Fonctions, programmes et démonstrations

Journée "Les fonctions en mathématiques: sous le concept, Babel?"
I.H.P. 23 Mars 1994

Jean-Louis Krivine

Equipe de Logique mathématique, Université Paris VII, C.N.R.S.

e-mail krivine@logique.jussieu.fr

Dans son exposé, C. Houzel nous a montré l'évolution du concept de fonction. Je voudrais vous faire voir ici comment cette évolution a accompagné, jusqu'à aujourd'hui, celle de l'axiomatisation des mathématiques; et comment le tout dernier avatar de ce concept de fonction, qui est la notion moderne de *programme*, est devenu un outil fondamental de la théorie de la démonstration. En fait, le programme est devenu (et cela en relativement peu de temps) un outil théorique essentiel dans bien d'autres sciences, comme la biologie, les neurosciences, les sciences cognitives, etc. Et, avec la diffusion massive de l'informatique, il a aussi envahi notre vie de tous les jours. Mais je me contenterai de parler ici de son impact sur les mathématiques et la logique.

Revenons tout d'abord brièvement sur l'histoire de la notion de fonction, que nous a décrite C. Houzel, et ce qu'on pourrait appeler (de façon très schématique) les fonctions au sens d'Euler et les fonctions au sens de Riemann.

Pour Euler, une fonction est donnée par une "formule", que celle-ci permette, ou non, le calcul de la fonction. Par exemple, e^x, $1 + x + \cdots + x^n + \cdots$, $\int_0^\infty t^{x-1}e^{-t}dt$, ... sont considérées comme des fonctions d'une variable x. Le domaine n'est pas précisé a priori, et la formule représente une "tentative" de calcul de la fonction, qui peut ne pas aboutir. Par exemple, l'expression $1 + x + \cdots + x^n + \cdots$ représente la fonction $1/(1-x)$ même si $|x| > 1$, alors que cette série ne converge pas. De même $\int_0^\infty t^{x-1}e^{-t}dt$ représente la fonction Γ même pour des valeurs négatives de x, pour lesquelles cette intégrale n'existe pas. Bien entendu, on peut donner un sens à ces expressions, au moyen d'outils mathématiques créés bien après Euler (séries formelles, prolongement analytique, distributions, opérateurs non bornés, etc.). Mais on est obligé de procéder au cas par cas, d'employer une méthode particulière pour chaque fonction. Il a toujours paru impossible d'obtenir une définition précise de la notion de fonction "à la Euler". Du coup, cette notion a été abandonnée pendant fort longtemps, au profit de celle introduite par Riemann. Mais, comme nous le verrons, la notion eulérienne de fonction a, depuis peu, fait un retour en force.

Avec Riemann, la perspective change du tout au tout, et on en vient à une approche qu'on appellerait maintenant "axiomatique"; comme on le sait, cette tendance ne va faire

que croître et embellir, et nous verrons à quoi tout cela va aboutir.

Toute idée de calcul, ou de formule, a disparu: une fonction de $\mathbf{R}$ dans $\mathbf{R}$ est une "boîte noire", c'est-à-dire un objet indéterminé, dont on ne connaît qu'une seule propriété: quand on lui donne un réel, elle rend un réel. On ne s'autorise à raisonner qu'à l'aide de cette seule propriété. Quels avantages y a-t-il donc à brider ainsi le raisonnement?

En fait, cela signifie que l'on va pouvoir maintenant démontrer des propriétés sur des *classes* de fonctions, et non plus, comme Euler, sur des fonctions particulières. Ou encore démontrer des propriétés *négatives*: telle fonction est continue sur $\mathbf{R}$, et nulle part dérivable.

Le but de ce changement radical de point de vue est ce qu'on appelle la "rigueur": il s'agit d'avoir des définitions précises des notions de fonction continue, dérivable, analytique, etc. La rigueur, en mathématiques, n'est pas le but en soi, mais le moyen de permettre au raisonnement mathématique de s'épanouir. Les raisonnements et les calculs d'Euler n'étaient pas toujours rigoureux au sens où nous l'entendons aujourd'hui, mais c'était bien des mathématiques, comme chacun sait.

