Chapitre 8

Peut-on tout programmer ?

8.1 Que peut-on programmer ?

Vous voici au terme de votre initiation à la programmation. Vous avez vu comment représenter des données de plus en plus structurées à partir de données élémentaires comme les nombres, les booléens et les caractères. Vous avez vu comment mettre en œuvre des algorithmes plus ou moins sophistiqués. L'approche a été essentiellement celle de la programmation dite *impérative* dans laquelle la notion principale est celle d'état d'une machine que l'on transforme par des instructions que l'on peut composer en séquences, en instructions conditionnelles, en itération conditionnelle ou non. Mais vous avons aussi flirté avec la programmation dite *fonctionnelle* dans laquelle les notions principales sont celles de fonction, de composition de fonctions et de récursivité. Sans oublier bien sûr la programmation *modulaire* qui permet de la programmation séparée et la réutilisation facile de composants logiciels. La programmation *orientée objet* que vous aborderez prochainement si vous poursuivez en informatique sera pour vous l'occasion d'aller plus loin encore dans cette voie.

Mais il est peut-être temps de vous poser quelques questions : Que peut-on programmer ? Est-il possible de tout programmer ? Toute tâche est-elle programmable ? Suffit-il de donner la spécification d'une fonction ou d'une procédure pour que vous puissiez la réaliser à l'aide d'un algorithme adéquat ?

8.2 Qu'est-ce qui est programmable ?

Pour aborder cette question, nous allons nous restreindre à des fonctions à un paramètre entier et à valeurs entières, autrement dit des fonctions du type

$$f \ : \ \mathbb{N} \rightarrow \mathbb{N}.$$

Certaines de ces fonctions sont programmables, d'autres peut-être pas. Les fonctions programmables sont aussi appelées des fonctions *calculables*. Une fonction non calculable est donc une fonction f pour laquelle aucun programme P n'existe qui donne en sortie la valeur $f(n)$ lorsqu'on lui donne en l'entier n en entrée, et ceci pour tout $n \in \mathbb{N}$.

Nous connaissons de très nombreuses fonctions calculables. En particulier toutes celles que nous avons programmées ! La plupart des fonctions que l'on rencontre naturellement en mathématiques ou dans d'autres domaines le sont.

Existe-t-il des fonctions non calculables ? Il n'est pas facile de dénicher de telles fonctions si elles existent. Cependant, nous allons voir qu'un simple argument de dénombrement va révéler qu'elles sont bien plus nombreuses que les fonctions calculables.

8.2.1 Nombre de programmes

Commençons par déterminer le nombre de programmes de type `int → int` syntaxiquement corrects qu'il est possible d'écrire en Caml [1].

Il est clair qu'il y en a une infinité. Par exemple, rien que pour programmer la fonction constante égale à 0, nous avons une infinité de programmes possibles. Par exemple, les programmes de la forme

```
let f n =  n - n + x - x
```

où x est à remplacer par n'importe quel entier.

On peut préciser l'infinité des programmes syntaxiquement corrects du type souhaité en Caml : c'est la même que celle des entiers naturels, à savoir l'infini dénombrable. Autrement dit, il est possible de numéroter tous ces programmes à l'aide des entiers naturels.

Pour s'en convaincre, il suffit de classer les programmes par tailles croissantes, la taille d'un programme étant le nombre de caractères pour l'écrire, puis au sein d'une classe de programmes de taille donnée, il suffit de les trier par ordre lexicographique. Ceci étant fait il suffit de numéroter les programmes en commençant par les plus courts et en allant vers les plus longs, et pour une longueur donnée, en suivant l'ordre lexicographique. De cette façon tout programme se verra attribuer un numéro qui sera un nombre entier, et puisque ces programmes sont en nombre infini, tout nombre entier est le numéro d'un programme.

Le programme P_0 est ainsi un programme en Caml de type `int → int` et de longueur minimale, et pour cette longueur le premier dans l'ordre lexicographique. P_1 est le programme qui suit ...

Il y a donc autant de programmes qu'il y a de nombres entiers. En désignant par $\mathcal{P}$ l'ensemble des programmes Caml de type `int → int`, on a donc

$$\mathrm{card}(\mathcal{P}) = \mathrm{card}(\mathbb{N}) = \aleph_0,$$

où $\aleph_0$ désigne l'infini dénombrable.

8.2.2 Nombre de fonctions calculables

Étudions maintenant le nombre de fonctions calculables.

