ECOLE NORMALE SUPERIEURE

De la logique à informatique

Giuseppe LONGO

LIENS - 98 - 18

Département de Mathématiques et Informatique

CNRS URA 1327

De la logique à informatique

Giuseppe LONGO

LIENS - 98 - 18

Décembre 1998

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 01 44 32 30 00

Adresse électronique : longo@dmi.ens.fr

De la Logique à Informatique

Giuseppe Longo
http://www.dmi.ens.fr/users/longo
CNRS et Dépt. de Mathématiques et Informatique
Ecole Normale Supérieure, Paris

De la rigueur mathématique aux langages pour les machines, Conférence invitée au Colloque "Voir, Entendre, Raisonner, Calculer", SPI et SPM, CNRS, La Villette, Juin, 1997. Appendice: **Démonstrations et Programmes**.

De la rigueur mathématique aux langages pour les machines

Giuseppe Longo
CNRS et Dépt. de Mathématiques et Informatiques
École Normale Supérieure
45, Rue d'Ulm, 75005 Paris

1. La Rationalité et la déduction

1.1 Voir et entendre: du regard dans l'espace aux lois de la pensée

«Par *raisonnement j'entend calcul* ... tout raisonnement se base sur ces deux opérations de l'esprit, la somme et la soustraction» [**Hobbes**, Computation sive Logica *dans* De Corpore, 1655]. Pour le plus illustre des nominaliste anglais, la connaissance est dans les noms et dans la "manipulation arithmétique" des concepts. On y reconnaîtrait presque les teneurs de l'hypothèse de l'Intelligence Artificielle forte de notre époque. Essayons, toutefois, d'arriver graduellement à cette histoire du "raisonnement comme calcul".

Chez Hobbes, la fracture par rapport à la rationalité du "voir" grecque est évidente: "théorème", le coeur du raisonnement, mathématique en particulier, veut dire "voir", "spectacle", en grecque. **Saint Augustin**, en effet, prétend que nous "voyons" les vérités éternelles des mathématiques dans la mémoire, où Dieu les a placées à coté de la connaissance de Lui même [Confessions, 401]. L'analyse de l'entendement de **Descartes** non plus néglige la géométrie et le "voir pour entendre": l'intuition de l'évidence, présente à chaque pas de la déduction, est aussi bien à la base du raisonnement que la certitude du mécanisme algébrique de l'inférence.

Mais quel essors pouvait-on s'attendre du nominalisme de Hobbes, qui paraît exclure toute entendement directe, par vision, s'il n'y avait pas eu un problème crucial: la rigueur des nouvelle mathématiques du XVIII siècle et, plus tard, la crise des géométrie non-euclidienne?

En fait, le XVIII siècle est l'époque de la naissance du calcul infinitésimal. La certitude du fini est perdue, le regard dans l'espace des grecques ne suffit plus: il faut une nouvelle rigueur pour trouver des certitudes face à l'explosion conceptuelle des mathématiques, face à la manipulation de l'infini en acte, des limites, du continu. Cette *exigence de rigueur* va de pair avec la croissance du corpus mathématique. Pour **Leibniz** les mathématiques sont d'abord un "ars demonstrandi" basée sur la Logique et un langage pour la logique. Il faut

les exprimer, ainsi que le raisonnement en général, dans une *Lingua Characteristica* (une "idéographie logique") [De Arte Combinatoria, 1666], forme générale de la pensée, mathématisée si possible, *mathesis* universelle. Voila le projet de l'entendement sans vision qui commence à prendre forme, à proposer, petit à petit, une nouvelle rigueur formelle/linguistique.

Mais la rigueur va de pair aussi aux théories mécanicistes et, en fait, aux mécanismes fantastiques de l'époque. Les horloges, les automates de Vaucanson et maints autres sont des machines merveilleuses. Ils sont aussi mécaniques que le raisonnement certain, celui qui se déroule pas à pas, calcul algébrique de symboles. Le clavecin est comme le logicien, le logicien comme le clavecin, pour **D'Alembert** et **Diderot**.

Toutefois la "pratique" des mathématiques restera encore confuse, au tournant du siècle: l'infini, les limites, le continu sont encore flous. Beaucoup d'invention, mais aussi de désaccords et d'incertitudes: **Cauchy**, malgré son génie mathématique, donne une preuve du Théorème des Valeurs Intermédiaires qui n'en est pas une. Son théorème, sur la continuité de la somme d'une série de fonctions *uniformément* continues, n'explicite pas l'hypothèse de l'uniforme continuité. **Poinsot** croit avoir démontré que toute fonction continue est différentiable. La richesse grandissante des mathématiques fait ultérieurement croître l'exigence de rigueur, mais on continue à faire confiance au regard dans l'espace, sur le plan, sur le feuil: à l'intuition du "voir". **Laplace**, **Gauss** et maints autres "pensent" encore les mathématiques dans l'espace. Le fondement de l'algèbre est dans la géométrie: on interprète même les nombres complexes, dont l'existence est une conséquence de la manipulation algébrique, sur le plan cartésien d'Argand-Gauss. C'est encore là qu'on les comprend, qu'ils trouvent leur "fondement". Mais comment faire pour manier avec certitude les concepts difficiles d'infini en acte, désormais omniprésent en mathématiques, et de continu? Il faudra attendre **Weierstrass**, **Cantor** et **Dedekind** pour arriver à *re*-construire le continu, son infinité, ses opérations de limite, à partir du discret, de façon algébrique, synthèse formidable entre le continu du mouvement dans l'espace et le discret linguistique des nombres entiers.

Il y toutefois une exception à cette vision qui garde, malgré les horloges et la certitude du raisonnement linguistique, toute primauté à la géométrie: les algébristes anglais. Pour **Woodhouse** [1801] la certitude est dans la manipulation "mécanique" des symboles algébriques. L'algèbre ne se "fonde" pas sur la géométrie, mais sur la manipulation de symboles linguistiques (on se souviendra de Hobbes). Voila donc, dans cette école, **Boole** et son algèbre de la Logique [Les Lois de la Pensée, 1854]. Et **Babbage**, un algébriste aussi, avec ses machines à calculer. L'algèbre est dans la machine, dans les engrenages mus

par la puissance de la vapeur. Des vraies machines à penser. Encore des "horloges", mais qui permettent de résoudre des équations différentielles (presque par des méthodes "aux différences fines"). Plus tard, Power ajoutera une certaine flexibilité à ce hardware rigide: des fiches en carton, remplaçables, permettent d'utiliser les mêmes engrenages pour des tâches différentes.

1.2 La crise de la Géométrie.

Le continent toutefois ne pourra pas continuer à faire référence à l'espace pour fonder ses certitudes mathématiques. **Lobacevskij**, **Bolyai** e **Riemann**, entre 1835 et 50, confirment que l'espace n'est pas si certain, canonique, unique: il n'y a pas une "intuition pure de l'espace en soi", en tant que variété euclidienne. Ils démontreront le bien fondé des deux négations possibles du cinquième axiome d'Euclide et proposerons les géométries non-euclidiennes.

Voila donc le langage, bien au-delà des intuitions de Leibniz, se proposer comme le seul fondement possible de la certitude. Pourvu que il soit basé sur une logique, sur La Logique. **Frege** fonde les mathématiques sur un calcul logique extrêmement puissant. Vrai percé mathématique, il décrit avec rigueur l'emploi des variables en mathématiques [Idéographie, 1879: Fond. de l'Arithmétique, 1884]: il formalise la *quantification*, les "pour tout", "il existe" utilisés si mal dans le langage commun, si souvent informellement importé dans la pratique mathématique. Les mathématiques sont "logisées". En fait, le calcul arithmétique, coeur des mathématiques, est une déduction logique. L'Arithmétique coïncide avec sa propre logique. Le *fondement* et la *signification* du calcul algébrique sont dans la logique.

Hilbert ira plus loin: la certitude du calcul est dans l'*absence de signification*. La manipulation des suites finies de symboles par des règles entièrement formalisées donne au langage mathématique la certitude et le fondement, elle exprime et garantie l'invariance et la stabilité des concepts mathématiques, elle les soustrait aux ambiguïtés sémantiques. Plus précisément certitude et fondement sont par les *preuves de cohérence* [1900]. Celles-ci doivent être données dans une mathématique qui *travaille sur* les mathématiques (les **métamathématiques**). La métathéorie est décrite dans un **métalangage** qui permet de parler du langage objet des mathématiques. Dans ce cadre on pourra démontrer aussi la *décidabilité* et la complétude des formalismes adoptés pour les mathématiques. Hilbert reprend l'hypothèse de Laplace sur la prédictibilité (ou complétude descriptive des mathématiques) à un niveau plus haut: les théories formelles des mathématiques permettent de "recouvrir complètement" et *décider* les propriétés des mathématiques, exactement comme

les mathématiques de Laplace devaient "recouvrir" ou *prévoir* (décider dans l'espace et le temps) l'évolution des structures physiques.

1.3 Raisonner et calculer: la déduction effective

Au cours des années '30 on arrive à préciser le rêve hilbertien de "déduction formelle", calcul certain puisque sans signification. **Herbrand, Gödel, Church** et autres définissent la déduction comme *fonction* calculable, une notion enfin rendue précise. On peut coder les formules du langage par des nombres: les déductions seront alors une classe définie rigoureusement de fonctions des entiers dans les entiers, des algorithmes mécanisables. Une déduction n'est qu'une fonction calculable ou récursive, lambda-calculable, elle va *des axiomes aux thèses, définie par les règles*. Vraie percé de Gödel, car une preuve (métathéorique) devient alors une fonction (théorique), coeur de la preuve diagonale d'incomplétude: en codant dans la théorie la phrase métathéorique "cette phrase n'est pas démontrable" il démontre que ... on ne peut pas la démontrer (et sa négation non plus).

Church, **Curry** (et Gödel) donnent donc une Théorie Formelle (et constructive) des fonctions, en tant que preuves. Voila l'origine de la
Programmation Fonctionnelle: une preuve qui va (des hypothèses) A à la thèse B est une fonction, $f : A \rightarrow B$.

Herbrand, à la même époque, donne un "modèle minimal", basé sur la syntaxe. La démonstration alors devient un contrôle de vérité sur des "instances", dans ce modèle. Voila l'origine (ou les bases mathématiques) de la
Programmation Logique: la preuve de "il existe x tel que P" devient une vérification constructive sur des instances: $\exists x.P \leftrightarrow \{P(a), P(f(a)), ... \}$.

Mais c'est **Turing** (1936) qui arrive à concevoir des *machines* (mathématiques, pure abstraction conceptuelle) qui manipulent des suites de symboles sans significations. Il les construit en prolongeant la distinction entre langage et métalangage et entre syntaxe et sémantique: il fait une distinction cruciale entre hardware et software (matériel/logiciel). Il utilise aussi une autre idée de la logique, la "gödélisation" (codage des programmes qui peuvent être manipulés par des programmes): sa Machine "Universelle", conçue grâce à la gödélisation, est un compilateur, dans notre terminologie. Ces idées sont à l'origine de la
Programmation Impérative: "do ... until", "go to .... ".

Voila donc un parcours possible de ce rationalisme qui est à l'origine de la science moderne. Son "objectivation" dans le langage, au cours des trois derniers siècles, jusqu'à arriver, dans

ce siècle, à la manipulation mécanique de suites finies de symboles, est à l'origine de ces machines qui ont changé notre vie. Maintenant que (cette interprétation de) la rationalité "est là", réifiée dans la machine, on peut reprendre un discours sur la rationalité humaine, dans son sens le plus vaste, là où Descartes l'avait laissé: l'analyse de l'intuition, le regard dans l'espace. Et on pourrait aller plus loin, jusqu'aux métaphores, les analogies, le rôle de l'intentionnalité, de l'affectivité dans l'intelligence ... . Mais cela est un autre discours.

2. Les Machines: les mathématiques de leurs langages

Depuis Turing, donc, les machines se décomposent en matériel et logiciel. Le premier pose des problèmes d'ingénierie et de physique qui deviendront, dans le temps, de plus en plus complexes. Toutefois, jusqu'aux problèmes contemporains, dus à la physique et l'informatique des systèmes distribués, interactifs et asynchrones, la "logique" de ces machines était toute dans la tête des logiciens des années '30: jusqu'à tout récemment, aucun calcul faisable échappait à leur caractérisation mathématique. Il n'en est plus ainsi à cause de la distribution des ordinateurs sur des grands réseaux, de leur vitesse, qui pose des problèmes à la communication limitée par la vitesse de la lumière, de l'absence d'une mesure absolue du temps (asynchronie essentielle des différents systèmes de référence). Mêmes les langages qui essayent de s'adapter à ces nouvelles donnes échappent aux paradigmes des pères fondateurs: les Langages Orientés aux Objets, par exemple, où concurrence et interaction jouent un rôle central.

Revenons toutefois aux grands systèmes hérités de la Logique, tout en restant au seul logiciel. On peut alors résumé les trois grands styles de programmation modernes, en les mettants en rapport à leur origine et fondement logique:

Herbrand (1930): la *Programmation Logique*: $\exists x.P \leftrightarrow \{P(a), P(f(a)), \ldots\}$

Church (1932): la *Programmation Fonctionnelle*: $f : A \rightarrow B$
(*dépendance fonctionnelle*: $\lambda y.f(x,y)$, i.e. $(\lambda y.f(x,y))a = f(x,a)$ ou $(\lambda y.f(y))a = f(a)$)

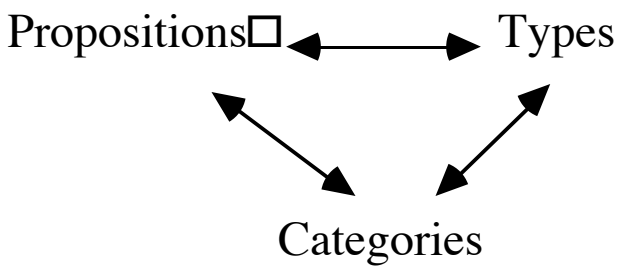
Turing (1936): *Programmation impérative*: "do ... until", "go to .... ".

On mentionnera maintenant seulement certains aspects du deuxième style et de ses développements mathématiques, car, ce style de programmation, le jeu entre structures mathématiques (Catégories), types et programmes a été particulièrement riche (voir aussi l'appendice).

2.1 Preuves comme programmes et comme morphismes

Si un programme prend comme arguments des "éléments" de type A et donne des valeurs dans B, dira qu'il est de type $A \to B$, ou qu'il calcule une fonction f de type $A \to B$. Mais ces calculs, ces programmes, sont des preuves, on l'a vu, des preuves de propositions logiques: en fait, f est une preuve qui prend toute preuve de A, où A est maintenant une proposition, et la transforme dans une preuve de B. Mais alors, f est une preuve de l'implication logique $A \to B$. Voila donc comme A, B, $(A \to B)$ sont des types, mais aussi des propositions. Et f est (le nom de) un programme, mais aussi une fonction et une preuve.

