règle de typage des applications, on type donc récursivement les sous-termes u et v, si le type de u est une flèche $A \to B$ et que le type de v est A alors on retourne le type B. Sinon le terme n'est pas typable.

- Si $t = \lambda x : A\ u$, alors t peut être typé uniquement par la règle de typage des abstraction, on type donc récursivement le sous-terme u dans le contexte $\Gamma, x : A$, soit B le type de ce terme. On retourne le type $A \to B$.

Dans le $\lambda\Pi$-calcul, à cause de la règle de conversion, il n'y a plus unicité du type. En effet si t a le type T alors t a aussi pour type tous les équivalents bien typés de T. Pour que l'application $(u\ v)$ soit bien typée, il n'est plus nécessaire que le type calculé de u soit un produit $\Pi x : A\ B$ et que le type calculé de v soit A, mais il est uniquement nécessaire que le type de u soit *équivalent* à un produit $\Pi x : A\ B$ et que le type de v soit *équivalent* à A.

La décidabilité du typage repose donc sur la décidabilité de l'équivalence de deux termes, donc sur la normalisation et la confluence de la réduction. On commence par montrer les lemmes suivants.

Proposition Si $\Gamma \vdash t : T$ et $\Gamma \vdash t : T'$ alors $T \equiv T'$.
Démonstration Par récurrence sur la somme des longueurs des dérivations de $\Gamma \vdash t : T$ et $\Gamma \vdash t : T'$.

Proposition Si $\Gamma \vdash t : T$ alors ou bien $T = Kind$ ou bien $\Gamma \vdash T : Type$ ou bien $\Gamma \vdash T : Kind$.

Démonstration Par récurrence sur la structure de la dérivation de $\Gamma \vdash t : T$.

Proposition Si t et u sont deux termes bien typés et $t \equiv u$ alors t et u ont même forme normale. L'équivalence entre deux termes bien typés est donc décidable.
Démonstration D'après la normalisation forte et la confluence.

Proposition Si t est équivalent à un produit alors t se réduit sur un produit.
Démonstration D'après la normalisation forte et la confluence.

Définition On obtient donc l'algorithme suivant

- Si t est une variable, on recherche cette variable dans Γ. Si on la trouve, on retourne le type associé à cette variable. Sinon le terme n'est pas typable.
- Si $t = (u\ v)$, on type récursivement les sous-termes u et v, on normalise ces deux types, si la forme normale du type de u est un produit $\Pi x : A\ B$ et que la forme normale du type de v est A alors on retourne le type $B[x \leftarrow u]$. Sinon le terme n'est pas typable.
- Si $t = \lambda x : A\ u$, on type récursivement le sous-terme A dans le contexte Γ. Si la forme normale de ce type est distincte de $Type$ le terme n'est pas typable. Sinon on type le sous-terme u dans le contexte $\Gamma, x : A$, soit B le type de ce terme. On retourne le type $\Pi x : A\ B$.
- Si $t = \Pi x : A\ u$, on type récursivement le sous-terme A dans le contexte Γ. Si la forme normale de ce type est distincte de $Type$ le terme n'est pas typable. Sinon on type le sous-terme B dans le contexte $\Gamma, x : A$. Si la forme normale de ce type est distincte de $Type$ et $Kind$ alors le terme n'est pas typable. Sinon on retourne ce terme.
- Si $t = Type$, alors on retourne $Kind$.
- Si $t = Kind$, alors il n'est pas typable.

Selon Curry

Les termes purs typables dans le $\lambda\Pi$-calcul et ceux typables dans le λ-calcul simplement typé étant les même, et l'ensemble des termes purs typables dans le λ-calcul simplement typé étant décidable, l'ensemble des termes purs typables dans le $\lambda\Pi$-calcul est décidable. Il existe donc un algorithme qui prend en argument un terme pur u et ou bien retourne un contexte Γ et deux termes t et T tels que $\Gamma \vdash t : T$ et $|t| = u$ ou répond que le terme n'est pas typable dans le $\lambda\Pi$-calcul.

En revanche on peut montrer que si on impose le type des variables libre de u, le problème devient indécidable, c'est-à-dire qu'il n'existe pas d'algorithme qui prend en argument un contexte Γ et un terme pur u et qui retourne deux termes t et T tels que $\Gamma \vdash t : T$ et $|t| = u$ ou répond que le terme n'est pas typable dans Γ dans le $\lambda\Pi$-calcul.

6.2.8 L'isomorphisme de Curry-De Bruijn-Howard pour la logique minimale

La *logique minimale* est le formalisme obtenu en considérant des propositions formées à partir des propositions atomiques, l'absurde ($\bot$), l'implication, et le quantificateur universel.

Pour démontrer des propositions on se donne les règles de déduction naturelle

$$\frac{P \in \Gamma}{\Gamma \vdash P} \text{hypothèse}$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \Rightarrow\text{-intro}$$

$$\frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q} \Rightarrow\text{-élim}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x\ P} \forall\text{-intro} \qquad x \text{ non libre dans une proposition de } \Gamma$$

$$\frac{\Gamma \vdash \forall x\ P}{\Gamma \vdash P[x \leftarrow t]} \forall\text{-élim}$$

On peut également inclure la négation, en considérant $\neg A$ comme une abréviation pour $A \Rightarrow \perp$. Car d'après les règles de déduction naturelle sur l'implication et la négation $\Gamma \vdash \neg A$ si et seulement si $\Gamma \vdash A \Rightarrow \perp$.

En revanche on ne considère pas la règle d'élimination de $\perp$ qui permet de déduire n'importe quelle proposition de $\perp$.

On représente les démonstrations par des termes du $\lambda\Pi$-calcul. En effet selon la sémantique de Heyting et Kolmogorov une démonstration de $\forall x\ A$ est une fonction qui associe à tout objet t, une démonstration de $A[x \leftarrow t]$. Le type de cette fonction est dépendant au sens où le type du résultat de la fonction dépend de la valeur de l'argument.

Soit un langage de premier ordre (par exemple l'arithmétique), composé de symboles de fonctions et de prédicats. Soit un jugement $A_1 ... A_n \vdash A$ dans ce langage.

On se place donc dans un contexte Γ qui contient :

- une variable ι de type $Type$,
- pour chaque variable x apparaissant dans les propositions $A_1, ..., A_n, A$, une variable x' de type ι,
- pour chaque symbole de fonction f d'arité n une variable f' de type $\iota \to ... \to \iota \to \iota$,
- pour chaque symbole de prédicat P d'arité n une variable P' de type $\iota \to ... \to \iota \to Type$,
- pour chaque proposition A_i une variable de type $(\Phi\ A_i)$, où la fonction Φ est définie par
  - $(\Phi\ x) = x'$,
  - $(\Phi\ (f\ u_1\ ...\ u_n)) = (f'\ (\Phi\ u_1)\ ...\ (\Phi\ u_n))$,
  - $(\Phi\ (P\ u_1\ ...\ u_n)) = (P'\ (\Phi\ u_1)\ ...\ (\Phi\ u_n))$,
  - $(\Phi\ (A \Rightarrow B)) = (\Phi\ A) \to (\Phi\ B)$,
  - $(\Phi\ (\forall x\ A)) = \Pi x : \iota\ (\Phi\ A)$.

La fonction Φ est une bijection de l'ensemble des termes de la théorie de premier ordre vers l'ensemble des termes normaux de type ι, et une injection de l'ensemble des propositions de la théorie de premier ordre vers l'ensemble des termes normaux de type $Type$.

Proposition Soient des propositions $A_1, ..., A_n, A$. Soit Γ le contexte associé aux propositions $A_1, ..., A_n$.

Si il existe une démonstration en déduction naturelle du jugement $A_1, ..., A_n \vdash A$ alors il existe un terme du $\lambda\Pi$-calcul de type $(\Phi\ A)$ dans ce contexte.

Démonstration Par récurrence sur la structure de la démonstration.

Comme toujours on utilise les mêmes notations pour les objets isomorphes. On dira donc de façon équivalente que

$$\lambda a : \Pi x : \iota\ ((x + 0) = x)\ (a\ 0)$$

est un terme de type $\Pi x : \iota\ ((x + 0) = x) \to (0 + 0 = 0)$ ou que c'est une démonstration de la proposition $\forall x : \iota\ ((x + 0) = x) \Rightarrow (0 + 0 = 0)$.

6.2.9 L'élimination des coupures en logique minimale

Définition Une *coupure* dans une démonstration, est une sous-démonstration qui consiste

- ou bien à démontrer une proposition $(P\ x)$ dans le cas général pour en déduire (avec la règle $\forall$-intro) $\forall x\ (P\ x)$ puis (avec la règle $\forall$-élim) la proposition $(P\ t)$,
- ou bien à démontrer une proposition B sous une hypothèse A pour en déduire (avec la règle $\Rightarrow$-intro) la proposition $A \Rightarrow B$, puis à démontrer la proposition A et à en déduire (avec la règle $\Rightarrow$-élim) la proposition B.

Remarque Habituellement, les démonstrations des mathématiques informelles comportent de nombreuses coupures. Par exemple, si on veut démontrer

$$\forall x\ (x^2 - 1) = (x + 1)(x - 1)$$

il vaut mieux utiliser le résultat bien connu

$$\forall a\ \forall b\ (a^2 - b^2) = (a + b)(a - b)$$

que redémontrer ce résultat dans le cas particulier $a = x$, $b = 1$. Néanmoins, il est important de montrer que l'on peut en théorie éliminer les coupures et démontrer tous les théorèmes directement. En particulier comme il est facile de montrer qu'il n'y a pas de démonstration directe de $\perp$, on pourra déduire du théorème d'élimination des coupures qu'il n'y a pas de démonstration (directe ou indirecte) de $\perp$, c'est-à-dire que la logique minimale du premier ordre est cohérente. Par ailleurs, quand nous aurons introduit le quantificateur existentiel, nous verrons qu'une démonstration intuitionniste avec coupure d'une proposition de la forme $\exists x\ (P\ x)$ contient un programme dont la sortie est un objet a qui vérifie la propriété P, alors qu'une démonstration sans coupure, contient directement cet objet a. L'élimination des coupures est donc une manière de calculer la sortie d'un programme, c'est-à-dire de l'exécuter.

Remarque Une coupure est un détour dans une démonstration en effet, dans le premier cas, on voit que la proposition $(P\ t)$ a une démonstration plus directe obtenue en spécialisant celle de $(P\ x)$ en substituant partout la variable x par le terme t. Dans le second cas la proposition B a une démonstration plus directe consistant à montrer B sans hypothèse comme dans la première démonstration en redémontrant la proposition A à chaque fois que la démonstration initiale faisait appel à l'hypothèse A.

Néanmoins cette remarque ne suffit pas pour montrer qu'on peut toujours éliminer les coupures dans les démonstrations. En effet, dans le second cas, si la proposition A la forme $P \Rightarrow Q$, et que sa démonstration se termine par l'introduction de cette implication, alors remplacer l'invocation à l'hypothèse A par la démonstration de cette hypothèse peut créer une nouvelle coupure qu'il faut alors éliminer. Pour montrer que toute proposition démontrable a une démonstration sans coupure, il faut donc montrer que ce processus d'élimination des coupures termine.

Proposition Si une démonstration comporte une coupure, alors le λ-terme correspondant contient un radical.
Démonstration Soit une démonstration qui comporte une coupure, montrons que le λ-terme associé comporte un radical.

- Premier cas : La démonstration comporte une sous-démonstration de la proposition $(P\ x)$ dont on déduit (avec la règle $\forall$-intro) $\forall x\ (P\ x)$ puis (avec la règle $\forall$-élim) la proposition $(P\ t)$. Appelons u le terme associé à la démonstration de $(P\ x)$, la démonstration de $\forall x\ (P\ x)$ a la forme $\lambda x : \iota\ u$ et celle de $(P\ t)$ la forme $((\lambda x : \iota\ u)\ t)$.
- Second cas : La démonstration comporte une sous-démonstration de la proposition B sous une hypothèse A, dont on déduit (avec la règle $\Rightarrow$-intro) la proposition $A \Rightarrow B$ puis la proposition B (avec la

règle $\Rightarrow$-élim) en utilisant une démonstration de A. Appelons u la démonstration de B sous l'hypothèse $x : A$ et t celle de A. La démonstration de $A \Rightarrow B$ a la forme $\lambda x : A\ u$ et celle de B la forme $((\lambda x : A\ u)\ t)$.

Dans les deux cas le sous-terme associé à la sous-démonstration qui est une coupure est un radical.

Remarque Les démonstrations plus directes obtenue en spécialisant dans le premier cas la démonstration de $(P\ x)$ en substituant partout la variable x par le terme t et dans le second en remplaçant les invocations à l'hypothèse A par la démonstration de la proposition A ont la forme $u[x \leftarrow t]$. Ces démonstrations sont obtenues en réduisant les radicaux associés aux coupures. La remarque que l'élimination d'une coupure peut créer de nouvelles coupures est simplement celle que la réduction d'un radical peut créer de nouveaux radicaux.

