Initiation à la calculabilité

David Monniaux

24 février 2013

Résumé

Le problème de savoir si un système d'équations diophantiennes, c'est-à-dire polynomiales et à solutions entières, a des solutions n'a *a priori* rien à voir avec les *bugs* des logiciels informatiques. La théorie de la *calculabilité* établit pourtant un lien : un résultat classique est que savoir si un système d'équations diophantiennes a une solution est un problème du même type que déterminer si un programme informatique va atteindre un certain état.

1 Équations diophantiennes

En 1900, au Congrès international des mathématiciens de Paris, David Hilbert, peut-être le mathématicien le plus fameux de son temps, propose à ses collègues présents et futurs de s'attaquer à dix problèmes irrésolus particulièrement importants ; la liste publiée en comportera 23 (Hilbert 1900 ; Hilbert 1902). Son X$^{\text{e}}$ problème, portant sur les *équations diophantiennes*, est de

> concevoir un procédé permettant de déterminer, en un nombre fini d'opérations, si l'équation admet une solution dans les entiers naturels.

Rappelons tout d'abord qu'une équation diophantienne est une équation polynomiale à coefficients entiers, dont on ne s'intéresse qu'aux solutions entières. Ainsi, $x^2+3x=-2$ et $3x+5y=7$ sont des équations diophantiennes, la dernière étant en plus *linéaire*.

1.1 Quelques cas simples

Regardons tout d'abord la seconde équation, que nous généraliserons en $ax+by=c$ pour a, b et c trois constantes données. Il est évident que si cette équation admet des solutions, alors le plus grand diviseur commun (PGCD) de a et b doit également diviser c. Cette condition est non seulement nécessaire, mais suffisante ; examinons pourquoi.

L'*algorithme d'Euclide* du calcul du PGCD est basé sur la remarque suivante : notant $\text{pgcd}(x, y)$ le PGCD de x et y et $x \bmod y$ le reste de la division euclidienne (entière) de x par y, alors pour tout $a > b$, $\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b)$. En effet, notant q le quotient de la division euclidienne de a par b, $a = qb + (a \bmod b)$; or pour tout x, $\text{pgcd}(a, b) = \text{pgcd}(a + xb, b)$ d'où le résultat. Sur notre exemple, nous avons ainsi $\text{pgcd}(5, 3) = \text{pgcd}(3, 2)$; nous pouvons poursuivre le procédé et obtenir $\text{pgcd}(3, 2) = \text{pgcd}(2, 1)$, et nous arrêter là car $\text{pgcd}(x, 1) = 1$ pour tout x. Si nous étions partis de $a = 45$ et $b = 35$, nous aurions fait $\text{pgcd}(45, 35) = \text{pgcd}(35, 10) = \text{pgcd}(10, 5)$ et nous pouvons nous arrêter là car 5 divise 10 : le PGCD est donc égal à 5.

Ce qu'Hilbert appelait « procédé », nous l'appelons actuellement *algorithme*, terme que Wikipédia définit ainsi : « une suite finie et non-ambiguë d'opérations ou d'instructions permettant de résoudre un problème ». La méthode appliquée ci-dessus sur deux exemples peut-elle s'exprimer par une telle suite d'opérations ? Oui, par exemple par le programme Objective Caml [1] suivant :

```
let rec pgcd_rec x y =
  if y=0
  then x
  else pgcd_rec y (x mod y)

let pgcd1 x y = if x > y
                then pgcd_rec x y
                else pgcd_rec y x

let pgcd x y = pgcd1 (abs x) (abs y)
```

Pour le lecteur qui ignorerait Caml, cela veut dire que nous commençons par calculer la valeur absolue des arguments de la fonction pgcd et les passons à pgcd1, qui appelle pgcd_rec en s'assurant que le premier argument est bien supérieur ou égal au second. La fonction pgcd_rec est définie par récurrence, et calcule $\text{pgcd}(x, y)$ comme $\text{pgcd}(y, x \bmod y)$ tant que $y \neq 0$.

L'exigence d'une suite *finie* d'instructions à suivre s'applique non seulement au texte du programme (qui est effectivement de longueur finie), mais aussi aux exécutions possibles de ce programme : autrement dit, il doit toujours *terminer*. Est-ce le cas pour notre fonction pgcd_rec ? Oui, car son second argument est un entier naturel qui décroît strictement lorsqu'elle s'appelle récursivement : pour tout x et y, $x \bmod y < y$ — on ne peut donc avoir de suite infinie d'appels récursifs. Ceci nous permet de majorer par y le nombre de pas (les appels récursifs). Une analyse plus fine, due à Gabriel Lamé, montre que le nombre de pas de calcul ne peut excéder cinq fois le nombre de chiffres décimaux de y (Lamé 1844) ; Knuth (1998, §4.5.3, pp. 356–373) présente des analyses plus détaillées. En tout état de cause, le

1. Le langage Objective Caml est le successeur du langage Caml Light étudié en France en classes préparatoires. `http://caml.inria.fr/`

nombre d'opérations élémentaires réalisées par la machine est majoré par un polynôme en les tailles de x et y, si nous appelons taille d'un nombre la longueur de son écriture en base 10 (si nous nous en tenons à des considérations comme : linéaire, polynomial, etc., la base n'importe pas). [2]

Remarquons au passage que le langage Caml permet de définir des fonctions « récursives » qui ne terminent jamais, par exemple **let rec** f x = f x (autrement dit, définir $f(x)$ circulairement par $f(x)$), ou terminent parfois seulement. Nous verrons plus loin que cette caractéristique est inévitable pour un langage de programmation, sauf à imposer des restrictions aux algorithmes qu'il permet de décrire.

Quel rapport avec la résolution de l'équation diophantienne initiale ? Chacun des entiers naturels a et b successivement manipulés (les arguments de la fonction pgcd_rec) s'exprime comme une combinaison linéaire à coefficients entiers relatifs des arguments initiaux a_0 et b_0, et l'on peut d'ailleurs calculer explicitement ces coefficients : si $a = \alpha a_0 + \beta b_0$ et $b = \gamma a_0 + \delta b_0$, alors, notant q le quotient de a par b, $a \bmod b = (\alpha - q\gamma)a_0 + (\beta - q\delta)b_0$. Par cet *algorithme d'Euclide étendu*, nous pouvons donc calculer en fonction de a et b, non seulement $\mathrm{pgcd}(a, b)$, mais aussi le *couple de Bézout* (α, β) tel que $\mathrm{pgcd}(a, b) = \alpha a + \beta b$. La fonction pgcd_et suivante calcule le triplet $(\mathrm{pgcd}(a, b), \alpha, \beta)$.

```
let rec pgcd_et_rec x alpha beta y gamma delta =
  if y=0
  then (x, alpha, beta)
  else let q = x/y in
    pgcd_et_rec y gamma delta
      (x mod y) (alpha-q*gamma) (beta-q*delta)

let pgcd_et1 x y = if x > y
                   then pgcd_et_rec x 1 0 y 0 1
                   else pgcd_et_rec y 0 1 x 1 0

let pgcd_et x y = pgcd_et1 (abs x) (abs y)
```

Pour résoudre une équation diophantienne $ax + by = c$, il suffit donc de calculer $(\mathrm{pgcd}(a, b), \alpha, \beta)$ tel que $\mathrm{pgcd}(a, b) = \alpha a + \beta b$, puis, si $\mathrm{pgcd}(a, b)$ divise c (condition nécessaire pour qu'il y ait une solution), produire $x = \frac{c}{\mathrm{pgcd}(a,b)}\alpha$ et $y = \frac{c}{\mathrm{pgcd}(a,b)}\beta$ (ce qui établit que cette condition est également suffisante).