Cette axiomatisation a eu des effets bénéfiques tout à fait évidents: elle a permis l'apparition d'un domaine privilégié, celui des fonctions analytiques (holomorphes) où les les deux conceptions des fonctions se marient exceptionnellement bien. Mais elle a aussi permis toutes les extensions de la notion de fonction, qui deviennent alors indispensables en analyse: mesures ("fonctions" de Dirac), distributions (où l'opération de dérivation est libre), espace de Hilbert, puis espaces fonctionnels divers et variés, ...

On peut donc retenir que c'est à propos de la notion de fonction qu'est apparu le besoin de rigueur, d'où est sortie la méthode axiomatique et, finalement, la théorie des ensembles: on veut savoir précisément ce qu'est une fonction, c'est-à-dire expliciter complètement les propriétés qu'on utilise quand on raisonne avec les fonctions. Puis, on se posera le même problème pour la notion de nombre réel, puis celle de nombre entier, et finalement la notion d'ensemble. Toutes ces questions, qui peuvent paraître de prime abord insolubles, car elles s'attaquent à des notions de plus en plus élémentaires, de plus en plus fondamentales, seront pourtant successivement résolues (Dedekind, Peano, Cantor, Zermelo), et l'aboutissement de ce travail sera la théorie axiomatique des ensembles de Zermelo-Fraenkel, où tous les objets mathématiques trouvent leur place, et se prêtent, sans rechigner, à la rigueur du raisonnement mathématique.

Est-on arrivé alors au bout de cette quête? Pas du tout, et la notion de fonction, dont on s'était bien éloigné, va refaire surface de plus belle.

En effet, ce qui est maintenant dans le collimateur, c'est le sacro-saint raisonnement mathématique lui-même! La question est dorénavant: qu'est-ce que ça veut dire de raisonner correctement? et, qui plus est, pourquoi faut-il donc, à tout prix, raisonner correctement?

Problèmes difficiles s'il en est, et sans aucun rapport, semble-t-il, avec la notion de fonction. Et pourtant, l'inconscient des mathématiciens avait déjà, depuis fort longtemps, fait le rapprochement entre raisonnement et fonction: en effet, on emploie le même symbole $\rightarrow$ pour l'*implication* (et on sait que l'implication symbolise le raisonnement mathématique)

et pour *l'application*, c'est-à-dire pour désigner une fonction (on écrit $f : A \to B$ pour dire que f est une fonction de domaine A et d'image B, c'est-à-dire une application de A dans B).

Ce qui semble n'être qu'une coïncidence de notation est, en fait, un indice qui a permis la découverte d'un outil extraordinairement puissant pour analyser le raisonnement mathématique: il s'agit de ce qu'on a appelé la *correspondance de Curry-Howard*.

Mais ne brûlons pas les étapes, et reprenons l'histoire dans l'ordre. Les premiers systèmes axiomatiques (Frege, Russell) sont plutôt des descriptions du raisonnement mathématique. Ce qui est remarquable, c'est qu'ils sont déjà exhaustifs, c'est-à-dire qu'ils prétendent, à juste titre, mais sans en apporter la preuve, le représenter dans son intégralité. La justification viendra, avec le théorème de complétude de Gödel-Herbrand, qui est sans doute le théorème le plus important et le plus central de la logique. Il montre, en effet, que l'on peut décrire les preuves mathématiques de façon purement formelle, au moyen d'un système de règles de déduction, comme les règles d'un jeu: une démonstration n'est alors pas autre chose qu'une partie jouée à ce jeu, en observant correctement les règles.

Ce théorème est remarquable, d'une part à cause de ce qu'il démontre, mais aussi, et surtout parce qu'il réussit à *énoncer* que certains systèmes axiomatiques représentent l'intégralité du raisonnement mathématique. En effet, il est bien loin d'être évident que l'on puisse seulement énoncer rigoureusement une chose pareille.