Proposition (Élimination des coupures) Soit P une proposition et π une démonstration de P, alors le processus d'élimination des coupures dans π termine.
Démonstration S'il ne terminait pas on pourrait construire une suite infinie de démonstrations telle que chacune s'obtient en éliminant une coupure dans la précédente. Par l'isomorphisme de Curry-De Bruijn-Howard on obtiendrait une suite de réductions infinie dans le $\lambda\Pi$-calcul en contradiction avec le théorème de normalisation forte.

Corollaire Toute proposition P démontrable en logique du premier ordre admet une démonstration sans coupure.

Proposition Soit Γ un contexte comprenant

- un symbole ι de type $Type$,
- des symboles de type $\iota \rightarrow ... \rightarrow \iota \rightarrow \iota$,
- des symboles de type $\iota \rightarrow ... \rightarrow \iota \rightarrow Type$.

Alors il n'y a pas de terme normal de type $\perp$ dans Γ.
Démonstration Considérons un terme normal bien typé dans Γ. Ecrivons $t = (u\ a_1\ ...\ a_n)$ où u n'est pas une application.

- Si u est une abstraction alors comme le terme est normal, $n = 0$, donc t est une abstraction le type de t a la forme $\Pi x : A\ B$, ce n'est donc pas $\perp$.
- Si u est une variable, alors cette variable a le type $Type$, $\iota \rightarrow ... \rightarrow \iota \rightarrow \iota$ ou $\iota \rightarrow ... \rightarrow \iota \rightarrow Type$ et donc le type de t n'est pas $\perp$.

Corollaire Dans le contexte Γ il n'y a pas de terme (normal ou non) de type $\perp$.

Corollaire (Cohérence) En logique minimale, il n'y a pas de démonstration de $\vdash \perp$.

Remarque Soit L un système logique (la logique du premier ordre sans axiomes, l'arithmétique du premier ordre, la théorie des types, *etc.*) Si un λ-calcul C permet de représenter les démonstrations de L alors la normalisation de C implique trivialement la cohérence de L. D'après le second théorème d'incomplétude de Gödel, si L est cohérent alors il est impossible de démontrer dans L la cohérence de L, donc il est impossible de démontrer dans L la normalisation de C. Cette remarque explique la relative difficulté des démonstrations des théorèmes de normalisation, puisque ces démonstrations doivent toujours s'exprimer dans un système logique plus fort que celui dont elles permettent de montrer la cohérence.

6.3 La logique intuitionniste

6.3.1 La sémantique de Heyting et Kolmogorov

Nous avons exprimé ainsi des démonstrations des propositions formées avec l'implication et le quantificateur universel.

- Une démonstration de $A \Rightarrow B$ s'exprime comme une fonction qui associe à toute démonstration de A, une démonstration de B.
- Une démonstration de $\forall x\ A$ s'exprime comme une fonction qui associe à tout objet t, une démonstration de $A[x \leftarrow t]$.

La sémantique de Heyting et Kolmogorov propose d'exprimer ainsi les démonstrations des propositions formées avec la conjonction, la disjonction, l'absurde, et le quantificateur existentiel.

- Une démonstration de $A \wedge B$ s'exprime comme un couple formé d'une démonstration de A et d'une démonstration de B.
- Une démonstration de $\exists x A$ s'exprime comme un couple formé d'un terme t et d'une démonstration de $A[x \leftarrow t]$.
- Une démonstration de $A \vee B$ s'exprime ou bien comme une démonstration de A ou bien comme une démonstration de B.
- Une démonstration de $\perp$ s'exprime comme un élément de l'ensemble vide.

Pour représenter ces démonstrations en λ-calcul il faut étendre le $\lambda\Pi$-calcul en ajoutant un type $A \times B$ pour les couples composés d'un élément de type A et d'un élément de type B, un type $A + B$ union disjointe des types A et B et un type vide $\emptyset$.

De même que le types des fonctions $A \rightarrow B$ devait se généraliser en $\Pi x : A\ B$ quand le type de la valeur retournée par la fonction dépendait de la valeur de l'argument de la fonction, le type des couples $A \times B$ doit se généraliser en $\Sigma x : A\ B$ quand le type du second élément du couple dépend de la valeur du premier (pour exprimer les démonstrations des propositions formées avec le quantificateur existentiel). Cela mène au $\lambda 1$-calcul.

6.3.2 Le $\lambda 1$-calcul

- On note $(p^{\Sigma x:A\ B}\ a\ b)$ le couple de type $\Sigma x : A\ B$ dont la première composante est a et la seconde b. Si c est un terme de type $\Sigma x : A\ B$, on note $(\pi_1^{\Sigma x:A\ B}\ c)$ la première composante de c et $(\pi_2^{\Sigma x:A\ B}\ c)$ la seconde.
- On note $(i^{A,B}\ a)$ l'élément de l'union disjointe de $A + B$ correspondant à l'élément a de A et $(j^{A,B}\ b)$ l'élément de l'union disjointe de $A + B$ correspondant l'élément b de B. Si c est un élément de $A + B$, f une fonction de A dans C et g une fonction de B dans C. On peut définir l'objet d qui est $(f\ a)$ si c correspond à l'objet a de A et $(g\ b)$ si c correspond à l'objet b de B. On note cet objet $(\delta^{A,B,C}\ f\ g\ c)$.
- Si a est un terme de type $\emptyset$, on note $(R^A\ a)$ un objet de type A.

On ajoute au $\lambda\Pi$-calcul les règles de typage suivantes.

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Type}{\Gamma \vdash \Sigma x : T\ T' : Type}$$

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Type \quad \Gamma \vdash a : T \quad \Gamma \vdash b : T'[x \leftarrow a]}{\Gamma \vdash (p^{\Sigma x:T\ T'}\ a\ b) : \Sigma x : T\ T' : Type}$$

$$\frac{\Gamma \vdash a : \Sigma x : T\ T' : Type}{\Gamma \vdash (\pi_1^{\Sigma x:T\ T'}\ a) : T}$$

$$\frac{\Gamma \vdash a : \Sigma x : T\ T' : Type}{\Gamma \vdash (\pi_2^{\Sigma x:T\ T'}\ a) : T'[x \leftarrow (\pi_1^{\Sigma x:T\ T'}\ a)]}$$

$$\frac{\Gamma \vdash T : Type \quad \Gamma \vdash T' : Type}{\Gamma \vdash T + T' : Type}$$

$$\frac{\Gamma \vdash T : Type \quad \Gamma \vdash T' : Type \quad \Gamma \vdash a : T}{\Gamma \vdash (i^{T,T'}\ a) : T + T'}$$

$$\frac{\Gamma \vdash T : Type \quad \Gamma \vdash T' : Type \quad \Gamma \vdash b : T'}{\Gamma \vdash (j^{T,T'}\ b) : T + T'}$$

$$\frac{\Gamma \vdash a : T + T' \quad \Gamma \vdash U : Type \quad \Gamma \vdash f : T \to U \quad \Gamma \vdash g : T' \to U}{\Gamma \vdash (\delta^{T,T',U}\ f\ g\ a) : U}$$

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash \emptyset : Type}$$

$$\frac{\Gamma \vdash t : \emptyset}{\Gamma \vdash (R^T\ t) : T}$$

6.3.3 L'isomorphisme de Curry-De Bruijn-Howard

Définition

- $(\Phi\ (A \wedge B)) = \Sigma x : (\Phi\ A)\ (\Phi\ B)$
- $(\Phi\ (\exists x\ A)) = \Sigma x : \iota\ (\Phi\ A)$
- $(\Phi\ (A \vee B)) = (\Phi\ A) + (\Phi\ B)$
- $(\Phi\ \bot) = \emptyset$

Proposition Si la proposition A est démontrable en logique intuitionniste, alors le type $(\Phi\ A)$ est habité.

type	cas non dép.	interpr. logique	interpr. log. non dép.	nom mathématique	nom dans les langages de programmation
$\emptyset$		$\bot$		ensemble vide	
$+$		$\vee$		union disjointe	type concret (ML), type union (C)
Σ	$\times$	$\exists$	$\wedge$	produit cartésien	type record
Π	$\to$	$\forall$	$\Rightarrow$	ens. de fonctions	type fonctionnel

6.3.4 La normalisation et la confluence

Définition Règles de réduction

- $((\lambda x : A\ a)\ b) \rhd a[x \leftarrow b]$
- $(\pi_1\ (p\ a\ b)) \rhd a$

- $(\pi_2\ (p\ a\ b))\ \triangleright b$
- $(\delta\ f\ g\ (i\ a))\ \triangleright (f\ a)$
- $(\delta\ f\ g\ (j\ b))\ \triangleright (g\ b)$

Ces règles peuvent être complétées par des règles homologues à la η-réduction pour ces symboles.

- $\lambda x : A\ (a\ x)\ \triangleright a$ si x n'apparaît pas dans a
- $(p\ (\pi_1\ a)\ (\pi_2\ a))\ \triangleright a$
- $(\delta\ \lambda x : A\ (i\ x)\ \lambda y : B\ (j\ y)\ a)\ \triangleright a$
- $(R^{\emptyset}\ t)\ \triangleright t$

Proposition Tout terme est fortement normalisable.
Démonstration On adapte la méthode de Tait.

Proposition Soit Γ un contexte comprenant

- un symbole ι de type $Type$,
- des symboles de type $\iota \to ... \to \iota \to \iota$,
- des symboles de type $\iota \to ... \to \iota \to Type$.

Soit t un terme normal bien typé dans Γ,

- le terme t n'a pas le type $\perp$,
- si le terme t a le type $\Sigma y : U\ V$ alors t a la forme $(p\ v\ w)$,
- si le terme t a le type $U + V$ alors t a la forme $(i\ v)$ ou $(j\ w)$.

Démonstration Considérons un terme normal bien typé dans Γ. Ecrivons $t = (u\ a_1\ ...\ a_n)$ où u n'est pas une application. On considère successivement les cas suivants :

- u est une variable de Γ,
- u est de la forme $\lambda x : T\ v$,
- u est de la forme $(p\ v\ w)$,
- u est de la forme $(\pi_1\ v)$ ou $(\pi_2\ v)$,
- u est de la forme $(i\ v)$ ou $(j\ w)$,
- u est de la forme $(\delta\ v\ f\ g)$,
- u est de la forme $(R\ v)$.

Corollaire 1 (Cohérence) En logique intuitionniste, il n'y a pas de démonstration de $\vdash \perp$.

Corollaire 2 En logique intuitionniste si la proposition $A \vee B$ est démontrable alors A est démontrable ou B est démontrable.

Corollaire 3 En logique intuitionniste si la proposition $\exists x P$ est démontrable alors il existe un terme t tel que $P[x \leftarrow t]$ soit démontrable. Ce terme est la forme normale de $(\pi_1\ u)$ où u est la démonstration de $\exists x P$. Une démonstration de $P[x \leftarrow t]$ est la forme normale de $(\pi_2\ u)$.

Remarque Ces règles de réduction ne permettent pas d'identifier certaines démonstrations qui expriment pourtant le même argument. On ajoute parfois d'autres règles appelées *règles de commutation*. Voir le chapitre 10 de [5].

6.3.5 L'expression des démonstrations de la théorie prédicative des types

Le $\lambda\Pi$-calcul permet donc d'exprimer les démonstration de la logique minimale du premier ordre et son extension le $\lambda 1$-calcul les démonstrations de la logique intuitionniste du premier ordre. Il n'est pas possible d'exprimer dans ce calcul les démonstrations de la théorie des types. En effet, le type des preuves de la proposition $\forall P : o\ (P \Rightarrow P)$ devrait être $\Pi P : Type\ (P \to P)$ et ce terme n'est pas bien typé en $\lambda 1$-calcul, car un produit $\Pi x : A\ B$ peut être formé uniquement quand A est un type (ι, $\iota \to \iota$, *etc.*) mais pas quand c'est un genre ($Type$, $\iota \to Type$, *etc.*).

Cette remarque est à rapprocher de celle qu'en $\lambda 1$-calcul on peut définir une famille de types paramétrée par un objet (par exemple par un entier), mais pas une famille de types paramétrée par un type (on peut définir le type des tableaux paramétrés par leur taille, mais pas le type des tableaux paramétrés par le type de leurs éléments). Des extensions du $\lambda 1$-calcul permettant la formation de tels types *polymorphes* et permettant de ce fait la représentation de toutes les démonstrations de la théorie des types seront étudiées dans la suite du cours.

Si la quantification imprédicative (c'est-à-dire la quantification sur les variables de prédicat, ou encore sur une variable dont le type contient une occurrence du symbole o) demande le polymorphisme, la quantification sur les variables de fonction ne pose aucun problème. On appelle *théorie prédicative des types* le fragment de la théorie des types qui permet la quantification sur une variable de type T uniquement quand le symbole o n'a pas d'occurrence dans le type T. Les démonstrations de la théorie prédicative des types peuvent s'exprimer dans le $\lambda 1$-calcul.