2. Nous avons écrit notre programme en utilisant le type int de Caml, qui, suivant l'architecture de l'ordinateur, limite les entiers à l'intervalle $[-2^{31}, 2^{31} - 1]$ ou $[-2^{63}, 2^{63} - 1]$; nous considérons que chaque opération arithmétique (addition, multiplication, soustraction, division) sur ces *entiers machine* est une opération élémentaire. Si nous avions voulu permettre de calculer sur des entiers de taille arbitraire, il nous aurait fallu recourir à une bibliothèque de calcul sur des grands entiers, qui réalise sur ceux-ci les opérations arithmétiques par combinaison d'opérations élémentaires sur les entiers machine. Cependant, on montre que le nombre d'opérations élémentaires resterait alors polynomial.

Cette approche se généralise aisément au cas d'une équation diophantienne linéaire à un nombre quelconque de variables. Pour des *systèmes* d'équations diophantiennes linéaires, l'algorithmique est plus compliquée, mais on parvient à savoir s'ils ont une solution et si oui à en exhiber une par exemple en mettant le système en *forme normale de Hermite* à l'aide d'une variante du pivot de Gauss.

Penchons-nous maintenant sur notre exemple $x^2 + 3x = -2$, que nous généralisons à un polynôme à une variable de degré $d > 0$ arbitraire $P(x) = \sum_{k=0}^{d} a_k x^k$, avec $a_d \neq 0$. Quand $|x|$ est grand, $P(x)$ vaut environ $a_d x^d$, et donc $P(x) \neq 0$. Voyons cela plus précisément. Comme pour x entier positif ou nul, $x^i \geq x^j$ pour $i \geq j$, alors $\left|\sum_{k=0}^{d-1} a_k x^k\right| \leq |x|^{d-1} \sum_{k=0}^{d-1} |a_k|$, d'où, encore par inégalité triangulaire, $|P(x)| \geq |a_d|.|x|^d - |x|^{d-1} \sum_{k=0}^{d-1} |a_k|$. Donc, $|P(x)| > 0$ si $|x| > K$ avec $K = \left\lfloor \frac{\sum_{k=0}^{d-1} |a_k|}{|a_d|} \right\rfloor$, en notant $\lfloor y \rfloor$ la partie entière de y.

Ceci nous suggère donc un algorithme : calculer K, et essayer tous les entiers x dans $[-K, K]$. Notre algorithme cherche donc une aiguille dans une botte de foin : la taille de l'espace de recherche est bornée par un polynôme en les coefficients de P...mais pas en leurs *tailles*. Notre algorithme semble donc d'une nature bien moins *efficace* que celle de l'algorithme d'Euclide ; mais bien qu'inefficace, il n'en reste pas moins un algorithme.

1.2 Variations

Notre succès sur les équations diophantiennes linéaires et les équations diophantiennes à une inconnue justifie que nous nous posions, comme Hilbert, la question de l'existence d'un algorithme pour les équations diophantiennes générales.

Commençons tout d'abord par remarquer que le problème pour des systèmes d'équations diophantiennes se ramène à celui d'une unique équation de degré supérieur : $P_1(x, y, \dots) = 0 \wedge \cdots \wedge P_n(x, y, \dots) = 0$ équivaut à $P_1^2(x, y, \dots) + \cdots + P_n^2(x, y, \dots) = 0$ ($\wedge$ note le « et » logique). De même, le problème des systèmes d'équations de degré quelconque se ramène à celui des systèmes d'équations de degré au plus 2 : il suffit d'associer une nouvelle variable à chacun des produits non linéaires qui le composent — par exemple, $xy + 5y^3 + 3x^2 = 0$ se réécrit en $(\alpha = xy) \wedge (\beta = 5y^2) \wedge (\gamma = \beta y) \wedge (\delta = 3x^2) \wedge (\alpha + \gamma + \delta = 0)$. Cette pratique de *réduire* un problème à un autre en exhibant un algorithme qui transforme une instance du premier problème en une instance du second est fondamentale en théorie de la calculabilité ; nous y reviendrons.

Si nous savons décider si un système d'équations diophantiennes a une solution, alors nous savons également le faire pour des systèmes d'inégalités : d'après le théorème de Lagrange, tout entier positif s'écrit comme la

somme de quatre carrés d'entiers, on peut donc remplacer toute inégalité $P(x, y, \dots) \geq 0$ par une égalité $P(x, y, \dots) = \alpha^2 + \beta^2 + \gamma^2 + \delta^2$ où $\alpha, \beta, \gamma, \delta$ sont des variables fraîches (n'apparaissant ni dans P ni dans aucune autre (in)égalité). On a donc réduit à notre problème d'équations diophantiennes un problème apparemment plus général.

Voyons maintenant ce qui arrive si l'on fait varier l'espace dans lequel on recherche les solutions. Considérons tout d'abord le problème de déterminer si un système d'équations polynomiales $P_1(x, y, \dots) = 0 \wedge \dots \wedge P_n(x, y, \dots) = 0$ à coefficients entiers (ou rationnels, peu importe : il suffit de mettre tous les coefficients au même dénominateur) a des racines *complexes*. Il est clair que si l'on peut exhiber $Q_1, \dots, Q_n$ tels que $\sum P_i Q_i = 1$, ce système ne peut avoir de solution dans quelque corps que ce soit (appliquer les deux membres de l'équation à une solution, on obtient $0 = 1$). Le théorème dit *Nullstellensatz* de Hilbert (encore lui) garantit que si le système n'a pas de racines complexes, alors de tels polynômes Q_i existent. Il existe d'ailleurs un algorithme qui va calculer de tels Q_i s'ils existent, et répondre sinon qu'il n'y en a pas : cet algorithme va donc décider si notre système a ou non des solutions. [3]

Le problème de déterminer si un tel système a des solutions *réelles* est considérablement plus pénible. Il existe cependant des algorithmes d'*élimination des quantificateurs* qui transforment une formule logique quelconque construite à partir d'inégalités polynomiales à coefficients entiers (ou rationnels), des connecteurs logiques (« et » $\wedge$, « ou » $\vee$, « négation » $\neg$), et des quantificateurs existentiel ($\exists$) et universel ($\forall$), en une formule ayant exactement les mêmes solutions et la même forme, mais *sans quantificateurs*. Appliqué à une formule de la forme $\exists x \exists y \exists z \dots P_1(x, y, \dots) = 0 \wedge \dots \wedge P_n(x, y, \dots) = 0$, un tel algorithme va produire une formule « vrai » ou « faux ». [4]

Vu notre succès sur les équations linéaires, les équations à une variable, les solutions complexes, les solutions réelles, nous pourrions nous attendre à ce que le cas des « bêtes » entiers soit plus simple. Ce n'est pas le cas, comme nous le verrons au §3.3. . .mais pour comprendre ce qui est en cause,

3. Il suffit de diviser 1 par une base de Gröbner de l'idéal engendré par $P_1, \dots, P_n$; une telle base peut se calculer par exemple par l'algorithme de Buchberger. Nous ne rentrerons pas dans les détails de ces processus, qui sont sans importance pour la suite de l'article, et renvoyons le lecteur par exemple à Cox, Little et O'Shea 2007.

4. L'algorithme de *décomposition cylindrique algébrique* peut être compris comme une généralisation des « tableaux de signes » familiers des lycéens au cas de plusieurs polynômes à plusieurs variables. Il est notamment implémenté dans le logiciel universitaire QepCad et dans le logiciel commercial de calcul formel Mathematica (fonctions `Reduce` et `Resolve`).

Remarquons au passage que comme la géométrie élémentaire peut se coder en coordonnées cartésiennes et donc en (in)égalités polynomiales, cet algorithme fournit une procédure pour résoudre tous les problèmes de géométrie de l'enseignement secondaire qui appellent une réponse par « oui » ou « non ». Élèves et professeurs ne doivent cependant pas se précipiter pour acheter Mathematica : la complexité de cet algorithme est trop élevée pour résoudre des problèmes géométriques ainsi exprimés.

il nous faudra tout d'abord évoquer les notions de calculabilité dues à Turing, Church et d'autres grands mathématiciens !