Or il y a, en mathématiques, une autre "théorie des fonctions" bien importante: la Théorie des Catégories. Elle ne demande, pour commencer, que des objets, A, B, ... et des morphismes $f : A \to B$ entre objets, qui se composent, qui contiennent l'identité. L'analogie est profonde et donne en faite des "isomorphismes" bien précis entre Types, Propositions et Objets dans le sens des catégories:



En bref, si on pose:

Notations: " |— " déduction (métalinguistique),

" → " implication (linguistique)

Formules du langage: A, B .... $A \to B$

Deux règles suffisent à nous donner une "logique" minimale et le coeur de la notion de catégorie mathématique:

Règle d'identité **Règle de composition**

(Id) A |— A (Comp) $\dfrac{A \vdash B \qquad B \vdash C}{A \vdash C}$

Alors:
Une **Catégorie** est donnée par
- identité (droite et gauche) et
- composition (associative) de morphismes

D'autre part, le **calcul des termes** (lambda-calcul), ou des **preuves** comme termes (typés), est donnée par
Termes: x, y, z .... b(c), λx.b
Notations: [b/y]c = "remplace y par b dans c"
"a : A" = "a est un *terme* de ou une *preuve* de A".

Insérons maintenant les termes dans les deux règles dessus: par cette technique on donne la structure des preuves et on défini aussi les termes "bien formés" ou **typés** du langage. Ces termes sont des programmes (fonctionnels) ainsi que des preuves:

(Id) $$x : A \vdash x : A$$

(Comp) $$\frac{x : A \vdash b : B \qquad y : B \vdash c : C}{x : A \vdash [b/y]c : C}$$

Nous avons donc des "analogies" ou isomorphismes importants:

Catégorie = Système déductif = Système fonctionnel

Mais pour avoir une logique expressive, l'identité et la composition ne suffisent pas. Deux règles permettent d'atteindre l'expressivité du calcul propositionnel (intuitionniste):

$$(\rightarrow E)\qquad \frac{\vdash a : A \qquad \vdash c : A \rightarrow B}{\vdash c(a) : B}\qquad \text{(Modus Ponens)}$$

$$(\rightarrow I)\qquad \frac{x{:}A \vdash b : B}{\vdash \lambda x{:}A.b : A \rightarrow B}$$

Voila donc des termes, des programmes, $c(a)$ et $\lambda x{:}A.b$ qui codent des preuves données par ($\rightarrow$E), Modus Ponens ou élimination de la flèche (flèche = implication), et par ($\rightarrow$I), introduction de la flèche.

L'analogie entre proposition, types et objets se poursuit, car on peut aussi démontrer que:

Conjonction logique = Produit cartésien;

Disjonction = Somme (coproduit)

2.2 Applications: extensions et variantes; synthèses et démonstrations.

Le fait de pouvoir considérer les programmes fonctionnels comme des termes et comme morphismes de catégories a des avantage bien évidents, car:

résultats sur les termes = résultats sur les preuves = résultats sur les catégories.

Par exemple, des démonstrations sur les *termes* typés permettent de prouver ce qu'on appelle, en Logique, l'élimination des "coupures", une sorte de preuve de "pureté de méthodes": on peut éliminer, dans les *preuves*, tout ce qui n'est pas explicitement utilisé dans la thèse. D'autre part, les termes inversibles caractérisent, par exemple, les *types* isomorphes, en fait les *objets* isomorphes dans les catégories.

En plus, les structures catégoriques donnent des modèles, c'est à dire des interprétation des systèmes formels qui enrichissent les langages de structures et ... d'idées. Une propriété des modèles, qui n'est pas démontrable dans le langage, peut suggérer des *extensions* du langage (e.g. les définition récursives dans le langages ML d'Edinburgh, suggérées ou permises par la sémantique sur des domaines "à la Scott"). Mais l'interprétation peut

suggérer des extensions "fortes" des systèmes de départ: on étudiant des modèles d'une extension du lambda-calcul de Church, J.Y. Girard eu l'idée d'examiner des *variantes* des systèmes logiques classique et intuitionnistes (dont la Logique Linéaire), car ces modèles contenaient des fonctions "linéaires".

Mais si les preuves sont des programmes (et viceversa), alors une **preuve** est une **synthèse de programmes**. La déduction, pas à pas, construit un programme: par exemple, dans la règle $(\rightarrow E)$, à partir des programmes c et a on "synthétise" le terme ou programme $c(a)$. Si, par contre, on part avec des termes ou des programmes sans types, une déduction leur "donne" des types. Cette dernière technique fournit un contrôle efficace, quoique partiel, de la correction d'un programme, comparable au "contrôle de dimension" en physique: si, dans le déroulement des calculs à partir d'une équation, les "dimensions physiques" (force, vitesse, accélération ...) sont préservées, alors il y a des fortes chances que les calculs soient correctes.

D'autre part, un programme développe une preuve: voila un outil essentiel à la **démonstration automatique**. Des techniques puissantes entrent alors en jeu: l'unification, les méthodes de résolution .... Toutefois, dès que l'on veut aller à peine au delà du calcul propositionnel, l'interaction homme-machine devient essentielle à la preuve. Considérez, par exemple, l'induction arithmétique:

$$(\mathrm{Ind}) \qquad \frac{A(0) \qquad \forall x.(A(x) \rightarrow A(x+1))}{\forall x.(A(x)}$$

Bien rarement, dans un théorème intéressant, on arrive à démontrer le pas inductif, c'est à dire $\forall x.(A(x) \rightarrow A(x+1))$, tel qu'il est. Il se pose alors le problème de la "charge inductive", c'est à dire, du choix d'un B plus "fort" que A et qui permet de faire le pas inductif. Il faut donc, en principe, l'intérvention de l'homme pour choisir (par "analogie" par exemple) un B tel que

$\forall x.(B(x) \rightarrow A(x))$ et, ensuite, démontrer $\forall x.(B(x) \rightarrow B(x+1))$

(voir appendice).

2.3 La récursion: programmes comme fonctions, types comme domaines[1]

Dans la définition de programmes comme fonctions on a mis en évidence la *dépendance fonctionnelle*: si $f(x,y)$ est une fonction de deux variables, $\lambda y.f(x,y)$ "mets en évidence" la dépendance de f de la deuxième variable. On peut alors "appliquer" cette fonction à un argument a et on obtient: $(\lambda y.f(x,y))a = f(x,a)$ (dans le cas d'une seule variable on pose $(\lambda x.f(x))a = f(a)$). Cet outil de définition, employé plus haut pour introduire les termes donnés par la règle $(\rightarrow E)$, est particulièrement puissant si on renonce à l'analogie "types = propositions" (et aux types tout court). C'est à dire si on se permet de considérer bien formés aussi les **termes sans types**, comme $x(x)$, une variable appliqué à elle même, un terme qui, apparemment n'a pas de sens mathématique: quel élément, quelle fonction "s'applique" à soi-même? Cet élargissement de la classe des termes "acceptables" permet de donner dans le langage les définitions récursives. Qu'est que c'est qu'une définition de ce gendre? Considérez la fonction "factoriel": $n! = 1\times2\times3 \ldots \times n = n\times(n-1)!$

Vous pouvez aussi la définir comme une fonction f telle que $f(n) = n\times f(n-1)$. Dans un certain sens, f est la solution d'une équation où elle apparaît à la gauche et à la droite de l'égalité. Ou, encore, elle est un point fixe d'un "foncteur" ou fonctionnel F tel que: $F(f) = \lambda x.x\times f(x-1)$. En fait, elle est la plus petite solution f_o de $f = F(f)$, telle que $f_o(0) = 1$.

Or, on peut donner des extensions même des langages typés qui admettent des telles définitions (les modèles en suggèrent la nature et démontrent leur cohérence, comme on disait plus haut). Mais, dans les langages sans types, on sait explicitement et directement résoudre $f = H(f)$ pour n'importe quel fonctionnel définissable H. En effet, posez $Y = \lambda y.(\lambda x.y(x(x)))(\lambda x.y(x(x)))$. Ce terme un peu compliqué est un **opérateur de point fixe** et permet de démontrer $Y(H) = H(Y(H))$; c'est à dire, Y donne, pour tout H, un point fixe $Y(H)$ de H. Cette idée fut à la base du "paradoxe de Curry", en 1932: en fait, si dans la langage on veut avoir la négation, neg, alors $Y(neg) = neg(Y(neg))$, ce qui contredit la "signification" de neg. Il suffit alors de restreindre le camp d'application de la négation (ou ne pas l'avoir du tout dans le langage) pour garder Y et avoir un puissant outil de définition, largement utilisé en programmation depuis la naissance du langage LISP (1960).

Mais attention, d'autres difficultés nous guettent. Si on peut appliquer un terme à soi même, un fonction f à soi même et poser $f(f)$ ou l'obtenir comme dans $(\lambda x.x(x))f = f(f)$, quelle est le sens mathématique de cette notion de "fonction"? Y a-t-il un univers mathématique où les fonctions peuvent s'appliquer à elles mêmes, où il n'y a pas de différence entre fonctions et arguments?

Oui, ca suffit de le construire, c'est à dire de trouver une **solution de l'équation**:

---

[1] Voir l'Appendice pour plus de détails.

$$X = X \to X \quad \text{ou} \quad X = X^X.$$

où $X \to X$ est un "domaine" de fonctions de X dans X. Ce domaine doit contenir assez de fonctions pour interpréter les calculs exprimables par les termes, donc toutes les fonctions calculables ou récursives. On sait, depuis Cantor, qu'aucun ensemble est égal, isomorphe plus précisément, à son espace de fonctions (sauf le cas inintéressant du singleton {a}). L'erreur consiste à penser en termes d'ensemble, c'est à dire en termes d'espaces non structurés. En mathématiques, ce qui compte sont les structures, algébriques, d'ordre, topologiques ..., en particulier dans ce cas. En fait, suite à la théorie de Scott des années '70, on sait construire des espaces topologiques, des ordres partiels plus précisément, tels que l'ensemble, $X \to X$, des fonctions continues sur l'un de ces espaces est isomorphe à l'espace lu-même, c'est à dire on sait résoudre l'équation $X = X \to X$, où $X \to X$ est le domaine mathématiquement structuré des fonctions topologiquement continues sur X (structuré par la topologie de la convergence par points, par exemple). Grâce à ces "domaines de calculabilité", on sait en fait résoudre une large classe d'équations, comme

$$X = A + B \times X + (X \to X)$$

où A et B sont des domaines arbitraires.

Ces topologies, ces catégories avec structure, semblent essentielles à la solution de ce gendre de problèmes. Elles sont nais en Logique et développés en profondeur en Informatique et on ne connaît pas de solution de ces équations sans qui n'utilise pas de structure topologique pour X. Le point, au contraire, est que, pour l'instant, ces structures sont plutôt "faibles": elles se basent sur des topologies sans notion de distance (sans métrique), ce qui est partiellement insatisfaisant en mathématiques. Un tournant important est en cours, qui parait indiquer des variantes (restrictions à des sous-espaces plus structurés) bien expressives. Depuis les solutions d'équations comme $X = X \to X$, la Géométrie est revenue en force pour tempéré les excès du formalisme et du logicisme de ce siècle, tout centrés sur le langage. Ces mêmes excès toutefois qui nous ont permis de concevoir des machines pour la manipulation de symboles sans significations, des pures machines logiques, des machines qui ont changé notre vie.

Autres applications

On a concentré la présentation seulement sur certains aspects de la Logique en Informatique. Des aspects toutefois particulièrement riches, du point de vue mathématique et informatique: l'analogie "types = propositions = objets de catégories" est un remarquable exemple de l'unité des mathématiques et des possibilités d'application de cette unité. Il faudrait mentionner maints autres liens entre Logique et Informatique. La **Logique Modale**, avec

ses aspects de logique temporelle, permet de traiter des problèmes pointus concernant le temps dans des systèmes concurrents (linear time, branching time ... pour la concurrence et le parallélisme). Le **mu-calcul**, qui en suit, donne des points fixes "maximaux", strictement reliés aux structures des ensembles "mal-fondés", c'est à dire des ensembles qui admettent des chaînes descendantes d'ensembles l'un inclus dans l'autre, comme certains processus infinis de calcul (les systèmes d'exploitation, par exemple). Apparaît aussi l'utilité des **Catégories Fibrées**, comme modèles, de structures algébriques style les algèbres de processus (pour les processus asynchrones). Souvent ces systèmes logiques ou catégoriques servent juste pour une unification de la pluralité des langages utilisés, ce qui n'est pas un moindre succès. Parfois on obtient des vraies percés, des extensions, des langages nouveaux et intéressants. De plus en plus des méthodes géométriques paraissent essentielles, avec leur autonomie par rapport à la logique centrée sur le langage et ses structures finitaires. Maintenant que les systèmes distribués, concurrents et asynchrones sont à l'ordre du jour, faudra-t-il jeter un coup d'oeil sur les systèmes dynamiques, les lieux du temps physique, pour les décrire et les concevoir?

QUELQUES REFERENCES

Abramsky S. et al. (eds) **Handbook of logic in computer science**, vol. 1 - 4, Oxford Claredon, 1992 - 1995.

Asperti A., Longo G. **Categories, Types and Structures**: *an introduction to Category Theory for the working computer scientist*, M.I.T. Press, 1991.

Barendregt, H. **The lambda calculus: its syntax and semantics**, Revised edition, N. H., 1984].

Constable R. L. et al. **Implementing Mathematics with the Nuprl Proof Development System**. Prentice-Hall, 1986.
`

Coquand T., Huet G. "The calculus of Constructions" **Information and Computation**, 76, 95 - 120, 1988.

Davis M., **Computability and Unsolvabilty**, MacGraw-Hill, 1958

Davis M., “Mathematical Logic and the Origin of Modern Computing” and “Influences of Mathematical Logic in Computer Science” *in* [Herken,1995].

Dowek G., **Démonstration automatique dans le Calcul des Constructions**, Thè se, Univ. Paris VII, 1991.

Graubard S. (ed.), **The Artificial Intelligence Debate**, M.I.T. Press, 1989.

Girard J.-Y. "Linear Logic" Theoretical Comp. Sci., 50 (1-102), 1987.

Girard J.Y., Lafont Y., Taylor R. [1989] **Proofs and Types,** Cambridge University Press.

Hasslacher L., “Beyond the Turing Machine” *in* [Herken,1995].

Herken R., **The Universal Turing Machine**, Springer-Verlag, 1995.

Lakatos I., **Proofs and Refutations**, Cambridge U. Press, 1976

Lambek J., Scott P.J. [1986] **Introduction to higher order Categorical Logic**, Cambridge University Press.

Longo G., "Set-Theoretical Models of Lambda-Calculus: Theories, Expansions, Isomorphisms", **Annals Pure Applied Logic**, 24, 1983.

Longo G. "Some aspects of impredicativity: notes on Weyl's philosophy of Mathematics and on todays Type Theory" **Logic Colloquium 87**, Studies in Logic (Ebbinghaus et al. eds), North-Holland, 1989 (downloadable from htpp://www.ens.fr/users/longo).

Longo G., "The mathematical continuum, from intuition to logic" **New Trends in Phenomenology**, (J. Petitot et al., eds) Stanford U.P., 1998 (to appear: downloadable from htpp://www.ens.fr/users/longo).