Remarque Si toutes les démonstrations exprimées en déduction naturelle, peuvent être traduites comme des termes, il n'est pas vrai que tous les termes dont le type exprime une proposition de la théorie prédicative des types expriment une démonstration en déduction naturelle. Par exemple, l'axiome du choix

$$\forall x\ \exists y\ (P\ x\ y) \Rightarrow \exists f\ \forall x\ (P\ x\ (f\ x))$$

n'a pas de démonstration en théorie des types simples En revanche, il en a une dans le $\lambda 1$-calcul

$$t = \lambda u\ (p\ \lambda x\ (\pi_1\ (u\ x))\ \lambda x\ (\pi_2\ (u\ x)))$$

Le $\lambda 1$-calcul est donc un système logique plus fort que la théorie prédicative des types (il permet de démontrer davantage de théorèmes), cela est du au fait que la seconde projection est un outil plus puissant que la règle d'élimination du quantificateur existentiel. Néanmoins, il est cohérent, comme le montre le théorème de normalisation.

6.3.6 Les démonstrations objets de la théorie

Dans la conception d'un système logique, on peut distinguer deux façons d'utiliser l'interprétation fonctionnelle des démonstrations. Selon une première conception, le système logique (logique du premier ordre, théorie des types, *etc.*) forme un langage, et le λ-calcul permettant d'exprimer les démonstrations forme un autre langage. Dans ce cas, en tant que terme $\lambda x : \iota\ x$ désigne la fonction identité de type $\iota \to \iota$, en tant que démonstration $\lambda x : A\ x$ est une démonstration de la proposition $A \Rightarrow A$, mais ces deux expressions appartiennent à deux langages distincts.

Selon une autre conception (après De Bruijn et Martin-Löf) les termes démonstrations doivent êtres des objets de la théorie, au même titre que 0 ou la fonction identité. Ce point de vue amène à concevoir non seulement des extensions du λ-calcul simplement typé, mais également des extensions de la théorie des types elle-même.

On considère à présent le $\lambda 1$-calcul comme un langage permettant d'exprimer les objets mathématiques, les propositions et les démonstrations de ces propositions (et non plus comme précédemment, uniquement les démonstrations). Pour cela il faut faire le même travail que celui qui nous avons fait pour passer du λ-calcul simplement typé à la théorie des types simples : donner un type aux propositions. En théorie des types simples on donne le type o aux propositions et o est un type. En théorie des types de Martin-Löf, on donne le type $Type$ à ces propositions. Le symbole o est devenu un synonyme de $Type$. En théorie des types simples on donnait le type $o \to o \to o$ aux symboles $\wedge$, $\vee$, *etc.* A présent, on voudrait leur donner le type $Type \to Type \to Type$, mais terme n'est pas bien formé dans le λ1-calcul. On donne donc de nouvelles règles de formation des types.

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : Type}{\Gamma \vdash A \vee B : Type}$$

etc.

Mais en fait cette règle est redondante avec la règle

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : Type}{\Gamma \vdash A + B : Type}$$

On préférera donc dire que $\vee$ est une synonyme de $+$. La théorie ainsi obtenue peut être vue comme le fragment fini de la Théorie des Types de Martin-Löf, décrite dans [8] jusqu'à la pages 70.

Cette théorie identifie les symboles o et $Type$, il faut donc abandonner la règle $o : Type$ qui s'exprimerait alors $Type : Type$ et qui mènerait à un système incohérent, car il serrait possible d'y exprimer le paradoxe de Girard. Les prédicats sont maintenant des familles de types (et non des objets comme en théorie des types simples), et il est impossible de quantifier sur une variable de prédicat.

Il est, par exemple, impossible d'exprimer dans cette théorie le principe de récurrence sur les entiers (puisque celui-ci fait intervenir une quantification sur un prédicat). Il y a alors deux possibilités pour étendre ce langage : ajouter le principe de récurrence sous forme d'une règle de typage particulière (on obtient alors la théorie des types de Martin-Löf), ou ajouter le polymorphisme pour exprimer toute la théorie des types (on obtient alors le Calcul des constructions).

De manière générale, comme les règles $o : Type$ (de la théorie des types simples) et $o = Type$ (isomorphisme de Curry-de Bruijn-Howard) sont incompatibles (puisque la règle $Type : Type$ mène à un calcul incohérent, il y a deux façon d'inclure l'isomorphisme de Curry-De Bruijn-Howard à la théorie des types simples.

- Ou bien le type des propositions est le type des types ($o = Type$) et dans ce cas les prédicats sont des familles de types et non des objets. Même si on autorise la quantification sur les symboles de prédicats, le type nat des naturels et le type $real$ des réels (qui peuvent être construits comme des ensembles de naturels) n'ont pas me même type : nat a le type $Type$, alors que $real$ a le type $Kind$.

- Ou bien le type des propositions est un type comme un autre ($o : Type$) et dans ce cas propositions et types ne sont pas identifiés, puisque les premières ont le type o et les second le type $Type$.

Une troisième possibilité consiste à prendre $o : Type$ et $o \subset Type$. c'est-à-dire à identifier les propositions avec certains des types, mais en se laissant la possibilité d'avoir d'autres types, comme nat ou o lui-même.

Bibliographie

La sémantique de Heyting et Kolmogorov est apparue en 1931-1932. En 1958 Curry remarque dans [1] que les types des λ-termes bien typés sont des tautologies intuitionnistes et établit la correspondance entre démonstrations et termes d'une part et propositions et types d'autre part. En 1965, dans [9] Tait remarque la correspondance entre normalisation et élimination des coupures. Suivant Curry et Tait, Howard propose en 1969 dans [7] un λ-calcul avec types dépendants et étend l'isomorphisme à la logique du premier ordre. Suivant Heyting, De Bruijn propose indépendamment en 1968 dans [2] une autre version du λ-calcul avec types dépendants. Ce λ-calcul étant la base d'un des premiers systèmes informatique de vérification de démonstrations : le système AUTOMATH. Une présentation moderne du $\lambda\Pi$-calcul est donnée dans [6]. Le paradoxe de Girard qui montre que la règle $Type : Type$ permet de typer des termes non normalisables (et que la théorie des types correspondante est incohérente) est exposé dans [4]. L'élimination des coupures pour la logique du premier ordre est montrée par Gentzen en 1934 [3]. En 1969, Howard donne une démonstration de l'élimination des coupures basée sur la normalisation de son système de λ-calcul. L'élimination des coupures est exposée en détails dans [5]. Une théorie des types basée sur le λ-calcul avec types dépendants est présentée dans [8].

Bibliographie

[1] H.B. Curry, R.Feys, Combinatory Logic, Vol. 1, *North Holland*, Amsterdam, 1958.

[2] N.G. De Bruijn, The Mathematical Language AUTOMATH, its Usage, and some of its Extensions. *Symposium on Automatic Demonstration*, Versailles, 1968. Lecture Notes in Mathematics 125, Springer-Verlag, 1970, pp. 29-61.

[3] G. Gentzen, The Collected Work of Gehrard Gentzen, M.E. Szabo (Ed.), *North Holland*, 1969.

[4] J.Y. Girard, Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur, *Thèse d'État*, Université de Paris VII, 1972.

[5] J.Y. Girard, Y. Lafont, P. Taylor, Types and Proofs, *Cambrige University Press*, 1989.

[6] R. Harper, F. Honsel, G. Plotkin, A Framework for Defining Logics, *Journal of the Association for Computing Machinery*, 40, 1, 1993, pp. 143-184.

[7] W.A. Howard, The Formulæ-as-type Notion of Construction, 1969, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J.R. Hindley, J.P. Seldin (Eds.), Academic Press, 1980, pp. 479-490.

[8] P. Martin-Löf, Intuitionistic Type Theory, *Bibliopolis*, Napoli, 1984.

[9] W.W. Tait, Infinitely long terms of transfinite type. *Formal Systems and Recursive Functions*, J.N. Crossley and M.A.E. Dummett (Eds.), North-Holland, Amsterdam, 1965.

Chapitre 7

Les types inductifs

Dans le chapitre consacré au λ-calcul simplement typé nous avons vu que quand on se donnait un type de base nat et des symboles $0 : nat$, $S : nat \to nat$, on ne pouvait β-exprimer que très peu de fonctions des entiers dans les entiers (les fonctions constantes et les fonctions ajoutant une constante à l'un de leurs arguments). Quand on exprime les entiers comme des itérateurs (les entiers de Church) on peut β-exprimer un peu plus de fonctions (les polynômes étendus) mais pas beaucoup plus. En tant que langage de programmation, le λ-calcul simplement typé est un langage très pauvre.

En revanche, dans le chapitre consacré à la théorie des types simples nous avons vu qu'il était possible de montrer dans cette théorie l'existence de fonctions qui n'étaient pas exprimables par un λ-terme. Par exemple on peut montrer l'existence de la fonction caractéristique des entiers pairs. Pour montrer l'existence de cette fonction, on montre par récurrence sur x la proposition

$$\forall x\ \exists y\ ((\exists z\ (x = 2 \times z) \to y = 1) \wedge (\neg\ \exists z\ (x = 2 \times z) \to y = 0))$$

puis on en déduit en utilisant l'axiome du choix

$$\exists f\ \forall x\ ((\exists z\ (x = 2 \times z) \to (f\ x) = 1) \wedge (\neg\ \exists z\ (x = 2 \times z) \to (f\ x) = 0))$$

L'ajout d'un opérateur de choix (ou de description) à la théorie des types donne une notation pour tous les objets dont on peut démontrer l'existence. En effet si P est un prédicat tel que la proposition $\exists x\ (P\ x)$ soit démontrable alors $(C\ P)$ est une notation pour un objet tel que $(P\ (C\ P))$.

Mais, cette notation n'est associée à aucune notion de calcul. La fonction caractéristique de l'ensemble des entiers pairs est exprimée par le terme

$$\chi_{2N} = (C\ \lambda f : nat \to nat\ \forall x\ ((\exists z\ (x = 2 \times z) \to (f\ x) = 1) \wedge (\neg\ \exists z\ (x = 2 \times z) \to (f\ x) = 0)))$$

et on peut démontrer la proposition $(\chi_{2N}\ 0) = 1$ mais le terme $(\chi_{2N}\ 0)$ ne se réduit pas sur le terme 1.

La raison pour laquelle on peut montrer l'existence de cette fonction alors qu'on ne peut pas la β-exprimer par un λ-terme simplement typé est qu'il est possible de *démontrer* une proposition par récurrence, mais qu'il est impossible de β-exprimer par un terme une fonction *définie* par récurrence. L'idée du système T de Gödel est d'étendre le λ-calcul simplement typé de manière à autoriser de telles définitions.

7.1 Le Système T de Gödel

7.1.1 Définition

La forme d'une définition par récurrence d'une fonction f de type $nat \to T$ est

$$(f\ 0) = a$$

$$(f\ (S\ n)) = (g\ n\ (f\ n))$$

où a est un objet de type T et g une fonction de type $nat \to T \to T$. On notera cette fonction $(Rec^T\ a\ g)$. Le symbole Rec^T a donc le type $T \to (nat \to T \to T) \to nat \to T$. L'intention est donc que $(Rec\ a\ g\ 0)$ vaille a et $(Rec\ a\ g\ (S\ n))$ vaille $(g\ n\ (Rec\ a\ g\ n))$.

Définition Les types du *système T* sont ceux du λ-calcul simplement typé à un type de base : nat. Les termes du système T sont ceux du λ-calcul simplement typé, avec des variables distinguées $0 : nat$, $S : nat \to nat$, $Rec^T : T \to (nat \to T \to T) \to nat \to T$. La réduction du système T est donnée par

- $(\lambda x : T\ t\ u) \rhd t[x \leftarrow u]$,
- $(Rec\ a\ g\ 0) \rhd a$,
- $(Rec\ a\ g\ (S\ n)) \rhd (g\ n\ (Rec\ a\ g\ n))$.

