MATHÉMATIQUES ET SCIENCES HUMAINES

BERNARD JAULIN

Sur un aspect du calcul

Mathématiques et sciences humaines, tome 14 (1966), p. 1-8

<http://www.numdam.org/item?id=MSH_1966__14__1_0>

Bernard JAULIN

SUR UN ASPECT DU CALCUL

Il est bien connu qu'un programme de calculatrice est un schéma de définition d'une semi-fonction récursive* et réciproquement qu'une telle machine ne peut calculer que des semi-fonctions de cette nature si l'on suppose que sa mémoire est infinie (en ce sens que ses éléments de mémoire pourront contenir un nombre aussi grand que l'on veut). En ne précisant pas plus la notion de calculatrice ni celle de langage de programmation cette réciproque correspond, pour cette idée intuitive d'algorithme de calcul, à un des aspects de la thèse de A. Church. Compte tenu de ce fait il est alors facile d'expliciter des modèles "théoriques" simples de calculatrices dans les termes desquels il sera possible d'expliciter le calcul de n'importe quelle semi-fonction effectivement calculable. Par exemple supposons que l'on dispose d'une quantité aussi grande que l'on veut de registres de "mémoire" dans chacun desquels il est possible d'enregistrer un nombre quelconque. Pour modifier le contenu de ces registres on dispose des 4 types suivants d'instruction.

A (r) : augmenter de 1 le nombre contenu dans le registre n°r
D (r) : diminuer de 1 le nombre contenu dans le registre n°r
E (r_1, r_2) : Porter dans le registre de numéro r_1, le nombre contenu dans le registre de numéro r_2.

(*) Les semi-fonctions recursives sont des semi-fonctions de $\mathbb{N}^p$ dans $\mathbb{N}$. ($\mathbb{N}$ désignant l'ensemble des entiers) qui sont définis de la façon suivante - f: $\mathbb{N}^p \longrightarrow \mathbb{N}$ est une semi fonction recursive si l'on peut la définir à partir des fonctions "successeur", suc, "projections" pr_i^p, nulles C^p, en utilisant les opérations de superposition, recurrence simple, minimilisation.

$suc(x) = x + 1$

$pr_i^p(x_1, \ldots, x_p) = x_i$

$C^p(x_1, \ldots, x_p) = 0$

- f est défini par *superposition* à partir de g, h_1, ... h_n si
  $f(x_1, \ldots, x_m) = g(h_1(x_1 \ldots, x_m), \ldots, h_n(x_1 \ldots x_m))$

- f est défini par *récurrence simple* à partir de g et h
  si $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$,
  $f(x_1, \ldots, x_n, y) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$.

- f est défini par *minimilisation* à partir de g si
  $f(x_1, \ldots, x_n)$ = le plus petit y tel que $g(x_1, \ldots, x_n, y) = 0$.

(Nota: On voit que la dernière opération, celle de minimilisation, peut conduire, à partir d'une fonction g à une semi fonction f, c'est-à-dire une application non partout défini) - (Cf(1) pour plus de détails).

$T\ (q_1, q_2)\ (r)$: Dans un programme, c'est-à-dire une suite finie d'instruction qui chacune porte un nom, q_i, lorsque l'on rencontre cette instruction, on regarde le nombre contenu dans le registre de numéro r, si celui-ci est zéro, on effectuera après l'instruction de nom q_1, sinon celle de nom q_2.

On vérifiera trivialement que pour toute semi-fonction récursive $\varphi: \mathbb{N}^p \to \mathbb{N}$ définie de façon arithmétique selon la définition de Gödel-Herbrand donnée ci-dessus, on peut construire un programme qui ne comportera que des instructions d'un de ces 4 types et telle que si $x_1, \ldots, x_p$ sont les nombres placés initialement dans les registres n° 1 à p, la machine ne s'arrêtera pas si $f\ (x_1, \ldots, x_p)$ n'est pas défini et sinon s'arrêtera avec $y = f\ (x_1, \ldots, x_p)$ dans le registre n° 1.