2 Les fonctions calculables

Constatons tout d'abord une difficulté. Lorsque nous avons voulu montrer que tel ou tel problème était *décidable*, c'est-à-dire qu'un algorithme le résout, il nous a suffit de décrire informellement cet algorithme ainsi qu'une justification de son bon fonctionnement (qu'il termine forcément, et qu'il produit alors le résultat attendu). En revanche, si nous devons montrer qu'aucun algorithme ne donne le résultat attendu, il nous faut poser d'avance une définition plus stricte de ce que nous considérons ou non comme un algorithme, et il faut que cette définition corresponde à l'intuition que nous avons de ce qui est calculable ou non, de la même façon qu'il a fallu poser une définition du corps des réels afin qu'elle corresponde à notre intuition d'une droite continue.

2.1 Les fonctions primitives récursives

Nous sommes toutefois avantagés par rapport aux mathématiciens du début du XX^{e} siècle : nous avons des ordinateurs, et nombreux parmi nous sont ceux qui les ont déjà programmés. Nous avons donc une notion intuitive de ce qu'est un algorithme : c'est un processus que l'on peut formaliser sous la forme d'un programme. Nous n'apporterons qu'un bémol à cette intuition : la mémoire d'un ordinateur est bornée (ne serait-ce que par le nombre de particules dans l'Univers, si toutefois ce nombre est fini — mais je ne suis pas physicien !) tandis que nous supposerons que nous disposons d'un ordinateur à mémoire non bornée et donc capable de calculer sur des nombres infiniment grands. Ceci évitera notamment des résultats peu intéressants, du style « si je vous fournis un système d'équations diophantiennes plus grand que la mémoire de l'ordinateur, celui-ci ne peut même pas le lire et encore moins le résoudre ».

Qu'inclure dans un langage informatique censé représenter tout algorithme ? Les algorithmes présentés au §1.1 peuvent s'écrire à l'aide d'opérations arithmétiques élémentaires ($+$, $-$, $\times$), de tests et de boucles « for » (rappelons qu'une boucle **for** i=a **to** b **do** f **done** exécute f pour les valeurs successives de i dans l'intervalle entier $[a, b]$).

Certes, nous avons écrit l'algorithme d'Euclide à l'aide d'une fonction récursive et non d'une boucle, mais nous pouvons facilement le transformer en une boucle, par exemple en langage C ou Pascal : [5]

5. On peut transformer tout *appel récursif terminal* en une boucle vers le début de la fonction. Cette optimisation est réalisée par le compilateur `ocamlopt` et par divers compilateurs pour d'autres langages, tels que `gcc`.

```c
int pgcd(int x, int y) {
  if (x < 0) x = -x;
  if (y < 0) y = -y;
  if (x < y) {
    int tmp = x;
    x = y;
    y = tmp;
  }
  while (y > 0) {
    int r = x % y;
    x = y;
    y = r;
  }
  return x;
}
```

```
FUNCTION pgcd(i, j : longint)
  : longint;
VAR
   tmp : longint;
BEGIN
   IF i < j THEN
   BEGIN
      tmp := i;
      i := j;
      j := tmp;
   END;
   WHILE (j > 0) DO
   BEGIN
      tmp := i MOD j;
      i := j;
      j := tmp
   END;
   pgcd := i;
END;
```

On m'objectera que ma boucle **while** n'est pas une boucle « for » avec la définition précédente. Cependant, comme nous l'avons vu, on peut très facilement borner son nombre de tours et la transformer en une boucle « for » :

```c
int pgcd_for(int x, int y) {
  ...
  int tours_max = x;
  for (int i=0; i<tours_max; i++) {
    if (y > 0) {
      int r = x % y; /* modulo */
      x = y;
      y = r;
    }
  }
  return x;
}
```

Avons-nous besoin d'inclure dans notre langage les opérateurs $+$, $\times$, etc. ? Non, puisque on peut les définir à l'aide de boucles « for » et de l'opération « successeur » $x \mapsto x+1$. Nous pourrions ainsi croire que toutes les fonctions calculables au sens intuitif pourraient l'être par des programmes manipulant un nombre fini de variables entières non bornées à l'aide d'un petit nombre d'opérations arithmétiques élémentaires, et de boucles « for », que l'on appelle les fonctions « primitives récursives ». Hélas, ce n'est pas le cas.

Prenons par exemple la fonction d'Ackermann–Péter :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{si } m > 0 \text{ et } n > 0. \end{cases} \qquad (1)$$

Cette fonction est calculable, par exemple par le programme Caml suivant [6] :

```
let rec ackermann m n =
  if m=0
  then n+1
  else
    if n=0
    then ackermann (m-1) 1
    else ackermann (m-1) (ackermann m (n-1))
```

Ce programme termine forcément : notez qu'à chaque fois que ackermann m n fait un appel récursif ackermann a b, le couple d'entiers naturels (a, b) est strictement inférieur à (m, n) pour l'ordre lexicographique (autrement dit, $a < m$, ou alors $a = m$ mais $n < b$).

On montre que pour toute fonction primitive récursive $f : \mathbb{N}^n \to \mathbb{N}$, il existe b tel que $f(x_1, \ldots, x_n) < A(b, x_1 + \cdots + x_n))$, ce b dépendant, grosso modo, de la structure des boucles dans un programme exprimant f (Th. 1). Ceci exclut que A soit primitive récursive : si elle l'était, $x \mapsto A(x, x)$ le serait aussi, donc il existerait b tel que $A(x, x) < A(x, b)$ pour tout x, ce qui donne pour $x = b$ l'inégalité absurde $A(b, b) < A(b, b)$.

Nous pourrions croire qu'il suffirait d'ajouter dans notre langage d'autres constructions d'itérations plus puissantes que la simple boucle « for », et que nous pourrions alors définir toute fonction calculable. Hélas, non, et voici pourquoi.

Un programme est une suite de caractères sur un certain alphabet (on parle parfois du *code source* d'un programme pour désigner cette suite). Par souci de simplicité, nous nous limiterons à des programmes prenant en entrée un ou plusieurs entiers. Ceci n'est pas gênant, parce que tous les types de données utiles (chaînes, etc.) peuvent se coder dans les entiers. Ainsi, il est possible d'énumérer successivement toutes les suites de caractères (en commençant par celles de 1 caractère, puis celles de 2...), et de ne garder que celles représentant des programmes « compilant correctement » : ceci nous permet donc d'assigner un entier naturel $\ulcorner P \urcorner$ à chaque programme P (on appelle parfois ceci une *numérotation de Gödel*).

Il est également possible de programmer un *interpréteur* pour un pareil langage, c'est-à-dire une fonction qui prend en argument un programme (donné comme code source) et l'argument (ou les arguments d'entrée) du programme, et retourne le résultat de l'évaluation du programme sur l'entrée : par exemple, la command `ocaml` *`fichier.ml`* exécute le programme Objective Caml contenu dans *`fichier.ml`*. On peut également définir cet

6. On m'objectera que la fonction d'Ackermann croît tellement vite qu'elle aura vite fait de dépasser les capacités du type entier de Caml. Il suffit alors d'utiliser une bibliothèque de calcul en précision arbitraire, par exemple `mlgmp` ou `Zarith`, qui sont des interfaces sur la bibliothèque `gmp`. `http://gmplib.org/`

interpréteur à l'aide de la numérotation de Gödel, sous la forme d'une fonction $I(i, j)$ qui interprète le programme numéro i sur l'entrée j et retourne la valeur calculée. Cette fonction est calculable (intuitivement) ; il en est donc de même de la fonction $D : i \mapsto I(i, i) + 1$.

Si notre langage est suffisamment expressif pour permettre de définir n'importe quelle fonction calculable, il existe au moins un programme qui calcule D ; notons i_0 son numéro. Alors, $I(i_0, i) = D(i) = I(i, i) + 1$. Appliquons cette fonction en $i = i_0$, nous obtenons $I(i_0, i_0) = I(i_0, i_0) + 1$. Contradiction. [7]

Nous concluons donc qu'il n'existe pas de langage de programmation qui permette de définir toutes les fonctions calculables au sens intuitif, puisque l'interpréteur pour ce langage de programmation ne peut être défini en son sein !