Les expressions réductibles par ces deux dernières règles s'appellent des *radicaux de récurrence*. On peut également ajouter la règle η

- $\lambda x : T\ (t\ x) \rhd t$ (si x n'apparaît pas dans t)

Remarque Au lieu d'ajouter une variable Rec^T de type $T \to (nat \to T \to T) \to nat \to T$, on peut préférer étendre le langage des termes par une construction primitive Rec^T et ajouter une règle de typage

$$\frac{\Gamma \vdash a : T \quad \Gamma \vdash g : nat \to T \to T \quad \Gamma \vdash k : nat}{\Gamma \vdash (Rec^T\ a\ g\ k) : T}$$

7.1.2 Exemples

La fonction double est définie par

$$d = \lambda a : nat\ (Rec^{nat}\ 0\ \lambda x : nat\ \lambda y : nat\ (S\ (S\ y))\ a)$$

L'addition est définie par le terme

$$+ = \lambda a : nat\ \lambda b : nat\ (Rec^{nat}\ a\ \lambda x : nat\ \lambda y : nat\ (S\ y)\ b)$$

La multiplication par le terme

$$\times = \lambda a : nat\ \lambda b : nat\ (Rec^{nat}\ 0\ \lambda x : nat\ \lambda y : nat\ (+\ y\ a)\ b)$$

La fonction puissance par le terme

$$\uparrow = \lambda a : nat\ \lambda b : nat\ (Rec^{nat}\ (S\ 0)\ \lambda x : nat\ \lambda y : nat\ (\times\ y\ a)\ b)$$

Le prédécesseur par le terme

$$pred = \lambda a : nat\ (Rec^{nat}\ 0\ \lambda x : nat\ \lambda y : nat\ x\ a)$$

La fonction caractéristique de $\{0\}$ par le terme

$$\chi_{\{0\}} = \lambda a : nat\ (Rec^{nat}\ (S\ 0)\ \lambda x : nat\ \lambda y : nat\ 0\ a)$$

La fonction caractéristique des entiers non nuls par le terme

$$\chi_{N^*} = \lambda a : nat\ (Rec^{nat}\ 0\ \lambda x : nat\ \lambda y : nat\ (S\ 0)\ a)$$

La fonction caractéristique des entiers pairs par

$$\chi_{2N} = \lambda a : nat\ (Rec^{nat}\ (S\ 0)\ \lambda x : nat\ \lambda y : nat\ (\chi_{\{0\}}\ y)\ a)$$

7.1.3 Propriétés

Proposition La réduction du système T est confluente.
Démonstration Par la méthode habituelle.

Proposition Tout terme du système T est fortement normalisable.
Démonstration On adapte la méthode de Tait.

Remarque Une définition par récurrence d'une fonction à plusieurs arguments est souvent exprimée sous la forme

$$(f\ x_1\ ...\ x_{p-1}\ 0) = (a\ x_1\ ...\ x_{p-1})$$

$$(f\ x_1\ ...\ x_{p-1}\ (S\ n)) = (g\ x_1\ ...\ x_{p-1}\ n\ (f\ x_1\ ...\ x_{p-1}\ n))$$

Par exemple l'addition est définie par

$$(+\ x\ 0) = x$$

$$(+\ x\ (S\ n)) = (S\ (+\ x\ n))$$

De telles fonctions peuvent être exprimées dans le système T, bien que les paramètres $x_1\ ...\ x_{p-1}$ ne soient pas explicitement pris en compte, car il est possible d'appliquer le récurseur Rec^T à des termes comportant des variables libres. On peut donc exprimer dans le système T toutes les fonctions *récursives primitives*, c'est-à-dire toutes les fonctions construites par composition et récurrence (comme ci-dessus avec des paramètres $x_1, ..., x_{n-1}$ de type *nat*) à partir de la fonction nulle, de la fonction successeur et des projections. De plus il est possible de définir des *fonctionnelles récursives de type fini* c'est-à-dire des fonctions construites par composition et récurrence (avec des paramètres et un résultat de types quelconques) à partir de la fonction nulle, de la fonction successeur et des projections.

Par exemple on peut définir dans le système T la fonction d'Ackermann A telle que

$$(A\ 0\ m) = (S\ m)$$

$$(A\ (S\ p)\ 0) = (A\ p\ (S\ 0))$$

$$(A\ (S\ p)\ (S\ q)) = (A\ p\ (A\ (S\ p)\ q))$$

par le terme

$$A = \lambda n : nat\ (Rec^{nat \to nat}\ S\ \lambda p : nat\ \lambda f : nat \to nat\ \lambda m : nat\ (Rec^{nat}\ (f\ (S\ 0))\ \lambda q : nat\ \lambda s : nat\ (f\ s)\ m)\ n)$$

7.2 La théorie des types de Martin-Löf

7.2.1 Les axiomes de l'égalité et les axiomes de Peano

Dans le λ1-calcul, on peut exprimer le premier axiome de l'égalité,

$$refl^T : \forall x : T\ (x = x)$$

en revanche on ne peut pas exprimer le second

$$\forall P : T \to Type\ \forall x : T\ \forall y : T\ ((x = y) \to ((P\ x) \to (Py)))$$

car il est impossible de quantifier sur un symbole de prédicat. On peut néanmoins ajouter un schéma d'axiomes c'est-à-dire, pour chaque terme P un axiome

$$L^{T,P} : \forall x : T\ \forall y : T\ ((x = y) \to ((P\ x) \to (Py)))$$

De même, on peut exprimer les deux premiers axiomes de Peano

$$p_1 : \forall x : nat \; \forall y : nat \; ((S \; x) = (S \; y)) \to (x = y)$$

$$p_2 : \forall x : nat \; (0 = (S \; x)) \to \perp$$

mais on remplace le principe de récurrence

$$\forall P : nat \to Type \; (P \; 0) \to (\forall n : nat \; (P \; n) \to (P \; (S \; n))) \to \forall n : nat \; (P \; n)$$

par un schéma d'axiomes c'est-à-dire, pour chaque terme P de type $nat \to Type$, un axiome

$$Rec^P : (P \; 0) \to (\forall n : nat \; (P \; n) \to (P \; (S \; n))) \to \forall n : nat \; (P \; n)$$

La normalisation du λ1-calcul ne suffit plus pour montrer la cohérence de cette théorie. En effet, il se pourrait que l'on puisse démontrer $\perp$ en utilisant ces axiomes. Pour montrer la cohérence de cette théorie, on introduit donc de nouvelles sorte de coupures *les coupures d'égalité et de récurrence.*

7.2.2 Les coupures d'égalité

Soit P un terme de type $T \to Type$. Si a est un terme de type T et b un terme démonstration de $(P \; a)$, alors on peut construire le terme $(refl^T \; a)$ démonstration de $(a = a)$. Ensuite on peut construire le terme $(L^{T,P} \; a \; a \; (refl^T \; a) \; b)$ qui est une démonstration de $(P \; a)$.

Il y a moralement une coupure dans cette démonstration puisqu'on part de $(P \; a)$ pour remplacer a par lui-même, la démonstration de $a = a$ étant donnée par le premier axiome de l'égalité. De telles coupures sont appelées *coupures d'égalité.* Les coupures traditionnelles étaient obtenues en utilisant une règle d'élimination sur un symbole ($\to$, $\forall$, etc.) produit par une règle d'introduction. Ici on peut considérer le premier axiome de l'égalité comme un principe d'introduction de l'égalité et le second comme un principe d'élimination. Une démonstration contient donc une coupure d'égalité quand on élimine par le second axiome une égalité introduite par le premier axiome.

On veut donc ajouter une règle de réduction pour éliminer les coupures d'égalité. Cette règle est

- $(L^{T,P} \; a \; a \; (refl^T \; a) \; b) \rhd b.$

7.2.3 Les coupures de récurrence

Soit P un terme de type $nat \to Type$. Si on a deux termes a et g démonstrations des propositions $(P \; 0)$ et $\forall n : nat \; ((P \; n) \to (P \; (S \; n)))$

$$a : (P \; 0)$$

$$g : \forall n : nat \; (P \; n) \to (P \; (S \; n))$$

On peut en déduire par récurrence la proposition $\forall n : nat \; (P \; n)$. Cette démonstration s'exprime par le terme $(Rec^P \; a \; g)$.

Si on déduit ensuite de la proposition $\forall n : nat \; (P \; n)$, la proposition $(P \; 0)$ (par la règle $\forall$-élim) on obtient la démonstration $(Rec^P \; a \; g \; 0)$. Il y a moralement une coupure dans cette démonstration puisqu'on part de $(P \; 0)$ et $\forall n : nat \; (P \; n) \to (P \; (S \; n))$ pour déduire $\forall n : nat \; (P \; n)$ et enfin retomber sur $(P \; 0)$.

De même, si on déduit de la proposition $\forall n : nat \; (P \; n)$, par exemple la proposition $(P \; (S \; (S \; 0))))$ (par la règle $\forall$-élim) on obtient la démonstration $(Rec^P \; a \; g \; (S \; (S \; 0)))$. Il y a moralement une coupure dans cette démonstration, puisque en utilisant a et g on aurait pu déduire $(P \; (S \; 0))$ puis $(P \; (S \; (S \; 0)))$ sans utiliser la récurrence. Cette démonstration est exprimée par le terme $(g \; (S \; 0) \; (g \; 0 \; a))$.

Ces coupures sont appelées *coupures de récurrence.* Les coupures traditionnelles étaient obtenues en utilisant une règle d'élimination sur un symbole ($\to$, $\forall$, etc.) produit par une règle d'introduction. Ici on peut considérer les symboles 0 et S comme des symboles d'introduction des entiers et le schéma de récurrence comme un symbole d'élimination des entiers. Une démonstration contient donc une coupure de récurrence quand on élimine un entier introduit par les symboles 0 et S.

On veut donc ajouter des règles de réductions pour éliminer les coupures de récurrence. Ces règles sont

- $(Rec^P\ a\ g\ 0) \rhd a$
- $(Rec^P\ a\ g\ (S\ n)) \rhd (g\ n\ (Rec^P\ a\ g\ n))$

Ce sont donc exactement les règles de réduction du système T. Définitions par récurrence et démonstrations par récurrence se correspondent donc par l'isomorphisme de Curry-De Bruijn-Howard. De même se correspondent la réduction des radicaux de récurrence et l'élimination des coupures de récurrence.

7.2.4 La théorie des types de Martin-Löf

Définition La *théorie des types de Martin-Löf* est le λ1-calcul, étendu par les axiomes de l'égalité, les axiomes de Peano et la réduction des radicaux d'égalité et de récurrence.

Remarque Le symbole *Rec* sert aussi bien à définir des objets par récurrence, qu'à démontrer des propositions par récurrence. Par exemple la fonction prédécesseur est exprimée par le terme

$$pred = \lambda a : nat\ (Rec^{\lambda x:nat\ nat}\ 0\ \lambda x : nat\ \lambda y : nat\ x\ a)$$

Remarque L'axiome de Peano

$$\forall x : nat\ \forall y : nat\ ((S\ x) = (S\ y)) \to (x = y)$$

est en fait superflu dans la théorie des types de Martin-Löf puisqu'on peut le démontrer ainsi. Si on suppose $(S\ x) = (S\ y)$ alors par les axiomes de l'égalité on démontre $(pred\ (S\ x)) = (pred\ (S\ y))$, et cette proposition se réduit sur $x = y$.

Proposition La réduction est confluente dans la théorie des types de Martin-Löf.
Démonstration Par la méthode habituelle.

Proposition Tout terme est fortement normalisable dans la théorie des types de Martin-Löf.
Démonstration On adapte la méthode de Tait.

Proposition Soit t un terme normal bien typé dans la théorie des types de Martin-Löf sans autres variables libres que les variables distinguées.

- Le terme t n'a pas le type $\emptyset$.
- Si le terme t a le type $a = b$ alors il est de la forme $(refl^T\ a)$ et de ce fait a et b sont des termes équivalents.
- Si le terme t a le type $\Sigma y : U\ V$ alors il est de la forme $(p\ v\ w)$.
- Si le terme t a le type $U + V$ alors t il est de la forme $(i\ v)$ ou $(j\ w)$.
- Si le terme t a le type nat alors il est de la forme $(S^n\ 0)$.

Démonstration Par récurrence sur la structure de t. Le terme t s'écrit $(u\ c_1\ ...\ c_n)$ où u n'est pas une application. On considère successivement les cas suivants :

- u est la variable $nat : Type$,
- u est la variable $=^T : T \to T \to Type$,
- u est la variable $0 : nat$,
- u est la variable $S : nat \to nat$,

- u est la variable $p_2 : \forall x : nat\ (0 = (S\ x)) \rightarrow \emptyset$,
- u est une variable $refl^T : \forall x : T\ (x = x)$,
- u a la forme $\lambda x : T\ v$,
- u a la forme $(p\ v\ w)$,
- u a la forme $(\pi_1\ v)$ ou $(\pi_2\ v)$,
- u a la forme $(i\ v)$ ou $(j\ w)$,
- u a la forme $(\delta\ v\ f\ g)$,
- u a la forme $(R\ v)$,
- u a la forme $(Rec^P\ a\ b\ k)$,
- u a la forme $(L^{T,P}\ a\ b\ k)$.

Corollaire 1 La théorie des types de Martin-Löf est cohérente.

Corollaire 2 Si la proposition $A \vee B$ est démontrable alors A est démontrable ou B est démontrable.

Corollaire 3 Si la proposition $\exists x : A\ B$ est démontrable alors il existe un terme t tel que la proposition $B[x \leftarrow t]$ soit démontrable.

Corollaire des Corollaires Ces propriétés valent aussi pour l'arithmétique intuitionniste du premier ordre.