On sait donc maintenant que le raisonnement mathématique est complètement mécanisable, et on connaît diverses façons de le mécaniser. Tout le monde comprend qu'il s'agit là d'une découverte très belle et très importante: on peut construire, théoriquement, des machines à faire des mathématiques. Bien sûr, dès que cela devient possible, on s'empresse d'en construire effectivement, et ce sont toutes les recherches sur ce qu'on appelle la *démonstration automatique*.

Maintenant que le raisonnement est complètement analysé et mécanisé, y a-t-il encore quelque chose à chercher? Que pourrait-on dire de plus?

En fait, cette analyse a fait apparaître la notion de machine, et des calculs qu'on peut, ou qu'on ne peut pas, faire sur machine. Une nouvelle sorte de fonction a fait son entrée: ce sont les fonctions récursives, ou fonctions calculables sur machine. Beaucoup de définitions vont en être données (Gödel, Turing, Church, ...) qui se révèlent toutes équivalentes. Mais il faut noter qu'on ne pourra pas démontrer de "théorème de complétude" disant que les fonctions récursives sont toutes les fonctions calculables sur machine: cet énoncé, qu'on appelle la *thèse de Church* n'est, en fait, qu'une définition mathématique de la notion de fonction calculable.

Comme par hasard, (mais, bien sûr, ce n'en est pas un), les véritables machines, c'est-à-dire les ordinateurs, font, au même moment, leur apparition. Or, pour un ordinateur, il ne suffit pas, mais pas du tout, qu'une fonction soit calculable pour qu'il la calcule. Il faut, en plus, qu'on lui donne un moyen (un algorithme) pour la calculer, c'est-à-dire un *programme*. Et nous voilà revenus à la notion de fonction à la Euler, où la fonction est donnée par une formule mathématique, qui représente le calcul de la fonction (calcul qui

peut aboutir ou non).

Bien entendu, cette notion s'est quelque peu modifiée en cours de route: la formule a été remplacée par un programme (mais, là aussi, le calcul peut aboutir ou non, on en verra plus loin des exemples), et il ne s'agit plus de fonctions de variable réelle ou complexe, mais de fonctions d'entiers (plus généralement, de fonctions définies sur des structures de données finitaires, comme les entiers, les mots, les formules, les arbres, ...).

La notion centrale va être, dorénavant, celle de programme, c'est elle qui va jouer le rôle essentiel. En effet, elle a un rôle naturel de référence ultime, c'est-à-dire de notion qui peut servir à définir toutes les autres. La raison en est la suivante: une notion peut être considérée comme expliquée, c'est-à-dire complètement analysée, s'il est possible de l'expliquer à une parfaite brute (ou, si l'on préfère, à un parfait Candide), c'est-à-dire à un ordinateur.

Il faut donc, pour cela, pouvoir traduire cette notion en termes de programmes, puisque c'est la seule chose que comprend un ordinateur.

C'est cette méthode qu'on va maintenant employer pour poursuivre l'analyse du raisonnement mathématique: on va analyser la notion de preuve à l'aide de celle de programme. L'idée nouvelle, c'est de tenter d'identifier les preuves à des programmes. Elle trouve son origine dans ce qu'on appelle la sémantique de Heyting pour la logique intuitionniste.

Voici un exemple simple qui montre bien de quoi il s'agit: qu'est-ce qu'une preuve de la proposition $A \to B$, si c'est un programme. La réponse donnée par Heyting est simple et intuitive: c'est un programme qui, à chaque fois qu'on lui donne une preuve de A (à l'aide d'un système d'axiomes quelconque), fournit une preuve de B (à l'aide de ces mêmes axiomes).

Et nous voilà bien en train d'identifier la flèche de l'implication $A \to B$ avec celle d'une fonction: $Pr(A) \to Pr(B)$, $Pr(A)$ étant l'ensemble des preuves de A.

Par ailleurs, comme on veut identifier les preuves et les programmes, on voit que la proposition A correspond à l'ensemble des preuves de A, donc à un ensemble de programmes, qu'on appellera programmes de type A. Autrement dit, un théorème A correspond à ce qu'on appelle un *type* en informatique.