Longo G., Milsted K. and Soloviev S., "The genericity theorem and the notion of parametricity in the polymorphic Lambda-calculus" **Theor. Com. Sci.** vol. 121, 1993 (downloadable from htpp://www.ens.fr/ users/longo).

Longo G., Moggi E., "A category-theoretic characterization of functional completeness" **Theor. Com. Sci.** vol. 70, 2, 1990.

Longo G., Moggi E., “Constructive Natural Deduction and its omega-Set Interpretation” **Mathematical Structures in Computer Science**,vol.1, n.2, 1991 (downloadable from htpp://www.ens.fr/users/longo)

Prawitz D, **Natural Deduction**, Almquist&Wiksell, 1965.

McCorduck P., **The Universal Machine: Confessions of a Technological Optimist**, McGraw Hill, 1986.

Makowsky Y.A., “Mental Images and the Architecture of Concepts” *in* [Herken, 1995].

Mitchell J.C. **Introduction to Programming Language Theory**, M.I.T. Press, 1993.

Rogers H., **Theory of Recursive Functions and Effective Computability**, 1967.

Scott D. [1972] "Continuous lattices" **Toposes, algebraic Geometry and Logic**, (Lavwere ed.), SLNM 274, (pp.97-136) Springer-Verlag.

Scott D. [1976] "Data types as lattices," **SIAM Journal of Computing**, 5 (pp. 522-587).

Turing A, “Computing Machines and Intelligence”, **Mind**, LIX, 1950.

van Heijenoort, J. **From Frege to Godel**, Harvard University Press,Cambridge, Massachusetts, 1967

Appendice: démonstrations et programmes

1. Introduction.

Pour que des machines puissent faire des mathématiques, il faut avant toute chose un langage qui décrive, les mathématiques en termes adéquats pour les machines. Ce langage doit être totalement *formalisé* et sans aucune ambiguïté sémantique. Les ordinateurs ne peuvent pas faire de choix opérationnels en fonction de la signification d'une phrase, surtout si elle s'avère incertaine ou si elle dépend du contexte, mais seulement en analysant la structure syntaxique. De plus, le langage doit être suffisamment simple et, bien qu'artificiel, compréhensible par le programmeur humain.

Il y a deux orientations principales pour "faire faire" des mathématiques aux ordinateurs. D'un côté l'étude de systèmes "calculatoires", c'est-à-dire l'étude de langages et de techniques qui aident l'algébriste ou l'analyste à faire des "comptes" lorsque ceux-ci s'avèrent trop ardus ou hors de portée d'un être humain. De l'autre, le développement de langages qui remplacent ou complètent l'activité logico-déductive du mathématicien (démonstration automatique et calcul symbolique). Dans ce bref exposé, nous présenterons la deuxième orientation, dans l'optique particulière, mais importante ou du moins paradigmatique, des calculs logico-fonctionnels qui se regroupent sous le nom de "lambda-calcul". Nous verrons surtout l'aspect linguistique ou le problème de l'élaboration d'un langage pour faire des déductions et manipuler des symboles, et plus généralement pour programmer de façon rigoureuse, modulaire et traduisible. En effet, la conception de langages avancés et de programmes fiables est l'objectif principal du "rapport" que nous allons décrire, entre machines et démonstrations mathématiques, et c'est aussi le motif de cette exposé.

Observons pour commencer, que si aujourd'hui, l'existence des machines encourage le développement de la démonstration automatique et du calcul symbolique, on peut dire aussi que les idées de base pour la conception des ordinateurs modernes ont trouvé leur origine dans l'étude de la démonstration, en tant qu'activité humaine abstraite. La notion de calcul effectif et les langages dont nous parlerons datent des années 20 et 30, donc bien avant les ordinateurs numériques et faisaient partie intégrante de la théorie de la démonstration hilbertienne, c'est-à-dire du

projet de formaliser la déduction mathématique. Le but était de donner une rigueur ou un fondement logico-formel à l'activité mathématique. Malgré les échecs du projet initial, les idées des logiciens et mathématiciens comme Turing, Kleene, Church, Gödel, Herbrand... ont fourni les bases de l'informatique. Développées par Von Neumann, les machines de Turing ont constitué le paradigme des premiers ordinateurs et donc, des langages dits de programmation impérative (basés sur des ordres tels que "do", "go to"...). Les calculs dont nous parlerons ont surtout eu une influence sur la programmation fonctionnelle et la programmation logique. Il s'agit de styles de programmation récents qui se distinguent du style impératif et se rattachent encore plus directement aux développements de la logique mathématique de l'entre deux guerres.

La logique mathématique, souvent appelée métamathématique a pour *objet l'étude* des mathématiques, ses langages et ses méthodes déductives, comme la géométrie, pour prendre l'exemple d'une discipline proprement mathématique, a pour objet l'étude de figures et de structures de l'espace. On peut donc imaginer une stratification à trois niveaux : les structures géométrico-algébriques, les théories mathématiques qui les étudient (algèbre, géométrie...) et, enfin, les métathéories qui traitent des théories mathématiques. En d'autres termes, du point de vue des calculs et des langages, l'algèbre linéaire et la géométrie analytique étudient des expressions qui représentent des lignes du plan ou des surfaces de l'espace ; le lambda-calcul, en tant que langage de la Théorie de la Démonstration, manipule des termes ou des expressions qui représentent des démonstrations formelles. En fait, les expressions de ce langage codifient des démonstrations mathématiques abstraites et, donc, les calculs effectués sur celles-ci correspondent à des opérations formelles effectuées sur des démonstrations, plutôt que sur des lignes ou des surfaces. Le fait que le lambda-calcul soit programmable, et qu'il soit en fait un langage de programmation paradigmatique, permettra de décrire le passage de la théorie de la démonstration, en tant que théorie abstraite de logique mathématique, à la démonstration automatique et au calcul symbolique, comme méthode mathématique pour ordinateurs.

Bien que l'on parle ailleurs dans cette étude de la théorie de la démonstration, nous allons souligner ici l'approche "constructiviste" afin

de présenter les "démonstrations en tant que lambda-termes" et d'étudier une version informatique de la prouvabilité. En utilisant le lambda-calcul, nous mentionnerons le rôle de la théorie des catégories dans la *sémantique mathématique* de la déduction et des langages formels et de programmation (voir ci-dessous). En effet, les résultats qui font de pont entre les différents secteurs des mathématiques et de l'informatique placent le lambda-calcul et la logique combinatoire, système équivalent, au point de rencontre de vastes secteurs de la Logique et de ses applications, donnant à ces théories une importance qui va au delà de leurs origines en tant que système pour la calculabilité ou la prouvabilité effective. Comme nous le disions, en effet, les deux théories, qui ont comme base commune le "calcul algébrique sans variables" de Shoenfinkel du début des années 20 et que l'on doit à Church et Curry (1928-1936), se proposent essentiellement de "formaliser" la notion de fonction calculable ou processus, et de démonstration effective, aussi bien pour en donner une définition mathématique que pour fonder les mathématiques sur les "unshakable certainties" de systèmes symboliques minimaux. L'équivalence que l'on doit à Church, Turing et Kleene, avec les autres systèmes de calculabilité (fonctions récursives et machines de Turing, en particulier, présentées à d'autres rubriques de cette encyclopédie), donnait une généralité complète, déjà dans les années 30, à ces systèmes en tant qu' instruments pour le calcul. A cette époque, ce fut justement le lambda-calcul qui joua un rôle central dans la démonstration des théorèmes d'équivalence : on prouva en fait que les fonctions calculables au sens de Turing, sont exactement celles qui sont définissables en lambda-calcul et que celles-ci, à leur tour, coïncident avec ces fonctions partielles récursives (voir Barendregt [1984]). La coïncidence entre les pouvoirs expressifs de ces divers systèmes formels suggéra à Turing et Church une hypothèse de travail, connue sous le nom de *Thèse de Church* : toutes les fonctions intuitivement calculables de façon finitaire (nombre fini d'instructions, nombre fini de pas de calcul...) sont représentables dans un des systèmes mentionnés et donc, grâce à l'équivalence, dans chacun d'eux.

De même aujourd'hui , le lambda-calcul joue un rôle central en devenant un *médium* important, comme nous essayerons de le préciser, dans les applications de la théorie de la démonstration et de la théorie des catégories à l'informatique, et aussi un langage pour la

démonstration automatique, surtout grâce à son extension récente, le Calcul des Constructions (voir section 8), dont cette présentation s'inspire largement. En général, les méthodes automatiques de démonstration permettent de déduire des théorèmes dans un système logique, et de synthétiser les démonstrations et les théorèmes. Dans cette étude, nous omettrons un aspect fondamental : les méthodes dites de *résolution* et *d'unification*. La difficulté technique de ces méthodes rend leur présentation synthétique ardue; de plus, un coup d'oeil sur les autres aspects de l'élaboration automatique (le calcul symbolique, la programmation fonctionnelle et sa sémantique mathématique), plus que l'approfondissement de techniques spécifiques, permet de mieux cerner le transfert de certaines tâches mécaniques de l'homme à la machine, en les traitant du point de vue des différentes formes de connaissance mathématique.

Aux chapitres 2 et 3, nous présenterons les "types en tant que propositions", c'est-à-dire que l'on étudiera un calcul logique très simple, dont le système des preuves est un calcul de termes, les termes du lambda-calcul. Le but de cette présentation n'est pas seulement de donner une unité à des thèmes apparentés, mais également de produire une "sémantique" simple, à la fois pour les formules logiques et pour les types. En fait, d'une part, la signification logique des types est certainement riche en informations, en particulier pour le lecteur familier avec le Calcul propositionnel. D'autre part, une interprétation des propositions et preuves en tant que types et termes (§.4) peut être de grande utilité pour le programmeur, qui a l'habitude des langages fonctionnels mais ignore la logique. En d'autres termes, nous soulignerons que la traduction entre des formalismes divers est une "sémantique" en soi. Toutefois, au chapitre 5, nous attaquerons le point plus complexe d'une vraie "sémantique mathématique" d'un langage de programmation, même si c'est dans un cadre plus restreint, tel que le lambda-calcul. Par sémantique mathématique, on entend en fait quelque chose de plus qu'une traduction d'un langage ou d'un système formel dans un autre. Il faut dire que, l'étude de la sémantique des langages de programmation a eu un grand essor, justement grâce au lambda-calcul, à partir des travaux de D.S. Scott dans les années 70. En résumé, un calcul formel des signes prend une signification mathématique quand des instruments mathématiques essentiellement autres, par exemple des structures

géométriques ou algébriques familières, conçues dans un but, et avec des techniques tout à fait indépendants du calcul donné, fournissent une interprétation ou traduction, dans ces structures, des termes et des opérations formelles. En général, plus les structures mathématiques proposées pour l'interprétation sont "culturellement lointaines" du formalisme en question, plus le sens fourni est riche d'informations, parce qu'il établit des ponts inattendus, et requiert des théories unifiantes.

De plus, et ceci est une expérience concrète en informatique, une signification mathématique novatrice peut suggérer des extensions ou des variantes du langage en question, inspirés par les constructions présentes dans les modèles et non dérivables dans le formalisme donné.

Aux chapitres 6 et 7 nous étudierons le polymorphisme et sa sémantique, c'est-à-dire la possibilité pour un lambda-terme ou un programme fonctionnel d'avoir de nombreux *types* ou de démontrer des *schémas* de propositions. Le polymorphisme est une forme paradigmatique de modularité en programmation, directement dérivée des logiques d'ordre supérieur.

Le chapitre 8 sera consacré aux méthodes générales et aux limites de la démonstration Automatique. La conclusion, chapitre 9, sera l'occasion d'une réflexion méthodologique. Le lecteur qui désire seulement réfléchir sur la thèse "philosophique" de cet article peut passer directement aux chapitres 8 et 9. La motivation réelle des considérations que l'on y trouve, réside toutefois dans les notions et dans les résultats techniques présentés aux paragraphes précédents.

2. Déduction Naturelle et termes.

L'idée de base des systèmes de déduction naturelle, auxquels nous nous référons, est la formalisation de la notion de *dérivation logique*, entendue comme abstraction de la déduction mathématique. Le pas déductif minimum est donné par l'application d'*une règle d'inférence* qui décrit la déduction d'une conséquence, disons C, à partir de prémisses données, par exemple A_1, A_2....A_n :

$$\frac{A_1 \quad A_2 \qquad A_n}{C}$$

Les règles peuvent être composées verticalement, c'est-à-dire, étant donné les règles

$$\frac{A \quad B}{D} \qquad \frac{C}{E} \qquad \frac{D \quad E}{C}$$

il est licite de les composer dans une déduction (ou *arbre déductif*) de E sous les hypothèses A, B, C, de la façon suivante :

$$\cfrac{\cfrac{A \quad B}{D} \qquad \cfrac{C}{E}}{\cfrac{C}{E}}$$

Des points de suspension verticaux sous-entendent une déduction formée de la composition verticale de plusieurs règles :

$$\cfrac{\begin{array}{c}A_1 \quad A_2 \qquad A_n \\ \vdots \\ B\end{array}}{C}$$

Dans ce cas on peut utiliser une notion fondamentale, celle *de déchargement* On déchargera l'hypothèse A de laquelle dérive B, si une telle dérivation est une prémisse dans la déduction de la formule $A \rightarrow B$. En fait, la vérité de $A \rightarrow B$ ne dépend pas de celle de A : on observe que $A \rightarrow B$ est vraie, même lorsque A est faux (ex falso quodlibet). Supposons par exemple avoir déduit que s'il pleut, alors le temps est humide. Dans n'importe quel langage formel avec implication "$\rightarrow$" , cette déduction métalinguistique aura pour conséquence formelle "il pleut $\rightarrow$ le temps est humide". Or, une telle implication est vraie, même s'il ne pleut pas. On peut donc omettre ou décharger l'hypothèse "il pleut" dans la déduction

de "il pleut → le temps est humide", étant donné que l'implication formelle subsiste dans tous les cas, indépendamment de l'hypothèse, et peut être affirmée en toute vérité, même en plein soleil.

Le système intuitionniste minimum a pour formules, les formules atomiques, A, B, C..., et les implications entre formules, (A→B), et rien d'autre. Celui-ci est en outre basé, en termes de déduction naturelle, seulement sur deux règles d'inférence : la "flèche" introduction, (→I), où [A] indique que A est déchargée, et la flèche élimination, (→E) :

Règle d'introduction **Règle d'élimination**

$$(\rightarrow I)\ \frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A \rightarrow B} \qquad (\rightarrow E)\ \frac{A \qquad A \rightarrow B}{B}$$

Le lecteur reconnaîtra dans (→E) le classique "modus ponens" : si A et A implique B, alors B. Dans (→I), A est déchargée, dans le sens que nous citions ci-dessus, c'est-à-dire que ce n'est pas une hypothèse nécessaire à la validité de (A → B). La règle (→I) transfère dans le langage des formules la déduction métalinguistique A |-- B, c'est-à-dire affirme que de la déduction de B à partir de A, je peux déduire la formule (A → B).