7.3 Les démonstrations d'existence et les notations pour les fonctions

7.3.1 Les démonstrations d'existence en théorie des types de Martin-Löf

Proposition Une fonction f a une démonstration d'existence en théorie des types de Martin-Löf si et seulement si elle est exprimable en théorie des types de Martin-Löf.
Démonstration Soit une fonction f représentée par une proposition P. Soit une démonstration t normale de $\forall x : nat\ \exists y : nat\ P$. Le terme $F = \lambda x : nat\ (\pi_1\ (t\ x))$ exprime la fonction f. Réciproquement, soit f une fonction calculable exprimée par un terme F en théorie des types de Martin-Löf. La proposition $y = (F\ x)$ représente la fonction f et on peut démontrer la proposition $\forall x\ \exists y\ (y = (F\ x))$.

7.3.2 L'équivalence calculatoire de la théorie des types de Martin-Löf et du système T de Gödel

On cherche maintenant à montrer que l'ensemble des fonctions exprimables en théorie des types de Martin-Löf est exactement l'ensemble des fonctions exprimables dans le système T de Gödel.

L'élimination des types dépendants, des axiomes, du type vide et de l'union disjointe

Proposition Une fonction exprimables en théorie des types de Martin-Löf est également exprimable dans dans un λ-calcul typé comprenant un unique type de base nat, un produit non dépendant $A \rightarrow B$, une somme non dépendante $A \times B$, des symboles $0 : nat$, $S : nat \rightarrow nat$, des constructeurs de termes $p^{a,b}$, $\pi_1^{A,B}$ et $\pi_2^{A,B}$ un récurseur pour les entiers et les règles de réductions associées.

Démonstration On commence par éliminer les types dépendants. On définit une fonctions β qui associe à chaque famille de type de la théorie des types de Martin-Löf un type sans dépendance de cette théorie.

$$(\beta\ Kind) = nat$$

$$(\beta\ Type) = nat$$

$$(\beta\ x) = nat$$

$$(\beta\ (\Pi x : A\ B)) = (\beta\ A) \to (\beta\ B)$$

$$(\beta\ (\Sigma x : A\ B)) = (\beta\ A) \times (\beta\ B)$$

$$(\beta\ (A + B)) = (\beta\ A) + (\beta\ B)$$

$$(\beta\ \emptyset) = nat$$

$$(\beta\ =) = nat$$

$$(\beta\ (\lambda x : A\ t)) = (\beta\ t)$$

$$(\beta\ (t\ u)) = (\beta\ t)$$

On définit de même une fonction α qui associe à chaque objet de la théorie des types de Martin-Löf un objet de cette théorie.

$$(\alpha\ x) = x$$

$$(\alpha\ \lambda x : A\ t) = \lambda x : (\beta\ A)\ (\alpha\ t)$$

$$(\alpha\ (u\ v)) = ((\alpha\ u)\ (\alpha\ v))$$

$$(\alpha\ (p^{A,B}\ u\ v)) = (p^{\beta A,\beta B}(\alpha\ u)\ (\alpha\ v))$$

$$(\alpha\ (\pi_1^{A,B}\ u)) = (\pi_1^{\beta A,\beta B}(\alpha\ u))$$

$$(\alpha\ (\pi_2^{A,B}\ u)) = (\pi_2^{\beta A,\beta B}(\alpha\ u))$$

$$(\alpha\ (i^{A,B}\ u)) = (i^{\beta A,\beta B}(\alpha\ u))$$

$$(\alpha\ (j^{A,B}\ u)) = (j^{\beta A,\beta B}(\alpha\ u))$$

$$(\alpha\ (\delta^{A,B,C}\ u\ f\ g)) = (\delta^{\beta A,\beta B,\beta C}(\alpha\ u)\ (\alpha\ f)\ (\alpha\ g))$$

$$(\alpha\ (R^{A}\ u)) = (R^{\beta A}(\alpha\ u))$$

$$(\alpha\ (Rec^{A}\ a\ g\ k)) = (Rec^{\beta A}(\alpha\ a)\ (\alpha\ g)\ (\alpha\ k))$$

$$(\alpha\ (L^{T,P}\ a\ b\ c\ d)) = (L^{\beta T,\beta P}(\alpha\ a)\ (\alpha\ b)\ (\alpha\ c)\ (\alpha\ d))$$

Si t est un terme de type $nat \to nat$ alors t et $(\alpha\ t)$ expriment la même fonction (car le terme pur contenu est le même).

On remplace ensuite les termes de la forme $(L^{T,P}\ a\ b\ c\ d)$ par d. Puis on remplace le symbole démonstration du premier axiome de l'égalité, celui démonstration du premier axiomes de Peano et les termes de la forme $(R\ t)$ (élimination du type vide) par un terme quelconque du bon type.

On remplace enfin les termes de la forme $(i\ a) : A+B$ par $(p\ 0\ (p\ a\ b_0)) : nat \times (A \times B)$ où b_0 est un terme quelconque du type B, les termes de la forme $(i\ b) : A+B$ par $(p\ (S\ 0)\ (p\ a_0\ b)) : nat \times (A \times B)$ où a_0 est un terme de type A quelconque et les termes de type $(\delta\ u\ f\ g)$ par $(Rec\ (\pi_1\ (\pi_2\ u))\ \lambda n\ \lambda x\ (\pi_2\ (\pi_2\ u))\ (\pi_1\ u))$.

On montre ainsi que les fonctions dont l'existence est prouvable en théorie des types de Martin-Löf sont exprimables dans le système T avec somme.

L'élimination de la somme

Proposition (Schütte) Soient $A_1, ..., A_n$ des types du système T, il existe des termes $R_1, ..., R_n$ tels que pour tous $a_1, ..., a_n$ termes du système T de types respectifs $A_1, ..., A_n$ et $g_1, ..., g_n$ termes du système T de types respectifs $nat \to A_1 \to ... \to A_n \to A_1, ..., nat \to A_1 \to ... \to A_n \to A_n$, pour tout i,

$$(R_i\ a_1\ ...\ a_n\ g_1\ ...\ g_n\ 0) \rhd a_i$$

$$(R_i\ a_1\ ...\ a_n\ g_1\ ...\ g_n\ (S\ z)) \rhd (g_i\ z\ (R_1\ a_1\ ...\ a_n\ g_1\ ...\ g_n\ z)\ ...\ (R_n\ a_1\ ...\ a_n\ g_1\ ...\ g_n\ z))$$

Démonstration Voir [8].

Remarque Soient

$$f_1 = (R_1\ a_1\ ...\ a_n\ g_1\ ...\ g_n)$$

$$...$$

$$f_n = (R_n\ a_1\ ...\ a_n\ g_1\ ...\ g_n)$$

Les fonctions $f_1, ..., f_n$ sont définies par récurrence mutuelle.

$$(f_i\ 0) = a_i$$

$$(f_i\ (S\ z)) = (g_i\ z\ (f_1\ z)\ ...\ (f_n\ z))$$

Proposition Toute fonction exprimable dans le système T avec paires est exprimable dans le système T.
Démonstration A chaque type du système T avec paires on associe une suite finie de types du système T.

- $|nat| = nat$,
- $|A \to B| = A_1 \to ... \to A_n \to B_1, ..., A_1 \to ... \to A_n \to B_p$ où $A_1, ..., A_n = |A|$ et $B_1, ..., B_p = |B|$.
- $|A \times B| = A_1, ..., A_n, B_1, ..., B_p$ où $A_1, ..., A_n = |A|$ et $B_1, ..., B_p = |B|$.

On associe ensuite à chaque terme a de type A dans le système T avec paire, une suite de termes $|a| = a_1, ..., a_n$ de types respectifs $A_1, ..., A_n$ où $A_1, ..., A_n = |A|$.

- $|0| = 0$,
- $|S| = S$,
- $|x| = x_1, ..., x_n$,
- $|(u\ v)| = (u_1\ v_1\ ...\ v_p), ..., (u_n\ v_1\ ...\ v_p)$ où $u_1, ..., u_n = |u|$ et $v_1, ..., v_p = |v|$,
- $|\lambda x : A\ u| = \lambda x_1 : A_1\ ...\ \lambda x_n : A_n\ u_1, ..., \lambda x_1 : A_1\ ...\ \lambda x_n : A_n\ u_p$ où $A_1, ..., A_n = |A|$ et $u_1, ..., u_p = |u|$,
- $|(p\ u\ v)| = u_1, ..., u_n, v_1, ..., v_p$ où $u_1, ..., u_n = |u|$ et $v_1, ..., v_p = |v|$,
- $|(\pi_1\ u)| = u_1, ..., u_n$ où $A \times B$ est le type de u, n la longueur de $|A|$, p la longueur de $|B|$ et $u_1, ..., u_n, u_{n+1}, ..., u_{n+p} = |u|$.
- $|(\pi_2\ u)| = u_{n+1}, ..., u_{n+p}$ où $A \times B$ est le type de u, n la longueur de $|A|$, p la longueur de $|B|$ et $u_1, ..., u_n, u_{n+1}, ..., u_{n+p} = |u|$.
- $(Rec\ a\ b\ k) = (R_1\ a_1\ ...\ a_n\ b_1\ ...\ b_p\ k), ..., (R_m\ a_1\ ...\ a_n\ b_1\ ...\ b_p\ k)$ où A est le type de $(Rec\ a\ b\ k)$, $A_1, ..., A_m = |A|$, $R_1, ..., R_m$ les termes de Schütte associés aux types $A_1, ..., A_m$, $a_1, ..., a_n = |a|$ et $b_1, ..., b_p = |b|$.

On montre ensuite que si t et u sont deux termes tel que $t \rhd u$ alors $t_1 \rhd u_1$, ..., $t_n \rhd u_n$ où $t_1, ..., t_n = |t|$ et $u_1, ..., u_n = |u|$. Soit t un terme du système T de type $nat \rightarrow ... \rightarrow nat \rightarrow nat$, on a $|T| = T$ et si $n_1, ..., n_p$ sont des entiers alors les formes normales des termes $(t\ n_1\ ...\ n_p)$ et $(|t|\ n_1\ ...\ n_p)$ sont le même entier. Donc toutes les fonctions exprimables dans le système T avec paires sont exprimables dans le système T.

Corollaire Ces trois ensembles de fonctions sont identiques :

- les fonctions dont l'existence peut être montrée en théorie des types de Martin-Löf,
- les fonctions exprimables en théorie des types de Martin-Löf,
- les fonctions exprimables dans le système T.

7.3.3 L'arithmétique intuitionniste du premier ordre

Trivialement si l'existence d'une fonction f peut être montrée en arithmétique intuitionniste du premier ordre, alors l'existence de f peut être montrée en théorie des types de Martin-Löf.

Réciproquement on montre, en arithmétisant la démonstration de normalisation du système T jusqu'à un ordre donné, que si une fonction est exprimable dans le système T alors on peut montrer son existence en arithmétique intuitionniste du premier ordre. Les ensembles de fonctions suivants sont donc égaux aux trois ci-dessus :

- les fonctions dont l'existence peut être montrée en arithmétique intuitionniste du premier ordre,
- les fonctions dont l'existence peut être montrée en arithmétique intuitionniste d'ordre supérieur prédicative.

7.3.4 L'optimisation de la construction du terme, la réalisabilité

Soit P une propriété (spécification) d'une fonction, par exemple

$$P = \forall x\ ((\exists z\ (x = 2 \times z) \rightarrow (f\ x) = 0) \wedge (\neg\ \exists z\ (x = 2 \times z) \rightarrow (f\ x) = (S\ 0)))$$

D'une démonstration intuitionniste de $\exists f\ P$ on peut donc extraire un terme du système T (ou du système F_w) qui vérifie la propriété P. L'isomorphisme de Curry-De Bruijn-Howard a donc comme conséquence l'isomorphisme des tâches consistant à

- écrire une démonstration d'une proposition,
- écrire un programme vérifiant une spécification.

Néanmoins, démontrer a certains avantages sur programmer.

- L'expression du *quoi programmer* est concentrée dans l'écriture de la spécification (proposition), et l'expression du *comment le programmer* dans l'écriture de la démonstration.
- Les programmes extraits sont garantis vérifier leur spécification.
- On peut vérifier mécaniquement qu'une démonstration est une démonstration d'une proposition.
- Les programmes extraits des démonstrations terminent toujours (normalisation forte).
- Malgré cela le système T est très expressif d'un point de vue calculatoire, puisqu'il permet d'exprimer toutes les fonctions dont on peut montrer l'existence en logique intuitionniste du premier ordre ou dans l'arithmétique intuitionniste d'ordre supérieur prédicative.
- La démonstration d'une proposition P peut se synthétiser (partiellement) automatiquement.

Néanmoins les programmes extraits naïvement ne sont pas très efficaces. En effet, si t est une démonstration de $\forall x \exists y P$ et a un entier alors la normalisation du terme $(t\ a)$ calcule non seulement l'entier b mais aussi une démonstration normale de la proposition $P[x \leftarrow a, y \leftarrow b]$. Autrement dit, l'exécution du programme, calcule non seulement la valeur de sortie, mais aussi une démonstration de la correction de l'exécution. Pour rendre les programmes ainsi construits plus efficace, il faut supprimer le calcul de la démonstration de correction et extraire uniquement l'*information calculatoire* de la démonstration.