Ce résultat peut apparaître comme paradoxal : l'interpréteur pour mon langage est clairement une fonction calculable, que l'on pourrait au besoin programmer. Le paradoxe est levé si l'on considère que nous nous sommes limités aux programmes dont l'évaluation termine toujours (autrement dit, ne part jamais en boucle infinie). Si nous rajoutons dans notre langage des constructions permettant au besoin de partir en boucle infinie, notre argument disparaît : si nous notons $\perp$ la valeur spéciale « ne termine pas », notre équation paradoxale $I(i_0, i_0) = I(i_0, i_0) + 1$ a pour unique solution $I(i_0, i_0) = \perp$. [8]

2.2 Une définition rigoureuse

Il nous faut rajouter quelque chose à nos fonctions primitives récursives, mais quoi ? Examinons les langages de programmation « réels » : ceux-ci prévoient des fonctions récursives [9] ainsi que des boucles « while », qui s'opposent aux boucles « for » au sens que leur nombre de tours de boucles n'est pas

7. Un tel argument, où l'on montre qu'un objet ne peut exister dans une certaine classe parce que son application à son propre indice dans la classe produit une contradiction s'appelle un *argument diagonal*. On trouve des arguments diagonaux par exemple dans la preuve de Cantor que $\mathbb{R}$ n'est pas dénombrable, dans la preuve des théorèmes d'incomplétude de Gödel, dans celle du théorème de l'arrêt de Turing, dans la preuve du théorème de la hiérarchie du temps en complexité algorithmique…

8. D'une façon générale, l'introduction sans précaution de fonctions dont l'évaluation ne termine pas forcément dans un cadre logique débouche sur des paradoxes. Par exemple, le programme Caml **let rec** f x = (f x)+1 correspond à une fonction mathématique $f(x) = f(x) + 1$, ce qui est paradoxal ! Ceci explique que les systèmes permettant de raisonner mathématiquement sur les programmes, par exemple Coq, exigent généralement de montrer la terminaison de toute boucle et de toute fonction récursive en exhibant un ordre bien fondé, comme nous l'avons fait pour la fonction d'Ackermann–Péter.

9. Il y a malheureusement ici un problème de terminologie. L'expression « fonction récursive » a un sens en logique mathématique, qui précède l'existence des ordinateurs et des langages de programmation, et qui est l'équivalent de « fonction calculable », et un sens en programmation, qui désigne des fonctions se rappelant elles-mêmes. C'est ce second sens que nous utilisons ici.

connu a priori. [10] Dans les deux cas, nous pouvons exprimer des programmes qui ne terminent pas, ce qui satisfait nos conclusions précédentes, à savoir qu'un langage de programmation ne saurait être « complet » sans permettre l'écriture de programmes qui ne terminent pas.

Les fonctions récursives peuvent se simuler à l'aide d'une pile (c'est-à-dire une suite finie de valeurs au bout de laquelle on ajoute ou l'on récupère une valeur), ce qui est d'ailleurs la méthode la plus courante pour les mettre en œuvre dans les ordinateurs — qu'est-ce que fait le microprocesseur, sinon une boucle « tant que l'ordinateur n'a pas l'instruction de s'éteindre, exécuter la prochaine instruction » ? Nous pouvons donc nous en passer, en les simulant à l'aide d'une pile et d'une boucle, et en simulant l'empilement et le dépilement par des opérations de codage et décodage arithmétique.

L'ensemble des fonctions calculables par des programmes comportant des opérations élémentaires sur les entiers, [11] des tests, des boucles « for » et des boucles « while » semble correspondre à notre intuition de ce qui est calculable.

On m'objectera que cette notion de « programme » est très floue, aussi je vais donner maintenant une caractérisation précise de ce qu'est une « fonction récursive primitive » et une « fonction μ-récursive ».

2.3 Fonctions récursives à la Church–Rosser

Nous nous intéressons à des fonctions $\mathbb{N}^n \to \mathbb{N}$, où n s'appelle l'*arité* de la fonction ; on parle de fonctions unaires, binaires, et plus généralement n-aires, et une fonction 0-aire est une *constante*. L'ensemble des fonctions primitives récursives (p.r.) est défini comme le plus petit ensemble contenant la constante 0, la fonction unaire successeur $x \mapsto x + 1$, les projections $(x_1, \ldots, x_n) \mapsto x_i$, et stable par :

- la composition : si f est une fonction p.r. n-aire et $g_1, \ldots, g_n$ sont des fonctions p.r. m-aires, alors $(x_1, \ldots, x_m) \mapsto f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$ est aussi une fonction p.r.
- la récursion primitive : si f est n-aire et g est $(n+2)$-aire, la fonction $(n+1)$-aire h définie par $h(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$ et $h(k+1, x_1, \ldots, x_n) = g(k, h(k, x_1, \ldots, x_n), x_1, \ldots, x_n)$ est primitive récursive.

On remarquera que la récursion primitive correspond à la définition familière d'un entier $h(k, x_1, \ldots, x_n)$ par récurrence sur k, en laissant $x_1, \ldots, x_n$ constants. Pourquoi notre définition en Caml de la fonction d'Ackermann–Péter ne rentre-t-elle pas dans ce cadre ? Parce que nous l'avons définie non

10. En langage C ou C++, la boucle **for** est une simple alternative syntaxique au **while**, au rebours du « for » de Caml, Pascal ou Ada. C'est à ce dernier que nous faisons allusion.

11. On peut d'ailleurs se passer d'une partie de ces opérations : $\times$ se définit par récurrence primitive à l'aide de $+$, $+$ se définit par récurrence primitive à l'aide de l'opération successeur $x \mapsto x + 1 \ldots$

pas par récurrence sur les entiers naturels, mais sur un ordre lexicographique. Nous pouvons d'ailleurs la reformuler ainsi :

```
let rec ackermann m =
  if m=0
  then fun n-> n+1
  else
    let rec ack2 n =
      if n=0
      then ackermann (m-1) 1
      else ackermann (m-1) (ack2 (n-1))
    in ack2
```

Si la fonction ack2 définit bien un entier par récurrence sur n, la fonction ackermann définit une *fonction des entiers dans les entiers* par récurrence sur m. C'est fondamentalement différent.

Cette définition rigoureuse des fonctions primitives récursives permet de prouver des propriétés par récurrence bien fondée sur leur définition, par exemple :

Théorème 1. *Soit A la fonction d'Ackermann–Péter telle que définie par l'équation 1. Pour toute fonction primitive récursive $f : \mathbb{N}^n \to \mathbb{N}$, il existe b tel que $f(x_1, \ldots, x_n) < A(b, x_1 + \cdots + x_n))$.*

L'idée de la preuve est la suivante : on montre

1. que la constante 0, la fonction unaire successeur $x \mapsto x + 1$, les projections $(x_1, \ldots, x_n) \mapsto x_i$ sont des fonctions f vérifiant la propriété ci-dessus ;
2. que cette propriété est préservée par composition et récursion primitive.

Il faut pour cela prouver quelques lemmes (monotonie stricte...) et majorations, qu'il serait trop long de donner ici. Le lecteur se reportera au besoin à Taylor (1998a) ; Taylor (1998b, ch. 3).

Pour obtenir les fonctions *μ-récursives partielles*, ou *récursives partielles* tout court, ou encore *calculables* (puisqu'elles correspondent à la notion intuitive de calculabilité), il faut clore par une opération additionnelle, *l'opérateur de minimisation* : pour toute fonction récursive f $(n+1)$-aire, $\mu_f(x_1, \ldots, x_n)$ est défini comme le plus petit i tel que $f(i, x_1, \ldots, x_n) = 0$, et par $\perp$ sinon (autrement dit, l'évaluation ne termine pas). Cet opérateur correspond à une boucle

```
int i=0;
while (f(i, x1, ..., xn) != 0) {
  i = i+1;
}
```

La présence de calculs qui ne terminent pas complique quelque peu la définition des opérateurs de composition : $f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$

vaut $\perp$ l'un des $g_i(x_1, \ldots, x_m)$ vaut $\perp$ ou si f ne termine pas sur $g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m)$.