Une *démonstration* est un arbre fait d'applications successives de règles d'inférence. La racine, qui est en bas, est le théorème démontré. Ce qui nous intéressera plus particulièrement ce sont les métathéorèmes, c'est-à-dire les propriétés du calcul déductif ou, plus exactement, du calcul des termes associés aux théorèmes.

La signification constructive de ce système minimal, basé seulement sur l'implication, est mise en évidence par ce que l'on appelle *l'interprétation intentionnelle* de Heyting-Kleene : une démonstration de (A → B) est une procédure de calcul qui transforme toute démonstration de A en une démonstration de B. Nous verrons ensuite que les termes du λ–calcul (λ–termes) formalisent cette interprétation, fournissant explicitement un calcul de preuves. En fait, c : C signifiera que le λ–terme c est (le code d') une démonstration effective de la formule C.

Construisons donc un langage pour les preuves. Autrement dit, définissons les λ–termes. En premier lieu, les variables x, y,... sont des termes, et x : A signifie que x est une démonstration arbitraire de A et que celle-ci peut être utilisée dans une hypothèse éventuellement déchargée. Supposons ensuite, que d'une preuve arbitraire x de A, c'est-à-dire x : A, on déduise une preuve b de B, c'est-à-dire, b : B (lire: b démontre B). Alors, la règle (→I) donne A → B: dans notre calcul, on notera λx:A.b le terme qui démontre A → B, c'est-à-dire (λx:A.b) : A → B. Si par contre c : A → B et a : A, nous écrirons (c a) pour le terme qui dénote l'application de la preuve c de A → B à la preuve a de A; celle-ci, comme nous l'avons dit plus haut , est une preuve de B, donc c a : B.

Les règles d'inférence définissent donc les λ–termes comme étant des variables, x, y..., des *λ–abstraction* (λx:A.b) d'un terme b par rapport à une variable quelconque x, et des *applications* (c a) d'un terme c à un terme a. Nous omettrons les parenthèses en l'absence d'ambiguïté. Nous pouvons alors réécrire les règles d'introduction et d'élimination comme suit

$$(\rightarrow I)\ \frac{\begin{array}{c}[x : A]\\ \vdots\\ b : B\end{array}}{\lambda x{:}A.b : A \rightarrow B} \qquad (\rightarrow E)\ \frac{a : A \qquad c : A \rightarrow B}{c\ a : B}$$

Les règles explicitent ou donnent un nom aux transformations qui font passer, par exemple, d'une démonstration c ≡ λx:A.b de A → B à la démonstration (λx:A.b)a de B, pour a : A, grâce à (→ E). On observe que "λx:A" est une opération d'*abstraction* qui *lie* dans λx:A.b la variable x, éventuellement libre dans b, c'est-à-dire qu'elle peut apparaître sans être déjà liée dans b. En fait, λx:... correspond à {x | ... } dans la théorie des ensembles ou à l'intégrale ∫...dx en analyse : la signification ou la valeur du terme, de l'ensemble ou de l'intégrale, ne dépend pas du nom de la variable liée, donc {x | P(x)} est totalement équivalent à {y | P(y)}, ∫ f(x)dx à ∫ f(y)dy, comme λx:A.b est identique à λy:A.b', pourvu que b' soit obtenu à partir de b *en substituant correctement* y à x (nous écrirons : b' ≡ [y/x]b, et nous dirons également que x est *renommée* par y dans b).

Nous indiquerons par $\vdash a : A$ la prouvabilité de $a : A$ dans ce système minimal ; d'éventuelles hypothèses non effacées seront placée à gauche de "$\vdash$": par exemple, $x : A \vdash b : B$. Pour simplifier, nous pourrons omettre d'expliciter A dans le terme $(\lambda x{:}A.b) : (A \to B)$, en écrivant $\lambda x.b$. Un résultat, mentionné ci-dessous, sur la décidabilité de l'affectation d'une preuve à une proposition justifiera cette convention. On observe aussi que les variables libres dans un terme dépendent toujours d'une hypothèse non déchargée : $\lambda y.yz : (C \to D) \to D$, par exemple, s'écrira à la place de $(\lambda y{:}(C \to D).yz) : (C \to D) \to D$, sous l'hypothèse non déchargée $z : C$. Pour ne pas abuser des λ, nous convenons d'abréger $\lambda x.\lambda y.\lambda z.(...)$ en $\lambda xyz.(...)$. Le lecteur intéressé peut étudier et compléter les deux exemples qui suivent, en observant que ceux-ci développent les démonstrations de deux propositions du calcul propositionnel et, en même temps construisent les λ-termes qui en codent les preuves.

Exemples : $\vdash \lambda xyz.xz(yz) : (A \to (B \to C)) \to ((A \to B) \to (A \to C))$; $\vdash \lambda xy.x : A \to (B \to A)$.

Démonstration : les règles utilisées sont indiquées à côté de la ligne d'inférence :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[z : A] \qquad [x : A \to (B \to C)]}{xz : B \to C}\,(\to E) \qquad \dfrac{[z : A] \qquad [y : A \to B]}{yz : B}\,(\to E)}{xz(yz) : C}\,(\to E)}{\lambda z.xz(yz) : A \to C}\,(\to I)}{\lambda yz.xz(yz) : (A \to B) \to (A \to C)}\,(\to I)}{\lambda xyz.xz(yz) : (A \to (B \to C)) \to ((A \to B) \to (A \to C))}\,(\to I)$$

Nous laissons au lecteur le deuxième exemple, plus simple. ⊗

Dans l'exemple traité, les hypothèses sont toutes déchargées dans les trois derniers pas déductifs ; en particulier, l'antépénultième décharge deux occurrences de l'hypothèse $z : A$. Notons de plus que la structure

du terme $\lambda xyz.xz(yz)$ code bi-univoquement l'arbre de la démonstration de

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C)).$$

En fait, de façon tout à fait en générale, l'ordre des applications et des λ–abstractions correspond exactement à l'ordre dans lequel viennent d'être utilisées les règles ($\to$E) et ($\to$ I).

2.2. Note: 1. **(La Logique Combinatoire)**. Le lecteur expérimenté en logique élémentaire se sera aperçu que les deux propositions démontrées dans l'exemple sont exactement les deux axiomes du calcul propositionnel (positif), dont les formules ne contiennent que l'implication et utilisent seulement la règle d'inférence "Modus Ponens", que nous appelons ($\to$ E). Donc, grâce à 2.1, avec les deux seules règles ($\to$E) et ($\to$ I) et sans axiome, nous avons la possibilité de déduire les axiomes du calcul propositionnel. Appelons alors $S \equiv \lambda xyz.xz(yz)$ et $K \equiv \lambda xy.x$ les deux termes associés aux démonstrations des deux axiomes dans notre système d'inférence. Et bien, S et K sont les deux "combinateurs" de base, ou constantes qui, avec la seule application (a b), c'est-à-dire la règle ($\to$ E), constituent la logique combinatoire.
Réciproquement, un théorème du calcul propositionnel, le théorème de déduction, restitue la règle ($\to$ I), démontrant ainsi l'équivalence logique entre le λ-calcul et la logique combinatoire.

2. (Produits) On peut facilement élargir le système minimal avec la conjonction ou produit logique. Les règles suivantes introduisent et éliminent le produit, en leur associant des termes qui dans ce cas sont également formées des couples <...,...> et de la première et la deuxième projection , p_1 et p_2, en plus des applications et des λ–abstractions.

$$(\times I) \qquad \frac{a : A \qquad b : B}{\langle a,b\rangle : A \times B}$$

$$(\times E_1) \qquad \frac{c : A \times B}{p_1(c) : A} \qquad\qquad (\times E_2) \qquad \frac{c : A \times B}{p_2(c) : B}$$

Observons la signification constructive de l'introduction de la conjonction : la preuve de la conjonction de deux formules est construite à partir des

preuves de chaque composante. Il faut observer que, puisqu'il n'y a pas de négation, la conjonction n'est pas dérivable de l'implication.

3. Le calcul des preuves et des termes.

Une variable dans un terme peut être *instanciée* par un autre terme. Par exemple, avec la même notation que celle utilisée pour renommer les variables, nous écrirons $[a/x](xzx) \equiv aza$ pour instancier x en a dans xzx. De façon inductive : $[a/x]x \equiv a$; si ensuite x n'est pas libre en b, ou mieux si elle n'apparaît pas ou n'est pas liée par λ, alors $[a/x]b \equiv b$; enfin, $[a/x]\lambda y.c \equiv \lambda y.[a/x]c$ et $[a/x](cd) \equiv ([a/x]c)([a/x]d)$. Il faut aussi supposer que, dans $\lambda y.[a/x]c$, la variable y , n'apparaisse pas libre dans a, autrement elle serait liée de façon impropre par λ (nous dirons que a est *libre pour* x dans $\lambda y.c$). Cette condition n'est pas restrictive: elle oblige seulement à renommer y, qui apparaît lié dans $\lambda y.c$, avec une variable non libre dans a, si nécessaire. Le calcul sera essentiellement basé sur l'opération de substitution d'un terme à la place des occurrences, probablement nombreuses, d'une variable libre. Voyons d'abord la signification logique de l'opération qu'il faut introduire formellement.
Considérons la démonstration suivante :

$$\dfrac{\dfrac{\begin{array}{c}[\,x : A\,]\\ \vdots\\ b : B\end{array}}{\lambda y.b : A \to B}(\to I) \qquad a : A}{(\lambda y.b)a : B}(\to E) \qquad \textbf{(P.1)}$$

Puisque x : A est une démonstration quelconque ou hypothétique de A et x peut avoir une occurrence en b, la preuve (P.1) se simplifie *(se réduit)* de la façon suivante:

a : A

:

(P.2)

[a/x]b : B

Le passage de la démonstration P.1 à la démonstration P.2 est connue sous le nom *d'élimination d'une coupure* et correspond au raisonnement mathématique suivant : si je sais que d'une preuve arbitraire x de A je peux toujours déduire une preuve b de B, alors, en particulier, je peux déduire d'une preuve spécifique a de A, une preuve spécifique, [a/x]b, de B. Pour être précis, le raisonnement inverse est particulièrement pertinent et commun en mathématiques : pour démontrer B à partir d'une preuve spécifique de A, je démontre d'abord un lemme général, qui m'assure que de chaque preuve x de A je peux déduire une preuve b de B, c'est-à-dire que je démontre que $\lambda x.b : A \rightarrow B$; donc, j'obtiens le cas particulier, [a/x]b : B, comme instance d'un cas général, $(\lambda x.b)a : B$. En conclusion, la preuve générale $(\lambda x.b)a$, donnée par le lemme $\lambda x.b : A \rightarrow B$, se réduit à la preuve particulière de [a/x]b. Nous écrirons $c > d$ pour dire que le terme c se *réduit* au terme d.

3.1 Définition. Le *λ–calcul* $(\lambda\beta_{>})$ est le calcul des termes introduits ci-dessus, basé sur l'axiome de réduction suivant :

$(\lambda x.b)a > [a/x]b$ où a est libre pour x dans b

(lire : substituer a à la place de toutes les occurrences libres de x dans b).

Le calcul $\lambda\beta\eta_{>}$ est obtenu en ajoutant l'axiome :

$\lambda x.cx > c$ pourvu que x ne soit pas libre dans c.

La signification logique de (η), n'est pas aussi pertinente que celle de (β). Elle peut par contre servir à rappeler la fonctionnalité du calcul que nous sommes en train de définir. $\lambda x.cx$ s'entend comme fonction qui dépend explicitement de x, dont le *corps de définition* est cx. Maintenant si c ne contient pas x, appliquer la fonction $\lambda x.cx$ ou directement c à un terme a, du même type que x, c'est la même chose : en effet, à cause de (β), on a $(\lambda x.cx)a > ca$. Donc (η) réduit $\lambda x.cx$ à c.

La signification opérationnelle du calcul symbolique que nous venons de présenter devrait être claire. L'axiome fondamental (β) n'est qu'une règle mécanique de réécriture : le symbole a est pris et réécrit

ou mis à la place de toutes les occurrences libres de x dans b. Il est en fait effacé, s'il n'y en a pas et copié n-fois si x apparaît n-fois dans b. En accord avec l'observation 2.2.2, il devient facile d'élargir le calcul aux termes par la conjonction logique : il faut simplement décrire formellement que la première et la deuxième projection, p_1 et p_2 choisissent le premier ou le second élément de chaque couple et que les couples sont toujours obtenus en accouplant les premières et les deuxièmes projections. Le calcul est complété par une règle qui exprime le caractère transitif de la réduction et par des règles qui garantissent ainsi la possibilité de l'application des axiomes dans les sous-termes (substituabilité). En résumé, on veut que si $c > d$, alors $\lambda x.a(cb) > \lambda x.a(db)$.

3.2 Note (Les théories de l'égalité). Une extension immédiate de la théorie de la réduction que l'on vient de présenter est donnée en considérant l'égalité entre formules comme la congruence minimum induite par ">". C'est-à-dire, $a > b$ implique $a = b$ et "=" est la relation minimale, ainsi faite, qui soit réflexive, symétrique, transitive et substitutive minimale. $\lambda\beta\eta_{=}$ est la théorie de l'égalité.

La notion fondamentale est alors la notion de *forme normale* pour les λ-termes. Un terme est en forme normale, s'il ne contient pas de sous-termes de la forme $(\lambda x.b)a$, donc des sous-termes auxquels ont puisse appliquer l'axiome (β). Il possède une forme normale, s'il se réduit à un terme en forme normale. Nous avons alors pour le calcul typé :

3.3 Théorème (Normalisation). Chaque terme de $\lambda\beta_{>}$ possède une forme normale.

Du point de vue de la déduction naturelle, une démonstration est en forme normale quand elle ne contient pas d'application de la règle $(\rightarrow I)$ suivie d'une application de $(\rightarrow E)$, deux règles qui, lorsqu'elles se suivent, introduisent respectivement $(\lambda x.b)$ et $(\lambda x.b)a$. Donc grâce à la correspondance biunivoque qui s'est établie par construction, entre termes et preuves, la démonstration de 3.3, effectuée sur le calcul des termes, fournit un résultat logique, ayant trait à la théorie de la

démonstration intuitionniste : toute démonstration peut être réduite à une autre, sans coupures, de la même assertion.

Il faut noter maintenant qu'un terme peut contenir divers sous-termes auxquels on peut appliquer (β), et peut donc donner lieu à différentes chaînes de réduction. Cependant, le théorème suivant garantit que celles-ci sont toujours confluantes.

Théorème (Church-Rosser). Si un terme a de $\lambda\beta_{>}$ se réduit à b et c, il existe alors d auquel se réduisent aussi bien b que c.

Les deux démonstrations demandent quelques précisions (voir Hindley&Seldin[1986]). Elles s'étendent facilement, toutefois, aux calculs avec (η) et les projections par le produit, puisque les réductions (η) et les autres peuvent être différées et n'interfèrent pas avec celles de (β).