- On peut alors, à partir de ce type d'algorithme, imaginer ceux équivalents du point de vue du calcul comme les systèmes de Post, les algorithmes de Markov, les machines de Turing*. Pour ces derniers, nés de cette analyse métamathématique bien connue des systèmes formels, et de façon plus nette pour les machines de Turing où l'on peut vraiment parler de programme de fonctionnement au sens traditionnel du terme, les instructions que l'on utilise pour construire un programme de calcul d'une semi-fonction récursive se situent au niveau des chiffres, c'est-à-dire des symboles utilisés pour représenter les nombres que l'on manipule. De plus dans ces divers types d'algorithme on utilise un seul registre de mémoire, appelé "bande" dans le cas des machines de Turing. Notons que cette unicité de registre de mémoire implique l'existence, dans l'alplabet que l'on utilise pour représenter les nombres, d'un signe supplémentaire, par exemple $\#$, qui servira à séparer, puisque l'on "manipule" exclusivement des séquences de symboles, les représentations de deux nombres.
- Dans tous les cas, c'est-à-dire quelque soit la forme particulière de l'algorithme que l'on utilise pour exprimer le calcul d'une semi-fonction récursive, que ce soit une liste d'instructions d'un langage de programmation ou bien les productions d'un système de Post, etc..., la présentation du calcul d'une fonction calculable est donnée par un nombre, puisqu'elle peut toujours être faite explicitement par une écriture finie sur un alphabet fini.

On dispose alors du premier résultat suivant, équivalent à un résultat dû à Rice.

Résultat 1.

Etant donné un langage de programmation suffisamment riche (en ce sens qu'il permet d'exprimer le calcul de n'importe quelle semi-fonction récursive), alors pour tout ensemble A de semi-fonctions récursives (effectivement calculable), différent de l'ensemble vide ou de l'ensemble de toutes les semi-fonctions récursives, l'ensemble B de tous les programmes permettant de calculer une semi-fonction de A est non récursif.

(On précise cet énoncé dans l'appendice où en outre on pourra trouver une démonstration).

Ce résultat signifie, puisque comme nous venons de le voir un programme dans un langage de programmation peut-être considéré comme un nombre, ce qui correspond

(*) On trouvera les définitions de ces termes dans (9), par exemple.

d'ailleurs à la réalité tangible dans les calculatrices modernes où l'on enregistre un programme dans une mémoire, qu'il n'est pas possible de construire un programme dans ce langage de programmation, qui opérant sur un programme quelconque comme donnée (sur le nombre représentant ce programme), nous permettrait de savoir si ce programme exprime le calcul d'une semi-fonction de l'ensemble A que l'on considère. Par exemple l'ensemble infini dénombrable des programmes qui permettront de calculer le p.p.c. m. de deux nombres est dans tout langage de programmation non récursif. De même si A est l'ensemble des semi-fonctions récursives définies partout sur un sous ensemble $C \subseteq \mathbb{N}$, l'ensemble B des programmes permettant de calculer ces semi-fonctions n'est pas récursif. En l'occurence ceci signifie dans le cas pratique où C serait la capacité de mémoire d'une calculatrice réelle, qu'il ne nous est pas possible d'écrire un programme qui nous permettrait de savoir pour un programme quelconque P écrit dans un langage de programmation si ce programme P "s'arrêtera" pour n'importe quelle "donnée" que l'on peut enregistrer dans la machine.