On peut reprendre l'ensemble de nos raisonnements précédents rigoureusement en utilisant cette caractérisation mathématique. Certains sont cependant assez pénibles à faire ; par exemple, il va nous falloir là encore définir un codage de Gödel des fonctions (c'est-à-dire associer un numéro à chaque définition syntaxique de fonction récursive f un entier, noté $\ulcorner f \urcorner$ avec des demi-crochets de Quine), et une fonction interpréteur, ou *fonction universelle*, u telle que $u(\ulcorner f \urcorner, x) = f(x)$.

2.4 Machines de Turing

Une autre caractérisation célèbre des fonctions calculables est que ce sont celles qui le sont par une *machine de Turing*. Une machine de Turing est un automate de calcul doté d'un état interne pris dans un ensemble fini Σ et de n bandes infinies où elle peut lire et écrire des symboles pris dans un alphabet A fini, chaque bande étant munie d'une tête de lecture/écriture qui peut bouger d'un cran au maximum à chaque pas de calcul. L'action de la machine de Turing est définie par une table de règles de la forme $(\sigma, l_1, \ldots, l_n) \to (\sigma', l'_1, \ldots, l'_n, d_1, \ldots, d_n)$ où les $d_i \in \{G, P, D\}$: cela veut dire que quand les têtes de lecture sont sur des cases portant respectivement $l_1, \ldots, l_n \in A$, et que l'état interne est σ, la machine fait un pas de calcul en écrivant $l'_1, \ldots, l_n$ sur les cases en question, en faisant bouger ses têtes suivant les d_i (gauche, sur place, droite), et en passant dans l'état σ'. Un état est désigné comme état initial σ_0, un autre comme état final σ_F. L'exécution se termine dès que la machine atteint l'état final.

Les machines de Turing sont souvent représentées par des graphes, de la même façon que les automates finis : chaque nœud représente un état, on indique l'état initial par une mention ou une forme particulière (carré), on indique le ou les états acceptants par une mention ou un entourage particulier (double trait), et les arêtes (σ, σ') sont étiquetées par les transitions allant de l'état σ à l'état σ'. Si toutes les transitions ne sont pas indiquées pour un état, on peut supposer l'existence d'un « état puits » non acceptant vers où vont les transitions non indiquées, et dont toutes les transitions bouclent sur lui-même. La figure 1 présente ainsi une machine de Turing qui calcule l'addition en notation binaire.

On considère des machines de Turing à trois bandes sur l'alphabet $\{0, 1, B\}$, B représentant un « blanc » (on peut montrer que le nombre de bandes et la taille ≥ 2 de l'alphabet importent peu). On écrit un nombre n en binaire sur la première bande, entouré de blancs, en positionnant la tête de lecture/écriture sur son premier chiffre, on exécute la machine de Turing, et si celle-ci termine on récupère sur la troisième bande un nombre en binaire que l'on nomme $f(n)$, la deuxième bande servant de bande de calcul. S'il y a le moindre problème (machine de Turing qui ne termine pas, nombre

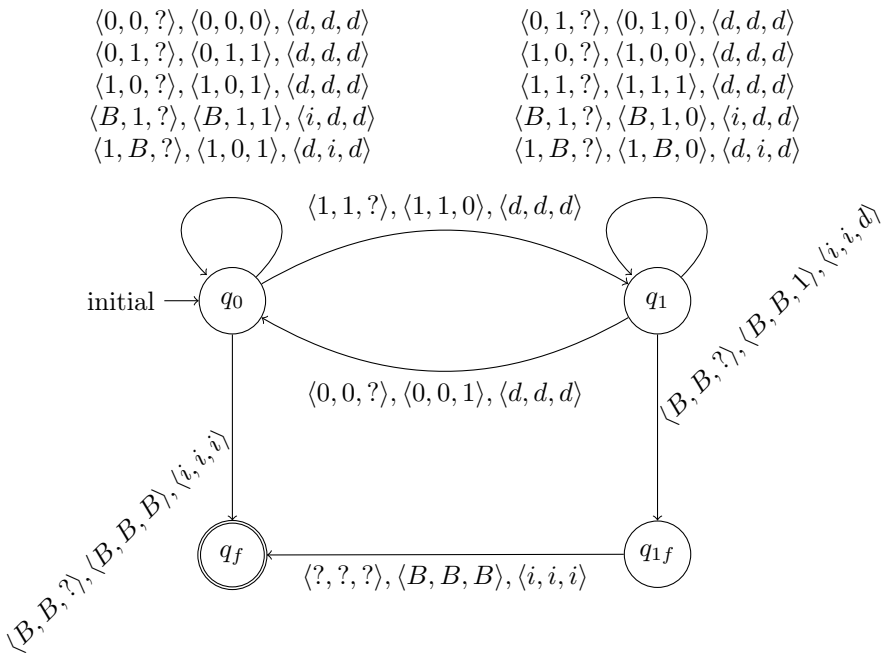

FIGURE 1 – Une machine de Turing sur l'alphabet $\{0, 1, B\}$, à 3 rubans, qui met sur la troisième bande le résultat de l'addition des nombres écrits en binaire sur les deux premières, chaque nombre étant écrit de la gauche vers la droite à partir de la position initiale de la tête sur ce ruban et suivi d'un B. À la fin du calcul, les têtes sont sur les B finaux des trois nombres. L'état final est entouré d'un double trait. Une transition dénotée $\langle l_1, l_2, l_3\rangle, \langle e_1, e_2, e_3\rangle, \langle d_1, d_2, d_3\rangle$ s'interprète ainsi : si les cases sous les têtes de lectures sont étiquetées l_1, l_2, l_3 alors la machine écrit sur ces cases e_1, e_2, e_3 puis chaque tête numéro i fait le pas dicté par d_i, i dénotant l'immobilité et d le mouvement vers la droite. Par souci de compacité, on note ? pour signifier que la valeur lue n'importe pas : une telle transition est une abréviation pour plusieurs transitions obtenues en remplaçant ? par $0, 1, B$. L'état q_0 (resp. q_1) dénote l'absence (resp. la présence) de retenue entrante, l'état q_f est final, et l'état $q_{1,f}$ sert à écrire un B final sur la troisième bande une fois qu'on y a écrit comme chiffre le plus significatif un 1 issu d'une retenue.

incorrectement écrit sur la bande de sortie...), on pose $f(n) = \perp$.

On appelle fonction Turing-calculable les fonctions partielles (on peut avoir la valeur spéciale $\perp$) ainsi définies. Pour pouvoir définir des fonctions Turing-calculables à plusieurs variables, on peut coder un n-uplet d'entiers naturels $(x_1, \ldots, x_n)$ sous forme d'un unique entier $\langle x_1, \ldots, x_n \rangle$, par exemple en utilisant les produits de puissances de nombres premiers :$2^{1+x_1}.3^{1+x_2}.5^{1+x_3}.\ldots$. Le codage utilisé importe peu, tant qu'il est « raisonnable » (p.ex. les projections doivent être Turing-calculables).

On montre que les fonctions Turing-calculables contiennent les fonctions récursives précédemment définies : elles contiennent la constante 0, la fonction unaire successeur, les projections, sont stables par composition, récursion primitive, minimisation. La preuve est assez fastidieuse ; nous nous contenterons de mentionner ici

- Une machine calculant sur la troisième bande la somme des nombres figurant sur les deux premières bandes (Fig. 1).
- La composition de deux machines d'ensembles d'états Σ_1 et Σ_2 (disjoints, sinon on les renomme) donne une machine à états $\Sigma_1 \cup \Sigma_2$ par simple concaténation des tables de transitions, chaque transition vers l'état final de la première étant remplacée par une transition vers l'état initial de la seconde.