Pour cela, on considère un terme t de type $nat \rightarrow ... \rightarrow nat \rightarrow nat$ de la théorie des types de Martin-Löf, on élimine les types dépendants comme ci-dessus mais en gardant trace de la provenance des variables de types : on pose un nouveau type de base Ω et une nouvelle variable ω de type Ω, au lieu de poser

$$(\beta\ x) = nat$$

pour toutes les variables, on pose

$$(\beta\ nat) = nat$$

$$(\beta\ =) = \Omega$$

$$(\beta\ \emptyset) = \Omega$$

On élimine ensuite les axiomes, le type vide les unions disjointes et les sommes comme ci-dessus. On obtient un terme de type $nat \rightarrow ... \rightarrow nat \rightarrow nat$ dans le système T étendu par le type de base Ω.

Intuitivement un terme de type Ω est un terme sans contenu calculatoire et peut donc être effacé. Pour cela on définit une fonction de l'ensemble des types du système T dans lui même :

- $|nat| = nat$,
- $|\Omega| = \Omega$,
- $|A \rightarrow B| = \Omega$ si $|A| = \Omega$ et $|B| = \Omega$,
- $|A \rightarrow B| = \Omega$ si $|A| \neq \Omega$ et $|B| = \Omega$,
- $|A \rightarrow B| = |B|$ si $|A| = \Omega$ et $|B| \neq \Omega$,
- $|A \rightarrow B| = |A| \rightarrow |B|$ si $|A| \neq \Omega$ et $|B| \neq \Omega$,

On montre par récurrence que ou bien $|A| = \Omega$ ou bien Ω n'a pas d'occurrence dans $|A|$.
On définit ensuite une fonction de l'ensemble des termes du système T dans lui même :

- Soit x une variable de type A,
  - $|x| = x$ si $|A| \neq \Omega$
  - $|x| = \omega$ si $|A| = \Omega$.
- Soit u un terme de type $A \rightarrow B$ et v un terme de type A,
  - $|(u\ v)| = \omega$ si $|A| = \Omega$ et $|B| = \Omega$,
  - $|(u\ v)| = \omega$ si $|A| \neq \Omega$ et $|B| = \Omega$,
  - $|(u\ v)| = |u|$ si $|A| = \Omega$ et $|B| \neq \Omega$,
  - $|(u\ v)| = (|u|\ |v|)$ si $|A| \neq \Omega$ et $|B| \neq \Omega$.
- Soit x une variable de type A et v un terme de type B,
  - $|\lambda x : A\ u| = \omega$ si $|A| = \Omega$ et $|B| = \Omega$,
  - $|\lambda x : A\ u| = \omega$ si $|A| \neq \Omega$ et $|B| = \Omega$,
  - $|\lambda x : A\ u| = |u|$ si $|A| = \Omega$ et $|B| \neq \Omega$,

- $|\lambda x : A\ u| = \lambda x : |A|\ |u|$ si $|A| \neq \Omega$ et $|B| \neq \Omega$.

- Soit a un terme de type A, b un terme de type $nat \to A \to A$ et k un terme de type nat,
  - $|(Rec\ a\ b\ k)| = (Rec\ |a|\ |b|\ |k|)$ si $|A| \neq \Omega$,
  - $|(Rec\ a\ b\ k)| = \omega$ si $|A| = \Omega$.

Si t est un terme de type $nat \to ... \to nat \to nat$ alors $|t|$ est un terme de même type et si $n_1, ..., n_p$ sont des entiers alors les formes normales des termes $(t\ n_1\ ...\ n_p)$ et $(|t|\ n_1\ ...\ n_p)$ sont le même entier. Le terme $|t|$ ne contient que l'information calculatoire du terme t.

7.4 Les types inductifs

Outre les entiers, d'autres types de données peuvent être définis inductivement : les listes, les arbres, etc.

Pour les listes d'entiers, par exemple, on a deux constructeurs

$$nil : list$$

$$cons : nat \to list \to list$$

Et un récurseur

$$Rec : \forall P : list \to Type\ (P\ nil) \to (\forall a\ \forall l : list\ (P\ l) \to (P\ (cons\ a\ l))) \to \forall l : list\ (P\ l)$$

qui permet de construire des fonctions par récurrence sur la structure de la liste (longueur, concaténation, etc.) et de démontrer des propriétés par récurrence sur la structure de la liste.

Bibliographie

[1] J.R. Hindley, J.P. Seldin, Introduction to Combinators and λ-Calculus, *Cambridge University Press* (1986).

[2] J.Y. Girard, Y. Lafont, P. Taylor, Types and Proofs, *Cambrige University Press* (1989).

[3] J.L. Krivine, Lambda-calcul, types et modèles, Masson (1990).

[4] D. Leivant, Reasoning about Functional Programs and Complexity Classes Associated with Type Disciplines, Annual Symposium on Foundations of Computer Science (1983) pp. 460-469.

[5] P. Martin-Löf, Intuitionistic Type Theory, *Bibliopolis*, Napoli (1984).

[6] Ch. Paulin-Mohring, Extraction de Programmes dans le Calcul des Constructions, Thèse de Doctorat, Université de Paris 7 (1989).

[7] Ch. Paulin-Mohring, Inductive Definitions in the System Coq, Rules and Properties, Typed Lambda Calculi and Applications, Lecture Notes in Computer Science 664, Springer-Verlag (1993) pp. 328-345.

[8] A.S. Troelstra, Metamathematical Investigation of Intuitionistic Arithmetic and Analysis, Lecture Notes in Mathematics 344, Springer-Verlag (1973).

Chapitre 8

Le polymorphisme

Le λ-calcul simplement typé permet d'exprimer les démonstrations du calcul des propositions minimal. Le λ-calcul avec types dépendants permet d'exprimer les démonstrations du calcul des prédicats minimal du premier ordre ainsi que de la théorie prédicative des types, c'est-à-dire de la théorie dans laquelle il est possible de quantifier sur des fonctions, mais pas sur des prédicats.

Dans ce chapitre, nous étendons le λ-calcul avec types dépendants de manière à pouvoir exprimer toutes les démonstrations de la théorie des types simples.

8.1 Le Calcul des constructions

Dans le λ-calcul avec types dépendants, les règles permettant de former un produit sont les suivantes.

$$\frac{\Gamma \vdash T : Type \quad \Gamma[x:T] \vdash T' : Kind}{\Gamma \vdash \Pi x : T\ T' : Kind}$$

$$\frac{\Gamma \vdash T : Type \quad \Gamma[x:T] \vdash T' : Type}{\Gamma \vdash \Pi x : T\ T' : Type}$$

La première règle permet de former, par exemple, le produit $nat \to Type$, qui est utile, entre autres, pour déclarer une variable de famille de types tab de manière à ce que $(tab\ 0)$, $(tab\ 1)$, *etc.* soient des types : le type des tableaux de taille 0, le type des tableaux de taille 1, *etc.* La seconde de former le produit $\Pi n : nat\ (tab\ n)$ qui est, par exemple, le type de la fonction qui à chaque entier associe le tableau de longueur n contenant n fois la valeur 0.

Du point de vue de la représentation des propositions comme des types. La première règle permet de former le type $nat \to Type$, qui est nécessaire, par exemple, pour déclarer une variable de prédicat $pair$, où $(pair\ n)$ est la proposition "n est un nombre pair". La seconde permet de former le type $\Pi n : nat\ (pair\ n)$ qui est la proposition (fausse) "tout entier est pair".

De même, les règles correspondantes pour les abstractions permettent de former des familles de type, des fonctions, des prédicats et des démonstrations.

Le fait que le type du terme T soit $Type$ et non $Kind$ empèche de former le produit $Type \to Type$ qui serait utile, par exemple, pour former une famille de types *paramétrée par un type* comme la famille $list$ telles que $(list\ nat)$ soit le type des listes d'entiers. Et de même, quand on représente les propositions comme des types, le produit $(nat \to Type) \to Type$ qui est le type des ensembles d'ensembles d'entiers.

La règle

$$\frac{\Gamma \vdash T : Kind \quad \Gamma[x:T] \vdash T' : Kind}{\Gamma \vdash \Pi x : T\ T' : Kind}$$

et la règle correspondantes pour les abstractions pallient ce manque. Cette règle est appelée la règle des *constructeurs de types*.

Dans la seconde règle, cette même restriction empèche de former le type $\Pi T : Type\ (T \to T)$ qui est le type de la fonction qui associe à chaque type la fonction identité sur ce type ou la proposition $\Pi E : nat \to Type\ (ppe\ E)$ qui exprime que tout ensemble d'entier a un plus petit élément.

De même la règle des *types polymorphes*

$$\frac{\Gamma \vdash T : Kind \quad \Gamma[x : T] \vdash T' : Type}{\Gamma \vdash \Pi x : T\ T' : Type}$$

et la règle correspondantes pour les abstractions pallient ce manque.

Le système ainsi obtenu est appelé *Le Calcul des constructions.*

8.2 Le cube des systèmes de types, systèmes de types purs

Toutes ces règles ont la même forme :

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma[x : T] \vdash T' : s_2}{\Gamma \vdash \Pi x : T\ T' : s_3}$$

Seuls changent les symboles s_1, s_2, s_3 qui paramètrent ces règles. Avec $< s_1, s_2, s_3 > = < Type, Type, Type >$ on a la règle de base du λ-calcul qui permet de former des types fonctionnels. Cette règle existe déjà dans le λ-calcul simplement typé. La règle $< Type, Kind, Kind >$ caractérise les types dépendants, la règle $< Kind, Type, Type >$ les types polymorphes, et la règle $< Kind, Kind, Kind >$ les constructeurs de types.

Soit R un ensemble de règles. Le système de types caractérisé par l'ensemble des règles R est défini par les règles de typage suivantes.

Définition (Système de types du cube)
Le contexte vide est bien formé :

$$\frac{}{[\ ] \text{ bien formé}}$$

Déclaration d'une variable:

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T] \text{ bien formé}}\ s = Prop \text{ ou } s = Kind$$

Type est un genre :

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash Type : Kind}$$

Les variables sont des termes :

$$\frac{\Gamma \text{ bien formé} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

Produit :

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma[x : T] \vdash T' : s_2}{\Gamma \vdash \Pi x : T\ T' : s_3} < s_1, s_2, s_3 > \in R$$

Application :

$$\frac{\Gamma \vdash t : \Pi x : T\ T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

Abstraction :

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma[x : T] \vdash T' : s_2 \quad \Gamma[x : T] \vdash t : T'}{\Gamma \vdash \lambda x : T\ t : \Pi x : T\ T'} < s_1, s_2, s_3 > \in R$$

Conversion :

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : Type \quad \Gamma \vdash T' : Type \quad T \equiv T'}{\Gamma \vdash t : T'}$$

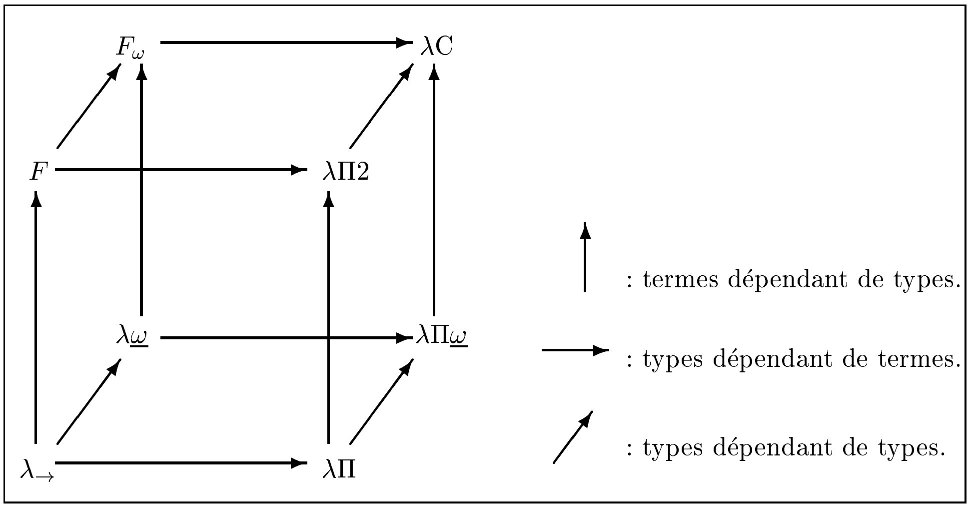
Les huit systèmes obtenus en prenant la règle

$$< Type, Type, Type >$$

et certaines des règles

$$< Type, Kind, Kind >, < Kind, Type, Type >, < Kind, Kind, Kind >$$

sont appelés les systèmes du cube [1] [1].