Donc, du fait de la correspondance parfaite entre termes et preuves, chaque démonstration, dans les systèmes déductifs correspondants, possède une forme canonique, sans coupures, par 3.3, et celle-ci , de plus, est unique, grâce à 3.4. Si en fait un terme a se réduit à deux formes normales b et c, celles-ci ne pouvant pas se réduire ultérieurement, en particulier pas à un terme commun, elles ne peuvent qu'être identiques. Ce fait garantit aussi la non contradiction ou *consistance* des systèmes d'équations présentés où, en l'absence de négation, par consistance on entend que non toutes les équations sont démontrables. En fait, 3.4 équivaut à l'affirmation suivante : b = c implique qu'il existe d tel que b > d et c > d. Il n'est donc pas possible de déduire l'égalité de deux formes normales distinctes, disons b et c, puisque, à nouveau, celles-ci ne peuvent se réduire à un d commun.

En conclusion, la Théorie de la Démonstration nous a suggéré un langage dont les termes codifient des démonstrations. Le langage peut être écrit par une machine, implémenté et manipulé automatiquement, il suffit que l'automate applique les axiomes (β) et (η). Nous avons donc présenté en même temps un langage pour les mathématiques et un langage de programmation dont les programmes sont les λ-termes. D'un côté, en fait on est capable de manipuler des démonstrations en tant que termes du λ-calcul : de les synthétiser à partir de propositions et, à l'inverse, d'écrire la proposition démontrée par un terme ; et de l'autre, d'effectuer des calculs symboliques de termes a priori sans signification

logique. Dans le prochain chapitre, nous examinerons ces faits du point de vue de la programmation.

4. Formules comme Types; le calcul sans Type.

Le λ–calcul a acquis un rôle important en programmation, surtout grâce au langage de programmation LISP (List Processing), très utilisé en intelligence artificielle, au langage ML (Meta Langage) d'Édimbourg et leurs dérivés, voir Mitchell[1993]. En fait, le λ-calcul est devenu un paradigme pour tous ces langages qu'on appelle langages fonctionnels ou applicatifs, souvent obtenus à partir de ce calcul, seulement grâce à des extensions avec des constructions qui les rendent plus efficaces du point de vue de la programmation. Les langages fonctionnels sont basés sur l'écriture des programmes comme fonctions à appliquer à des arguments ($\lambda x.b$ qui s'applique à a et se réduit à $[a/x]b$) au lieu d'un ensembles d'ordres (les "do", "go to" des langages de programmation impérative). La manipulation est purement symbolique, elle ne traite pas de chiffres, a priori, même s'il est évidemment possible de codifier les chiffres avec des chaînes appropriées de symboles : comme nous l'avons expliqué, les axiomes base formalisent des opérations très simples d'effacement et de copie de symboles.

Du point de vue de la programmation, les λ-termes sont programmés et les propositions (A, B, $(A \to B)$...), que nous avons considérées comme des formules démontrées par les λ-termes sont appelés les *types* des programmes, reprenant en ceci une notion développée par Russell. Intuitivement, un type est un ensemble de termes ; dans la terminologie de la Physique, ceci peut être compris comme la "dimension" d'une expression : dans $f = ma$, en Mécanique, les expressions des deux membres ont le "type" d'une force. De toute évidence les formules ne sont pas toutes des propositions : seulement les formules ayant une démonstration le sont. Ainsi considérons (ex : $A \to B$ et $A \to (B \to A)$: seulement la seconde formule est démontrable et la démonstration est codifiée par $K \equiv \lambda xy.x$). Nous appellerons donc *habité* un type qui est une proposition, c'est-à-dire qui en tant que formule, a une démonstration, celle codifiée par l'un des termes sans variables libres qui l'habitent. D'après ce que nous avons vu, le type d'un terme, sans variable libre ou qui ait une variable explicitement typée est unique,

alors qu'un type peut contenir plusieurs termes même s'ils sont tous en forme normale (un théorème peut avoir plusieurs démonstrations). Ceci correspond à l'intuition pratique, qu'un type de programme peut contenir de nombreux programmes : si par exemple, Int est le type (l'ensemble) des nombres entiers, Int → Int contient tous les programmes des entiers à valeurs dans les entiers. Les types de calcul présentés jusqu'ici s'appellent *simples* et, comme nous l'avons vu, correspondent aux formules (et propositions) du calcul propositionnel positif.

Il faut toutefois mentionner que aussi bien le λ–calcul que le LISP, en tant que langages de programmation, ont été conçus sans type. En fait, si l'on considère seulement les termes qui admettent des types, les fonctions mathématiques définissables sont très peu nombreuses : c'est pour cela que nous parlerons des extensions du λ-calcul avec types, dans le §.5. Pour nous qui avons commencé avec la correspondance entre λ-calcul et logique, il est possible de retourner en arrière vers le calcul sans types, simplement en relisant les règles de bonne formation de termes sans aucune information ou restriction de type. Alors, l'application (ab) est autorisée pour chaque a et b, a = b inclus, sans les restrictions imposées par les hypothèses dans la règle (→E). Évidemment, on ne peut pas attribuer un type à tous les termes : justement, xx est un terme, tandis qu'il ne l'est pas dans le calcul avec types puisque dans l'hypothèse de la règle (→E) il ne se peut pas que x ait à la fois un type A et un type A → B, comme cela serait nécessaire pour appliquer x à lui-même. Les axiomes de réduction (β) et (η) sont identiques. Toutefois, pour le calcul sans type, le théorème de normalisation (3.3) n'est pas valable. On voit tout de suite par exemple que le terme o = (λx.xx)(λx.xx) n'a pas la forme normale : il se réduit à lui même indéfiniment. Cela vaut la peine de noter l'analogie entre λx.xx et l'ensemble non-fondé {x | x∈x }, complémentaire de l'ensemble paradoxal, qui suggéra à Russell le paradoxe pour le système de Frege et l'introduction de la théorie des types pour les ensembles : il suffit de substituer l'abstraction "ensembliste" {x | ...} à la λ-abstraction λx.... et l'auto-appartenence x∈x à l'auto-application xx. La non convergence de o = (λx.xx)(λx.xx), un fait "négatif", si l'on veut, est en réalité liée à l'entière expressivité calculatoire du calcul sans types. En fait une variante de o est d'un très grand intérêt. On considère θ ≡ λy.((λx.y(xx))(λx.y(xx))) (comme dans LISP, les parenthèses sont très

importantes !). Alors, en termes d'égalité et en appliquant plusieurs fois l'axiome (β), on obtient $\theta a = [a/y]((\lambda x.y(xx))(\lambda x.y(xx)) = (\lambda x.a(xx))(\lambda x.a(xx)) = [(\lambda x.a(xx))/x](a(xx)) = a(\theta a)$. Ce résultat est très important, car il assure que pour chaque terme a on peut trouver un *point fixe* θa, tel que $a(\theta a) = \theta a$. De plus, le point fixe est fourni de façon *uniforme* et *effective*, c'est-à-dire interne au langage, grâce au terme θ. De ceci on déduit la représentativité, dans le λ-calcul sans types, de toutes les fonctions partielles récursives, que nous avons mentionnée dans l'introduction. Celles-ci, justement, sont définies au moyen d'équations récursives, dont les équations à point fixe sont une version généralisée.

Toutefois, ayant perdu toute relation avec la logique (les termes ne codifient pas nécessairement des démonstrations, étant donné qu'ils peuvent ne pas avoir de types) le problème qui se pose est celui de la consistance de la théorie de la réduction ou équationnelle; celles-ci sont définies exactement de la même façon que pour le calcul avec types, laissant ainsi de côté toute restriction de type dans la formulation des axiomes (β) et (η). De nouveau, en l'absence de négation, la consistance s'exprime en termes de non-démontrabilité... de toutes les équations entre termes. Le théorème de Church-Rosser (v. 3.4), qui subsiste aussi pour le calcul $\lambda\beta\eta$ sans types, garantit justement le fait que non toutes les équations ne sont pas toutes déductibles: comme dans le cas avec types, ainsi il n'est pas possible de déduire l'égalité de deux formes normales distinctes.

Comme nous l'avons déjà dit, les termes en forme normale ont une importance particulière. Le théorème de normalisation est l'application fondamentale du calcul avec types dans la logique, important surtout dans le cas, que nous mentionnerons au chapitre 6, des logiques d'ordre supérieur. Dans le calcul sans types, les termes avec une forme normale représentent les calculs qui terminent ; certains auteurs, et d'abord Church et Böhm considèrent comme doués de signification seulement ces termes (nous reviendrons sur la notion de "signification", pas seulement calculatoire, en parlant des modèles). A ce propos, du fait d'un résultat de Böhm [1968] (voir aussi Barendregt[1984]) il n'est pas possible d'égaliser deux calculs différents qui terminent. Plus précisément si : a et b possèdent des formes normales distinctes, $\lambda\beta + (a = b)$ n'est pas consistant. Le théorème de Böhm assure ainsi qu'aucun calcul de symboles, qui soit une extension du λ-calcul, ne peut être ambigu sur les

calculs qui terminent : s'ils sont exprimables dans le λ-calcul, on ne pourra pas les confondre entre eux. Du point de vue sémantique, le théorème de Böhm est donc un résultat de complémentarité relatif aux formes normales : une fois qu'un modèle quelconque de calcul sans types est fixé, (v. §.5), une égalité entre formes normales est vraie si et seulement si elle est démontrable.

Un autre résultat, extrêmement intéressant pour la programmation, qui lie le calcul non typé avec le calcul typé, est le suivant : on peut décider si un terme avec type est bien typé, et si à un terme non typé on peut lui assigner un type (algorithme de Hindley-Milner, v. Hindley&Seldin[1986]). Autrement dit, étant donné un programme fonctionnel, ou un terme écrit librement, sans faire attention aux types, un type-checker automatique peut dire si ce programme est bien typé ou s'il peut admettre des types. On se souvient de l'analogie que nous avons mentionnée, entre la notion de type en programmation et de dimension en Physique, l'algorithme de type checking pour les programmes fonctionnels peut être assimilé, par sa nature et pour son côté pratique, au contrôle dimensionnel des équations de la Physique. On sait qu'on ne peut pas avoir d'algorithmes généraux de contrôle de l'exactitude des programmes (théorème de Rice), c'est-à-dire qu'on démontre en général, qu'il n'est pas possible de contrôler effectivement qu'un programme calcule la fonction que l'on veut implémenter. Les types fournissent alors un outil efficace de contrôle partiel de la correction des programmes, tout à fait analogue au "contrôle dimensionnel" en Physique : étant donnée une équation en Physique, on fait des calculs, on développe, et à la fin on contrôle que si à gauche on trouve une force, à droite aussi on trouve bien une force, à une énergie, doit correspondre une autre énergie, etc. Dans ce cas aussi, il s'agit d'un contrôle partiel : en aucun cas le contrôle des dimensions n'assure l'exactitude des calculs faits. La même chose est vraie pour le contrôle des types d'un programme. Toutefois, presque toutes les erreurs de calcul dans une équation en Physique, ou d'implémentation de calcul dans un programme, est révélée par une erreur dimensionnelle ou de type. L'algorithme de type checking est en effet le coeur des langages de programmation, du style ML: en fait il est divisé en un contrôle de la "typabilité" et un algorithme d'assignation des types, basé sur la règle d'inférence logique du §.1.

Revoyons maintenant la relation entre termes et types du point de vue de la logique. Dans ce but l'assignation d'un type à un programme est la démonstration d'une proposition, c'est-à-dire son type. L'association d'un terme à un type est la synthèse d'une démonstration, celle codifiée par le terme.

5. Sémantique.

La formalisation de types et de termes présentée jusqu'à présent a déjà eu une interprétation : les types comme propositions, les termes comme démonstrations (ou le contraire). Raisonnons maintenant sur la possibilité d'une signification mathématique, non formelle ou de pur calcul des signes, par les règles et les termes introduits. Il est souhaitable de faire une place à cet aspect, apparemment non essentiel de l'élaboration mécanique, pour au moins deux motifs. Les formalismes abstraits de la logique peuvent être adaptés à des machines qui élaborent sans "donner de signification", mais qui sont souvent hostiles à l'intelligence humaine. La compréhension d'un système logique, même essentiel ou minime dans ses parties formelles, devient meilleure s'il est immergé dans des structures mathématiques, pas nécessairement constructives, ni élémentaires, mais basées sur des expériences connues de synthèse conceptuelle ou des intuitions spacio-temporelles non formalisées. Enfin le rôle joué par le λ-calcul en informatique, en tant que manipulateur symbolique et langage décrivant des fonctions mathématiques, est dû aussi aux études de sémantique des langages de programmation que celui-ci a inspiré.
On se rappellera que les deux règles d'inférence du λ-calcul, $(\rightarrow I)$ et $(\rightarrow E)$, ont deux rôles bien précis. La première introduit dans les langage comme implication formelle "A $\rightarrow$ B" la déduction métalinguistique

$$\begin{array}{c} A \\ : \\ B \end{array}$$

et les termes qui la codifient. Ce passage est essentiel pour un traitement formel, linguistique, de la logique en tant que métamathématique : elle a pour objet d'étude la démonstration mathématique et doit, justement, donner une forme linguistique rigoureuse à ce qui est en Mathématique la déduction, souvent

informelle, toujours métalinguistique, d'une assertion d'un langage spécifique ou théorie mathématique (le langage ou la théorie des groupes, des espaces topologiques...). L'autre règle, ($\rightarrow$E), codifie avec des termes de λ-calcul le classique "modus ponens" en soulignant son caractère fonctionnel, comme déjà décrit par ce qu'on appelle interprétation de Heyting-Kleene. C'est-à-dire que, la signification intuitive de $A \rightarrow B$ est celle d'être un ensemble de fonctions ou procédés effectifs qui transforment des éléments (preuves) de A en éléments (preuves) de B.

Afin de donner une signification mathématique rigoureuse à cette signification intuitive de la syntaxe, nous rappelons la définition mathématique de *catégorie* comme collection d'objets, A, B... et de morphismes entre objets, f,g,... Les morphismes incluent l'identité id_A, sur chaque objet A et sont fermés par composition, $f \circ g$; l'associativité, $(f \circ g) \circ h = f \circ (g \circ h)$, et les propriétés de l'identité, $f \circ id = f$ et $id \circ g = g$, complètent la définition (v. Asperti & Longo[1991]). La catégorie des ensembles (sans structure) avec les fonctions classiques entre ensembles comme morphismes, la catégorie des groupes avec les homomorphismes entre groupes comme morphismes et celle des espaces topologiques, avec les fonctions continues comme morphismes, sont les exemples habituels de catégories. En fait, une catégorie est une collection "d'ensembles structurés", dont les propriétés structurelles sont décrites par les ensembles, non nécessairement structurés, des morphismes entre chaque couple d'objets. Le lecteur, même inexpérimenté peut comprendre de façon intuitive, que la notion, explicite ou implicite de catégorie est primordiale en mathématiques.