Remarquons ici que ce résultat de Rice tel que nous l'avons énoncé semble dépendre d'une part du langage de programmation et d'autre part de la façon que l'on a choisi de représenter un programme par un nombre. En fait il n'en est rien comme l'on pourrait le montrer facilement. Il faudrait par exemple définir de façon précise la notion de numérotation principale (cf. Annexe) ou encore définir de façon intrinsèque la notion de fonction récursive dans un monoïde libre ce qui ne présente pas de difficulté. D'autre part, de ce résultat on peut déduire trivialement d'autres résultats classiques du type de ceux des deux exemples que nous avons donné: Ceux relatifs aux divers problèmes de halte des programmes (dans l'exemple précédent faire $C = \{0\}$ ou $C = \{a\}$ ou $C = \mathbb{N}$ par exemple) aux problèmes sur les diverses équivalences que l'on peut définir entre programmes* (cf. appendice) etc.... La portée pratique pour l'utilisation des calculatrices de ces trop nombreux résultats négatifs est discutable: Ce sont des exemples naturels dans ce cadre, de fonctions de $\mathbb{N}^p$ dans $\mathbb{N}$ qui ne sont pas programmables. Or de telles fonctions il y en a une infinité non dénombrable!...

Un aspect important de ces problèmes qui enlève à leur étude un caractère pratique est leur généralité. Reprenons par exemple le cas du problème de la halte sur un sous-ensemble $C \subset \mathbb{N}$. Il s'agit de l'étude de la fonction :

$$\theta(x) = \begin{cases} 1 & \text{si le programme de numéro } x \text{ s'arrête pour toutes les valeurs} \\ & \text{des arguments } x \in C \\ 0 & \text{sinon.} \end{cases}$$

Il serait naturel, si l'on interprète C comme étant la capacité de mémoire d'une calculatrice réelle d'étudier la semi-fonction θ' définie pour les programmes dont les numéros sont inférieurs à x c'est-à-dire :

$$\theta'(x) = \theta(x) \text{ si } x \in C \quad \theta'(x) \text{ indéfini sinon.}$$

(*) Signalons qu'en transposant ces résultats pour des formes d'algorithme ayant un léger caractère algébrique comme les systèmes de Post, etc...; on pourra arriver à des résultats d'indécidabilité pour des problèmes que l'on pourrait se poser en algèbre. Par exemple il n'existe pas d'algorithme valable pour tous les systèmes finis de relations entre les mots d'un monoïde libre permettant de décider, étant donné un tel système, si le monoïde quotient associé est fini etc... (Ce résultat reste vrai pour les présentations de groupe et est un cas particulier d'un théorème plus général dû à M. O. Rabin).

On voit donc apparaître la nécessité, si l'on veut étudier de façon théorique certains problèmes pratiques, d'introduire ou de définir précisément les caractéristiques de plusieurs notions comme l'importance de la mémoire utilisée par un programme, sa rapidité, ... etc.... Nous regrouperons maintenant sous le terme générique de complexité ces différentes notions, car, et nous le verrons, elles se laissent caractériser dans le cadre général des fonctions récursives à l'aide des mêmes axiomes. Il est alors utile de faire une remarque afin d'éviter des confusions. Un problème de calcul c'est la programmation dans un certain langage d'une semi-fonction récursive. La notion de complexité d'un problème n'a donc pas de sens si l'on ne spécifie pas en même temps la forme de l'algorithme (ou langage de programmation) que l'on utilisera pour calculer la semi-fonction récursive définie par le problème. Une fois ceci acquis, quelles seront les mesures de complexité que l'on pourra associer au calcul d'une semi-fonction récursive pour un algorithme d'un certain type? On peut en concevoir facilement un grand nombre selon la forme de l'algorithme que l'on considère. Par exemple si l'on effectue le calcul avec une machine de Turing ou un algorithme de Markov, ou un système de Post, on pourra mesurer la quantité de "mémoire" utilisée sur le "ruban", c'est-à-dire la longueur maximum des mots considérés au cours du calcul, ou bien encore pour les algorithmes de calcul tels que celui présenté au début de cet exposé, le nombre de fois où l'on effectuera une opération logique du type $T(q_1, q_2)(r)$, ou encore le nombre d'instructions effectuées, etc...etc.... Le premier exemple est on le voit une tentative pour mesurer la mémoire, le dernier est relatif à la rapidité, le second la complexité logique du programme etc...