On peut également montrer que la fonction calculée par toute machine de Turing est une fonction récursive partielle. Nous ne donnerons là encore que l'intuition de la preuve : on code chaque bande par deux entiers naturels l et r, l étant l'entier dont le développement en binaire est écrit sur la bande à gauche de la tête de lecture écriture (le chiffre le moins significatif correspond au symbole situé immédiatement à gauche de la tête), et r de même pour à droite de la tête (si la présence des blancs vous trouble, on peut dire qu'on considère des nombres en développement ternaire). Chaque pas de calcul de la machine de Turing s'exprime par des manipulations arithmétiques codables par des fonctions récursives partielles. Le nombre d'opérations N nécessaires à l'exécution de la machine s'obtient à l'aide de la minimisation (donc si l'exécution de la machine de Turing ne termine pas, on obtient $\perp$) ; ensuite, pour obtenir son résultat final, on exécute N pas de calculs (fonction primitive récursive de N et de l'entrée).

Les notions de fonctions (partielles) calculables par machine de Turing et comme fonction récursive sont donc identiques.

2.5 La thèse de Church–Turing

Au final, deux formalisations a priori très différentes de la même notion de « fonction calculable » caractérisent le même ensemble de fonctions. Nous aurions pu également évoquer d'autres modèles, comme les « machines RAM », les automates à deux piles, ou le λ-calcul pur : tous définissent exactement la même classe de fonctions calculables ; et même, il existe des moyens

algorithmiques pour passer un « programme » d'un modèle dans l'autre en préservant la fonction qu'il calcule, ou, en termes plus informatiques, pour *compiler* les programmes d'un modèle vers un autre.[12] Cette convergence entre diverses notions apparemment différentes n'est pas si surprenante que cela pour un mathématicien. Prenons par exemple le corps des réels : on peut le définir à l'aide des suites de Cauchy de rationnels ou avec les coupures de Dedekind, mais on obtient la même structure à isomorphisme près.

Ce constat d'équivalence entre des définitions fort différentes justifie la thèse dite de Church–Turing : « toute notion raisonnable de calcul définit un ensemble de fonctions calculables équivalent à celle définie par les machines de Turing ou les fonctions récursives ». Il s'agit là d'une thèse philosophique et non mathématique, car nous ne définissons pas mathématiquement ce que nous entendons par « notion raisonnable de calcul », mais qui est confortée par l'expérience (voir Smith 2007 pour une discussion philosophique plus approfondie).

Les informaticiens théoriciens, les logiciens, et plus généralement les mathématiciens qui évoquent les fonctions calculables ou les algorithmes, font tous plus ou moins appel à cette thèse. Il est extrêmement fastidieux de décrire explicitement une machine de Turing réalisant une activité complexe (c'est un peu comme programmer un grand logiciel en langage assembleur), et encore très fastidieux de l'obtenir par composition explicite de machines de Turing plus simple. En pratique, dans les cas simples, on se contente de dire que telle ou telle fonction est clairement calculable, ou d'esquisser l'algorithme, quitte à justifier plus précisément les points délicats. On pourra détailler l'algorithme plus avant si nécessaire, par exemple si l'on veut déterminer le nombre de pas de calcul nécessaires.

Cela n'a, là encore, rien d'extraordinaire pour un mathématicien. Si l'on veut prouver, par exemple, que

$$f(x) = \sin(x^3 + x^2 + 3x - 5) + \sqrt{\sin^2(\sqrt{x^4 + x^2 + 1} + 5) + x^6 + 1}$$

est continue, personne ne va, en pratique, revenir à la définition de la continuité par $\forall x \forall \alpha \exists \eta > 0 \forall y \ |x - y| < \eta \Rightarrow |f(x) - f(y)| < \alpha$, et exprimer explicitement le module de continuité η en fonction de x et de α, ce qui serait aussi fastidieux et peu utile que détailler explicitement les états d'une machine de Turing. Dans une copie d'étudiant voyant pour la première fois la notion de continuité, on se contentera de dire qu'il s'agit de la composée de fonctions continues (sinus, polynômes, racine carrée…), et éventuellement qu'il n'y a pas de problème de définition de la racine carrée car les arguments sont clairement positifs ; dans un contexte plus expérimenté, on se contentera d'asséner que cette fonction est C^∞ sans plus d'explication.

12. On peut définir une notion de système acceptable de programmation, et montrer le théorème d'isomorphisme de Rogers : entre deux tels systèmes il existe un compilateur bijectif (Machtey et Young 1978, §3.4 ; Rogers 1987, §11.4, 5e illustration, p. 191).

3 Quelques résultats classiques de calculabilité

3.1 Démonstrations formelles

Finalement, le choix du modèle de calcul importe peu, vu qu'ils sont tous équivalents et que dans la plus grande partie des preuves, on n'utilise que quelques propriétés de haut niveau. On se base sur les propriétés suivantes, vérifiées par tous les modèles (fonctions récursives, machines de Turing, etc.) :

- Chaque fonction récursive (totale ou partielle) est représentée par au moins un programme (en fait, on peut montrer que l'ensemble des conditions imposent qu'elle le soit par une infinité de programmes).
- On peut associer à chaque programme P un numéro $\ulcorner P \urcorner$. La fonction calculée par le programme numéro x sera notée φ_x.
- On a un codage des paires d'entiers naturels dans les entiers naturels, noté $\langle x, y \rangle$, que l'on étend aux n-uplets.
- On a un interpréteur universel, de numéro u, tel que $\varphi_u \langle x, y \rangle = \varphi_x(y)$ — autrement dit, si on lance l'interpréteur universel sur le couple (x, y), il retourne le résultat de l'évaluation du programme numéro x sur l'entrée y, ce résultat pouvant être $\perp$, c'est-à-dire la non terminaison.
- On a une fonction récursive totale c telle que $\varphi_{c(x,y)} = \varphi_x \circ \varphi_y$, autrement dit cette fonction réalise la composition des programmes de numéro x et y.

On peut d'ailleurs sans grandes difficultés remplacer les entiers naturels par un autre ensemble dénombrable, par exemple les chaînes de caractères, ce qui permet d'identifier un programme P à son code source $\ulcorner P \urcorner$ (on peut considérer qu'un code source incorrect donne un programme qui rend toujours $\perp$).

On peut montrer que dans un tel système, il y a un algorithme de remplacement d'arguments dans les fonctions : il y a une fonction récursive totale s telle que pour tout i, tout $m \geq 1$, tout $n \geq 1$, pour tout $x_1, \ldots, x_m$ et tout $y_1, \ldots, y_n$,

$$\varphi_{s\langle n,m,i,\langle x_1,\ldots,x_m\rangle\rangle}\langle y_1, \ldots, y_n \rangle = \varphi_i \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle \tag{2}$$

Autrement dit, $s\langle n, m, i\langle x_1, \ldots, x_m\rangle\rangle$ est le codage d'une fonction correspondant à φ_i où l'on a remplacé les m premiers arguments par $x_1, \ldots, x_m$. Ce résultat est parfois nommé « s-n-m » ou « s-m-n ».

Rappelons que, derrière le formalisme, tout ce que l'on dit, c'est qu'il y a des algorithmes de manipulation de programmes qui, à partir de leur code source, impriment le code source de leur composition, ou le code source d'un programme dont l'on a remplacé une partie des arguments. Rien d'extraordinaire, donc, pour quiconque se doute un peu de comment fonctionne un compilateur !

L'existence de la fonction de composition, de la construction « s-n-m », et le fait que toutes les fonctions arithmétiques sont récursives totales, justifie

que dans les démonstrations, on puisse se contenter de définir une fonction f et immédiatement de dire que l'on prend un numéro x pour elle (donc $\varphi_x(y) = f(y)$), voire de dire que ce numéro x est une certaine fonction calculable $F(a, b, \dots)$ où a et b sont des numéros de fonctions intervenant dans la définition de f.