Nous comprendrons donc nos symboles formels et nos calculs logiques en interprétant les types comme objets et les termes comme morphismes de catégories appropriées. Toutefois, en général l'espace des morphismes entre deux objets d'une catégorie est une collection ou un ensemble "en dehors" de la catégorie, c'est-à-dire, comme nous le disions, il n'est pas nécessairement structuré comme les objets de la catégorie en question, exactement comme la déduction de la pratique mathématique est métalinguistique et donc en dehors de la théorie ou du langage mathématique objet de cette étude. La nécessité de corréler les deux notions est clairement suggéré par l'interprétation de Heyting-Kleene du type $(A \rightarrow B)$ comme collection de morphismes de A en B, v. §.2. Alors, pour donner une signification mathématique à la règle ($\rightarrow$I), qui

amène à l'intérieur du langage la déduction métalinguistitque, il faudra trouver des catégories dans lesquelles la notion de collection des morphismes entre deux objets puisse être internalisée, c'est-à-dire, qu'elle puisse être à son tour vue comme un objet de la catégorie donnée. En d'autres termes, si A et B sont des objets de la catégorie C, il faudra aussi que $\mathbf{C}[A,B]$, l'ensemble des morphismes de A en B, soit (représenté par) un objet de la même catégorie, *l'exposant* de A en B, que nous définirons avec B^A ou même $A \to B$. Dans le cas de la catégorie des ensembles (sans structure), il est clair que la notion d'espace des morphismes est immédiatement internalisée : l'ensemble des fonctions entre deux ensembles est encore un ensemble, c'est-à-dire, un objet de la catégorie. Ce n'est pas ainsi dans les deux autres exemples cités dans lesquels on veut que les objets soient des ensembles avec une structure : en général, les homomorphismes entre deux groupes ne forment pas un groupe. En ce qui concerne les espaces topologiques, même si à l'ensemble des fonctions continues entre deux espaces topologiques on peut lui donner une structure topologique, il n'est pas dit toutefois que celui-ci ait la propriété, non évidente, suivante, nécessaire pour définir des exposants suffisamment expressifs pour pouvoir interpréter dans une catégorie les types comme objets et les termes du λ-calcul comme morphismes. On observe en premier lieu, qu'une λ–abstraction permet de former une fonction de plusieurs arguments "un argument à la fois": étant donné un terme $a : A$, qui peut contenir deux variables libres $x : B$ et $y : C$, le terme $\lambda x{:}B.(\lambda y{:}C.a) : (C \to (B \to A))$ a la signification d'une fonction qui, en prenant un argument dans C, donne comme résultat une fonction $\lambda y{:}C.a$ dans $(B \to A)$. Mais les deux variables libres dans a, donnent également à $a : A$ la signification d'une fonction de deux arguments, à savoir $(\lambda\langle x,y\rangle{:}C\times B.a) : (C\times B \to A)$, pourvu qu'on ait une notion quelconque de produit dans la catégorie des significations. Mais ceci est facile : le produit cartésien de deux ensembles est un ensemble et l'on peut dire la même chose des groupes ou des espaces topologiques. La généralisation catégorielle de la notion de produit cartésien comme ensemble (structuré) des couples de deux ensembles (structurés) est simple et nous renvoyons le lecteur à la littérature ou au texte cité pour les détails. La difficulté réside justement dans l'opération fondamentale suivante du λ-calcul, appelée "currying" (de H.B.Curry, v. Hindley&Seldin[1980]): une fonction à plusieurs arguments peut être définie de façon équivalente par

abstraction d'un argument chaque fois. Par exemple, il faudrait, pour que les espaces topologiques fournissent une interprétation, qu'une fonction continue puisse être considérée telle, sachant seulement qu'elle l'est par rapport à chaque argument ; on sait au contraire qu'il existe, en analyse, des fonctions de plusieurs variables, continues par rapport à chaque argument, mais non globalement continues, c'est-à-dire non continues pour la topologie de l'espace produit. Le lecteur qui connaît les fonctions élémentaires continues sur des espaces produits et qui sait ne pas pouvoir démontrer la continuité en l'examinant variable par variable, a alors compris la signification mathématique réelle de λ-abstraction et son pouvoir expressif : s'il fait preuve de grande inventivité ou d'expérience mathématique, il peut localiser lui-même la classe des catégories qui peut lui fournir une sémantique rigoureuse. Présentons-les explicitement, par commodité, pour les autres lecteurs.

La propriété requise d'une catégorie pour interpréter le λ-calcul est celle d'être cartésienne fermée, c'est-à-dire d'avoir tous les produits $C \times B$ ainsi qu'un isomorphisme (uniformément interne à la catégorie ou naturel, v. Asperti&Longo[1991]) entre $(C \to (B \to A))$ et $(C \times B \to A)$, pour tout C, B et A, objets de la catégorie. Pour différentes raisons, comme nous le disions, les groupes et les espaces topologiques n'ont pas cette propriété. A nouveau, la catégorie des ensembles vient à notre secours : cet isomorphisme est banalement vrai entre ensembles.

Toutefois, dans la sémantique du λ-calcul il faut aller au delà de la simple catégorie des ensembles sans structure. D'abord parce que le λ-calcul est aussi un paradigme pour la programmation fonctionnelle et, si l'on veut écrire des programmes suffisamment expressifs avec un calcul à types simples, il faut l'étendre pour avoir la possibilité de donner des définitions récursives de fonctions. Celles-ci, comme nous l'avons dit au §.4, sont définissables dans le calcul sans types (et donc dans le langage de programmation LISP et dans ses dialectes). Elles doivent être ajoutées dans le cas des langages avec types comme le langage ML. En l'occurrence on se souvient qu'une définition récursive d'un terme ou d'une fonction f est donnée quand celle-ci est la solution d'une équation $x = a(x)$, ou bien quand f est un point fixe de a, §.4. Par exemple, la fonction factorielle peut être définie comme :

$$\underline{\text{fact}}(n) = \textit{si}\ \ n = 0\ \textit{alors}\ 1, \textit{autrement}\ \ n \cdot \underline{\text{fact}}(n-1).$$

Alors, en supposant avoir codifié dans le λ-calcul notre métalangage, (*si…alors…*) et en permettant quelques abus de langage, le terme a =

$\lambda xy.(si\ \ y = 0\ \ alors\ \ 1, autrement\ \ y \cdot x(y-1))$ a comme point fixe la factorielle, puisque $\underline{fact} = a\underline{fact}$. D'après ce que nous avons observé dans le §.4 sur le calcul sans types, en posant $\underline{fact} = \theta a$ on obtient mécaniquement la solution. Si l'on trouve une sémantique mathématique du calcul des types dans laquelle il existe un opérateur, une fonctionnelle, avec les propriétés de θ, ceci justifierait et garantirait également la consistance logique de l'extension de ce calcul avec un terme ayant la propriété de θ. Nous sommes intéressés, en second lieu, par une catégorie plus structurée que celle des ensembles parce que nous voulons trouver également en elle un modèle du calcul sans types. Au fond, celui-ci n'est "qu'un cas particulier" du calcul avec types : c'est le même calcul, sans les restrictions des types ou, si l'on veut, c'est un calcul avec un seul type "universel". Il faudra, pour lui donner une signification, trouver une structure avec un type "universel" dans le sens qu'il devra contenir toutes les fonctions sur ses propres éléments ; alors, chaque terme pourrait s'appliquer à chaque autre terme, et en particulier à lui-même. Or, aucun ensemble ne peut "contenir" l'ensemble des fonctions sur lui même, sauf un ensemble composé d'un seul élément, ou ensemble *banal*. En fait, un résultat classique de Cantor assure que l'ensemble des fonctions sur un ensemble non banal est strictement plus grand, par cardinalité, que l'ensemble donné. Nous trouverons, et ce n'est pas facile, un espace topologique non banal dans lequel on peut immerger isomorphiquement l'espace des endomorphismes (morphisme d'un objet en lui-même).
Avec ce double but en tête, la récursion pour des termes avec types et un modèle pour le calcul sans types, nous construirons une sous catégorie de la catégorie des espaces topologiques, qui soit cartésienne fermée et qui ait des points fixes pour chaque endomorphisme. De plus, elle contiendra un objet universel, dans lequel elle puisse immerger, en particulier, ses propres endomorfismes. La construction demande une certaine attention mathématique.

Prenons un ensemble partiellement ordonné $(A,\leq)$, un sous-ensemble D de A est appelé *direct* si chaque couple d'éléments de D admet un majorant dans D (c'est-à-dire : $\forall x,y \in D$ *(pour tout x et y dans D)* $\exists z \in D$ *(il existe z en D)* $x \leq z \ \&\ y \leq z$). Soit maintenant $\underline{A} = (A, A_o, \leq)$ un ensemble partiellement ordonné et A_o un sous-ensemble

de A ; $\underline{A}$ est un espace de Scott (*S-space)* si les conditions suivantes sont satisfaites :

(1) tout ensemble direct D admet une borne supérieure, sup D, dans A;

(2) A possède un élément minimum, disons ∇;

(3) pour tout x et y dans A, si $x \not\leq y$ (x n'est pas inférieur ou égal à y), il existe z_o dans A_o tel que $z_o \leq x$ et $z_o \not\leq y$ (A_o sépare les éléments de A);

(4) pour tout x et y dans A_o, si x_o et y_o ont un majorant dans A, alors ils ont un majorant minimum (ou borne supérieure) $z_o = \sup\{x_o, y_o\}$ dans A_o.

Le lecteur connaissant la Géométrie peut observer que chaque S-espace $\underline{A}$ a une structure topologique, étant donné l'ordre, qui a comme éléments de la base $\{z \in A \,/\, x_o \leq z\}$ pour $x_o \in A_o$, plus l'ensemble vide. Dans une telle topologie, les fonctions continues entre deux S-espaces sont toutes monotones (non décroissantes), c'est-à-dire, qu'elles préservent l'ordre et, en particulier, elles appliquent des ensembles directs dans des ensembles directs**.** En outre, si f est continu, et D est direct, alors f(supD) = sup f(D).

Un exemple intéressant de S-espace est constitué par l'ensemble des parties, PB, d'un ensemble infini quelconque B. Il suffit de prendre l'inclusion entre ensembles comme ordre partiel, la collection des ensembles finis comme sous-ensembles PB_o de PB, pour que $\underline{PB} = (PB, PB_o, \subseteq)$ satisfasse (1-4), avec l'ensemble vide en tant qu'élément minimum.

On peut maintenant vérifier que la catégorie des S-espaces est Cartésienne Fermée. En particulier, donc, une fonction à plusieurs arguments est continue, si elle l'est en tant que fonction de chaque argument. La fermeture cartésienne garantit de pouvoir interpréter les types comme des S-espaces. L'interprétation des λ–termes comme morphismes est une induction facile sur la structure des termes eux-mêmes : chaque variable de type A est un morphisme de l'espace banal {o}, avec un seul élément, dans l'interprétation de A ; l'abstraction $\lambda x{:}A.b : A \to B$ définit un morphisme de l'interprétation de A dans celle de B ; l'application formelle cd, pour $c : A \to B$ et $d : A$, est l'application fonctionnelle de c à d.

L'autre propriété de la construction qui nous intéresse est que la catégorie ait comme objets des espaces topologiques qui soient aussi des ordres partiels complets ce qui est exactement ce qui est demandé dans

l'hypothèse (1). Il est possible alors d'utiliser une construction due à Knaster et Tarski pour construire des points fixes minimums de fonctions monotones. Si f est une fonction continue quelconque, et donc monotone, d'un S-espace $\underline{A}$ dans $\underline{A}$, la chaîne $\nabla \leq f(\nabla) \leq f^2(\nabla) = f(f(\nabla)) \leq \ldots$, en tant qu'ensemble direct, admet une borne supérieure, $\sup f^n(\nabla)$. En effet puisque f est continu, $f(\sup_n f^n(\nabla)) = \sup_n f(f^n(\nabla)) = \sup_n f^n(\nabla)$; de plus, $\sup_n f^n(\nabla)$ est le point fixe minimum de f. Donc, la fonctionnelle $\Theta(f) = \sup_n f^n(\nabla)$, qui à chaque endomorphisme f de la catégorie associe son point fixe minimum, fournit une interprétation pour un opérateur de récursion θ du λ–calcul avec types.

Pour conclure, nous avons construit un modèle mathématique de calcul avec types et des opérateurs de points fixes. Les instruments utilisés proviennent de la Géométrie et sont tout à fait indépendants du λ-calcul.

Il nous faut maintenant trouver, dans la catégorie des S-espaces, un modèle du calcul sans type, dans lequel les termes puissent être à la fois des fonctions et des arguments de fonctions. Il faut pour cela construire un objet qui contienne ou dans lequel on puisse immerger l'espace de ses endomorphismes (les fonctions sont des éléments) et tel que chaque élément définisse un endomorphisme (les éléments sont des fonctions). Rappelons que l'ensemble puissance d'un ensemble infini est un S-espaces et considérons l'ensemble puissance familier, PN, de l'ensemble des nombres naturels N. La démonstration que la structure d'ordre et la structure topologique de PN, comme S-espace, ont les qualités voulues et utilise aussi la propriété des nombres entiers ; en particulier , la possibilité de codifier des couples et des ensembles finis avec des nombres (v. Scott[1976]). En particulier, donc, chaque $a \in PN$ peut être appliqué, comme une fonction, à chaque $b \in PN$: l'application ab entre éléments arbitraires de PN donne une signification à l'application formelle entre termes arbitraires, sans restrictions de type, comme elle est définie dans le calcul sans type.

La construction mentionnée complète la sémantique du λ-calcul avec et sans types, en donnant une signification mathématique à des manipulations symboliques abstraites, telles que l'internalisation de l'application métalinguistique, la λ-abstraction, l'auto-applicabilité d'un terme à lui-même. L'autonomie de la syntaxe des instruments utilisés a été soulignée non seulement pour des motifs épistémologiques, liés à la

notion de signification comme traduction, d'autant plus porteuse d'informations qu'elle a la possibilité de corréler des univers différents, mais aussi pour des motifs d'ordre pratique. Comme nous le disions dans l'introduction, le thème que nous étudions n'est pas seulement un cas particulier d'investigation mathématique en informatique ; en fait, la sémantique du λ-calcul a eu un rôle paradigmatique dans la façon de mener l'activité de recherche en sémantique des langages de programmation, en influant sur l'activité dans d'autres contextes et en s'enrichissant et en enrichissant des méthodologies d'autres origines. Dans certains cas, la signification de structures géométriques ou algébriques a suggéré des variantes ou des extensions de langages de programmation (le dialecte CAML de ML, des extensions du langage prototype Quest...). Quelquefois, d'obscures constructions de programmation, à peine claires pour l'auteur lui même, sont devenus intelligibles, et ont été modifiées dans la mesure du possible. L'effort d'immerger dans des modèles mathématiques solides, des langages et des programmes a certainement contribué à une importante amélioration du style de présentation de beaucoup de ceux-ci. Il est certain que ces dernières années, certains manuels de programmation sont devenus lisibles, ou presque, grâce à une influence croissante du style mathématique, qui pousse, en même temps, vers la rigueur, la généralité et la recherche de signification.

6. **Polymorphisme**.