Il est possible de paramétrer davantage ces règles de typage. On se donne alors un ensemble S (qui dans le cas des systèmes du cube est $\{Type, Kind\}$), une ensemble A de paires d'éléments de S et un ensemble R de triplets d'éléments de S. Les règles de typage sont alors les suivantes.

Définition (Système de types purs)
Le contexte vide est bien formé :

$$\overline{[\,]\ \text{bien formé}}$$

Déclaration d'une variable:

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \text{bien formé}}\ s \in S$$

Typage des éléments de S :

$$\frac{\Gamma\ \text{bien formé}}{\Gamma \vdash s_1 : s_2} < s_1, s_2 > \in A$$

Les variables sont des termes :

$$\frac{\Gamma\ \text{bien formé} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

Produit :

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma[x : T] \vdash T' : s_2}{\Gamma \vdash \Pi x : T\ T' : s_3} < s_1, s_2, s_3 > \in R$$

Application :

$$\frac{\Gamma \vdash t : \Pi x : T\ T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

[1] Merci à Vincent Prosper pour son dessin.

Abstraction :

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma[x : T] \vdash T' : s_2 \quad \Gamma[x : T] \vdash t : T'}{\Gamma \vdash \lambda x : T \ t : \Pi x : T \ T'} < s_1, s_2, s_3 > \in R$$

Conversion :

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : Type \quad \Gamma \vdash T' : Type \quad T \equiv T'}{\Gamma \vdash t : T'}$$

8.3 La normalisation en présence de types polymorphes

8.3.1 Le système F

Les termes typables dans l'un des huit systèmes du cube sont fortement normalisables.

Une difficulté nouvelle, due aux types polymorphes, apparaît dans la preuves de normalisation de ces systèmes. Pour illustrer cette difficulté, nous présentons la preuve de normalisation du système F, c'est-à-dire du λ-calcul polymorphe, sans constructeurs de types ni types dépendants. Cette preuve peut ensuite se généraliser à des systèmes de types plus riches.

Dans le système F, il n'y a qu'un seul genre, c'est le symbole $Type$. Donc toutes le familles de types sont des types. Un produit est donc ou bien de la forme $\Pi X : Type \ T$ où T est de type $Type$, ou bien $\Pi x : T \ U$ où T et U sont de type $Type$. Dans ce dernier cas la variable (d'objet) x ne peut pas apparaître dans le type U et le type peut s'écrire $T \to U$. Les seules dépendances sont donc par rapport aux variables de type.

8.3.2 Généraliser la preuve de Tait

En tentant de généraliser la preuve de Tait, on cherche à définir par récurrence sur la structure des types des ensembles Red_T de termes réductibles. On garde les clauses habituelles pour les cas où le type est atomique (s'il y a de tels types) et pour les types flèche.

- Si T est un type atomique alors Red_T est l'ensemble des termes de type T fortement normalisables.
- Si $T = U \to V$ alors Red_T est l'ensemble des termes t tels que pour tout u dans Red_U, le terme $(t\ u)$ est dans Red_V.

Il reste le cas où le type T a la forme $\Pi X : Type\ U$. On voudrait qu'un terme t de ce type soit réductible si pour tout type V, $(t\ V)$ est réductible de type $U[X \leftarrow V]$. Au lieu de faire cette substitution, on peut garder la variable X, en sachant que c'est un prête-nom pour le type V. On aboutit ainsi à une définition d'ensembles de termes réductibles pour les types ouverts (c'est-à-dire, contenant des variables de type libres), modulo une interprétation de ces variables de type.

Une *assignation* est une fonction ψ qui associe à chaque variable de type X, un ensemble de terme E (intuitivement, l'ensemble des termes réductibles du type pour lequel X est un prête-nom) et on définit l'ensemble des termes réductibles de type T modulo l'assignation ψ par les règles suivantes.

- Si T est un type atomique alors Red_T^ψ est l'ensemble des termes de type T fortement normalisables.
- Si $T = X$ alors $Red_T^\psi = \psi(X)$.
- Si $T = U \to V$ alors Red_T^ψ est l'ensemble des termes t de type T tels que pour tout terme u de Red_U^ψ, le terme $(t\ u)$ soit dans Red_V^ψ.
- Si $T = \Pi X : Type\ U$ alors Red_T^ψ est l'ensemble des termes t tels que pour tout type V, $(t\ V)$ soit dans $Red_U^{\psi + <X,R>}$ où $R = Red_V^\psi$ et où $\psi + < X, R >$ est la fonction qui assigne à X l'ensemble R et aux variables Y distinctes de X l'ensemble $\psi(Y)$.

Hélas, cette définition est circulaire. En effet pour définir l'ensemble des termes réductibles de type $\Pi X : Type\ U$, il faudrait avoir déjà défini l'ensemble des termes réductibles pour tous les types V, y compris le type $\Pi X : Type\ U$. Il ne s'agit donc pas d'une définition correcte par récurrence sur la structure des types.

8.3.3 Les candidats de réductibilité

Pour contourner cette circularité, l'idée consiste à définir l'ensemble des termes réductibles de terme ΠX : *Type* U comme l'ensemble des termes t tels que le terme $(t\ V)$ soit dans $Red_U^{\psi+<X,R>}$ pour tout type V *et pour tout ensemble* R *de termes normalisables de type* V. Ainsi l'ensemble Red_V^{ψ} sera parmi ces ensembles, mais il n'est pas besoin de le désigner dans la définition et la circularité est évitée.

En fait, il faut imposer à l'ensemble R de vérifier également deux autres conditions de fermeture. On pose donc la définition suivante.

Définition (Candidat de réductibilité)
Un ensemble R de termes de type T est un *candidat de réductibilité* de type T si

- tout terme de R est fortement normalisable,
- si $t \in R$ et t se réduit en t' alors $t' \in R$,
- si t n'est pas une abstraction et tous les termes t' obtenus en réduisant un redex dans t sont dans R alors t est dans R.

Définition (Ensemble de termes réductibles)

- Si T est un type atomique alors Red_T^{ψ} est l'ensemble des termes de type T fortement normalisables.
- Si $T = X$ alors $Red_T^{\psi} = \psi(X)$.
- Si $T = U \rightarrow V$ alors Red_T^{ψ} est l'ensemble des termes t tels que pour tout terme u de Red_U^{ψ}, le terme $(t\ u)$ soit dans Red_V^{ψ}.
- Si $T = \Pi X$: *Type* U alors Red_T^{ψ} est l'ensemble des termes t tels que pour tout type V, et pour tout candidat de réductibilité R de type V, $(t\ V)$ est dans $Red_U^{\psi+<X,R>}$.

La proposition suivante assure que l'ensemble Red_T^{ψ} est bien parmi les ensembles R de la définition des ensembles de termes réductibles.

Proposition Soit T un type contenant des variables libres $X_1, ..., X_n$. Soient $V_1, ..., V_n$ des types quelconques, et ψ une assignation qui associe à chaque X_i un candidat de réductibilité de type V_i. Alors l'ensemble Red_T^{ψ} est un candidat de réductibilité de type $T[X_1 \leftarrow V_1, ..., X_n \leftarrow V_n]$.

Comme dans la preuve de Tait on en déduit que les termes réductibles sont fortement normalisables.

Corollaire Soit T un type contenant des variables libres $X_1, ..., X_n$. Soient $V_1, ..., V_n$ des types quelconques, et ψ une assignation qui associe à chaque X_i un candidat de réductibilité de type V_i. Alors les termes de l'ensemble Red_T^{ψ} sont des termes fortement normalisables de type $T[X_1 \leftarrow V_1, ..., X_n \leftarrow V_n]$.

La proposition suivante montre qu'il est équivalent de faire la substitution dans un type, ou de garder un prête-nom.

Proposition $Red_{T[X \leftarrow V]}^{\psi} = Red_T^{\psi+<X,Red_V^{\psi}>}$

On termine ensuite la démonstration comme celle de Tait en montrant que tout terme bien typé est réductible et donc normalisable.

Proposition Soit un terme t de type T. Soient $x_1, ..., x_n$ les variables libres d'objets de t et $U_1, ..., U_n$ leur types. Soient $X_1, ..., X_p$ et des variables libres de type de $T, U_1, ..., U_n$. Soient $V_1, ..., V_p$ de types quelconques, $u_1, ..., u_n$ des termes de type $U_i[X_1 \leftarrow V_1, ..., X_p \leftarrow V_p]$ et ψ une assignation qui associe à chacun des X_i un candidat de réductibilité de type V_i. Alors le terme $t[X_1 \leftarrow V_1, ..., X_p \leftarrow V_p][x_1 \leftarrow u_1, ..., x_n \leftarrow u_n]$ est dans Red^{ψ}_T.

Et on en déduit le résultat final.

Theorème Tout terme du système F est fortement normalisable.

Pour une preuve plus détaillée, voir [6].

8.4 Les définitions par récurrence dans les λ-calcul polymorphes

L'entier de Church n est défini comme

$$n = \lambda x\ \lambda f\ (f\ ...\ (f\ x)\ ...)$$

avec n occurrences du symboles f.

Dans le λ-calcul simplement typé on peut donner le type $nat = T \to (T \to T) \to T$ à ces termes. Si g est une fonction de type $T \to T$ et a un objet de type T, la fonction f définie par récurrence par

$$(f\ 0) = a$$

$$(f\ (S\ n)) = (g\ (f\ n))$$

est exprimée par le terme $f = \lambda n\ (n\ a\ g)$. Il est donc possible de définir des fonctions par récurrences, mais uniquement des fonctions de type $T \to T$. Il n'est en particulier pas possible de définir une fonction de type $nat \to nat$ par récurrence puisque la multiplication est exprimable, mais pas la fonction qui à n associe 2^n.

Dans le λ-calcul polymorphe en revanche, on peut définir les entiers comme de Church comme

$$n = \lambda X : Type\ \lambda x : X\ \lambda f : X \to X\ (f\ ...\ (f\ x)\ ...)$$

Ces termes ont le type $nat = \Pi X : Type\ (X \to (X \to X) \to X)$. Il est à présent possible de définir une fonction par récurrence quelque soit son type. Par exemple, la fonction $n \mapsto 2^n$ est exprimée par le terme

$$f = \lambda n : nat\ (n\ nat\ 1\ (\lambda p : nat\ (\times\ 2\ p)))$$

Proposition Toutes les fonctions exprimables dans le système T sont exprimables dans le système F.

8.5 L'isomorphisme de Curry-De Bruijn-Howard pour la théorie des types simples

8.5.1 L'expression des démonstrations

L'identification entre types et propositions nous amène à identifier o le type des propositions et $Type$ le type des types. Mais comme nous avons par ailleurs la règle $o : Type$, on aurait $Type : Type$ et le système serait alors non normalisable.

On abandonne donc le principe $o : Type$ et on garde $o = Type$. Comme on a $Type : Kind$ on a donc $o : Kind$.

Les types des prédicats $nat \to o$, $nat \to nat \to o$, $(nat \to o) \to o$ sont donc de type $Kind$ alors que nat lui même ou le type des fonctions $nat \to nat$ sont de type $Type$. Comme il n'est pas très pratique d'avoir un type différent pour le type des objets selon qu'il s'agit de fonctions ou d'ensembles, on posera plutôt $nat : Kind$, ainsi $Type$ qui avait été crée pour être le type des types ne reste que le type des propositions, et le types des objets sont montés au niveau $Kind$. Ainsi pour exprimer la théorie des types simples, on donne le type $Kind$ au symbole o et au symbole ι (ou aux symboles nat, $bool$ *etc.* si on a plusieurs types de base).

Les autres symboles ont ensuite leur type habituel $0 : nat$, $S : nat \to nat$, $\leq: nat \to nat \to o$ c'est-à-dire $\leq: nat \to nat \to Type$, *etc.*

Theorème Si une proposition P de type $Type$ est démontrable en théorie des types minimale, alors le type P est habité.
Démonstration Par récurrence sur la structure de la démonstration de P.

8.5.2 L'élimination des coupures

La théorie des types minimale

L'élimination des coupures pour la théorie des types minimale est conséquence du théorème de normalisation du Calcul des constructions.

La contrepartie logique de l'impossibilité de définir les ensembles de termes réductibles par récurrence sur la structure des types est que, comme nous l'avons déjà remarqué au chapitre 5, quand on élimine, par exemple, une coupure de la forme

$$\dfrac{\dfrac{\dfrac{\pi}{A}}{\forall P : nat \to o\ A} \quad B : nat \to o}{A[P \leftarrow B]}$$

en

$$\dfrac{\pi[P \leftarrow B]}{A[P \leftarrow B]}$$

On peut introduire, en substituant P par B dans la preuve π, des coupures de tailles arbitraires car la proposition $(B\ x)$ est arbitraire.

Proposition Soit π une démonstration en théorie des types minimale (c'est-à-dire ne contenant que le connecteur $\Rightarrow$ et le quantificateur $\forall$), et π' une démonstration obtenue en éliminant une coupure dans π. Soit t l'expression de la preuve π sous forme d'un terme du Calcul des constructions, et t' l'expression de π'. Le terme t se β-réduit en au moins une étape sur t'.