L'on remarque alors que dans chacune de ces situations, pour une forme d'algorithme donnée, si i est le numéro de l'algorithme ou du programme permettant de calculer la semi-fonction f_i, on mesure la complexité de ce calcul par une semi-fonction récursive F_i, tel que :

1) $\forall$ n, f_i (n) défini $\Longleftrightarrow$ F_i (n) défini.

2) $\{(i, n, m) \ / \ F_i(n) = m\}$ est récursif. Cette dernière condition signifiant que étant donné un programme de numéro i, et un argument n on peut reconnaître si m est la valeur de la complexité du programme i appliqué à cet argument n. L'étude de ce système d'axiomes introduit par M. Blum a été faite par cet auteur. On peut en déduire un certain nombre de conséquences et en particulier le résultat suivant :

Résultat 2. (M. Blum. Thèse du M.I.T.- 1965)

Soit r une fonction récursive de 2 variables, alors il existe une fonction récursive f prenant les valeurs 0 et 1 avec la propriété que pour tout programme i permettant de la calculer, il existe un autre programme j écrit dans le même langage de programmation permettant de calculer f et tel que $F_i(n) > r(n, F_j(n))$ pour presque tout n, (c'est-à-dire sauf pour un ensemble fini de valeurs de n).

On voit l'importance dans ce résultat de l'expression "pour presque tout n". En effet si pour la fonction r on prend $r(x,y) = x^y$, ce résultat affirme l'existence d'un problème tel que pour tout programme i associé à ce problème il existe un autre programme j beaucoup moins complexe pour presque tout n, c'est-à-dire tel que $F_i(n) > n^{F_j(n)}$ pour presque tout n.

Si l'on remplace le mot complexe par rapidité ou quantité de mémoire utilisée

au cours d'un calcul, on se rend compte aisément de la signification de ce résultat.

Une démarche complémentaire à celle de Blum consiste à étudier la complexité que l'on pourrait définir sur les présentations des calculs c'est-à-dire sur la forme même des algorithmes. Par exemple si l'on associe à chaque programme p, un nombre m (p) qui exprime la quantité d'instructions du programme, etc..., on pourra associer à une semi-fonction f. le nombre c (f) égal au plus petit des m (p), lorsque p parcourt l'ensemble des programmes permettant de calculer f. Cette seconde façon d'aborder l'étude des caractéristiques des fonctions calculables mécaniquement, moins intéressante que la précédente, conduit cependant à quelques résultats négatifs qui sont des corollaires du résultat (1) de Rice. De façon plus précise soit n: $\mathbb{N} \rightarrow {}^{s}F_{R}^{(1)}$ une énumération principale (cf. annexe) de l'ensemble des semi-fonctions récursives.

(n^{-1} (f), $f \in {}^{s}F_{R}^{(1)}$ désignant par exemple l'ensemble des numéros des programmes permettant de calculer f) et soit m: $\mathbb{N} \longrightarrow \mathbb{N}$ une fonction récursive exprimant la pseudo-complexité des programmes (m (p) peut-être par exemple le nombre d'instructions logiques du programme de numéro p). On peut alors poser:

$$\text{Complexité de } f = c(f) = \min_{p \in n^{-1}(f)} m(p).$$

Dans ces conditions on peut se demander si l'application C*: $\mathbb{N} \longrightarrow \mathbb{N}$ définie par C* (p) = C (n (p)) est calculable. c'est-à-dire s'il est possible de calculer, étant donné un programme de numéro p, la pseudo-complexité minimum des programmes équivalents à celui donné.

Il apparaît clairement que les résultats que l'on peut obtenir dépendront d'une part de la numérotation choisie n, des semi-fonctions récursives et d'autre part des caractéristiques de fonctions de pseudo-complexité m que l'on veut étudier.