Il est clair qu'il y en a une infinité. Il suffit de considérer l'infinité des fonctions constantes.

De plus, il ne peut pas y avoir plus de fonctions calculables qu'il n'y a de programmes, puisqu'à chaque fonction calculable lui correspond par définition un programme. Donc l'infinité des fonctions calculables n'est pas plus grande que celle des programmes.

Et comme l'infini dénombrable est le plus petit infini, on en déduit que l'ensemble $\mathcal{F}_c$ des fonctions calculables est infini dénombrable.

$$\mathrm{card}(\mathcal{F}_c) = \mathrm{card}(\mathbb{N}) = \aleph_0.$$

1. Nous négligeons ici le fait que l'ensemble des `int` de Caml n'est pas égal à $\mathbb{N}$ (certains `int` sont négatifs, et les `int` sont en nombre fini).

8.2.3 Nombre de fonctions de $\mathbb{N}$ dans $\mathbb{N}$

Étudions enfin le nombre de fonctions de $\mathbb{N}$ dans $\mathbb{N}$.

Il y en a évidemment une infinité. Mais est-ce la même infinité que celle des fonctions calculables ?

Nous allons restreindre un peu les fonctions considérées ici pour ne dénombrer que celles qui ne prennent que les deux valeurs 0 et 1. Nommons $\mathcal{F}_{0,1}$ l'ensemble de ces fonctions.

À chaque partie A de $\mathbb{N}$, on peut associer sa fonction indicatrice définie par

$$\begin{array}{rcl} \mathtt{f_A} : \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \longmapsto & \left\{ \begin{array}{ll} 0 & \text{si} \quad n \notin A \\ 1 & \text{si} \quad n \in A \end{array} \right. \end{array}.$$

On définit ainsi une application Φ de l'ensemble $\mathcal{P}(\mathbb{N})$ des parties de $\mathbb{N}$ dans l'ensemble $\mathcal{F}_{0,1}$

$$\begin{array}{rcl} \Phi : \mathcal{P}(\mathbb{N}) & \longrightarrow & \mathcal{F}_{0,1} \\ A & \longmapsto & f_A \end{array}.$$

Cette fonction Φ est injective et surjective. Elle établit donc une bijection entre les ensembles $\mathcal{P}(\mathbb{N})$ et $\mathcal{F}_{0,1}$, et donc les cardinaux de ces deux ensembles sont égaux

$$\text{card}(\mathcal{P}(\mathbb{N})) = \text{card}(\mathcal{F}_{0,1}).$$

Or un théorème de Cantor établit que pour tout ensemble E, le cardinal de l'ensemble de ses parties est strictement plus grand que le sien

$$\text{card}(\mathcal{P}(E)) > \text{card}(E).$$

On en déduit que

$$\text{card}(\mathcal{F}_{0,1}) > \text{card}(\mathbb{N}),$$

et comme $\mathcal{F}_{0,1} \subset \mathcal{F}$, on en déduit que

$$\text{card}(\mathcal{F}) > \text{card}(\mathbb{N}).$$

Autrement dit l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ n'est pas dénombrable (il n'est pas possible de numéroter à l'aide des nombres entiers toutes ces fonctions).

8.2.4 Conclusion

D'une part nous avons vu qu'il y a une infinité dénombrable de programmes et donc de fonctions calculables. D'autre part il y a une infinité non dénombrable de fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Il existe donc des fonctions non calculables.

On peut même ajouter que dans un certain sens les fonctions non calculables sont infiniment plus nombreuses que les fonctions calculables. Les fonctions non calculables sont la règle générale, et les fonctions calculables l'exception.

Les fonctions calculables dont nous avons parlé ici sont les fonctions calculables en CAML. Mais bien entendu, nous pourrions tenir le même raisonnement avec n'importe quel autre langage informatique. Ce n'est donc pas une limitation quelconque du pouvoir d'expression du langage CAML que nous constatons ici, mais c'est une règle générale : quelque soit le langage utilisé, il existe des fonctions non calculables.

8.3 L'arrêt des programmes

Vous venez de voir que la non calculabilité est la règle générale. Presque toutes les fonctions sont non calculables. Mais pouvons-vous en citer au moins une ?