Nous avons ainsi établi les premiers éléments d'une correspondance très féconde et très profonde entre les notions de théorie de la démonstration, et celles de programmation: c'est ce qu'on appelle la *correspondance de Curry-Howard*. C'est cette correspondance qui va nous permettre de poursuivre l'analyse de la notion de preuve mathématique.

Ecrivons les premiers termes de cette correspondance, que nous venons d'obtenir (nous verrons plus loin comment cette liste se prolonge):

Théorie de la démonstration		*Programmation*
Preuve	$\Leftrightarrow$	Programme
Théorème	$\Leftrightarrow$	Type, spécification
Preuve de $A \to B$	$\Leftrightarrow$	Programme qui prend une preuve de A et rend une preuve de B

Par exemple, $A \to A$ est un théorème, quelle que soit la proposition A, et une preuve de $A \to A$ correspond à un programme qui prend une preuve de A et rend une preuve de A. Il existe bien un tel programme, à savoir celui pour la fonction identité (le programme qui rend exactement ce qu'on lui donne).

Un autre exemple, un peu plus compliqué, mais qui montre bien le rôle des fonctions, est le théorème $(A \to B) \to ((B \to C) \to (A \to C))$. Une preuve de ce théorème est un programme (s'il en existe) qui prend une fonction (programme) de A dans B, une fonction de B dans C, et rend une fonction de A dans C. Il existe bien un tel programme, qui consiste à composer les deux fonctions (programmes) données.

On voit, sur ces exemples triviaux, que les programmes ont une propriété curieuse: si on les considère comme des fonctions, leurs arguments et leur valeur sont aussi des programmes. Autrement dit, ils forment un monde fermé de fonctions dont on ne sort jamais. Et c'est bien ce qui se passe dans une machine: un programme est alors une partie de la mémoire de l'ordinateur, ses données, c'est-à-dire ses arguments, sont aussi des zones de la mémoire, et son résultat est également inscrit dans une zone de la mémoire. On a donc affaire à des segments de la mémoire de la machine, qui agissent les uns sur les autres (le moteur de cette action étant le processeur de l'ordinateur).

Pour décrire mathématiquement cette situation, il nous faut construire un univers dans lequel les objets représentent des fonctions dont les arguments et les valeurs sont aussi ces fonctions; autrement dit, un univers de fonctions agissant sur elles-mêmes.

Une telle structure a été inventée par A. Church dans les années 1930, c'est-à-dire bien avant l'avènement des ordinateurs. Le λ-calcul, comme il l'a appelée, est resté longtemps une théorie tout à fait confidentielle et plutôt ésotérique. On comprend facilement, d'après tout ce qui vient d'être dit, qu'elle occupe maintenant une place centrale à la fois en théorie des programmes (c'est-à-dire en informatique) et en logique (en théorie de la démonstration).

Je vais essayer d'en donner une brève description. C'est une structure munie de deux opérations, l'une très simple, l'autre plus subtile. Le λ-calcul sera alors la structure la plus simple possible (la structure libre, si l'on veut) où ces deux opérations sont définies.

La première opération est appelée l'*application*, et son sens intuitif est très simple: si j'ai une fonction f et un argument g (qui, comme on l'a vu est aussi une fonction), alors je peux appliquer f à g et j'obtiens $f(g)$ (qui est aussi une fonction).

On se donne donc, sur la structure considérée, une opération binaire, qu'on appelle *application*, et sur laquelle aucun axiome n'est postulé!

Maintenant, il y a une autre opération, un peu plus difficile à saisir intuitivement, comme son nom l'indique d'ailleurs, car elle est appelée *abstraction*. On peut indiquer approximativement son sens intuitif en disant que c'est la transformation d'une formule de calcul (comme celle des fonctions à la Euler) en objet mathématique. C'est ce qu'on fait, tout à fait couramment, en mathématiques, quand on dit: "Considérons la fonction définie par la formule e^{x+x}, ou $1/(1+x^2)$, ..."; ou encore: "Considérons la fonction $x \mapsto e^{x+x}$ ". A. Church a introduit la notation $\lambda x e^{x+x}$ pour désigner cette fonction.