Nous pouvons maintenant prouver des théorèmes intéressants. Nous entendrons par « décider P » nous voulons dire « retourne 1 si P est vrai, 0 sinon ».

Théorème 2 (arrêt de Turing)**.** *Il n'existe pas d'algorithme qui décide, au vu de x, si $\varphi_x(x) \neq \perp$.*

Relevons qu'il ne « suffit » pas de lancer $\varphi_x(x)$ et de regarder si le résultat est $\perp$, tout simplement parce que si $\varphi_x(x) = \perp$, notre algorithme de test serait forcé lui aussi de ne pas terminer. Tout ce que nous pouvons faire ainsi, c'est avoir un algorithme de *semi-décision*, qui répond 1 si la propriété est vrai, et qui répond 0 ou $\perp$ si elle est fausse. Passons maintenant à la preuve.

Démonstration. Supposons qu'il y ait un tel algorithme, soit t un numéro de programme tel que $\varphi_t(x) = 1$ si $\varphi_x(x) \neq \perp$ et 0 sinon. Il y a donc (via composition, s-n-m etc.) un programme numéro a tel que pour tout x $\varphi_a(x) = \perp$ si $\varphi_t(x) = 1$ (ce qui équivaut à $\varphi_x(x) \neq \perp$) et $\varphi_a(x) = 1$ sinon (ce qui équivaut à $\varphi_x(x) = \perp$). Appliquons ce programme en $x = a$. D'après les définitions, si $\varphi_a(a) \neq \perp$ alors $\varphi_a(a) = \perp$, et si $\varphi_a(a) = \perp$ alors $\varphi_a(a) = 1$ (encore un argument diagonal !). Contradiction. □

Corollaire 3. *Il n'existe pas d'algorithme qui décide, au vu de x et de y, si $\varphi_x(y) \neq \perp$.*

L'analogue formel de notre théorème informel « il n'existe pas de langage de programmation qui permette de programmer toutes les fonctions qui terminent » s'obtient facilement :

Théorème 4. *Il n'existe pas d'énumération des fonctions récursives totales, autrement dit il n'existe pas de fonction récursive totale e telle que l'ensemble des fonctions récursive totales s'obtient comme $\{\varphi_e(x) \mid x \in \mathbb{N}\}$.*

Démonstration. Imaginons qu'il y ait une telle fonction φ_e, alors $x \mapsto \varphi_{\varphi_e(x)}(x) + 1$ est une fonction récursive totale. Il existe donc x_0 tel que cette fonction s'exprime comme $\varphi_{\varphi_e(x_0)}$. Prenons $x = x_0$, alors $\varphi_{\varphi_e(x_0)}(x_0) = \varphi_{\varphi_e(x_0)}(x_0) + 1$. Contradiction. □

Théorème 5 (Rice)**.** *Soit S un sous-ensemble des fonctions récursives partielles, non trivial (S est non vide et de complémentaire non vide). Alors il n'existe pas d'algorithme décidant, d'après x, si $\varphi_x \in S$.*

Démonstration. La fonction qui ne termine jamais appartient soit à S soit à son complémentaire. Quitte à remplacer S par son complémentaire, et sans perte de généralité, supposons qu'elle n'appartient pas à S. Soit $\varphi_F \in S$.

Soit φ_x une fonction récursive partielle quelconque. On peut construire la fonction $\varphi_{G(x)}$ qui prend en argument y et exécute d'abord $\varphi_x(x)$, puis, si ce calcul termine, qui retourne $\varphi_F(y)$; et même (s-n-m, composition...) on obtient $G(x)$ comme fonction récursive totale de x.

Il est clair que si $\varphi_x x$ termine, alors $\varphi_{G(x)} = \varphi_F$ en tout point et appartient donc à S, et que si $\varphi_x x = \bot$, alors $\varphi_{G(x)}(y) = \bot$ pour tout y, et donc $\varphi_{G(x)} \notin S$. Autrement dit, nous avons *réduit* le problème de tester l'arrêt de $\varphi_x x$ (indécidable par un algorithme, d'après le Th. 2) à celui de tester si $\varphi_{G(x)} \in S$. Ce second problème est donc indécidable par un algorithme. □

3.2 Impact pratique

Interprétons le théorème de Rice sur les programmes informatiques : il signifie que pour toute propriété (ou *spécification*) P non triviale (non toujours vraie ou toujours fausse) reliant l'entrée et la sortie du programme (par exemple « le programme retourne la liste qu'on lui passe en entrée, triée »), il n'existe aucun algorithme qui dise, au vu du code source d'un programme, si celui-ci vérifie la propriété P. Autrement dit, en termes plus provocants, il impossible d'analyser automatiquement les programmes afin de voir s'ils sont buggués.

Ce théorème suppose toutefois que l'on se place dans un modèle

- ne bornant pas la mémoire : un programme opérant sur une mémoire à N bits, N constante, est un automate fini à au plus 2^N états ; on n'a plus alors affaire à un problème de calculabilité, mais de *complexité* !
- ne bornant pas le temps d'exécution : l'ensemble des comportements possibles d'une machine de Turing en N pas de calcul est borné (certes, de taille exponentielle en N), et donc les propriétés y sont décidables.

Par ailleurs, si ce théorème exclut que l'analyse automatique puisse répondre juste en toute généralité, il n'exclut pas que des analyses automatiques puisse répondre juste sur un très grand nombre de cas rencontrés en pratique. Ainsi, il existe des outils [13] capables de démontrer automatiquement l'absence d'erreurs à l'exécution (dépassements arithmétiques, erreurs de pointeurs, etc.) dans des programmes du monde réel (pilotage d'avion, de machines, etc.) ; simplement, dans certains cas, ces outils n'arrivent pas à montrer la propriété désirée et produisent alors des avertissements au sujet d'éventuels dysfonctionnements du logiciel analysé qui ne peuvent pas arriver en pratique ; diminuer la quantité de tels avertissements (fausses alarmes) est un des buts de la recherche en analyse statique de programmes.

13. Citons notamment Astrée `http://www.astree.ens.fr` `http://www.absint.de/astree`, dont l'auteur était un des développeurs.

De plus, il n'est pas évident qu'un utilisateur industriel ordinaire (nous excluons ici les industries critiques comme l'avionique) ait réellement envie de montrer l'absence de bugs dans les programmes. La plupart des logiciels contiennent de très nombreux bugs, et les industriels ont donc des bases entières de dysfonctionnements à résoudre. Dans ces circonstances, un outil qui signale non seulement des vraies violations de spécification, mais également des fausses alarmes, est peu utile. D'autres catégories d'outils, axés sur la recherche de dysfonctionnements vraiment réalisés (quitte à « oublier » des vrais dysfonctionnements), sont alors pertinentes. [14]

Examinons maintenant le problème de la gestion de la mémoire dans les programmes : dans les langages de programmation modernes (Caml, Java, C#, F#, Haskell,m, Python, PHP, Perl, etc.) et dans le vénérable Lisp, cette gestion est automatique (un dispositif dit « ramasse-miette » ou *garbage collector* récupère la mémoire inutilisée), tandis que dans d'autres langages (C, C++) cette gestion est manuelle : on doit explicitement rendre la mémoire (fonction free() ou opérateur **delete**) que l'on a explicitement allouée (fonction malloc() ou opérateur **new**), sinon elle est « perdue » et on a une fuite mémoire (ce qui peut se traduire, en pratique, par une quantité de mémoire utilisée par l'application qui croît avec le temps jusqu'à rendre le système lent, voire inutilisable, ou à provoquer l'arrêt de l'application par manque de mémoire). On entend parfois dire que la *garbage collection* supprime le problème des fuites mémoire ; c'est inexact, et nous allons voir pourquoi.