Toutes les fonctions classiques que vous avez rencontrées dans vos études antérieures (en maths, en physique, en économie, ...) sont des fonctions que vous pouvez programmer à partir des quatre opérations arithmétiques élémentaires, et des structures de contrôle usuelles : instructions conditionnelles, itérations, récursivité.

8.3.1 Le problème de l'arrêt des programmes

Quel programmeur ne s'est jamais interrogé face à une exécution interminable d'un programme qu'il vient de mettre au point ? Le programme boucle-t-il indéfiniment parce qu'il a mal été pensé ? ou bien le calcul en cours est-il vraiment très long ?

Dans le premier cas, boucle infinie, le programme ne s'arrêtera jamais, et il est nécessaire d'en interrompre l'exécution pour corriger l'erreur de mise au point qui s'y est manifestement logée.

Dans le second cas, il suffit d'être patient, le calcul finira bien par s'arrêter et donner la réponse attendue.

Mais en attendant quelle attitude adopter ? Arrêter le calcul ou le laisser travailler ?

Pour l'instant, laissons le calcul se poursuivre et réflechissons sur le problème de l'arrêt de certains programmes exemples.

Commençons par la classique fonction factorielle proposée en exercice à tout programmeur débutant. Cette fonction s'écrit simplement de manière itérative à l'aide d'une boucle pour

```
let factorielle n =
  let f = ref 1 in
    for k = 1 to n do
      f := !f * k
    done ;
    !f
```

ou bien de manière récursive.

```
let rec factorielle n =
  if n = 0 then
    1
  else
    n * factorielle (n - 1)
```

Dans les deux cas, il n'est pas difficile de « prouver » que quelque soit l'entier n passé en paramètre, pourvu qu'il soit positif, le calcul de l'expression `factorielle n` se termine. Dans la version itérative, c'est une quasi évidence puisqu'on a une simple boucle pour et par conséquent le nombre d'étapes de calculs se lit sur l'intervalle que parcourt l'indice de boucle. Dans la version récursive, une récurrence sur n établit aussi le résultat.

Envisageons un programme un peu moins facile : le calcul du pgcd de deux entiers par l'algorithme d'Euclide à l'aide de divisions successives. La version itérative utilise une boucle tant que

```
let pgcd a b =
  l  et a = ref a
```

```
  and   b = ref b
    and r = ref (a mod b) in
      while !r <> 0 do
        a := !b ;
        b := !r ;
        r := !a mod !b
      done ;
      !b
```

et on peut aussi en donner une programmation récursive

```
let rec pgcd a b =
  if b = 0 then
    a
  else
    pgcd b (a mod b)
```

Cette fois, l'arrêt du programme est moins évidente, et il faut recourir d'une part au fait que dans une division euclidienne le reste est toujours strictement plus petit que le diviseur, et d'autre part à l'inexistence de suites infinies d'entiers positifs strictement décroissantes pour conclure à l'arrêt de ce programme pour toutes valeurs de a et b.

Envisageons pour terminer une fonction simple à programmer.

```
let rec collatz n =
  if (abs n) <= 1 then
    1
  else if n mod 2 = 0 then
    collatz (n / 2)
  else
    collatz (3 * n + 1)
```

Cette fonction est manifestement une fonction de type `int → int`, et la seule valeur qu'elle peut renvoyer est l'entier 1. La question qui se pose est donc de savoir si elle renvoie l'entier 1 pour toute valeur de son paramètre. À cette question, personne ne connaît aujourd'hui la réponse. De très nombreuses explorations ont été menées, et tous les entiers sur lesquel on a testé cette fonction ont donné une réponse positive. C'est pourquoi on conjecture qu'il en est ainsi pour tous les entiers. C'est la conjecture de Collatz.

8.3.2 Une fonction programmable ?

Est-il possible d'écrire un programme qui teste si un autre programme s'arrête sur toutes les données qu'on lui fournit ? Autrement dit, existe-t-il une fonction qui prend une fonction `f` de type `'a → 'b` en paramètre et qui renvoie le booléen **`true`** si pour la valeur `x` de type `'a` le calcul de l'expression `f x` s'arrête, et renvoie le booléen **`false`** dans le cas contraire.

Appelons `arret_garanti` une telle fonction.

`arret_garanti : ('a -> 'b) -> 'a -> bool`.