A la fin du §.4 nous avons dit, qu'un terme sans types est "décidable" si on peut lui assigner un type. Celui-ci toutefois n'est pas nécessairement unique : l'identité $\lambda x.x$, par exemple a le type $A \to A$ *pour tout type* A. C'est-à-dire qu'il possède un *schéma* de type, noté d'habitude par des métavariables de type : $X \to X$. L'algorithme d'assignation des types de Hindley-Milner (§.4), implémenté en ML, attribue à chaque terme, s'il le possède, le schéma le plus général, pour lequel tout autre schéma ou type du terme en question est un exemple particulier. Par exemple, $\lambda x.x$ a aussi le schéma $(Y \to Y) \to (Y \to Y)$, cas particulier de $X \to X$. La notion de schéma de type est tout à fait analogue à celle du schéma d'axiomes ou de propositions en logique. Pour en revenir aux exemples de 2.1, $\lambda xy.x$ a un schéma de type (le plus général) $X \to (Y \to X)$ et

ceci est l'un des deux schémas d'axiome du calcul propositionnel positif dont nous avons parlé au sous-chapitre 2.2.1.
Brièvement donc, les termes sans types, mais auxquels on peut donner un type, sont *polymorphes*, car ils possèdent des schémas de type et, habitent donc plusieurs types, contrairement aux termes avec types du §.2 et 3. Les langages de la classe des ML sont polymorphes, exactement dans ce sens. Du point de vue de la logique, les programmes sont des preuves de schémas de propositions.

Dans le polymorphisme implicite, à la ML, la quantification par rapport aux variables de type est métalinguistique et seulement externe aux schémas de type. Rappelons maintenant la signification élémentaire ou intuitive des types comme ensembles ou, plus formelle, comme objets d'une catégorie. La quantification *explicite*, à l'intérieur du langage, sur des variables d'ensembles ou des objets d'une catégorie est au contraire à la base des systèmes du second ordre, où l'on quantifie sur des ensembles ou des objets de catégories. En particulier, il est à la base de l'analyse, interprétée comme une arithmétique du second ordre, puisque les réels sont des ensembles d'entiers. Le λ–calcul du second ordre, $\lambda\beta\eta^2$ (le système *F* de Girard[1971]), est obtenu justement en ajoutant la quantification par rapport à des variables de type. Avant de parler du polymorphisme explicite, observons que le polymorphisme d'un programme peut être vu comme une propriété d' *invariance* par rapport aux types en tant que structures. C'est-à-dire, le programme $\lambda x.x$, qui calcule l'identité , ou $\lambda xy.x$, qui calcule la fonction constante dans le premier argument, sont des invariants par rapport à chaque domaine d'arguments.

La lecture de la fin de ce paragraphe (et sa sémantique, au §.7) demande une certaine attention, bien que, formellement, celui-ci suppose seulement les instruments déjà introduits. Les logiques d'ordre supérieur se basent en fait sur une abstraction mathématique ultérieure.

Les types de $\lambda\beta\eta^2$ sont obtenus en étendant ceux de $\lambda\beta\eta_=$ avec variables de type, X, Y,... et avec des types quantifiés universellement, $\forall$X:Tp.A, où Tp est la collection de types (lire: *pour tout type X*, A est valable ; où il peut apparaître X dans A) ; les termes sont construits aussi avec l'abstraction par rapport aux variables de type, λX:Tp.a, et l'application de termes à types, bA. Les règles qui introduisent les nouveaux types et termes sont les suivantes :

$$(\forall I)\qquad \frac{\begin{array}{c}[X : Tp]\\ \vdots\\ b : B\end{array}}{(\lambda X : Tp\ .\ b) : (\forall X : Tp.\ B)}\ (*)$$

$$(\forall E)\qquad \frac{b : (\forall X : Tp\ .\ B) \qquad A : Tp}{(bA) : [A/X]B}$$

(*) ou en b aucune variable libre n'a le type qui dépend de X.

La première règle forme des fonctions qui vont de la collection de types aux termes. La seconde dit, justement, qu'un terme b peut être (fonctionnellement) appliqué à un type A et donner un terme bA qui habite dans le type B dans lequel on substitue A à X.

Les axiomes que l'on doit ajouter à (β) et (η) du §.3 sont les suivants (observons qu'ils en sont la version au second ordre) :

(∀β) $(\lambda X : Tp.b)A = [A/X]b$ avec A libre pour X dans b

(∀η) $(\lambda X : Tp.\ aX) = a :$ avec X non libre dans a

Par exemple, $(\lambda X{:}Tp.\lambda x{:}X.x) : (\forall X.X \to X)$ est l'identité du second ordre ou explicitement polymorphe ; $\lambda X{:}Tp.(\lambda Y{:}Tp.(\lambda x{:}X.\lambda y{:}Y.x))$ de type $\forall X \forall Y.X \to (Y \to X)$ est la fonction, explicitement polymorphe, constante dans le deuxième argument. En appliquant la première à un type A on obtient $(\lambda X{:}Tp.\lambda x{:}X.x)A = \lambda x{:}A.x$, l'identité de type $A \to A$. De façon analogue pour $(\lambda X{:}Tp.(\lambda Y{:}Tp.\lambda x{:}X.\lambda y{:}Y.x))AB : (A \to (B \to A))$.

Les types habités sont exactement les théorèmes de ce que l'on appelle le calcul propositionnel du second ordre, qui est le système logique sous-jacent à l'Analyse, comme Arithmétique de second ordre (les réels sont des ensembles de nombres entiers, d'où la quantification sur des ensembles). D'autre part, on peut comprendre le côté informatique mis en évidence dans ce passage au second ordre : les types sont "automatiquement mis à jour", puisque les termes peuvent prendre pour argument des types. C'est-à-dire, les types sont gérés à l'intérieur du langage ou manipulés par un calcul formel faisable sur une

machine, au lieu d'être particularisés en dehors du langage, de façon métalinguistique.

Observons que la théorie des types de $\lambda\beta\eta^2$ est essentiellement imprédicative. C'est-à-dire, en affirmant que $\forall X{:}Tp.A$ est un type, formellement $(\forall X{:}Tp.A) : Tp$, on définit la collection de types, Tp, en quantifiant sur la collection même que l'on est en train de définir, c'est-à-dire le type $\forall X{:}Tp.A$ contient la quantification $\forall X{:}Tp$. De telles définitions sont communes en Analyse : par exemple, quand on définit un ensemble comme intersection d'une collection d'ensembles qui peut inclure l'ensemble que l'on définit (bornes supérieures ou inférieures, mesure de Lebesgue...). L'imprédicativité des types est à la base de l'expressivité du langage et constitue un défit logique et sémantique non négligeable. De plus, même pour $\lambda\beta\eta^2$ les théorème de Normalisation et de Church-Rosser sont valables, énoncés comme dans 3.3 et 3.4 ; la démonstration, du premier en particulier, est rendue énormément complexe par l'impossibilité de stratifier les formules et d'utiliser une forme quelconque d'induction, à cause du caractère circulaire implicite dans la définition imprédicative des types (Girard[1971], v. Hindley&Seldin[1986]). Au moyen de l'analogie entre types et propositions (de second ordre), les théorèmes de normalisation et de Church-Rosser démontrent le résultat "d'élimination des coupures" et "d'uniticité des preuves canoniques (ou de forme normale)" pour les systèmes du second ordre, comme nous l'avons observé pour le calcul propositionnel au §.3. Ces résultats, en outre, garantissent la consistance logique du calcul des types et du calcul équationnel, en confirmant l'exactitude des constructions imprédicatives, qui sont à la base de l'analyse, en tant qu' arithmétique du second ordre. Le rapport étroit avec cette dernière théorie est confirmé par le théorème qui caractérise l'expressivité du calcul. Du fait du théorème de normalisation, il est évident que les fonctions exprimées sont toutes totales (toujours convergentes). En fait, les fonctions calculables en $\lambda\beta\eta^2$ sont exactement les fonctions récursives démontrablement totales dans l'arithmétique du second ordre (GirardLafontTaylor[1986]). Un tel ensemble de fonctions est beaucoup plus grand que celui des fonctions primitives récursives et l'on considère qu'il peut inclure largement toutes les fonctions totales dont on puisse avoir besoin pour les buts pratiques du calcul. Toutefois, les langages expérimentaux actuels de programmation basés sur le

polymorphisme explicite, étendent $\lambda\beta\eta^2$ avec des opérateurs de point fixe ou des récursions. Alors le théorème de normalisation n'est plus valable et l'on perd la correspondance entre programmes et démonstrations ; toutefois, l'efficacité de la récursion, en tant qu'instrument pour définir des fonctions, permet une plus grande simplicité de la programmation.

7. Sémantique du polymorphisme.

La signification mathématique du polymorphisme est relativement simple dans le cas du polymorphisme implicite : il faudra seulement réussir à corréler la signification des termes sans type avec leur version avec les types assignés. Considérons alors le modèle $\underline{PN} = (PN, PN_o, \subseteq)$ du calcul sans types mentionné à la fin du §.5 et rappelons que dans celui-ci, chaque élément $a \in PN$ est un endomorphisme, et vice versa : ab interprète l'application fonctionnelle entre termes, considérés comme des éléments du modèle. Pour interpréter les types, construisons la catégorie PER des relations d'équivalence partielles sur $\underline{PN}$ de la façon suivante. Les objets sont des relations, A, B... sur PN (c'est-à-dire des sous-ensembles du produit $\underline{PN} \times \underline{PN}$), d'équivalence (donc symétriques et transitives), partielles (les éléments de PN ne sont pas tous corrélés à eux-mêmes). PER est cartésienne fermée. La construction de base pour le vérifier est vite donnée : l'internalisation de l'espace des morphismes entre deux objets A et B, est la relation d'équivalence partielle d'équivalence $A \to B$ telle que $(d,d') \in (A \to B)$ si et seulement si , pour tout $(a,a') \in A$, on a $(da,d'a') \in B$; c'est-à-dire, d et d' sont équivalents dans $A \to B$ s'ils appliquent des éléments équivalents dans A sur des éléments équivalents dans B. C'est justement cette construction de l'espace interne des morphismes qui fait choisir des relations partielles : prenons d = d' et une relation target B assez petite , alors, les d n'impliquent pas tous des éléments équivalents dans A sur des éléments équivalents dans B. Donc, ce n'est pas généralement le cas que $(d,d) \in (A \to B)$, c'est-à-dire, qu'il n'est pas dit que tout d soit équivalent à lui-même dans $A \to B$. Dans PER l'assignation de types à des termes sans types a la signification suivante : si à un terme c on donne formellement un type C, alors l'interprétation dans PN de c, comme terme sans type,

est équivalente à elle même dans l'interprétation du type C comme relation partielle d'équivalence. Puisque ceci est vrai pour chaque interprétation des variables libres dans c avec des éléments de PN et des variables (de type) libres dans C avec des objets de PER, nous avons donné une signification correcte du polymorphisme : tout terme, vu comme un élément de PN, habite beaucoup de relations, et en particulier toutes celles qui interprètent ses types formels. On réussit aussi à démontrer que l'interprétation d'un terme sans type est dans la classe d'équivalence de son interprétation comme terme auquel ont été donnés tous les types. Ceci complète la corrélation sémantique entre termes, types, calculs sans type et assignation de schémas de types.

Pour passer au polymorphisme explicite, rappelons que celui-ci est basé sur une théorie imprédicative de types, une théorie donc, dans laquelle le type $\forall X{:}Tp.A$ est obtenu en quantifiant sur la collection de tous les types. La signification mathématique générale, en fait catégorielle de cette construction n'est pas évidente. En premier lieu, il s'agit d'interpréter la quantification universelle $\forall X{:}Tp$, comme *produit indexé (ou généralisé)* . En fait, pour la règle ($\forall$E), si $a : \forall X{:}Tp.A$, alors $aB : [B/X]A$; c'est-à-dire, le terme a est interprété comme une fonction qui, considérant (l'interprétation d') un type B comme argument, donne un résultat dans (l'interprétation d') un type A dans lequel B est substitué à la variable X. Ceci est justement la signification intuitive d'un produit indexé par un ensemble quelconque, notion qui généralise celle de produit ordinaire d'ensembles : un produit cartésien est un produit indexé par un ensemble fini. La difficulté consiste à trouver une catégorie qui soit fermée par rapport au produit indexé par la catégorie elle-même : en fait, si la collection Tp des types est interprétée comme une catégorie, $\forall X{:}Tp.A$ doit être interprété comme un produit à indice sur toute la catégorie et doit être lui aussi un objet de la catégorie. Il faudrait que le lecteur prête attention à la réflexivité forte ou propriété forte de fermeture que nous demandons ; sa compréhension rigoureuse est un typique et beau morceau de Mathématiques dans lequel les instruments géométrico-catégoriels donnent une signification à des symboles qui, autrement, pourraient n'être qu'un simple jeu de signes : on peut évidemment écrire $(\forall X{:}Tp.A) : Tp$, avec Tp, le definiendum qui apparaît dans le definiens, ce qui est intéressant est de comprendre si l'on est en train de signifier quelque chose. Beaucoup en effet trouvent

cette écriture inacceptablement circulaire, malgré les importants résultats syntaxiques cités de normalisation et de Church-Rosser.

Brièvement, la catégorie PER a la propriété de fermeture désirée. Pour le démontrer complètement, toutefois, il faut immerger PER dans un contexte plus large, dans lequel on puisse définir le produit indexé par PER elle-même. L'idée est de trouver une catégorie dans laquelle on puisse immerger PER en tant que sous-catégorie, et en même temps, comme objet sur lequel on puisse définir le produit . La réponse est donnée grâce à des *topos* appropriés. Ceux-ci sont des catégories avec une propriété forte de fermeture et , justement, ce sont des modèles de la théorie intuitionniste des ensembles. Dans un topo particulier dit "*effectif*" et construit en généralisant la construction de la catégorie PER, on peut définir le produit indexé par PER, considérée comme objet du "topo" . Le résultat, disons $\Pi_{A\in PER}F(A)$, non seulement est un objet du topo, mais carrément de PER, qui est aussi une sous-catégorie. De façon informelle, $(\Pi_{A\in PER}F(A)) \in PER$; c'est-à-dire, le produit indexé par PER est un objet de PER, en tant que (sous-)catégorie. En faisant la construction pas à pas, on "comprend" ce que peut signifier, mathématiquement, une obscure définition formelle imprédicative (v. Asperti &Longo[1991]). En particulier, les relations ou ensembles des parties sur les nombres naturels, les objets de PER, sont définis et se comprennent indépendamment du produit en question ; donc, la circularité syntaxique d'une collection de types, Tp, définie en listant entre les types, également ceux obtenus par des produits indexés par la collection elle-même (ou quantifications: $\forall X{:}Tp.A$), se réduit à démontrer une propriété de fermeture d'une structure mathématique pré-définie. Puisque la sémantique se base sur des catégories géométriques (les topos ont leur origine dans la Géométrique Algébrique), qui toutefois peuvent aussi fournir une interprétation à la théorie des ensembles intuitionniste, nous avons, dans un certain sens, fermé le cercle de notre compréhension. Nous étions partis d'un calcul formel des propositions et des termes du calcul propositionnel intuitionniste, simple et du second ordre, en passant par des structures d'ordre et topologiques, des relations sur des nombres naturels, des espaces quotients, etc... nous avons retrouvé un lien indépendant avec d'autres aspects de la logique, et de la théorie des ensembles intuitionniste. L'unité ainsi établie entre différentes théories donne une

signification, et ajoute à la compréhension, de chacune d'elles ; elle fait voir les démonstrations comme des programmes, des morphismes, des fonctions calculables, en proposant un cadre mathématique rigoureux à la programmation sur machines. C'est seulement une proposition possible, mais elle aide à dépasser la méthodologie empirique, et les solutions "ad hoc", intraduisibles, non généralisables et compréhensibles seulement par leur auteur, sources d'erreurs.