En outre on peut essayer, si la fonction C* n'est pas calculable ce qui est le cas pour les pseudo-complexités m intéressantes (cf. infra), d'approximer cette fonction par une fonction calculable. Ceci pourrait être fait de la manière suivante:

a) Une fonction C*: $\mathbb{N} \longrightarrow \mathbb{N}$ est approximée linéairement par une fonction C : $\mathbb{N} \longrightarrow \mathbb{N}$ si il existe M tel que / C* (n) - C (n) / < M - pour tout n.

b) Une fonction C* : $\mathbb{N} \longrightarrow \mathbb{N}$ est approximée logarithmiquement par une fonction C_a : $\mathbb{N} \longrightarrow \mathbb{N}$ si il existe M tel que $/ \frac{C^*(n)}{C_a(n)} / < M$ pour tout n.

On peut alors énoncer le résultat suivant dû à Kloss [4].

Résultat (3)

Si m est une fonction récursive (de pseudo-complexité) telle que m^{-1} (n) est fini pour tout n; alors la fonction de complexité C* : $\mathbb{N} \longrightarrow \mathbb{N}$ est non récursive. De plus cette fonction ne peut pas être approximée ni linéairement, ni logarithmiquement par une fonction récursive.

On trouvera en annexe une démonstration simplifiée de ce résultat.

Conclusion :

L'énoncé et la présentation des trois résultats 1 et 3 ne constituent pas un résumé exhausif des travaux faits dans ce domaine. Ce sont des résultats généraux qu'il est utile de connaître avant d'aborder plus en détail les nombreux problèmes qui peuvent se poser. Leur caractère négatif tient à leur généralité, on peut cependant espérer, mais nous n'avons pas la place pour le montrer ici qu'une restriction de cette généralité permet l'étude de problèmes pour lesquels il est possible d'obtenir des résultats positifs et même de définir explicitement des algorithmes de calcul assez général et intéressant pour la pratique des calculatrices électroniques.

ANNEXE : DEMONSTRATION DES RESULTATS DE RICE-USPENKHI ET DE KLOSS:

(La démonstration du résultat de M. Blum, trop technique et un peu longue ne sera pas reproduite ici).

Les démonstrations données sont informelles, il sera facile de les écrire avec toute la rigueur propre à une démonstration.

Soit $n : \mathbb{N} \longrightarrow sF_R^{(1)}$ une énumération principale des semi-fonctions récursives, (n^{-1} (f) est par exemple l'ensemble des numéros des programmes permettant de calculer f). On écrira $n(p) = f_p$ (la semi-fonction calculée par le programme de numéro p). Cette énumération possède la propriété caractéristique suivante: la semi-fonction $\varphi(p,y) = f_p(y)$ est récursive. (Cette semi-fonction φ est appelée de façon traditionnelle une semi application récursive universelle pour les semi applications récursives à 1 variable).

On montre d'abord que l'ensemble $D = \{ n / \varphi(n,n) \text{ est défini} \}$ est non récursif. En effet s'il l'était $\complement D$ le serait donc il existerait n_0 tel que $\complement D = \{ y / \varphi(n, y) \text{ soit défini} \}$. On aurait alors $n \in D \Longleftrightarrow \varphi(n_0, n_0)$ défini $\Longleftrightarrow n_0 \in \complement D$ ce qui est contradictoire. Donc D n'est pas récursif. Pour le résultat de Rice, on va montrer que si B est récursif (voir l'énoncé) alors D le serait. Pour ceci on associe à tout nombre x, un nombre k (x) tel que $k(x) \in n^{-1}(f)$ ou $f \in A$ si $\varphi(x, x)$ est défini et sinon $k(x) \in n^{-1}(u^*)$ où u^* est la semi-fonction récursive définie nulle part. Cette application $k : \mathbb{N} \longrightarrow \mathbb{N}$ va être construite explicitement et sera donc récursive. Si l'on considère alors l'application g o k ou g est la fonction caractéristique de B, on constate dans ces conditions que g o k est la fonction caractéristique de D :

$$g \circ k(x) = \begin{cases} 1 \text{ si } k(x) \in B \Longleftrightarrow \varphi(x,x) \text{ défini} \\ 0 \text{ si } k(x) \notin B \Longleftrightarrow \varphi(x,x) \text{ non défini} \end{cases}$$

Par suite si g était récursive, D serait récursif d'où la contradiction.