8.3.3 Une fonction paradoxale

Supposons que nous ayions programmé la fonction `arret_garanti`. Alors nous pouvons l'utiliser pour programmer d'autres fonctions. Par exemple la fonction ci-dessous.

```
let rec une_fonction () =
  if arret_garanti une_fonction () then
    une_fonction ()
  else
    ()
```

La lecture du code de cette fonction `une_fonction` montre que sa définition est récursive et que son type est `unit` $\rightarrow$ `unit`. De plus aucun effet de bord (transformation de l'état de la mémoire) ne peut être provoqué par un appel à elle.

Sa définition est une simple expression conditionnelle dont la condition est une application de notre fonction `arret_garanti`, que nous avons supposé être réalisée, à la fonction `une_fonction` et l'unique valeur `()` qu'elle peut prendre en paramètre. Ainsi la condition de cette expression conditionnelle est satisfaite si tout appel à la fonction `une_fonction` lance un calcul qui s'arrête en un nombre fini d'étapes, et elle ne l'est pas dans le cas contraire.

Alors que donne l'évaluation d'un appel à la fonction `une_fonction` ? De deux choses l'une : ou bien cette évaluation s'arrête ou bien elle ne s'arrête pas. Étudions ces deux cas.

1. Supposons que l'évaluation de l'expression `une_fonction ()` s'arrête. Alors la condition de la conditionnelle est satisfaite et l'évaluation de la même expression est lancée récursivement. Ainsi l'évaluation de cette expression est infinie. Et nous somme amenés à en conclure que si l'expression `une_fonction ()` lance un calcul qui s'arrête, alors il est infini, autrement dit il ne s'arrête pas ! Notre hypothèse de départ est donc impossible. L'évaluation de `une_fonction ()` ne peut qu'être infinie.
2. L'évaluation de `une_fonction ()` est donc infinie. Autrement dit, conformément à la spécification de la fonction `arret_garanti`, l'expression booléenne

   `arret_garanti une_fonction ()`

   ne peut prendre que la valeur **`false`**, et ainsi l'évaluation de `une_fonction ()` donne la valeur `()` est s'arrête. Cette fois-ci nous sommes amenés à en conclure que le non arrêt du calcul de `une_fonction ()` implique son arrêt.

Les deux cas que nous venons d'envisager nous obligent à conclure que

> le calcul de `une_fonction ()` s'arrête si et seulement s'il ne s'arrête pas.

Ce qui est évidemment une contradiction logique que nous ne pouvons pas admettre.

Alors ? La seule conclusion que la logique nous impose est que la réalisation algorithmique de la fonction `arret_garanti` est impossible.

Il existe donc des fonctions parfaitement spécifiables qui n'ont aucune solution algorithmique. Tout n'est donc pas programmable.

8.4 D'autres problèmes indécidables

8.4.1 Égalité des programmes

Un enseignant d'informatique souhaite automatiser la correction des travaux réalisés par ses étudiants. Le travail qu'il donne à ses étudiants consiste souvent à réaliser un programme pour une certaine fonction $f : A \rightarrow B$.

Pour cela, il réalise le programme suivant en Caml :

```
let corrige solution_prof solution_etudiant =
  if solution_etudiant = solution_prof then
```

```
    20 / 20
  else
    0 / 20
```

programme qui renvoie la valeur 1 si le programme que l'étudiant a écrit pour réaliser la fonction f est égal à celui que le professeur a réalisé (qui, bien entendu, est conforme à la spécification de f), et la valeur 0 dans le cas contraire.

Par égalité des deux programmes, nous ne voulons pas dire qu'ils sont syntaxiquement égaux (cela ressemblerait étrangement à un plagiat), ni même qu'ils suivent le même algorithme pour calculer les valeurs de f. Nous voulons seulement dire que les deux programmes produisent les mêmes sorties $f(x)$ pour toutes les entrées x pour lesquelles $f(x)$ est défini.

Cependant voici ce qu'il arrive lorsque ce professeur utilise sa fonction `corrige` pour vérifier que le programme `sa_fonction` réalisé par l'étudiant Raymond Calbuth est bien égal au sien qu'il a nommé `ma_solution`.

```
# corrige ma_solution  sa_fonction ;;
Exception: Invalid_argument "equal:_functional_value".
```

L'exception déclenchée provient de la tentative de comparer deux fonctions. Pourtant l'opérateur d'égalité en CAMLest polymorphe `'a` → `'a` → `bool`. Il permet de tester l'égalité de valeurs de tout type... sauf les types contenant des fonctions.