8. Démonstration automatique.

Nous avons commencé cette présentation en soulignant le rôle du λ-calcul comme langage pour les démonstrations : chaque terme du calcul avec types est la codification d'une démonstration et sa réduction en forme normale conduit à une démonstration canonique ou "minimum" de l'assertion correspondant à son type. La décision : si un terme sans types admet un type simple, l'assignation d'un type à un terme et sa réduction à la forme normale peuvent être faites de manière automatique. Ce sont donc des instruments pour la démonstration automatique et en particulier, pour la démonstration des propriété de programmes. En fait, c'est une propriété de programmes le fait d'admettre ou non des types ; mais non seulement cela, un type est en réalité une *spécification* ou une façon de spécifier un programme. En disant qu'un programme va des entiers dans les booleens , on en spécifie une propriété qui, partiellement, contribue à le définir. Dans certains cas, le type peut univoquement déterminer le programme : il existe un seul programme, qui s'entend comme terme sans variables libres et qui ait le type $\forall X.(X \to X)$, l'identité. C'est-à-dire, que si l'on spécifie un programme comme fonction qui applique chaque type sur lui-même, la description formelle de ce type, au second ordre, détermine univoquement le programme qui calcule la fonction identique.

Plus généralement parlant, comme on l'a déjà dit, l'assignation de types à un programme est, à la fois, une démonstration de propositions logiques, les types, et une méthode de synthèse (automatique) de programmes. Revenons au deuxième exemple dans 2.1 : en partant d'hypothèses sur des variables, on construit pas à pas un programme de type $A \to (B \to A)$. En fait, l'unique programme avec schéma de type $X \to (Y \to X)$.

Déjà donc, dans le contexte actuel, nous pouvons dire que nous avons une méthode, partielle, de démonstration de propositions et de propriétés de programmes, et aussi de synthèse de programmes. Une méthode partielle, parce qu'elle requiert l'intervention externe, humaine, dans le choix des hypothèses, qu'il peut être nécessaire d'ajouter également au cours de la déduction, et parce que, par rapport aux affirmations sur les programmes, on peut déduire ainsi, seulement quelques propriétés et quelques programmes.

Toutefois, pour parler de démonstration automatique, comme substitut de la démonstration mathématique humaine, ce qu'il manque aux notions que nous venons de présenter, ce sont ... les mathématiques. Ou mieux, il manque les instruments de formalisation des propriétés plus ou moins usuelles des mathématiques. Il est évident, en effet que le traitement mathématique se base d'abord sur l'utilisation de variables *individuelles*, pour les éléments des ensembles, et sur leur quantification. En d'autres termes, il faut pouvoir écrire des formules qui décrivent des affirmations comme : pour chaque nombre entier, il en existe un plus grand ; chaque élément d'un groupe admet un inverse, etc... donc de la forme $\forall x \exists y(x < y)$, $\forall x \exists y(xy^{-1} = 1)$... Dans les formules, ou types, introduites jusqu'à maintenant, il n'y a pas de variables de premier ordre, ou d'éléments et on est passé directement du calcul propositionnel simple à celui du second ordre, qui tout au plus, permet de traiter la partie logique de l'analyse mathématique.

La théorie Intuitionniste des Types de Martin-Löf (prédicative) et le calcul des constructions de Coquand et Huet (Coquand&Huet[1988]), de façon différente, étendent les types du λ-calcul avec la possibilité de définir aussi les formules de premier ordre comme types, c'est-à-dire avec la structure $\forall x{:}A.B$, où x est une variable de type A (v. Hindley&Seldin[1986]). En effet, le λ-calcul a déjà des variables de premier ordre : celles qui apparaissent dans les termes. Comme nous l'avons déjà dit, la λ-abstraction est une forme de quantification sur les termes, analogue à l'abstraction ensembliste : $\{x \mid P(x)\}$ est l'ensemble de tous les individus x, d'un certain univers, qui jouissent de la propriété P. Mais, seulement cela, les variables sont des termes, de plein droit, et elles n'ont pas exclusivement une signification acceptée comme en mathématiques. Dans la pratique, en mathématiques, en effet, "pour tout x entier..." signifie seulement que x peut être particularisée par un entier arbitraire. La variable x au contraire, dans

le λ–calcul, est un terme de forme normale qui peut être manipulé, traité comme les autres termes, fermés, constants ... Les systèmes cités utilisent cette richesse linguistique du λ–calcul pour donner un traitement unifié de formules et de termes. En bref, les formules mathématiques sont des termes et des types du premier ordre, sur lesquels on peut effectuer des calculs de façon uniforme et automatique. Par exemple, la formule $x(x + 1) = 12$ devient, par des calculs évidents, $x^2 + x = 12$, où l'on a déclaré que x est un entier (assomption encore valable x : Int). La solution de l'équation est donnée par la particularisation ou substitution de x par l'entier 3. Notons que les formules ou propositions mathématiques ne sont que des restrictions à satisfaire : à leur tour, les propositions sont spécifiques ou des types et un programme est la preuve d'une spécificité, un type, comme dans les cas déjà discutés. On peut noter dans ceci une différence substantielle avec la Programmation Logique où un programme est une proposition et son évaluation est une preuve de la proposition. L'intérêt actuel d'une intégration des méthodes de la programmation fonctionnelle et celles de la programmation logique, est justement due à la possibilité d'étudier la synthèse de programmes fonctionnels, à partir de spécifications logiques, en tant que méthode de compilation de programmes logiques en programmation fonctionnelle. Comme déjà annoncé dans l'introduction, nous avons toutefois omis l'instrument principal pour traiter les formules du premier ordre ou les restrictions qui décrivent les propriétés mathématiques usuelles: les techniques d'unification et les résolutions qui gèrent la particularisation uniforme des formules, que nous venons de mentionner. Ces téchniques, centrales en programmation logique ont été introduites dans le calcul des constructions, aussi bien du premier ordre, que, ce qui est nouveau, au second ordre.

Dans l'ensemble, les directions de recherche que nous venons de détailler ont donné de bons résultats, s'ils sont lus avec une certaine précaution. De Bruijn, par exemple, a développé, des "jargons mathématiques", dans des extensions du λ–calcul, d'une importante efficacité intuitive (v. Hindley&Seldin[1986]). Toutefois, les propriétés vraiment importantes qui ont pu être discutées, sont des propriétés de programmes. Les assertions purement mathématiques démontrées de façon tout à fait automatique ne sont pas nombreuses, (et toujours les mêmes) Le problème est que la richesse implicite dans une vraie démonstration mathématique se trouve dans les changements de

langage, dans les ponts et dans les analogies indirectes, dans la superposition des méthodes. Leur réduction stérilisante dans un seul langage, pauvre et statique, peut être d'une certaine importance dans l'interactif, si l'on est optimiste. De toute façon des instruments en plus, pour faire des démonstrations, sont les bienvenus, même les automatiques. L'algèbre calculatoire, les systèmes sur le type du langage "Mathematica" complètent avec une grande efficacité le travail de l'algébriste, en intervenant là où il faut faire des calculs très longs, des explorations d'innombrables cas et dans d'autres tâches pour lesquelles la complexité pratique devient impossible pour un être humain. Les méthodes que l'on présente ici, et inspirées du λ–calcul développent l'approche complémentaire, en cherchant à faire déduire le plus possible de mathématiques de la formalisation logique. Malgré le caractère limitatif et les échecs de cette mise en place, considérée comme un projet de base, du point de vue pratique, l'interaction homme machine peut faire des miracles, mais bien sur, si l'on accentue le caractère interactif des preuves. Prenons, par exemple la démonstration par induction. Le λ–calcul décrit parfaitement le schéma d'induction et les preuves inductives sont un candidat typique pour le traitement automatique. Toutefois, chacun sait que dans les cas non banals, le vrai problème mathématique, dans une preuve par induction, est le choix de l'hypothèse inductive. Souvent la "charge inductive" doit être beaucoup plus lourde que la thèse ; c'est-à-dire, pour démontrer $\forall x.P(x)$ inductivement, on ne réussit pas toujours à le faire en démontrant que, pour chaque n, de P(n-1) on déduit P(n) et il faut recourir à une propriété Q, plus forte que P, pour avoir que Q(n-1) implique Q(n).

Le problème du choix des hypothèses inductives est aujourd'hui un problème central en démonstration automatique. Il n'est pas clair en fait qu'il suffise d'explorer un nombre fini d'hypothèses inductives possibles ou que, pour obtenir des méthodes complètes, il soit nécessaire d'en considérer un nombre infini . C'est-à-dire, si dans des secteurs standards des mathématiques, on a seulement à faire avec des démonstration inductives longues ou si, plutôt, de telles démonstrations ne sont pas essentiellement difficiles : les critères de choix entre une infinité d'hypothèses inductives possibles sont difficiles parce qu'ils sont généralement extérieurs à une méthodologie préétablie, à un langage et un cadre formel gelé, comme on le disait. S'en suit la nécessité d'un développement interactif des programmes de synthèse automatique de

preuves où l'utilisateur indique à la machine les propositions à démontrer dans l'induction (le poids inductif) et laisse à la machine le travail de bas niveau. L'étude de méthodes interactives ou heuristiques le plus possible automatisées peut de toute façon conduire à des systèmes acceptables ou utiles, même s'ils sont incomplets.

9. Conclusion.

Dans cette présentation du λ-calcul on a cherché avec insistance à décrire les λ–termes, à la fois, comme programmes, comme codifications de démonstrations, et comme morphismes de catégories. Avec cela, nous avons voulu souligner que l'importance cognitive des mathématiques vient aussi, sinon surtout, des ponts, des corrélations entre contextes divers, où la suggestion informelle, par exemple, qui fait qu' un langage en construction inspiré par un modèle, peut avoir un grand intérêt pratique et gnoséologique . Souvent la proposition de nouvelles idées et structures, la formulation de conjectures, survient grâce à un équilibre réflexif de théories qui s'intègrent et s'expliquent mutuellement et se développent, en se modifiant , dans cette interaction. L'unité est donnée, non pas par une unicité logique ou linguistique métaphysique, mais par les rapports entre théories et langages divers. Heureusement le temps est fini, pour une fondation ultime et unique des mathématiques, dont s'inspiraient dans les années trente les inventeurs du λ-calcul et de la logique combinatoire. Seulement quelques formalistes et réductionnistes se complaisent encore dans de lourds programmes totalisants. Leurs efforts quelques fois ne sont pas inutiles, grâce aujourd'hui à l'informatique, où de purs calculs minimaux des signes peuvent suggérer d'autres langages adaptés à des machines. Du reste, ceci est exactement ce qui est arrivé pour les systèmes examinés : la mise en place formaliste à son origine a trouvé application, d'une part dans la rigidité des machines et elle s'est dissoute, d'autre part, à sa rencontre avec le platonisme ou réalisme mathématique. En fait c'est cela la vision implicite ou explicite des chercheurs qui en on cherché la signification dans des structures géométrico-catégorielles, en contribuant à l'entrelacement des significations auxquels on a fait allusion.

Plus en général, l'approche formaliste et le platonisme répandu en mathématiques, apparemment tellement différents, ont une justification

commune et leur origine dans l'observation qu'un concept mathématique, laissant de côté la structure spécifique d'où il émerge, acquiert une généralité et une indépendance qui le rend applicable à une pluralité de structures. Les calculs logiques dont nous avons parlé ici, dans leur parfaite autonomie linguistique, qui comprend la consistance démontrable (v.§.3), sont un exemple paradigmatique d'indépendance, généralité et abstraction qui font "oublier" les structures qui les ont suggérées : l'algèbre sans variables pour la logique combinatoire (Curry), la notion de fonction (effective) pour le λ-calcul (Church). Ensuite il n'était pas banal de réindividualiser des modèles mathématiques qui interprètent les formalisations auxquelles ont avait enfin abouti. Ces structures, d'autre part, se ramifient dans une pluralité de connexions et d'applications, souvent inattendues.

Il faut souligner, à ce sujet, que c'est justement la généralité et l'indépendance des significations spécifiques qui est à l'origine de cette fameuse universalité et objectivité des mathématiques : l'importance d'une notion et d'un théorème réside dans son invariance par rapport à des notations linguistiques et à des structures mathématiques particulières. L'expérience historique concrète de cette invariance par rapport à une pluralité d'univers pratiques, souvent aussi réels que le fait de compter avec les nombres, est à la base de toute fondation formaliste ou vision ontologique des notations dans les Mathématiques. Les deux, chacune à sa façon, attribuent l'universalité et l'existence à de pures, quoique très raffinées, constructions linguistiques et géométriques de l'homme, et sont dues à la stupeur du mathématicien devant la généralité de ces constructions, pourtant surgies des expériences de la vie et du monde réel.

Bibliographie Essentielle (Appendice)

Asperti A., Longo G. [1991] *Categories, Types, and Structures: an Introduction to Category Theory for the Working Computer Science*, MIT Press.

Barendregt H. [1984] *The Lambda Calculus, Its Syntax and Semantics*, revised edition, North-Holland.

Böhm C. [1968] "Alcune proprietà delle forme beta-eta normali nel Lambda-K-calcolo", Pubblicazioni dell'Instituto per le Applicazioni del Calcolo.n. 696, Roma

Church A. [1941] *The Calculi of Lambda Conversion*, Princeton University Press

Coquand T., Huet G. [1988] "The Calculus of Constructions" *Information and Computation*, 76, pp. 95-120.

Girard J.Y. [1971] "Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types", In *Proceedings of the Second Scandinavian Logic Symposium, Studies in Logic 63*, J.E. Fenstad (ed.), North-Holland, Amsterdam, pp.63-92.

Girard J.Y., Lafont Y., Taylor R. [1989] *Proofs and Types*, Cambridge University Press

Hindley J.R., Seldin J.P. [1980] *To H.B. Curry: Essays on combinatory logic, Lambda Calculus and Formalism*, Academic Press.

Hindley J.R., Seldin J.P. [1986] *Introduction to Combinators and Lambda-Calculus*, Cambridge University Press.

Mitchell J.C. [1993] *Introduction to Programming Language Theory*, M.I.T. Press.

Robinson J. [1965] "A machine-oriente logic based on the resolution principle" *Journal of the Association for Computing Machinery*, 12, pp. 23-41.

Scott D.S. [1976] "Data types as lattices", S.I.A.M.Journal of Computing 5.