Plus précisément, le sens intuitif de cette opération d'abstraction est le suivant: elle consiste à remplacer la fonction considérée par son nom, ou encore sa référence. Par exemple arctan est le nom de la fonction définie par $x \mapsto \int_0^x \frac{dt}{1+t^2}$. Cela veut donc dire que $\lambda x \int_0^x \frac{dt}{1+t^2}$ est "arctan". Si une fonction $\phi(x)$ est définie dans un texte mathématique sous la référence **Définition 25**, cela veut dire que $\lambda x \phi(x)$ est "Définition 25".

Cette opération n'est, évidemment, pas toujours possible pour les fonctions mathématiques, car elles n'ont pas toutes un nom, ou une référence où elles ont été définies. Par contre, elle est absolument indispensable si on veut pouvoir considérer les fonctions comme des programmes, c'est-à-dire si on se limite à ne considérer que des fonctions qui sont des programmes. En fait, l'opération d'abstraction correspond à des notions tout à fait élémentaires et fondamentales de programmation, de celles qu'on apprend dans les premiers cours d'informatique: ce sont les notions d'*adresse* et de *pointeur*.

Rappelons brièvement de quoi il s'agit: la mémoire d'un ordinateur est divisée en petites cases, qui contiennent chacune un mot, et qui ont chacune un numéro, qu'on appelle leur adresse. Chaque case a donc une adresse fixe, et un contenu variable. Or, le contenu d'une case a peut tout à fait être l'adresse d'une case b. La case mémoire a est alors appelée *pointeur* sur b.

Considérons alors un programme, que je désigne par $P(x)$, pour dire qu'il opère sur une variable x, qui n'est pas autre chose qu'une case mémoire. Ce programme occupe une zone mémoire, en général très grande, c'est-à-dire une longue suite de cases. S'il doit servir d'*argument* à un autre programme $Q(y)$, il faut pouvoir le désigner par une quantité qui tienne dans une seule case mémoire, c'est-à-dire par une adresse (qui est, par exemple, mais pas toujours, l'adresse du début de la zone mémoire occupée par le programme $P(x)$). Cette adresse sera notée $\lambda x P(x)$ qui se peut alors se lire "adresse du programme P dépendant de la variable x ". On peut alors mettre cette adresse dans la case mémoire y qui devient ainsi un pointeur sur le programme $P(x)$. Il ne reste plus alors qu'à lancer le programme $Q(y)$.

Revenons au λ-calcul comme structure mathématique, qui est donc défini par ces deux opérations d'application et d'abstraction. C'est donc un objet mathématique très simple et très élémentaire à définir, mais, par contre, très difficile à étudier, et qui pose de redoutables problèmes aux mathématiciens qui s'en occupent. Il ressemble beaucoup, en cela, à l'ensemble $\mathbf{N}$ des entiers naturels, et il est d'ailleurs tout aussi passionnant.

Les éléments de cette structure sont appelés les termes du λ-calcul, ou encore λ-termes. Par exemple, $\lambda x\, x$ (l'application identique), ou $\lambda x\, x(x)$ (l'opération d'appliquer un programme à lui-même) sont des λ-termes.

Puisque les termes du λ-calcul représentent des programmes, il doit y avoir une opération qui représente l'exécution d'un programme, autrement dit le calcul de la fonction pour une valeur donnée de l'argument. C'est ce qu'on appelle la β-réduction. Elle consiste, par exemple, à transformer $(\lambda x(x+x))(2)$ en $2+2$. En général, elle fait passer de $(\lambda x\, P(x))(A)$ à $P(A)$. Cela paraît tout à fait trivial, mais cela correspond, dans l'ordinateur, à l'opération que j'ai expliquée il y a quelques instants: pour appliquer le programme d'adresse $\lambda x P(x)$ à celui d'adresse A, on met dans la case mémoire x l'adresse

A (on transforme x en un pointeur sur A) et on exécute le programme P.