Corollaire L'élimination des coupures termine en théorie des types minimale.

Les connecteurs et les quantificateurs

Dans le $\lambda\Pi$-calcul (λ-calcul avec types dépendants) on doit ajouter des types sommes, des unions disjointes et un type vide pout pouvoir interpréter les preuves intuitionnistes (utilisant la conjonction, la disjonction, la négation et le quantificateur existentiel). Mais nous avons vu au paragraphe 5.3.9 que ces connecteurs et quantificateurs pouvaient être définis en théorie des types minimale. De plus l'élimination des coupures en théorie des types se ramène à l'élimination des coupures en théorie des types minimale, donc à la β-reduction dans le Calcul des constructions.

Autrement dit, les types sommes, les unions disjointes et le type vide peuvent se définir dans le Calcul des constructions, et leur règles de calcul se ramènent à la β-reduction.

$$\times = \lambda A : Type\ \lambda B : Type\ \Pi C : Type\ ((A \to B \to C) \to C)$$

$$<,>= \lambda A : Type\ \lambda B : Type\ \lambda x : A\ \lambda y : B\ \lambda C : Type\ \lambda u : A \to B \to C\ (u\ x\ y)$$

$$\pi_1 = \lambda A : Type\ \lambda B : Type\ \lambda u : (A \times B)\ (u\ A\ \lambda x : A\ \lambda y : B\ x)$$

$$\pi_2 = \lambda A : Type\ \lambda B : Type\ \lambda u : (A \times B)\ (u\ B\ \lambda x : A\ \lambda y : B\ y)$$

$$+ = \lambda A : Type\ \lambda B : Type\ \Pi C : Type\ ((A \to C) \to (B \to C) \to C)$$

$$i = \lambda A : Type\ \lambda B : Type\ \lambda x : A\ \lambda C : Type\ \lambda u : A \to C\ \lambda v : B \to C\ (u\ x)$$
$$j = \lambda A : Type\ \lambda B : Type\ \lambda x : B\ \lambda C : Type\ \lambda u : A \to C\ \lambda v : B \to C\ (v\ x)$$
$$\delta = \lambda A : Type\ \lambda B : Type\ \lambda C : Type\ \lambda x : (A + B)\ \lambda u : (A \to C)\ \lambda v : B \to C\ (x\ C\ u\ v)$$

$$\emptyset = \Pi C : Type\ C$$
$$R = \lambda C : Type\ \lambda u : \emptyset\ (u\ C)$$

$$\Sigma_T = \lambda P : T \to Type\ \Pi C : Type\ ((\Pi x : T\ ((P\ x) \to C)) \to C)$$
$$<, >'_T = \lambda P : T \to Type\ \lambda t : T\ \lambda u : (P\ t)\ \lambda C : Type\ \lambda v : (\Pi x : T\ (Px) \to C)\ (v\ t\ u)$$

En revanche, il est impossible de d'exprimer est la seconde projection dans le cas des sommes dépendantes, mais on peut coder la règle d'élimination du connecteur existentiel ainsi

$$ex\_elim_T = \lambda P : T \to Type\ \lambda C : Type\ \lambda u : (\Sigma_T P)\ \lambda v : (\Pi x : T\ (Px) \to C)\ (u\ C\ v)$$

On en déduit le résultat suivant.

Proposition L'élimination des coupures termine en théorie des types intuitionniste.

L'égalité

Dans le $\lambda\Pi$-calcul (λ-calcul avec types dépendants) pour ajouter l'égalité, on doit ajouter des règles de réduction pour les symboles $refl$ et L. Mais nous avons vu au paragraphe 5.3.9 que l'égalité pouvaient être définie en théorie des types minimale. On définit les termes

$$=_T = \lambda a : T\ \lambda b : T\ \Pi P : T \to Type\ ((P\ a) \to (P\ b))$$
$$refl_T = \lambda a : T\ \lambda P : T \to Type\ \lambda x : (P\ a)\ x$$
$$L_T = \lambda a : T\ \lambda b : T\ \lambda e : (a =_T b)\ \lambda P : T \to Type\ \lambda x : (P\ a)\ (e\ P\ x)$$

Ici encore, l'élimination des coupures d'égalité se ramène à de la β-réduction. Car

$$(L_T\ a\ a\ (refl_T\ a)\ P\ x) = ((\lambda P : T \to Type\ \lambda x : (P\ a)\ x)\ P\ x)$$

et ce terme se β-réduit sur x.

On en déduit le résultat suivant.

Proposition L'élimination des coupures termine en théorie des types intuitionniste avec égalité.

Les entiers

Dans le Calcul des constructions, si nat est le type des entiers de Church $\Pi X : Type\ (X \to (X \to X) \to X)$, on peut exprimer l'axiome de récurrence

$$\Pi P : nat \to Prop\ ((P\ 0) \to (\Pi n : nat\ (P\ n) \to (P\ (S\ n))) \to (\Pi n : nat\ (P\ n)))$$

Il n'est malheureusement pas possible de trouver un terme clos de ce type. En effet, si on a une preuve a de $(P\ 0)$ et une preuve g de $\Pi n : nat\ (P\ n) \to (P\ (S\ n))$, il n'est pas possible de construire la preuve de $(P\ n)$, $(g\ n\ (g\ ...\ (g\ 0\ a)))$ en itérant la construction qui permet de passer de $(P\ n)$ à $(P\ (S\ n))$ avec un entier de Church, car cette construction a un type différent à chaque itération (contrairement à ce qui se passe quand on itère, par exemple, la multiplication par 2 pour construire la fonction $n \mapsto 2^n$).

Mais, on a vu au paragraphe 5.3.9 que les entiers pouvaient se définir dans la théorie des types simples. On pose un type ι, des symboles $0 : \iota, S : \iota \to \iota$ puis on définit l'ensemble des entiers comme le plus petit ensemble qui contient 0 et qui est clos par S

$$N = \lambda n\ (\forall X\ (X\ 0) \Rightarrow \forall x\ ((X\ x) \Rightarrow (X\ (S\ x))) \Rightarrow (X\ n))$$

La preuve de $(N\ k)$ est le terme $\lambda X\ \lambda w\ \lambda f\ (f\ (k-1)\ (f\ 1\ (f\ 0\ w)))$ c'est, à peu de choses près, l'entier de Church k, mais son type est $(N\ k)$ c'est-à-dire

$$\forall X\ (X\ 0) \Rightarrow \forall x\ ((X\ x) \Rightarrow (X\ (S\ x))) \Rightarrow (X\ k)$$

et non

$$\forall X\ (X \Rightarrow (X \Rightarrow X) \Rightarrow X)$$

On peut alors démontrer le principe de récurrence,

$$\forall E\ ((E\ 0) \Rightarrow (\forall x\ ((E\ x) \Rightarrow (E\ (S\ x)))) \Rightarrow \forall n\ ((N\ n) \Rightarrow (E\ n)))$$

En suivant la preuve du paragraphe 5.3.9, on construit le terme-preuve

$$\lambda E\ \lambda w\ \lambda f\ \lambda n\ \lambda p\ (p\ E\ w\ f)$$

On peut remarquer que cette preuve consiste à démontrer $(E\ n)$ en itérerant n fois la preuve de $\forall x\ ((E\ x) \Rightarrow (E\ (S\ x)))$ en utilisant la preuve de $(N\ n)$ (l'entier de Church n) comme itérateur. Cet itérateur a, maintenant, le bon type.

Ici encore les coupures de récurrences se ramènent à de la β-réduction. Si a est une démonstration de $(P\ 0)$, g une démonstration de $\forall x\ ((P\ x) \Rightarrow (P\ (S\ x)))$, la démonstration de $\forall n\ (P\ n)$ est

$$(\lambda E\ \lambda w\ \lambda f\ \lambda n\ \lambda p\ (p\ E\ w\ f)\ P\ a\ g)$$

Ce terme se β-reduit sur

$$\lambda n\ \lambda p\ (p\ P\ a\ g)$$

Si on en déduit $(P\ k)$ en appliquant cette preuve au terme k et à la preuve de $(N\ k)$ on obtient le terme $(\lambda n\ \lambda p\ (p\ P\ a\ f)\ k\ \lambda X\ \lambda w\ \lambda f\ (f\ (k-1)\ ...\ (f\ 1\ (f\ 0\ w))))$ qui se β-réduit sur $(\lambda X\ \lambda w\ \lambda f\ (f\ (k-1)\ ...\ (f\ 1\ (f\ 0\ w)))\ P\ a\ f)$ et enfin sur $(f\ (k-1)\ ...\ (f\ 1\ (f\ 0\ a)))$ qui est la preuve directe.

8.6 Les démonstrations d'existence et les notation pour les fonctions

Proposition Une fonction f a une démonstration d'existence dans le Calcul des constructions si et seulement si elle est exprimable dans le Calcul des constructions.
Démonstration Soit une fonction f représentée par une proposition P. Soit une démonstration t normale de $\forall x : nat\ \exists y : nat\ P$. Le terme $F = \lambda x : nat\ (\pi_1\ (t\ x))$ exprime la fonction f. Réciproquement, soit f une fonction calculable exprimée par un terme F dans le calcul des constructions. La proposition $y = (F\ x)$ représente la fonction f et on peut démontrer la proposition $\forall x\ \exists y\ (y = (F\ x))$.

Remarque Une preuve de $\exists x\ ((N\ x) \wedge (P\ x))$ est un triplet qui contient le témoin k sous forme d'un entier et Peano, ce même témoin sous forme d'un entier de Church (la preuve de $(N\ k)$) et la preuve de $(P\ k)$. L'entier de Church est utilisé comme itérateur dans la construction des démonstrations, l'entier de Peano pour afficher le résultat.

Par ailleurs, en éliminant les types dépendants, le type $(N\ k)$ c'est-à-dire

$$\forall X\ (X\ 0) \Rightarrow \forall x\ ((X\ x) \Rightarrow (X\ (S\ x))) \Rightarrow (X\ k)$$

devient le type

$$\forall X\ (X \Rightarrow (X \Rightarrow X) \Rightarrow X)$$

des entiers de Church.

On montre ainsi que les fonctions exprimables dans le Calcul des constructions et dans le système F_ω sont les mêmes, on en déduit que les ensembles suivants sont sont égaux :

- les fonctions dont l'existence peut être montrée dans le Calcul des constructions,
- les fonctions exprimables dans le Calcul des constructions,
- les fonctions exprimables dans le système F_w

À ces ensembles on peut ajouter

- les fonctions dont l'existence peut être montrée en arithmétique intuitionniste d'ordre supérieur.

En effet si l'existence d'une fonction peut être montrée en arithmétique intuitionniste d'ordre supérieur, cette preuve peut être traduite dans le Calcul des constructions. Réciproquement, on montre que si l'existence d'une fonction est démontrable en arithmétique intuitionniste d'ordre supérieur, elle l'est également dans le Calcul des constructions.

Remarque Comme au chapitre précédent, il est possible d'optimiser le terme exprimant une fonction en éliminant les parties de la démonstration inutiles au calcul.

Remarque Si la fonction prédécesseur peut se programmer dans le Calcul des constructions (ou dans le système F_ω), le nombre de β-reductions nécessaire pour calculer le prédécesseur de n est proportionnel au nombre n. En revanche, il est constant dans la théorie des types de Martin-Löf. Cela motive une extension Calcul des constructions en ajoutant un type inductif primitif pour les entiers. On aboutit ainsi à un nouveau calcul : le *Calcul des constructions Inductives* qui est une synthèse entre la théorie des types de Martin-Löf et le Calcul des constructions [7, 9].

Bibliographie

[1] H. Barendregt, Lambda calculi with types, Handbook of Logic in Computer Science, S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (eds.) Clarendon Press, Oxford (1992).

[2] Th. Coquand, Une théorie des constructions, *Thèse de troisième cycle*, Université Paris 7 (1985).

[3] T. Coquand, G. Huet, The calculus of constructions, *Information and Computation*, 76 (1988) pp. 95-120.

[4] J. Gallier, On Girard's candidats de réductibilité, *Logic and Computer Science*, P. Odifreddi (Ed.), Academic Press, London (1990) pp. 123-203.

[5] J.Y. Girard, Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur, *Thèse de Doctorat d'État*, Université de Paris 7 (1972).

[6] J.Y. Girard, Y. Lafont, P. Taylor, Proofs and types, *Cambridge Universtity Press* (1989).

[7] Ch. Paulin-Mohring, Inductive definitions in the system COQ, Rules and Properties, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 664 (1993) pp. 328-345.

[8] J.C. Reynolds, Towards a theory of type structure, *Paris colloquium on programming*, Lecture Notes on Computer Science 19, Springer Verlag (1974).

[9] B. Werner, Une théorie des constructions inductives, *Thèse de Doctorat*, Université de Paris 7 (1994).

Table des matières