Il reste à construire k. On procède ici pour cette construction de façon heuristique. A tout nombre x on associe le programme de calcul suivant que l'on exprime dans le langage de programmation de la Page 1.

q_1 : A (2)
q_x : A (2)
} on place x dans le registre n° 2

q_{x+1} : A (3)
q_2 n : A (3)
} on place x dans le registre n° 3

q_{2n+1} : $\varphi[2,3] \longrightarrow 4$: on calcule $\varphi(x,x)$ le nombre x étant placé dans les registres 2 et 3.

q_{2n+2} : $f[1] \longrightarrow 1$: on effectue le calcul d'une semi-fonction f, f quelconque, f $\in$ A, sur l'argument y placé dans le registre n°1.

Il apparaît clairement que ce programme de numéro k (x) est celui d'une semi-fonction répondant la condition énoncée ci-dessus: Si $\varphi(x,x)$ est défini, ce programme calcule f $\in$ A, si $\varphi(x,x)$ n'est pas défini, alors pour tout y, le calcul défini par ce programme ne s'arrête pas. On a donc montré le résultat de Rice et Uspenkhi.

- Pour vérifier le résultat de Kloss on va constater que l'on se trouve dans les conditions d'application de celui de Rice et l'on en déduira une contradiction. En effet si C* est récursive et si $C^*(n) = y_o$ alors $C^{*-1}(y_o) = B$ est récursif. Or si $A = \left\{ f \in {}^{s}F_R \;/\; c(f) = y_o \right\}$, $B = n^{-1}(A)$. On va montrer que $A \neq \emptyset$, est différent de ${}^{s}F_R$, on en déduira d'après le résultat de Rice que B est non récursif, d'où la contradiction. En effet si m est à domaine fini $C = \left\{ p/m(p)=y_o \right\}$ est fini, et $A \subset n(C)$ car n (C) est l'ensemble des semi-fonctions ayant une complexité inférieure à y_o. n (C) étant fini, A est non vide et fini donc différent de ${}^{s}F_R$. On pourra procéder de la même façon pour montrer que toute fonction c_a, c_e approximant de façon linéaire ou algorithmique la fonction c, ne peut pas être calculable.

Bibliographie

(1) LACOMBE - Fonctions récursives et applications. Bulletin de la Société Mathématique de France.
Gauthier Villars (1962).

(2) USPENKHI (V.A.) - Lectures on computable functions. Matematika Logika i Osnovanija Matematiki.
Fizmatgiz, Moscow (1960).

(3) ARBIB (M.A.) and BLUM (M.) - Machine dependance of degree of difficulty.

(4) KLOSS - The definition of complexity of algorithms. Soviet mathematics, volume 5 n° 4, (july-august 1964).

(5) MARKOV (A.A.) - Normal algorithms which computes boolean functions. Soviet mathematics, volume 5 n° 4, (july-august 1964).

(6) GRZEGORCZYK (Andrej) - Some classes of récursive functions. Rozprany Matematycne, Warsawa (1953).

(7) CLEAVE (J.P.) - A hierarchy of primitive recursive functions. Zeitsch f. Math. Logik und Grunalagen d. Math.

(8) RITCHIE (R.) - Classes of recursive functions of predictable complexity. Thèse - Université de Princeton.

(9) DAVIS - Compatability and Unsolvability (Mac graw Hill).

(10) MARKOV (A.A.) - Theory of algorithms. (Distribué par Oldbourne Press).

(11) TRAHTENBROT (B.A.) - Algorithmes et Machines à calculer (Dunod).

(12) SMULLGAN - Theory of formal systems (Princeton Un. Press).

Pour une introduction à la théorie des algorithmes, voir (11), et pour plus de détails sur cette théorie (1), (2), (9), (10) et (12).