L'exécution d'un λ-terme consistera donc à effectuer des β-réductions, jusqu'à qu'il n'y en ait plus aucune de possible. On dit alors que le terme est normal. Par exemple, $(\lambda x\, xx)(\lambda x\, x)$ donne $(\lambda x\, x)(\lambda x\, x)$ puis $\lambda x\, x$ et on s'arrête là.

Or, en programmation, on écrit souvent des programmes qui ne s'arrêtent pas, par exemple des boucles infinies. Cette situation se retrouve dans le λ-calcul, où l'exemple le plus simple de boucle infinie est donné par $(\lambda x\, xx)(\lambda x\, xx)$. On voit aussi que l'on retrouve ici une expression qui ressemble fort aux séries ou intégrales divergentes que manipulait Euler.

Pour pouvoir, dans cette structure, calculer sur les entiers, il faut les représenter par des λ-termes. A. Church, l'inventeur du λ-calcul, a trouvé une façon naturelle de le faire. L'idée de Church pour cela est simple: si l'entier 3 doit être un λ-terme, c'est-à-dire un programme, ce doit être le programme qui prend une fonction et l'itère trois fois, c'est-à-dire la compose trois fois avec elle-même. Autrement dit, on doit écrire $3 = \lambda f \lambda x\, f(f(f(x)))$. Même chose, bien sûr, pour n'importe quel entier. On a ainsi défini ce qu'on appelle les *entiers de Church*.

Remarque. On voit particulièrement bien ici, comment la notion de programme peut être utilisée comme concept primitif servant à définir tous les autres: un entier est défini comme un programme qui se comporte de telle et telle façon vis-à-vis des autres programmes.

Grâce à cette représentation, on peut programmer, dans le λ-calcul, certaines fonctions d'entiers dans les entiers. Par exemple, l'addition, qui est $\lambda m \lambda n \lambda f \lambda x\, m(f)(n(f)(x))$, ou la multiplication $\lambda m \lambda n \lambda f\, m(n(f))$. Et l'un des premiers résultats établis par Church et Kleene, est que les fonctions d'entiers que l'on peut programmer dans ce langage sont toutes les fonctions récursives, c'est-à-dire toutes les fonctions d'entiers programmables sur machine. Autrement dit, le λ-calcul est, malgré sa simplicité, un langage de programmation universel. On est d'ailleurs très vite passé de la théorie à la pratique, et l'un des premiers langages de programmation, LISP, est directement inspiré du λ-calcul.

Mais revenons à la correspondance de Curry-Howard entre preuves et programmes. Nous allons pouvoir l'expliciter un peu mieux, maintenant que nous avons, avec le λ-calcul, une représentation mathématique des fonctions comme programmes. A chaque démonstration mathématique, effectuée dans un système formel convenable, on va faire correspondre un programme, sous la forme d'un terme du λ-calcul.

Pour cela, à chaque pas de la démonstration, on assemble des pièces détachées, qui, à la fin, constituent le programme cherché. Par exemple, supposons qu'à un moment donné dans la démonstration, on applique la règle du modus ponens: de A et de $A \to B$, déduire B. Cela veut dire qu'on a déjà démontré A, donc obtenu un programme (λ-terme) t_A, et aussi démontré $A \to B$, donc obtenu un autre programme $u_{A\to B}$. Alors le terme v_B que l'on construit à ce stade de la démonstration est $u(t)$. On voit ainsi que la règle de déduction essentielle de la logique, qui est le modus ponens, est associée à l'opération d'appliquer une fonction à son argument.

Toutes les autres règles de démonstration sont traitées d'une façon analogue, mais je ne le ferai pas, faute de temps, et pour éviter de trop entrer dans la technique.

Quand on a un théorème T, les programmes qui correspondent aux diverses démonstrations de T sont appelés programmes *de type* T. Il y a donc une infinité de programmes d'un type donné, ce type étant une formule qui est un théorème.

