Le langage mathématique et les langages de programmation

Gilles Dowek

Voir, Entendre, Raisonner, Calculer
Cité des sciences et de l'industrie
10-13 Juin 1997

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

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

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

Qu'est-ce qu'un programme ?

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

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

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

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

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

Les fonctions dans le langage mathématique

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

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

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

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

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

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

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

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

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

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

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

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

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

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

– On donne les propriétés que la fonction doit vérifier :

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

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

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

La définition de la racine carrée

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

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

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

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

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

La fonction *puissance* se définit donc à présent ainsi

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

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

Quand plusieurs objets vérifient la propriété utilisée, différentes conventions peuvent être adoptées. L'utilisation de l'opérateur de descriptions est en général prohibée dans ce cas, mais une règle plus libérale autorise cette utilisation et permet, par exemple, la formation de l'expression $[x \mid x \times x = 9]$ qu'on ne lit plus dans ce cas "le nombre x tel que $x \times x = 9$" mais plutôt "un nombre x tel que $x \times x = 9$". L'opérateur de descriptions est alors appelé *opérateur de choix*, *opérateur ε de Hilbert* ou *opérateur τ de Bourbaki*. Le nombre $[x \mid x \times x = 9]$ vaut ou bien 3 ou bien -3, sans qu'on sache s'il vaut 3 ou -3. Ainsi, on ne peut pas démontrer $[x \mid x \times x = 9] = 3$ ni $[x \mid x \times x = 9] = -3$ mais on peut démontrer $[x \mid x \times x = 9] \in \{3, -3\}$.

Les spécifications et les programmes

Quand un client commande un programme à un informaticien, il doit lui expliquer ce que doit faire ce programme. Cette explication s'appelle le *cahier des charges* ou encore la *spécification* du programme. Par

exemple, quand un industriel commande à une société de service un programme d'inversion de matrices, la spécification tient en une ligne

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

alors que le programme peut être beaucoup plus long.

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

L'exécution des programmes

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

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

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

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

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

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

est également vraie, et

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

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

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

Il nous faut donc concevoir un langage mathématique qui permette non seulement de démontrer que $12 + 23 = 35$ mais aussi de transformer l'expression $12 + 23$ en l'expression 35. La grammaire du langage

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

Quelques langages de programmation

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

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

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

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

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

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

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

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

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

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

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

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

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

À la différence du langage ML qui utilise l'opérateur de descriptions pour définir la fonction elle même, le langage Prolog (conçu en 1973 par Alain Colmerauer et son équipe) utilise l'opérateur de descriptions

uniquement pour définir la valeur prise par la fonction, c'est-à-dire la valeur de sortie du programme. Par exemple la fonction *puissance* n'est plus définie par un terme de la forme $[f \mid P(f)]$ où P est une propriété caractéristique de cette fonction, mais par un terme de la forme $x, n \mapsto [y \mid Q(x, n, y)]$ où Q est une propriété caractéristique du nombre x^n. La propriété $Q(x, n, y)$ peut être par exemple :

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

qui mène au programme Prolog

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

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

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

Les fonctions non calculables

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

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

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

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

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

Au début du vingtième siècle, une école de mathématiciens, appelés *intuitionnistes* ou *constructivistes*, a vivement critiqué cette règle de raisonnement. Comment peut-on prétendre savoir que la propositionă"P ou non P" est vraie, si on ne sait ni que "P" est vraie, ni que "non P" est vraie ? De même, comment peut-on prétendre avoir défini le nombre $h(p)$ si on ne sait même pas si ce nombre vaut 0 ou 1 ? Pour les intuitionnistes la définition de la fonction h ci-dessus est tout simplement incorrecte. Derrière Luitzen Egbertus Jan Brouwer, le chef de file de l'école intuitionniste, se sont ralliés des mathématiciens importants comme Hermann Weyl ou Andreï Kolmogorov.

Un théorème dû à Stephen Kleene montre que, quand on abandonne le tiers exclu, toutes les fonctions qu'on peut définir avec l'opérateur de descriptions sont calculables. Ce théorème prend naturellement des formes différentes en fonction de la théorie dans laquelle on se place pour démontrer l'existence et l'unicité de la fonction décrite. Autrement dit, le tiers exclu est indispensable pour montrer l'existence d'une fonction non calculable. Quand on abandonne le tiers exclu, il est donc possible de trouver des règles de calcul

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

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

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

Un exemple de raisonnement non constructif

pour l'opérateur de descriptions dans toute sa généralité. Quand on définit une fonction $[f \mid P(f)]$ après avoir donné une démonstration constructive p de l'existence d'une fonction vérifiant la propriété P, on peut exécuter ce programme. Pour cela on applique à la démonstration p et à la valeur d'entrée une transformation appelée l'*élimination des coupures*. Le premier procédé d'élimination des coupures a été proposé en 1934 par Gerhard Gentzen pour les démonstrations exprimées dans l'arithmétique. Depuis, il a été généralisé à bien d'autres théories. En particulier, Jean-Yves Girard a proposé en 1971 un procédé d'élimination des coupures pour l'arithmétique d'ordre supérieur, qui est une variante de la théorie des ensembles.

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

Les démonstrations comme objets

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

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

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

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

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

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

La gestion des ressources

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

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

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

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

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

Un outil essentiel pour résoudre ces questions est sans doute la *logique linéaire*, développée par Jean-Yves Girard en 1987. Selon cette théorie, les mathématiques ne sont pas très conscientes de la gestion des ressources parce que "les hypothèses ne s'usent pas quand on s'en sert". Ainsi dans les logiques traditionnelles les propositions "P" et "P et P" sont équivalentes. En logique linéaire ces deux propositions ne sont plus équivalentes, supposer la première permet d'utiliser une unique fois l'hypothèse P dans une démonstration alors que supposer la seconde permet d'utiliser deux fois cette hypothèse. Dans la démonstration de l'existence

d'une fonction de tri, une hypothèse utilisée une unique fois révèle qu'une fois la liste utilisée, elle peut être détruite, ce qui est exactement l'information nécessaire pour programmer le tri en place. La logique linéaire a déjà été utilisée pour gérer les ressources dans des compilateurs pour le langage ML. Ce type d'analyse est à étendre au cas de la programmation en langage mathématique.

Réactions sur le langage mathématique

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

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

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

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

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

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

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

Références

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

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

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

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

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