Est-ce une lacune du langage CAML ? Non, pas du tout. C'est un autre résultat fondamental en informatique : l'égalité des programmes est indécidable. Il est impossible de programmer le test d'égalité de deux programmes.

Cet enseignant en informatique aurait pourtant dû le savoir !

8.4.2 Pavage du plan

8.5 Le trombinoscope de la calculabilité

Il est intéressant de noter que ces questions de calculabilité, ainsi que l'existence de fonctions non calculables, dont l'impossibilité de l'arrêt d'un programme, ne sont pas nouvelles, et ont même précédé l'ère informatique. En effet, plusieurs mathématiciens se sont penchés sur ces questions théoriques et ont établis des résultats fondamantaux de l'informatique dans les années 1930 avant la construction des premiers ordinateurs.

David Hilbert (1862-1943), mathématicien allemand. En 1900, lors du deuxième congrès international des mathématiciens, il pose une série de 23 problèmes qu'il considère comme devant retenir toute l'attention des mathématiciens du XXème siècle qui débute. Parmi ces problèmes, le 10ème consiste à trouver un algorithme pour déterminer si une équation diophantienne admet des solutions, problème qui ne sera résolu qu'en 1970 par Matiyasevich. Même si les mathématiciens ont de tout temps utilisé des algorithmes (algorithme d'Euclide pour calculer le pgcd de deux nombres par exemple), cette notion d'algorithme n'avait jamais été définie correctement. En 1920, Hilbert lance un programme de recherche visant à assoir toutes les mathématiques sur des systèmes formels permettant des procédés mécaniques de démonstration. C'est un problème de décision (*Entscheidungsproblem*), dont le but est de déteminer de façon algorithmique si un énoncé mathématique est un théorème ou non.

Alonzo Church (1903-1995), mathématicien américain. En 1936, Alonzo Church qui étudiait l'Entscheidungsproblem de Hilbert, a montré l'existence de problèmes insolubles par des procédés mécaniques. Pour cela il a introduit un formalisme, le λ-calcul pour modéliser

FIGURE 8.1 – David Hilbert (1862-1943)

FIGURE 8.2 – Alonzo Church (1903-1995)

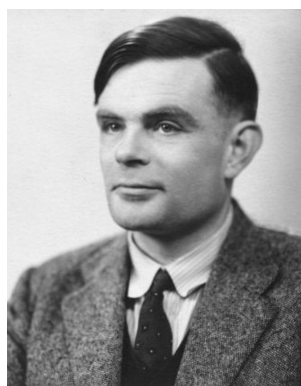

FIGURE 8.3 – Alan Turing (1912-1954)

cette notion floue de fonction calculable. Aujourd'hui, ce formalisme sert en informatique de fondement théorique à la programmation fonctionnelle. Des langages comme CAML en sont les héritiers.

Alan Mathison Turing (1912-1954), mathématicien anglais. Dans un article de 1936, Alan Turing décrit des machines abstraites, connues maintenant sous le nom de machines de Turing, capables d'effectuer tous les calculs arithmétiques connus. Parmi ces machines, il montre qu'il en existe capables de simuler toutes les autres à l'aide d'un « programme ». Ces machines sont donc en quelque sorte les versions idéales de nos ordinateurs actuels. Elles servent d'ailleurs encore aujourd'hui de cadre théorique pour discerner ce qui est calculable ou non, et pour étudier la complexité algorithmique des problèmes programmables. Dans son article, Turing montre en particulier qu'aucune de ses machines n'est capable de déterminer si le calcul qu'effectue une autre machine s'arrête ou non. L'arrêt des programmes est donc un exemple de problème non décidable. C'est une réponse négative à l'Entscheidungsproblem.

Le concept d'algorithme, ou de fonction calculable, n'est pas facile à définir. Les formalismes apportés par Alonzo Church et Alan Turing pour répondre (en partie) au programme de Hilbert, bien que très différents, s'avèrent en fait équivalents. Toute fonction calculable au sens de Church (c.-à-d. représentable dans le formalisme du λ-calcul) l'est au sens de Turing (c.-à-d. représentable par une machine de Turing), et réciproquement. Ce constat a abouti à la formulation d'une thèse [2], appelée *thèse de Church-Turing*, qui conduit à déclarer calculable toute fonction représentable par une machine de Turing, ou par un terme du λ-calcul.

2. C'est un résultat qu'il n'est pas possible de démontrer, ce n'est pas un théorème.