Intuitivement, le type d'un programme est, en quelque sorte, sa spécification c'est-à-dire ce qu'il fait, son but, ce pour quoi il a été écrit. On comprend qu'il y ait beaucoup de programmes différents pour réaliser le même but.

Ce qui est, en fait, tout à fait remarquable, c'est qu'on ait trouvé ainsi une approche mathématique de cette notion de spécification, qui semble pourtant particulièrement difficile à cerner avec précision (puisqu'il s'agit de définir ce qu'est l'*intention* du programmeur). Ajoutons qu'on est là dans un champ immense pour les applications pratiques, puisque la question de la conformité d'un programme à ses spécifications est un des problèmes clés de l'industrie informatique. Les déboires du système SOCRATE de réservation de la S.N.C.F. sont un bon exemple de ce qui se passe quand ce problème n'est pas résolu correctement.

Si nous reprenons les deux exemples simples que nous avons examinés précédemment, à savoir les deux "théorèmes" $A \to A$ et $(A \to B) \to ((B \to C) \to (A \to C))$, nous voyons que $\lambda x\, x$ ou $(\lambda x\, x)(\lambda x\, x)$ sont des exemples de programmes de type $A \to A$; $\lambda f \lambda g \lambda x\, g(f(x))$, qui est l'opérateur de composition des fonctions, est un exemple de programme de type $(A \to B) \to ((B \to C) \to (A \to C))$.

Pour donner un autre exemple, moins trivial, considérons un théorème d'arithmétique comme: "Il existe une infinité de nombres premiers". A chaque preuve de ce théorème va correspondre un programme, et tous ces programmes ont le même type, qui est ce théorème lui-même. Il est facile de deviner ce que font tous ces programmes: ce sont des fonctions qui, pour chaque entier n, fournissent un nombre premier $p > n$. Voilà donc la spécification associée à ce théorème de théorie des nombres.

Nous voyons donc que, maintenant, chaque théorème de mathématiques a pris une nouvelle signification: outre son sens habituel, celui que lui donnent tous les mathématiciens qui s'en servent, il a un sens comme type, ou spécification de programmes. Pour certains théorèmes, en gros ceux que l'on appelle constructifs, cette nouvelle signification est claire, et très proche du sens habituel. C'est le cas des exemples que nous avons donnés. Or, ce sont des théorèmes que l'on peut prouver dans une logique plus faible que la logique classique, qu'on appelle logique intuitionniste (introduite par Brouwer), dans laquelle on interdit le raisonnement par l'absurde, ou le tiers exclu (A ou non A). Elle a, en gros, la propriété suivante: si on démontre, dans cette logique, qu'il existe un objet ayant une certaine propriété, on donne, en même temps, un moyen de construire un tel objet.

C'est pourquoi on a cru pendant longtemps que seules les démonstrations constructives pouvaient donner lieu à des programmes, c'est-à-dire que la correspondance de Curry-Howard était limitée à la logique intuitionniste. Mais, depuis quelques années, on s'est aperçu que ce n'était pas le cas, et que le raisonnement par l'absurde correspondait à

des méthodes de programmation utilisées depuis fort longtemps. C'est une remarquable découverte qui a été faite par Felleisen et Griffin, qui sont, il faut le noter, deux informaticiens.

Toutefois, lorsqu'il s'agit de théorèmes non constructifs, on ne comprend pas, en général, quel est le type qui leur est associé, et il y a là un problème extrêmement intéressant, parce qu'encore tout à fait mystérieux: la recherche du sens caché des théorèmes!

Cependant, on avance dans cette direction, et la découverte même de Felleisen et Griffin en est un exemple: ce qu'ils ont trouvé, en effet, ce n'est pas autre chose que la signification informatique des théorèmes non constructifs les plus simples, à savoir le tiers exclu: A ou non A, et le raisonnement par l'absurde: (non non A) $\rightarrow A$. Leur idée est tout à fait étonnante, et il faut un certain temps de réflexion pour se convaincre qu'elle est correcte: les instructions qu'ils associent à ces deux théorèmes sont ce qu'on appelle, en informatique, les instructions d'échappement, qui ont été inventées par les programmeurs pour le traitement des exceptions et des erreurs. Vous les voyez en action, chaque fois que l'ordinateur émet une protestation, c'est-à-dire qu'il affiche un message d'erreur, généralement accompagné d'un bip, par exemple, parce que vous lui demandez de lire une disquette que vous n'avez pas mise dans le lecteur. Avouez qu'il n'était pas évident, a priori, que cette situation avait quelque chose à voir avec le raisonnement par l'absurde!