Soit P un programme quelconque où la variable ptr n'intervient pas. Considérons le programme :

```
1  char *ptr = malloc(1000);
2  P
3  *ptr = 1;
```

Quand un *garbage-collector* idéal pourrait-il au plus tôt libérer le bloc pointé par ptr ? A priori, le programme a besoin ligne 3 pour pouvoir écrire dedans, donc la libération ne pourrait intervenir qu'après cette ligne. . .mais ceci suppose que P termine ! Si P ne termine pas, il pourrait libérer le bloc après la ligne 1. Autrement dit, placer optimalement les points de libération de mémoire impose de résoudre le problème de l'arrêt !

Soit G_0 l'ensemble des données qu'un *garbage collector* idéal libérerait en un point de programme fixé, autrement dit les données qui ne serviront plus dans aucune exécution possible de la suite du programme. Les *garbage collectors* réels sont tous *conservateurs* : ils libèrent $G \subsetneq G_0$, et il est possible de produire des fuites de mémoire avec des données dans $G_0 \setminus G$.

14. Citons par exemple l'outil de Coverity `http://www.coverity.com/`

3.3 Retour sur les équations diophantiennes

Le problème de savoir si un système d'équations diophantiennes a une solution se ramène au problème de l'arrêt de Turing : étant donné un système S, on construit un programme qui essaye toutes les valeurs des variables de S et s'arrête dès qu'il trouve une solution, ce programme s'arrête donc si et seulement si le système a une solution.

Les travaux de Martin Davis, Hilary Putnam, Julia Robinson et Yuri Matiyasevitch ont abouti à la réciproque : il existe un algorithme qui, étant donné un programme φ_x, construit un système d'équations diophantiennes qui a des solutions si et seulement si $\varphi_x(x) \neq \perp$; plus généralement il existe un système d'équations diophantiennes dont la projection de l'ensemble des solutions donne exactement $\{x \mid \varphi_x(x) \neq \perp\}$. Ce résultat a nécessité d'importants développements, le lecteur pourra se reporter à Matiassevitch 1995.

Nous avons dit que le problème des équations diophantiennes devenait décidable si l'on regarde leurs solutions *réelles*. Nous allons maintenant voir que le problème redevient indécidable si on permet non seulement d'avoir des (in)équations polynomiales, mais également des équations faisant intervenir la fonction sinus (ou cosinus, peu importe).

Remarquons que π se définit comme l'unique solution de $\sin(\pi) = 0 \wedge 3 < \pi < 22/7$ et qu'un réel x est entier si et seulement si $\sin(\pi x) = 0$. Étant donné un système S en variables entières $x, y, \ldots$ nous pouvons obtenir en variables réelles en lui ajoutant $\sin(\pi) = 0 \wedge 3 < \pi < 22/7 \wedge \sin(\pi x) = 0 \wedge \sin(\pi y) = 0 \wedge \ldots$, dont les solutions en $x, y, \ldots$ sont exactement les solutions entières de S. Le problème des équations diophantiennes se ramène donc au problème des équations réelles mixtes polynômes et sinus.

Ce genre de réductions pose certaines limites aux capacités de simplification et de résolution d'équations des systèmes de calcul formel. Soulignons que, comme pour l'analyse de programmes, ce n'est pas parce qu'il y a un résultat d'impossibilité dans le cas général qu'il est impossible de construire des logiciels qui résolvent le plus souvent les problèmes qui nous intéressent !

4 Conclusion

En partant d'un problème relevant apparemment de la seule arithmétique, nous sommes parvenus à des problèmes de gestion de la mémoire dans les langages de programmation, sujet apparemment technique et sans rapport avec les mathématiques. Nous aurions pu, également, évoquer les liens très forts entre la calculabilité et les théorèmes d'incomplétude de Gödel, la possibilité d'énumérer algorithmiquement les formules démontrables jouant un rôle essentiel dans la preuve de ces théorèmes. . .mais cela devrait faire l'objet d'un autre article.

On pourra reprocher que la notion de calculabilité utilisée est très théorique : elle fait fi de la possibilité pratique de réaliser les calculs évoqués. Ainsi, la fonction d'Ackermann est calculable, mais à des coûts en temps et en espace (quantité de mémoire) vite prohibitifs, ne serait-ce que pour stocker le résultat. Lorsque nous faisons intervenir ces coûts, nous ne parlons plus de *calculabilité*, mais de *complexité*, qui en général étudie leur comportement asymptotique. Ainsi, on s'intéressera à savoir si un problème peut être résolu par une machine de Turing en temps, en espace polynomial en la longueur de son entrée...C'est dans ce cadre que s'exprime le fameux problème $\mathrm{P} \subsetneq \mathrm{NP}$, un des « problème du millénaire » mis à prix par l'Institut Clay,[15] mais c'est une autre histoire !

Références

Cox, David A., John B. Little et Donal O'Shea (2007). *Ideals, Varieties, And Algorithms. An Introduction to Computational Algebraic Geometry And Commutative Algebra*. 3e éd. Undergraduate Texts in Mathematics. Springer. ISBN : 0387356509.

Hilbert, David (1900). « Mathematische Probleme. Vortrag, gehalten auf dem internationalen Mathematiker-Kongreß zu Paris 1900 ». Allemand. Dans : *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, p. 253–297. URL : http://resolver.sub.uni-goettingen.de/purl?GDZPPN002498863.

— (1902). « Mathematical problems. Lecture delivered before the International Congress of Mathematicians at Paris in 1900 ». Anglais. Trad. par Mary Winston Newson. Dans : *Bulletin of the American Mathematical Society* 8. Traduction de Hilbert 1900, p. 437–479. DOI : 10.1090/S0002-9904-1902-00923-3. URL : http://en.wikisource.org/wiki/Mathematical_Problems.

Knuth, Donald Ervin (1998). *The Art of Computer Programming. Seminumerical Algorithms*. 3e éd. T. 2. Addison-Wesley. ISBN : 0-201-89686-4.

Lamé, Gabriel (1844). « Note sur la limite du nombre des divisions dans la recherche du plus grand commun diviseur entre deux nombres entiers ». Dans : *Compte rendu des séances de l'Académie des Sciences* 19.18, p. 867–870. URL : http://gallica.bnf.fr/ark:/12148/bpt6k2978z/f867.

15. Le *Clay Mathematics Institute*, fondé par l'homme d'affaires américain Landon T. Clay, est dédié à l'accroissement et à la dissémination des connaissances en mathématiques. Il finance la recherche dans ce domaine de diverses façons, notamment par des bourses de recherche postdoctorales, mais il est principalement connu pour avoir, en l'an 2000, annoncé une liste de sept « grand problèmes du millénaire » dont la résolution emporterait pour chacun un prix d'un million de dollars. Jusqu'à présent, seul un de ces problèmes a été résolu : la conjecture de Poincaré, par le mathématicien russe Grigoriy Perelman. http://www.claymath.org/millennium/

MACHTEY, Michael et Paul YOUNG (1978). *An introduction to the general theory of algorithms*. Computer science library, Theory of computation series. New York : North-Holland. ISBN : 044400226X.

MATIIASSEVITCH, Youri (1995). *Le dixième problème de Hilbert. son indécidabilité*. Trad. par Patrick CEGIELSKI et Denis RICHARD. Axiomes. Masson. ISBN : 2-225-84835-1.

ROGERS Jr., Hartley (1987). *Theory of recursive functions and effective computability*. MIT Press. ISBN : 0-262-68052-1.

SMITH, Peter (2007). *An introduction to Gödel's theorems*. Cambridge University Press. ISBN : 978-0-521-85784-0.

TAYLOR, R. Gregory (1998a). *Ackermann's Function is not Primitive Recursive*. URL : `http://home.manhattan.edu/~gregory.taylor/thcomp/pdf-files/ackerman.pdf`.

— (1998b). *Models of Computation and Formal Languages*. Oxford University Press.