La recherche sur la correspondance de Curry-Howard entre preuves et programmes, est donc très active (et ce d'autant plus qu'elle n'est pas seulement motivée par ses implications philosophiques, mais aussi, et peut-être surtout, par les applications industrielles dont j'ai parlé tout à l'heure). On est en train, petit à petit, de construire un véritable "dictionnaire" dans lequel chaque notion de programmation a une traduction en théorie de la démonstration, et vice-versa. C'est une situation tout à fait remarquable, dont on a déjà eu un exemple dans l'histoire des mathématiques: lors de la découverte, par Kolmogoroff, de l'axiomatique des probabilités à l'aide de la théorie de la mesure. Chaque notion probabiliste a trouvé alors sa traduction en une notion de théorie de la mesure. On obtient, par exemple:

Probabilité	$\Leftrightarrow$	Mesure
Evènement	$\Leftrightarrow$	Ensemble mesurable
Variable aléatoire	$\Leftrightarrow$	Fonction mesurable
Espérance ou valeur moyenne	$\Leftrightarrow$	Intégrale
Indépendance	$\Leftrightarrow$	Produit d'espaces mesurés
Espérance conditionnelle	$\Leftrightarrow$	Théorème de Radon-Nikodym

Donnons un aperçu de l'état actuel du "dictionnaire" pour la correspondance de Curry-Howard. Comme je viens de le dire, certaines de ces équivalences n'ont été obtenues que très récemment. Elles sont toutes assez surprenantes, et inattendues a priori:

Théorie de la démonstration		*Programmation*
Règle logique (règle de déduction)	$\Leftrightarrow$	Instruction
Preuve	$\Leftrightarrow$	Programme
Axiome, hypothèse	$\Leftrightarrow$	Déclaration de variable
(Preuve d'un) lemme	$\Leftrightarrow$	Procédure, fonction
Théorème (conclusion d'une preuve)	$\Leftrightarrow$	Type, spécification (d'un programme)
Raisonnement par récurrence	$\Leftrightarrow$	Boucle "for"
Réduction d'une preuve (élimination des coupures)	$\Leftrightarrow$	Exécution d'un programme
Négation	$\Leftrightarrow$	Continuation
Raisonnement par l'absurde	$\Leftrightarrow$	Instruction d'échappement (ou de contrôle) par exemple pour le traitement des erreurs
$\perp$ (Faux)	$\Leftrightarrow$	Type des programmes exécutables ou encore du "top-level"

En conclusion, on peut dire qu'on assiste actuellement à l'émergence d'un domaine tout à fait fascinant, où les concepts de fonction et de programme jouent un rôle clé. Il est clair qu'on tient là un fil conducteur extrêmement solide, qui est en train de nous mener à une compréhension en profondeur des mécanismes et de la *nature même* du raisonnement mathématique. Un autre trait étonnant de ce domaine est que, malgré son caractère extrêmement abstrait (quand même, il ne s'agit de rien de moins que l'analyse du raisonnement!), il soit en prise directe avec les applications. Il faut d'ailleurs remarquer que certaines des idées et des intuitions essentielles qui permettent cette analyse, nous viennent, non pas des mathématiques ou de la logique, mais directement de la programmation et de l'informatique.

Livres sur le sujet.

H. Barendregt. The lambda-calculus. North Holland Pub. Co.
J.Y. Girard, Y. Lafont, P. Taylor. Proofs and types. Cambridge Univ. Press
J.L. Krivine. Lambda-calcul, types et modèles. Masson.